UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

A Geometric Control Strategy for Unmanned Aerial Systems

Permalink

https://escholarship.org/uc/item/4pt6j38f

Author

Lamb, Zachary

Publication Date

2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A GEOMETRIC CONTROL STRATEGY FOR UNMANNED AERIAL SYSTEMS

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL & COMPUTER ENGINEERING

by

Zachary O. Lamb

June 2023

The thesis of Zachary O. Lamb
is approved:

Ricardo Sanfelice, Chair

Professor Gabriel Elkaim

Professor Dejan Milutinović

Peter Biehl
Vice Provost and Dean of Graduate Studies

Copyright © by

Zachary O. Lamb

2023

Table of Contents

List of Figures **vi**

List of Tables **viii**

Abstract **ix**

Dedication **x**

Acknowledgments **xi**

1 Introduction **1**

I Developing a Quadrotor Control Platform **6**

2 Basics of Quadrotor Flight Control **7**

3 ST Drone Platform **14**

3.1 Hardware 15
3.1.1 Accessories 16
3.1.2 Flight Controller Unit 18
3.2 Software 25
3.2.1 Attitude Commands 26
3.2.2 Attitude Heading Reference System (AHRS) 28
3.2.3 PIV Control 29
3.2.4 Motor Mixing 33
3.3 Drawbacks 35

4 ST Drone Platform Extension **37**

4.1 Hardware Accessories 37
4.1.1 Vibration Damping Gel 38
4.1.2 Mounted Battery Slot 38
4.2 Communication Requirements 39

4.2.1 Low Latency Communication Link . . . 41
4.2.2 Real Time Data Collection . . . 43
4.2.3 Onboard Data Modification . . . 45
4.3 Safety Requirements . . . 46
4.3.1 Mode Control Algorithm . . . 47

5 Ground Control 51
5.1 Position Control . . . 52
5.1.1 Verifying Attitude Control Performance . . . 53
5.1.2 PID Position Feedback Control . . . 56
5.1.3 Results . . . 62
5.2 Xbox Control . . . 64

II Geometric Estimation & Control 67

6 Theory & Simulation 68
6.1 Quadrotor Model . . . 70
6.1.1 Translational Dynamics . . . 75
6.1.2 Rotational Dynamics . . . 77
6.1.3 Complete Equations of Motion . . . 78
6.1.4 Model Validation . . . 79
6.2 Closed Loop Quadrotor Position Control . . . 83
6.2.1 Implementation . . . 84
6.3 Geometric Estimation . . . 86
6.3.1 Nonlinear Complementary Filtering on $SO(3)$ . . . 87
6.3.2 Implementation and Validation . . . 101
6.4 Geometric Control . . . 101
6.4.1 Controller Design . . . 102
6.4.2 Matrix Log Approximation . . . 106
6.5 Results . . . 110
6.5.1 Simple Dynamical System Simulation . . . 111
6.5.2 ST Drone Simulation . . . 114

7 Hardware Implementation 118
7.1 Geometric Estimator . . . 119
7.2 Geometric Controller . . . 122
7.2.1 Inertia/Torque Solution . . . 124
7.2.2 Large Steady State Error Solution . . . 124
7.2.3 Noisy Ω_d Solution . . . 125
7.2.4 Results . . . 126

8 Conclusion & Future Work 129

Bibliography **132**

A Attitude Heading and Reference System Using Quaternions **138**

B Attitude Heading and Reference System Using DCM **141**

C HC12 Code **143**

D Linear Algebra Functions in C **148**

List of Figures

2.1 Example Quadrotor Dynamics: v represents the current velocity vector of the drone, F_d represents the drag force, F_g represents the force due to gravity, F_1, F_2, F_3, F_4 represent the four forces generated by each motor respectively, and $\tau_1, \tau_2, \tau_3, \tau_4$ represent the four torques generated ny each motor respectively. 8
2.2 Motor Mixing Example from ST Drone Website 11
2.3 Standard Quadrotor Control Architecture Example 12

3.1 ST Drone 15
3.2 ST Drone Hardware Accessories 16
3.3 ST Drone Motor Spin Configuration 17
3.4 ST Drone FCU Components - Components which currently exist (shaded blue), components which are supported but do not exist (shaded grey with black outline). 19
3.5 BLE Communication Flow 22
3.6 ST Drone MCU Peripherals 24
3.7 ST Drone FCU Components Labeled 25
3.8 ST Drone Software 26
3.9 PIV Control Architecture 30
3.10 ST Drone Motor Configuration 33

4.1 Kyosho Vibration Damping Gel Pad 39
4.2 Battery Mount Solution 40
4.3 HC12 Radio Module 41
4.4 HC12 Connection Diagram 41
4.5 Bluetooth Write Time 42
4.6 HC12 Write Time 43
4.7 Mode Control State Machine 48

5.1 Attitude Estimate Comparison - MOCAP vs ST AHRS 54
5.2 PID Attitude Command Tracking 56
5.3 Infrared Markers Attached to ST Drone 57

5.4 Motive Software With Rigid Body Defined 57
5.5 Complete Closed Loop System Description 58
5.6 Closed Loop Position Control 59
5.7 GCS Position Control 60
5.8 ST Drone Attitude Control 61
5.9 Position Point Tracking 63
5.10 Position Circle Tracking 64
5.11 XBox Controller Commands 65

6.1 Quadrotor Forces and Moments 71
6.2 Experiment 1: Hover State 80
6.3 Experiment 2: Positive Roll 81
6.4 Experiment 2: Positive Pitch 82
6.5 Experiment 4: Positive Yaw 83
6.6 Closed Loop Position Control 85
6.7 Position Control Response 85
6.8 Complementary Filter [23] 96
6.9 Passive Complementary Filter [23] 98
6.10 Top: Mahony roll estimate (red) compared to MOCAP truth (blue). Bottom: Error between Mahony and MOCAP. Mean error = 0.56° . . . 102
6.11 Top: Mahony pitch estimate (red) compared to MOCAP truth (blue). Bottom: Error between Mahony and MOCAP. Mean error = 0.34° . . . 103
6.12 Power Series Log Approximation Experiment 109
6.13 Rodriguez Method Experiment 109
6.14 Geometric Attitude Tracking 113
6.15 Geometric Attitude Tracking Error 113
6.16 Closed Loop System with Geometric Control 115
6.17 Geometric Position Control Results 116
6.18 Geometric Attitude Control Results 117

7.1 Mahony DCM Hardware Estimation Comparison 121
7.2 Geometric Attitude Control Hardware Roll Results 126
7.3 Position Tracking Performance with Geometric Attitude Controller . . . 127
7.4 Position Tracking Performance with Geometric Attitude Controller . . . 128

List of Tables

3.1 BLE Readable Characteristic UUID's 22

4.1 BLE IMU Characteristic Data Alignment 44
4.2 BLE Environmental Characteristic Data Alignment 44
4.3 BLE Arming Status Characteristic Data Alignment 45

Abstract

A Geometric Control Strategy for Unmanned Aerial Systems

by

Zachary O. Lamb

Effective attitude control is a key component of any Unmanned Aerial System (UAS). Typical PID based control operates on the euler angle representation of attitude. This representation is susceptible to the issue of gimbal lock – a phenomenon in which the rotation axes of a three-axis system align, causing a loss of one degree of freedom and potential loss of orientation information. Moreover, PID based attitude control of UAS's suffer from the drawbacks associated with typical linear controllers. Namely, an ability to reliably control the system when not near the point of equilibrium. This thesis will cover an attitude control implementation directly on the Special Orthogonal group, $SO(3)$, where neither of the aforementioned issues exist. This controller is implemented in both simulation and on hardware.

To my family,

Mark, Susan, et al.,

Who have always had my back and given me the opportunities to succeed.

Without you all, none of this would be possible.

Acknowledgments

A special thank you again goes out to my family, but also the professionals I've worked closest with. Each member of this thesis committee has had a profound effect on my career path, and without the support of Ricardo Sanfelice, Gabriel Elkaim, and Dejan Milutinović, I would be nowhere close to where I am today. Thank you all for your efforts over the years.

Chapter 1

Introduction

Linear controllers exhibit drawbacks in controlling rigid body attitudes for aircraft. For one, they are only effective around some small flight envelope near the equilibrium point, and two, they often require extensive lookup tables to control a flight over a wider regime [10] [27] [1]. Many nonlinear control paradigms exist to address these issues [28] [12]; however, depending on the implementation of the nonlinear controller, these also have drawbacks that come in the form of the attitude representation. Generally, there are three methods to interpret an attitude – Euler angles, Quaternions, or as a rotation matrix on the Special Orthogonal group $SO(3)$. There are inherent drawbacks with each of these attitude representations – namely, gimbal lock for euler angles [24], the unwinding phenomena associated with the double cover of quaternions [26], and the computational cost of linear algebra functions required for an $SO(3)$ representation.

This thesis will focus on an $SO(3)$ (or "geometric") based control design. While

an attitude controller designed on $SO(3)$ will have the computational drawback discussed previously, it is better suited than euler angles because there exist no points of singularity, and quaternions because it does not face the unwinding phenomena. It should be noted here that a common solution to the quaternion unwinding phenomena is to include a simple conditional statement in the control logic that decides which quaternion to wrap to; however, as discussed in [25], this is not robust to sensor noise. A geometric interpretation addresses both of the aforementioned issues, making it a desirable choice for attitude control design.

While choosing a geometric attitude representation is preferable for the reasons stated, there does exist an issue inherent to any subsequent control design - that is, no continuous time-invariant feedback controller on $SO(3)$ can be globally asymptotically stable [9]. There will exist points on the rotation group which remain undefined for geometric control laws. Previous work on a class of rigid body attitude controllers on $SO(3)$ produced both almost-globally stable and localy stable controllers [2]. These controllers contain matrix logarithm operations which are undefined at plus or minus 180° for any given axis of rotation. Using Hybrid control techniques, it is possible to seamlessly stitch together a series of controllers on $SO(3)$ which may be robust against these undefined points, and work together to produce a globally convergent geometric controller. While a hybrid geometric controller is not the focus of this thesis, future work may include this concept. Details regarding the natural geometric controller design are found in Part II of this thesis.

While Part II covers the novel control design, Part I describes the hardware

platform necessary for testing. An important stage in developing any new control algorithm is the transition from theory to hardware. With respect to control techniques intended for aerospace applications, this can undeniably be a tough feat as most aerospace systems are safety-critical; safety-critical systems are commonly referred to as any system where failure/malfunction may lead to death or serious injury, damage to equipment/property, or environmental harm. Fixed wing aircraft need a large flight space for testing, and as such, unexpected failure may be more likely to result in serious consequences. Quadrotors, on the other hand, can be tested in a confined lab setting where barriers separate humans and the vehicle. This yields a much safer test-bed for novel aerospace-related control designs, and is the platform that this thesis focuses on developing.

Existing quadrotor platforms such as ROSflight [16], Crazyflie [14], and the QDrone2 by Quanser provide functionality to facilitate a smooth research experience; however, these systems have drawbacks. With respect to ROSflight, this system requires a companion computer capable of running Linux. This essentially demands that the drone be sufficiently large in order to carry the weight of this computational device. Quanser drones have a similar issue in that they are inherently built as a physically large drone platform. The Crazyflie platform is smaller, however, is very expensive. A requirement imposed on this thesis work is that the drone platform to be used must be both lightweight and inexpensive. This requirement allows for rapid experimentation even in the case a drone gets destroyed. ST Microelectronics is semiconductor company who produces the ST Drone. This drone is a small and lightweight option, with an

open-source codebase and CAD model design. This enables the ability to fly the drone in a smaller lab, as well as quickly rebuild frames and even create new frame designs. Moreover, the ST Drone is cost-effective, with a single unit being roughly $50. These reasons led to the choice of the ST Drone for the quadrotor control platform in this thesis. A few issues exist, however, with the ST Drone; namely, the platform is not well equipped for autonomous guidance and control experimentation. This is due to a lack of a stable communication channel, and a lack of safety constraints being imposed on the drone. These issues and solutions will be addressed in Part I.

This thesis aims to contribute the following: 1) A firmware extension for the ST Drone quadrotor control platform capable of developing advanced control techniques on, and 2) An implementation of a geometric attitude control scheme. These contributions split this work into two parts. Part I involves a discussion on developing the quadrotor control platform, and Part II describes the novel geometric control design which was implemented on this quadrotor control platform. Chapter 2 begins with a discussion on the basics of quadrotor flight control. This includes a brief discussion on quadrotor dynamics and control. Chapter 3 then described the hardware and software onboard the ST Drone, as well as current capabilities and drawbacks of the platform. Chapter 4 covers the necessary improvements which were made to the ST Drone platform to enable the development of more advanced control techniques, and to enable safe/reliable experiments. With a reliable control platform now configured, Chapter 5 describes the process into which a position-level feedback controller is designed and implemented. This includes a discussion on motion capture (MOCAP) systems, PID position control

techniques, and a custom control interface using an Xbox controller to streamline the experimentation process. In Part II, Chapter 6 will cover the process of simulating quadrotor control. It begins with a simple model derivation and position controller configuration, and then dives into the geometric observer/control design. This controller is verified in simulation, and position control is shown to be just as effective under a geometric attitude control framework. Finally, Chapter 7 presents hardware-based geometric attitude control results, and Chapter 8 provides a concluding analysis on the results of this thesis.

Part I

Developing a Quadrotor Control Platform

Chapter 2

Basics of Quadrotor Flight Control

Quadrotors have gained significant popularity in recent years due to their versatility and agility. These unmanned aerial systems are capable of performing various tasks, ranging from aerial photography and videography to search and rescue missions. The ability to control the flight of a quadrotor is vital for its safe and efficient operation. This chapter will explore the basics of quadrotor flight control, including key principles behind stability and maneuverability.

A quadrotor consists of four vertically oriented propellers mounted on a symmetrical frame. By adjusting the rotational speeds of these propellers, the quadrotor can generate the necessary lift and control forces to achieve stable flight. Understanding the dynamics of a quadrotor is crucial to developing effective flight control strategies. The primary forces acting on a quadrotor are thrust, weight, and drag. Thrust is generated by the propellers, countering the weight of the quadrotor, while drag opposes the forward motion of the quadrotor. Thrust is equal to the summation of the forces

generated by each propeller. Weight is equal to the drones mass multiplied by the gravity constant 9.81 m/s^2, and points inertially down. Drag is proportional to the drones velocity, and points opposite the forward velocity. An important feature which enables stable flight of quadrotors is the configuration of the motor spin directions. Each motor induces a yaw torque just by the act of rotating. If all motors rotate in the same direction, the drone will yaw uncontrollably. By integrating motors which spin in opposite directions, stable flight control is possible. Furthermore, by placing motors which rotate in the same direction across from eachother, roll, pitch, and yaw are decoupled. This is an important feature of quadrotors, as without it, controllability would be non-trivial. These forces and moments acting on the quadrotor is visualized in Figure 2.1.

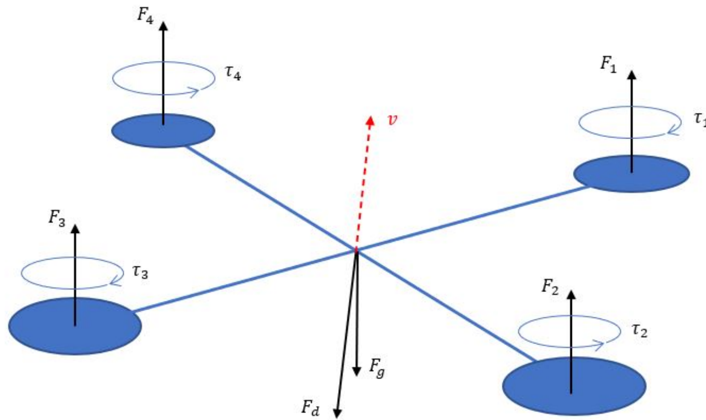

Figure 2.1: Example Quadrotor Dynamics: v represents the current velocity vector of the drone, F_d represents the drag force, F_g represents the force due to gravity, F_1, F_2, F_3, F_4 represent the four forces generated by each motor respectively, and $\tau_1, \tau_2, \tau_3, \tau_4$ represent the four torques generated ny each motor respectively.

A quadrotor has four control surfaces – the four motors. Actuating these control surfaces in various ways allows for four degrees of freedom of the quadrotor;

that is, by commanding the rate of each motor, one can control all three rotational axes independently, and translation along the vertical axis. A key feature which leads to the controllability of a quadrotor is the decoupling of the four controllers responsible for controlling the four degrees of freedom. Again, this is only possible because of the motor configuration. In the physical world, there may exist some coupling between the four degrees of freedom; however, in practice, the quadrotor Attitude Control System (ACS) is designed under the assumption that roll, pitch, yaw, and thrust are decoupled.

In order to control these four degrees of freedom, a control architecture is required which utilizes feedback provided by onboard sensors. Typical sensors found on a quadrotor include an Inertial Measurement Unit (IMU), pressure sensor, and camera. The IMU works to measure angular rates, linear accelerations, and magnetic fields via the gyroscope, accelerometer, and magnetometer respectively. The pressure sensor measures pressure which is typically then converted into an estimate of altitude, and a camera is used to aid in estimating horizontal motion. These sensors feed into algorithms which work to estimate the controllable states. The most important estimation algorithm running onboard is the Attitude Heading and Reference System (AHRS). The IMU feeds its measurements into this algorithm which then determines the quadrotors current orientation (or attitude). By estimating attitude correctly, the drone may control it's own orientation based on user attitude commands. Moreover, by utilizing altitude and horizontal position estimates from the pressure sensor and camera, a quadrotor can control its position.

With access to accurate state estimates, a variety of control algorithms may

be used to control the quadrotor. The most common form of control found onboard quadrotors are Proportional, Integral, Derivative (PID) controllers. These controllers operate by using feedback from the state estimates in order to form error, derivative of error, and integral of error terms. These error terms are then fed through the PID controller and multiplied by a series of gains (K_p, K_i, K_d) in order to generate control signals to actuate the vehicle to track user commands. PID control is discussed in further detail later on. Onboard a quadrotor, there are typically four PID controllers which control each degree of freedom - altitude, roll, pitch, and yaw. As stated previously, this is only possible because the configuration of the motors decouples these four degrees of freedom.

Utilizing the motor configuration, one more important feature is necessary to control the drone; namely, the motor mixing algorithm. The motor mixing algorithm tells the drone how to spin its motors in order to react to commands along the four degrees of freedom. For instance, if a roll command is produced, the drone needs to know how to reduce the speed of some motors, and increase the speed of other motors in order to induce a roll without affecting any of the other degrees of freedom. One may think that by simply increasing the speed of two motors on one side of the drone, a roll can easily be induced. An issue with this, however, is that if the two motors on the opposite side are not decreased by that same amount, then the total thrust has been effected by a roll command. Figure 2.2 illustrates how each motor must increase or decrease its angular rate in order to achieve a desired command, but also not effect any other degree of freedom.

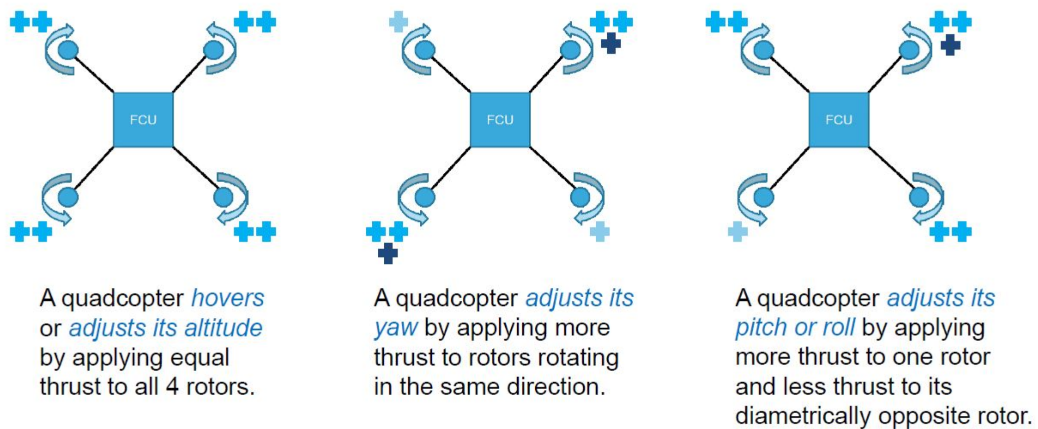

Figure 2.2: Motor Mixing Example from ST Drone Website

All of the previously described systems - AHRS, PID control, motor mixing algorithm - are key components to the successful design of a quadrotor. The AHRS provides working estimates of the controllable states, the PID controllers use the feedback of these state estimates in order to output desired torques to track the user's attitude commands, and the motor mixing algorithm tells the drone how to spin its motors to achieve the desired torque commands. A block diagram of all these systems (AHRS, PID controllers, motor mixer) working together is shown in Figure 2.3.

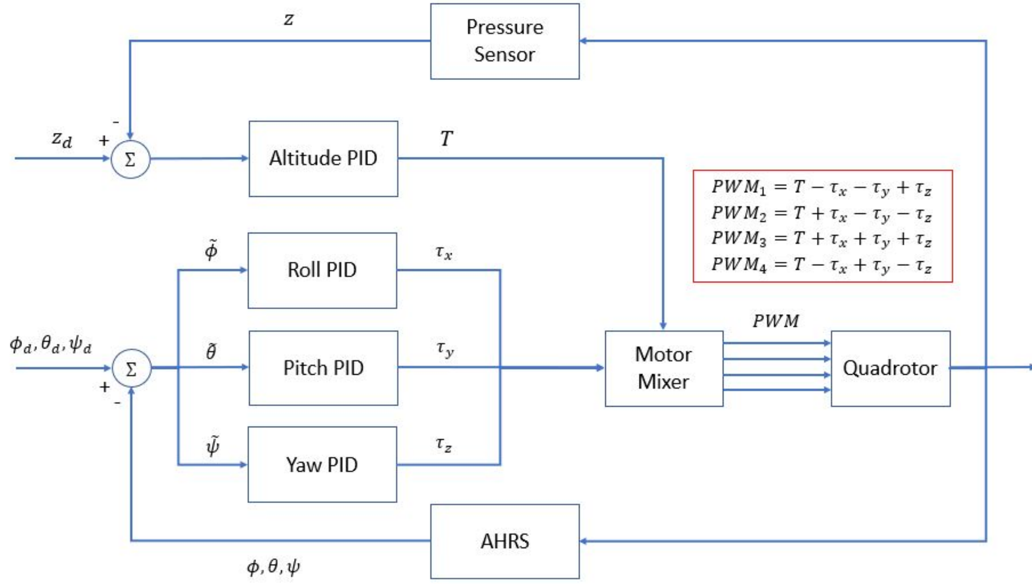

Figure 2.3: Standard Quadrotor Control Architecture Example

In this figure, $\phi_d, \theta_d, \psi_d, z_d$ represent desired roll, pitch, yaw, and thrust respectively (ie. the controllable states of the system). The tilde notation represents error; for example, $\widetilde{\phi} = \phi_d - \phi$. τ_x, τ_y, τ_z represent the desired torques, which are outputs of the three attitude PID controllers. T is the desired thrust which is the output of the height PID controller. Again, these four command signals are passed through the motor mixing block and get translated directly into PWM signals for each motor. An example motor mixing algorithm is outlined in red.

In addition to the basic PID control mechanisms described, several advanced flight control techniques have been developed to enhance the capabilities of quadrotors. A few of these techniques include model predictive control [18], adaptive control [33] [19], and nonlinear control [31] methods. Model predictive control employs a predictive model of the quadrotor's dynamics to optimize control inputs over a future time

horizon. Adaptive control algorithms adjust the control parameters based on changes in the quadrotor's dynamics or external conditions. Nonlinear control methods handle the complex interactions between the quadrotor's dynamics and control inputs more effectively than linear control approaches. Another advanced flight control strategy is to employ a so-called "Geometric" attitude control scheme. This sort of controller has benefits over typical linear PID attitude controller which lay in the nonlinear nature of the controller, as well as the underlying attitude representation of the controller. The second part of this thesis will focus on a geometric attitude control design. For a more detailed discussion on the basics of quadrotor flight control, see [5]

Chapter 3

ST Drone Platform

Quadrotors are used in many research labs due to their ease of use in testing new control strategies. Unlike fixed-wing UAS's, they can be experimented on in a controlled environment where proper safety requirements can be met. Many strong quadrotor platforms exist, however, they are typically built to be large/expensive, non-configurable/modifiable, or close source. The requirements for the drone platform in this thesis were previously stated in the Introduction. To reiterate, the drone must 1) be small enough to fly in the lab space provided, 2) be cost-effective to allow for quick turnaround on experiments in case the drone crashes, and 3) be open-source such that the firmware can be modified as needed. ST Microelectronics is a semi-conductor company who manufactures the ST Drone – a lightweight, cost-effective, and open-source drone option. The ST Drone framework is relatively standard, making it a platform with lots of opportunity for improvement. The drone is pictured below in Figure 3.1.

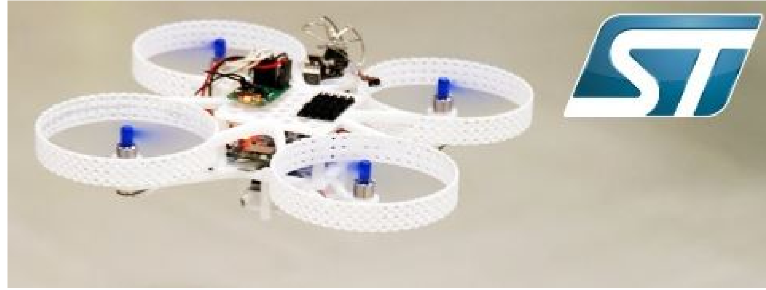

Figure 3.1: ST Drone

This chapter of the thesis aims to build an understanding of the standard (non-modified) ST Drone framework. By the end of the chapter, one should understand both the hardware and software onboard, as well as what some of the drawbacks of the system are. Section 3.1 begins with a description of the hardware found onboard a fully assembled ST Drone. This includes a sensors, communication modules, motors, and the microcontroller and its various peripherals. Section 3.2 then describes the existing software/algorithms onboard the ST Drone that enable it to fly. This includes details regarding the attitude control architecture, state estimation scheme, sensor filtering, and Bluetooth capabilities. Lastly, Section 3.3 covers some of the drawbacks of the ST Drone related to experimental controls work, and how these drawbacks are addressed.

3.1 Hardware

A quadrotor needs relatively few hardware components to fly. These hardware components can be classified under one of two categories. The first category is the accessories; this includes the motors, propellers, battery, and quadrotor frame. These items from the ST Drone will be discussed in Subsection 3.1.1. The next category is

the Flight Control Unit (FCU) and all hardware components that exist on it. These components are discussed in detail in Subsection 3.1.2

3.1.1 Accessories

Figure 3.2: ST Drone Hardware Accessories

The ST Drone comes with four DC motors (two CW and two CCW), four propellors (two CW and two CCW), a 4.2V Lithium Polymer (LiPo) battery, and a small air frame. These parts are all shown in Figure 3.2. This particular drone is very lightweight, meaning that there is no real requirement on strong motors or a powerful battery. A typical battery life for the 4.2V LiPo is around 5 minutes of flight time. Care should be taken to not allow the battery to dip under 3V, as this is the threshold where

these batteries may not be rechargeable anymore.

The four DC motors are all brushed motors. While this means they are not necessarily as powerful as their brushless counterparts, they work fine for the lightweight ST Drone. Additionally, there are no onboard Electronic Speed Controllers (ESC's), however, one may add them to the design externally in order to get more accurate rate control out of the motors. Two of the motors spin CW, and two spin CCW. These motors should connect to their corresponding propellers (which are meant to spin either CW or CCW). The reason for the varying spin directions is to cancel out the total yaw torques generated by each motor. This is illustrated well in Figure 3.3 below.

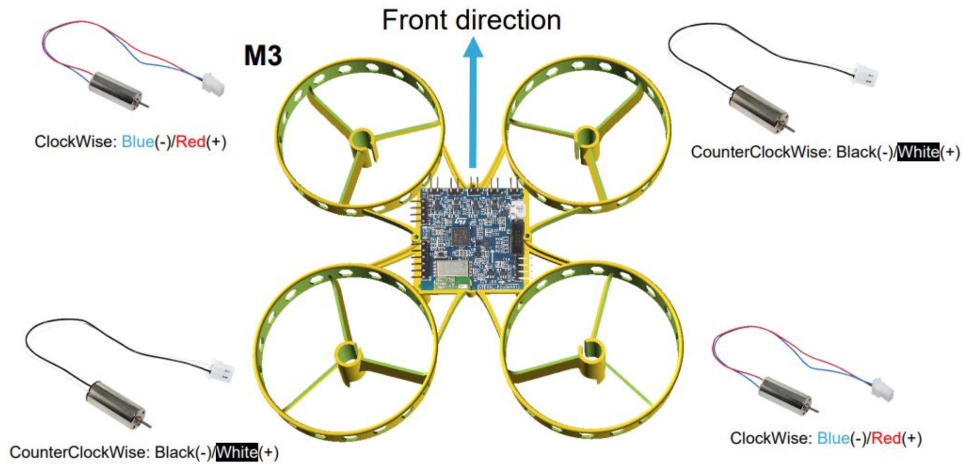

Figure 3.3: ST Drone Motor Spin Configuration

If all four motors spun in the same direction, a net torque along the vertical z-axis would be induced and cause the drone to keep spinning in circles. By using motors which spin in both directions and placing them diagonally apart from eachother, this net torque is canceled out. This means if the same PWM command is sent to each motor, they will produce a net yaw torque of zero, and a net thrust force causing the

drone to lift in the air.

The frame itself is designed for maximal airflow. Notice the holes in the propeller protection rings in Figure 3.1.1. These exist to allow the propellers to rotate with little resistance/interference from air blowing back into the rotor. The propeller rings are also designed such that they are parallel to the ground. It is important to have relatively flat propellers such that the net force is directly upwards.

The last hardware component that allows every other component to work together is the FCU. This is the brains of the drone, and allows it to fly based on control agorithms which run on the Microcontroller Unit (MCU).

3.1.2 Flight Controller Unit

The STEVAL-FCU001V1 is the FCU onboard the ST Drone. This is the system which houses components for communication, sensing of the environment, and interfacing with the motors. Importantly, it also houses the MCU which handles running all of the critical control algorithms that make the drone to fly. These algorithms will be discussed later in Section 3.2, however, the rest of this subsection will dive into details regarding each of the previously mentioned components; namely, the Bluetooth communication module, the Inertial Measurement Unit (IMU) used for sensing orientation, the power MOS used for handling power distribution to the motors, and finally the MCU and its various peripherals. Each of these components can be visualized in Figure 3.4 below.

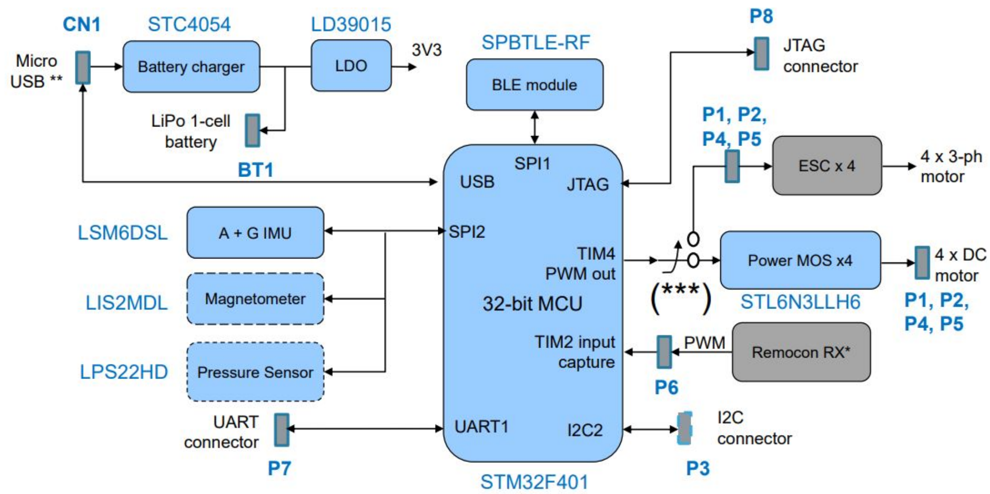

Figure 3.4: ST Drone FCU Components - Components which currently exist (shaded blue), components which are supported but do not exist (shaded grey with black outline).

3.1.2.1 Sensors

There are four primary sensors onboard the ST Drone - The first two are a 3-axis gyroscope and accelerometer which are housed together as one IMU, the third is a 3-axis magnetometer, and the fourth is a pressure sensor.

The accelerometer and gyroscope are housed on an IMU chip called the LSM6DSL. These are Micro-electromechanical system (MEMS) sensors who's data sensor data is polled at 800Hz onboard the ST Drone. These sensors together are used in the Attitude Heading Reference System (AHRS) in order to aid in estimating the orientation of the drone. The magnetometer (LIS2MDL) is also available for use in the AHRS, however, it is unused by default for the ST Drone. The gyroscope provides angular rate measurements, and the accelerometer provides acceleration measurements. These raw measurements are converted into $\frac{\text{rad}}{\text{s}}$ and $\frac{\text{m}}{\text{s}^2}$ respectively.

The magnetometer being used is the LIS2MDL chip. This is a digital output sensor with a range of 50 Gauss, however, it is is unused by default on the ST Drone. One should refer to the datasheet ot he LIS2MDL in order to enable it on the ST Drone.

The final sensor, which is also unused by default, is the pressure sensor. The chip that provides this functionality is the LPS22HD. This is a MEMS sensor with a digital output range of 260 - 1260 hPa. Pressure sensors are typically used to estimate altitude. This pressure sensor has significant noise, and thus requires heavy filtering in order to use it. The ST Drone codebase does provide an initialization function for this sensor if the user wishes to use it. Aside from attitude control which is enabled by the IMU sensors, altitude control is the only other possible form of onboard control which is enabled by the pressure sensor.

3.1.2.2 Bluetooth Communication

Bluetooth Low Energy (BLE) is a wireless communication technology designed for short-range connections and power-efficient applications. It operates on the same 2.4 GHz frequency band as traditional Bluetooth but uses a different modulation scheme and lower power consumption. BLE provides a reliable and energy-efficient means of transmitting data between devices. The low power consumption enables extended battery life, making it ideal for a quadrotor which requires long-term operation without frequent recharging.

The Bluetooth capabilities onboard are provided by a BLE module called the SPBTLE-RF. This module allows for both transmission and reception of data. For

reception, it is used to receive four commands: one thrust command and three attitude commands. For transmission, it is used to send out three different BLE characterstics – IMU data, pressure sensor / battery level data, and arming status data.

Bluetooth characteristics are groups of data with a specific length, meant to be received at the other end of an established bluetooth connection.

In the context of BLE, the messaging protocol revolves around the concept of "characteristics." A characteristic is a fundamental unit of data transfer in BLE and represents a specific piece of information or functionality within a Bluetooth device. Each characteristic has a unique identifier (UUID) and is associated with a set of properties that define its behavior.

The messaging protocol in BLE typically involves two roles: the central device (ie. the quadrotor) and the peripheral device (ie. a smartphone or tablet). The peripheral device initiates communication by discovering the available services and characteristics offered by the central device. Once a characteristic of interest is identified, the peripheral device can read its value, write a value to it, or subscribe to receive notifications or indications when the value changes.

To exchange data, the peripheral device sends requests to the central device using the Attribute Protocol (ATT) layer, which operates over the Generic Attribute Profile (GATT). The GATT defines the structure and organization of data within services and characteristics. The ATT layer handles read, write, and notification operations for individual characteristics.

When the peripheral device wants to read or write a characteristic's value,

it sends an appropriate ATT request with the relevant characteristic's handle and the desired operation. The central device responds with the requested data or acknowledges the write operation. Additionally, the central device can notify or indicate the peripheral device about characteristic value updates, allowing for real-time data streaming or event-driven communication.

With respect to the ST Drone, an Iphone running the ST Drone app is the peripheral device, and the ST Drone itself is the central device. The flow of communication to and from this app is presented in Figure 3.5.

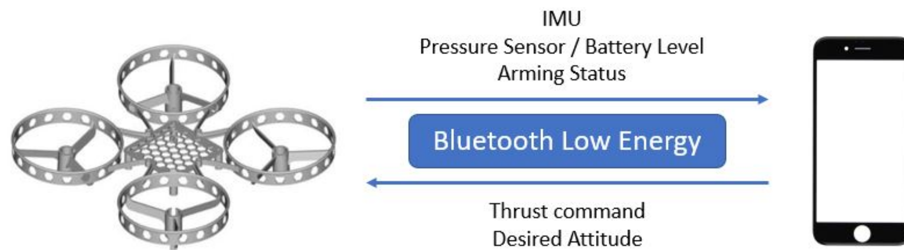

Figure 3.5: BLE Communication Flow

By connecting to the ST Drone BLE module from Matlab, one can list out all of the available characteristics. The relevant characteristics related to receiving data from the drone are the following:

Description	Service UUID (1st row) / Characteristic UUID (2nd row)
IMU Data	"00000000-0001-11E1-9AB4-0002A5D5C51B"
	"00E00000-0001-11E1-AC36-0002A5D5C51B"
Environmental Data	"00000000-0001-11E1-9AB4-0002A5D5C51B"
	"001D0000-0001-11E1-AC36-0002A5D5C51B"
Arming Status	"00000000-0001-11E1-9AB4-0002A5D5C51B"
	"20000000-0001-11E1-AC36-0002A5D5C51B"

Table 3.1: BLE Readable Characteristic UUID's

These BLE characteristics and their content are described in more detail in Section 4.2. Because BLE contains much overhead and runs on a packet acknowledgment protocol, the stability of the communication channel is not always reliable with respect to transmission rates. For a quadrotor control platform, this is obviously not desirable. Control platforms need a low latency and consistent mode of communication. This issue will be addressed in Chapter 4.

3.1.2.3 Microcontroller & Peripherals

The microcontroller found onboard the ST Drone comes from the STM32F4 family. This specific microcontroller, the STM32F401CC, is an ARM M4 cortex MCU. It provides the interface to every other component previously mentioned. Additionally, it provides an interface to various onboard peripherals. These peripherals include one external facing UART port, four external facing PWM input pins to be used with a seperate radio receiver, four PWM output pins connected to the four motors, one I2C port, one internally facing SPI port connected to the BLE module, and one USB port. The physical port locations of these peripherals is shown in Figure 3.6.

Details on some of the most important hardware components were discussed in this section. Below is a comprehensive list of all the hardware components onboard the STEVAL-FCU001V1. Figure 3.7 shows the location of each of these components on the STEVAL-FCU001V1.

1. **STM32F401CCU6** - ARM Cortex-M4 MCU with 256 KB of flash memory, 64 KB of RAM.

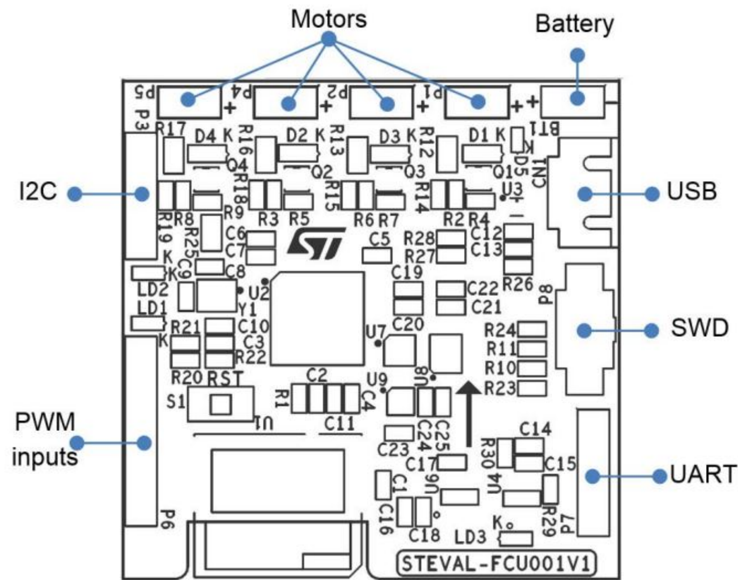

Figure 3.6: ST Drone MCU Peripherals

2. **SPBTLE-RF** - Bluetooth Low Energy (BLE) module with chip antenna. Running Bluetooth 4.1.

3. **LSM6DSL** - 3 axis MEMS accelerometer and gyroscope

4. **LIS2MDL** - 3 axis MEMS magnetometer

5. **LPS22HD** - MEMS pressure sensor

6. **STL6N3LLH6** - 30V 6A MOSFET for motors

7. **STC4054** - 800mA LiPo battery charger from USB

8. **LD39015** - Voltage regulator

9. **USBULC6-2M6** - Electro-static Discharge (ESD) protection

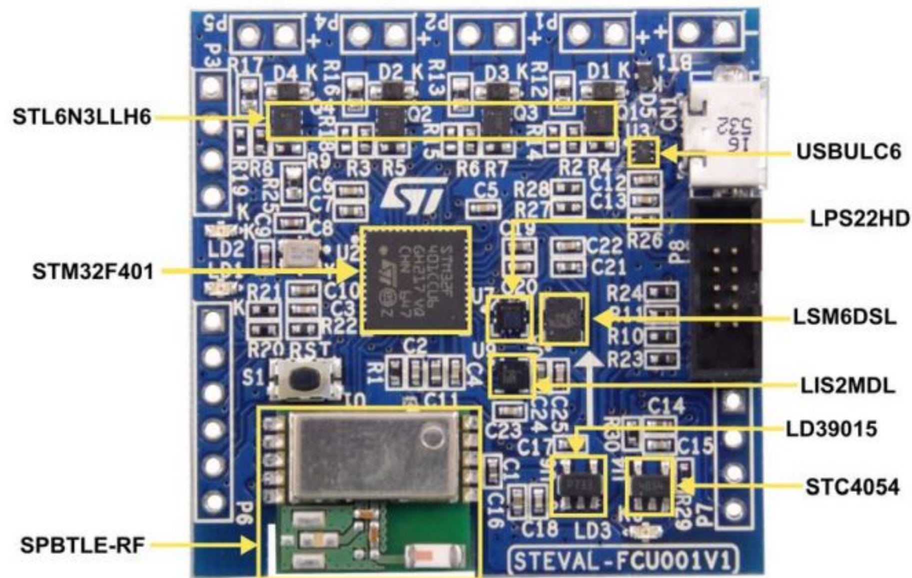

Figure 3.7: ST Drone FCU Components Labeled

3.2 Software

The STEVAL FCU001V1 provides a modest software stack developed to both fly the drone and communicate with other peripheral devices. Typically, the start of data flow to/from the drone comes from a smartphone device. Using the ST Drone app, a user can establish a bluetooth connection with the drone, calibrate and arm it, and then send flight commands. These flight commands are received onboard, interpreted, and then fed into a series of feedback controllers. These feedback controllers track desired commands using sensor measurements from the IMU and an AHRS. These controller outputs are finally fed into a so-called motor mixing algorithm, and PWM signals are generated for each of the four motors. This flow of data is visualized in Figure 3.8

This section will dive into individual blocks of this block diagram in an attempt to explain how the ST Drone is controlled exactly. Most quadrotors fly on the same

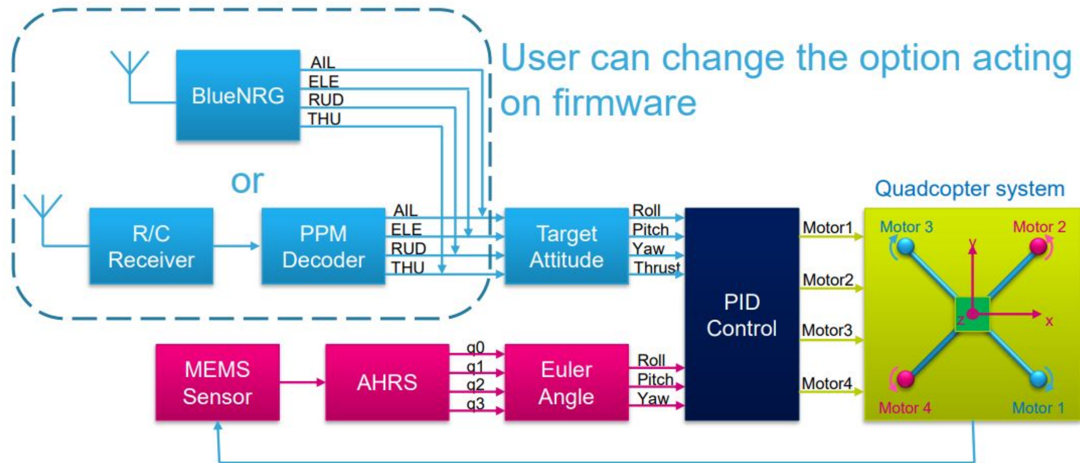

Figure 3.8: ST Drone Software

principles, so this overview of the ST Drone flight control unit is not only relevant to the ST Drone.

3.2.1 Attitude Commands

A short description of the BLE communication protocol was provided in the previous section. The BLE packet which is received onboard the drone consists of four bytes of data. These bytes are commanded thrust, commanded roll, commanded pitch, and commanded yaw. Each of these commands are then either scaled up or mapped back into an expected range. For instance, the thrust command is proportionally scaled, however, the attitude commands are mapped into a given range. This mapping occurs in the ***Target Attitude*** block, where the function *GetTargetEulerAngle()* converts an attitude command byte from the range 0 to 255 to the range -30 to +30 degrees. For

an arbitrary angle θ, this mapping is defined as

$$\theta = (\text{byte}/255) * 60 - 30 \tag{3.1}$$

This angle is then converted to radians and fed into the ***PID Control*** block. *GetTargetEulerAngle()* performs this mapping for roll, pitch, and yaw. -30 to +30 degrees is the maximum attitude command range allowed onboard; this can be changed in the macros of the *rc.h* file. The code related to the *GetTargetEulerAngle()* function is provided below for desired roll conversion:

```
void GetTargetEulerAngle(EulerAngleTypeDef *euler_rc, EulerAngleTypeDef
    *euler_ahrs)
{
    ...

    t1 = gAIL;
    if (t1 > RC_FULLSCALE)
        t1 = RC_FULLSCALE;
    else if (t1 < -RC_FULLSCALE)
        t1 = - RC_FULLSCALE;
    euler_rc->thy = -t1 * max_roll_rad / RC_FULLSCALE;

    ...
}
```

Source Code 3.1: GetTargetEulerAngle()

The *gAIL* term is a globally declared variable, and contains the latest pre-converted roll command. This command is then saturated, and converted to the max allowable attitude command range, specified by the variable *max_roll_rad*. By default, this is -30 to +30 for roll and pitch.

3.2.2 Attitude Heading Reference System (AHRS)

The ***AHRS*** block is in charge of fusing IMU sensor data from the gyroscope and accelerometer in order to estimate the drones current orientation. It accomplishes this through a quaternion implementation of a nonlinear complementary filter which will be described later in Section 6.3. This filter takes in gyroscope and accelerometer sensor measurements as input, and outputs a rigid body attitude estimate. In the current AHRS scheme, this output is in quaternions. Quaternions are an efficient attitude representation with respect to computation time, however, designing a quaternion based controller is more involved than designing a simple euler angle based PID controller. For this reason, the ***Euler Angle*** block exists, wich takes the current attitude estimate in a quaternion representation and converts it to an euler angle representation. These euler angle representations (roll, pitch, yaw) are then fed into the ***PID Control*** block in order to close the loop on the control structure. Code for both the AHRS algorithm is supplied in Appendix B, and the Quaternion to Euler angle conversion is provided below:

```
1 void QuaternionToEuler(QuaternionTypeDef *qr, EulerAngleTypeDef *ea)
2 {
3     float q0q0, q1q1, q2q2, q3q3;
4     float dq0, dq1, dq2;
5     float dq1q3, dq0q2/*, dq1q2*/;
6     float dq0q1, dq2q3/*, dq0q3*/;
7
8     q0q0 = qr->q0*qr->q0;
9     q1q1 = qr->q1*qr->q1;
10    q2q2 = qr->q2*qr->q2;
11    q3q3 = qr->q3*qr->q3;
12    dq0 = 2*qr->q0;
13    dq1 = 2*qr->q1;
14    dq2 = 2*qr->q2;
15    //dq1q2 = dq1 * qr->q2;
16    dq1q3 = dq1 * qr->q3;
```

```
17      dq0q2 = dq0 * qr->q2;
18      //dq0q3 = dq0 * qr->q3;
19      dq0q1 = dq0 * qr->q1;
20      dq2q3 = dq2 * qr->q3;
21
22      ea->thx = atan2(dq0q1+dq2q3, q0q0+q3q3-q1q1-q2q2);
23      ea->thy = asin(dq0q2-dq1q3);
24  }
```

Source Code 3.2: QuaternionToEuler() - A quaternion, *qr*, is evaluated and decomposed into a set of equivalent roll,pitch,yaw rotations. These outputs are passed by reference into *ea*

3.2.3 PIV Control

Proportional Integral Derivative (PID) control is a form of Single Input Single Output (SISO) feedback control which uses a set of gains and error terms to stabilize a system. Mathematically, a PID controller is defined as

$$u = K_p e + K_d \dot{e} + K_i \int e dt \tag{3.2}$$

where u is the control output, K_p is a proportional gain, K_d is a derivative gain, and K_i is an integral gain. K_p typically provides quick responses to the error, K_d can ease overshoot problems, and K_i aids in decreasing steady state error. With respect to the ST Drone, this is an attitude level PID control scheme which means that the error term e is equal to the difference between the commanded attitude and the estimated attitude.

While the exact control structure found onboard the ST Drone does contain PID controllers, it is actually referred to as a PIV (Proportional Integral Velocity) controller [4]. This form of controller nests two PID loops together; the outer loop acts on position and sends velocity commands to the inner loop, and the inner loop acts on

velocity and outputs torque commands. By zooming into the ***PID Control*** block from Figure 3.8, the following structure appears.

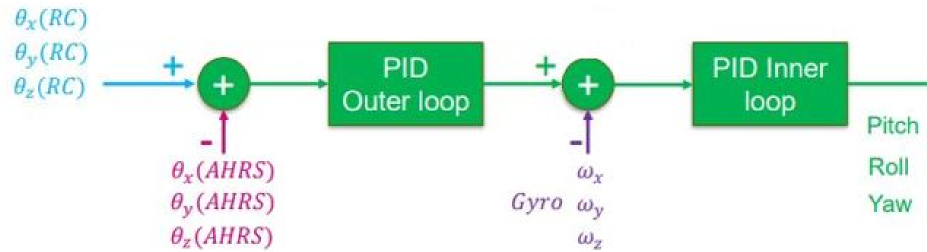

Figure 3.9: PIV Control Architecture

The PID outer loop implementation is actually a simple proportional controller operating at 160Hz on the attitude error, and outputs an attitude rate command. The PID inner loop is a complete PID controller operating at 800Hz on attitude rate, where the rate estimates are fed in directly from filtered gyroscope measurements. This inner loop then outputs roll, pitch, and yaw torque commands to be fed into the motor mixing algorithm which will be described in the next subsection. The PID control code for both the inner and outer loops are provided below:

```
1  void FlightControlPID_OuterLoop(EulerAngleTypeDef *euler_rc,
       EulerAngleTypeDef *euler_ahrs, AHRS_State_TypeDef *ahrs,
       P_PI_PIDControlTypeDef *pid)
2  {
3    float error;
4
5    if(gTHR<MIN_THR)
6    {
7      pid_x_integ1 = 0;
8      pid_y_integ1 = 0;
9      pid_z_integ1 = 0;
10   }
11
12   //x-axis pid
13   error = euler_rc->thx - euler_ahrs->thx;
14   pid_x_integ1 += error*pid->ts;
15   if(pid_x_integ1 > pid->x_i1_limit)
16     pid_x_integ1 = pid->x_i1_limit;
17   else if(pid_x_integ1 < -pid->x_i1_limit)
```

```
18      pid_x_integ1 = -pid->x_i1_limit;
19    pid->x_s1 =  pid->x_kp1*error + pid->x_ki1*pid_x_integ1;
20
21    //y-axis pid
22    error = euler_rc->thy - euler_ahrs->thy;
23    pid_y_integ1 += error*pid->ts;
24    if(pid_y_integ1 > pid->y_i1_limit)
25      pid_y_integ1 = pid->y_i1_limit;
26    else if(pid_y_integ1 < -pid->y_i1_limit)
27      pid_y_integ1 = -pid->y_i1_limit;
28    pid->y_s1 =  pid->y_kp1*error + pid->y_ki1*pid_y_integ1;
29
30    //z-axis pid
31    error = euler_rc->thz - euler_ahrs->thz;
32    pid_z_integ1 += error*pid->ts;
33    if(pid_z_integ1 > pid->z_i1_limit)
34      pid_z_integ1 = pid->z_i1_limit;
35    else if(pid_z_integ1 < -pid->z_i1_limit)
36      pid_z_integ1 = -pid->z_i1_limit;
37    pid->z_s1 =  pid->z_kp1*error + pid->z_ki1*pid_z_integ1;
38  }
```

Source Code 3.3: FlightControlPID_OuterLoop() - Attitude commands are fed in via the *euler_rc* variable, and current state estimates are fed in via *euler_ahrs*.

In this code, the outerloop PID controllers operate on attitude directly. The outputs are stored in *pid→x_s1*, *pid→y_s1*, *pid→z_s1*. These are then used as inputs to the innerloop controller, which is provided below.

```
1  void FlightControlPID_innerLoop(EulerAngleTypeDef *euler_rc, Gyro_Rad *
       gyro_rad, AHRS_State_TypeDef *ahrs, P_PI_PIDControlTypeDef *pid,
       MotorControlTypeDef *motor_pwm)
2  {
3    float error, deriv;
4
5    if(gTHR<MIN_THR)
6    {
7      pid_x_integ2 = 0;
8      pid_y_integ2 = 0;
9      pid_z_integ2 = 0;
10   }
11
12   dt_recip = 1/pid->ts;
13
14   //X Axis
15   error = pid->x_s1 - gyro_rad->gx;
16   pid_x_integ2 += error*pid->ts;
17   if(pid_x_integ2 > pid->x_i2_limit)
18     pid_x_integ2 = pid->x_i2_limit;
19   else if(pid_x_integ2 < -pid->x_i2_limit)
```

```
20     pid_x_integ2 = -pid->x_i2_limit;
21   deriv = (error - pid_x_pre_error2)*dt_recip;
22   pid_x_pre_error2 = error;
23   deriv = pid_x_pre_deriv + (deriv - pid_x_pre_deriv)*D_FILTER_COFF;
24   pid_x_pre_deriv = deriv;
25   pid->x_s2 = pid->x_kp2*error + pid->x_ki2*pid_x_integ2 + pid->x_kd2*
       deriv;
26
27   if(pid->x_s2 > MAX_ADJ_AMOUNT)  pid->x_s2 = MAX_ADJ_AMOUNT;
28   if(pid->x_s2 < -MAX_ADJ_AMOUNT)  pid->x_s2 = -MAX_ADJ_AMOUNT;
29
30   //Y Axis
31   error = pid->y_s1 - gyro_rad->gy;
32   pid_y_integ2 += error*pid->ts;
33   if(pid_y_integ2 > pid->y_i2_limit)
34     pid_y_integ2 = pid->y_i2_limit;
35   else if(pid_y_integ2 < -pid->y_i2_limit)
36     pid_y_integ2 = -pid->y_i2_limit;
37   deriv = (error - pid_y_pre_error2)*dt_recip;
38   pid_y_pre_error2 = error;
39   deriv = pid_y_pre_deriv + (deriv - pid_y_pre_deriv)*D_FILTER_COFF;
40   pid_y_pre_deriv = deriv;
41   pid->y_s2 = pid->y_kp2*error + pid->y_ki2*pid_y_integ2 + pid->y_kd2*
       deriv;
42
43   if(pid->y_s2 > MAX_ADJ_AMOUNT)  pid->y_s2 = MAX_ADJ_AMOUNT;
44   if(pid->y_s2 < -MAX_ADJ_AMOUNT)  pid->y_s2 = -MAX_ADJ_AMOUNT;
45
46   //Z Axis
47   error = pid->z_s1 - gyro_rad->gz;
48   pid_z_integ2 += error*pid->ts;
49   if(pid_z_integ2 > pid->z_i2_limit)
50     pid_z_integ2 = pid->z_i2_limit;
51   else if(pid_z_integ2 < -pid->z_i2_limit)
52     pid_z_integ2 = -pid->z_i2_limit;
53   deriv = (error - pid_z_pre_error2)*dt_recip;
54   pid_z_pre_error2 = error;
55   pid->z_s2 = pid->z_kp2*error + pid->z_ki2*pid_z_integ2 + pid->z_kd2*
       deriv;
56
57   if(pid->z_s2 > MAX_ADJ_AMOUNT_YAW)  pid->z_s2 = MAX_ADJ_AMOUNT_YAW;
58   if(pid->z_s2 < -MAX_ADJ_AMOUNT_YAW)  pid->z_s2 = -MAX_ADJ_AMOUNT_YAW;
59
60   ...
61 }
```

Source Code 3.4: FlightControlPID_innerLoop() - Attitude rate commands are fed in from the outerloop controller, and compared against rate estimates from the *gyro_rad* variable. Control output is calculated and then saturated before being fed into the motor mixing algorithm.

3.2.4 Motor Mixing

In order for a quadrotor to react to torque commands on all three axes, a motor mixing algorithm needs to be implemented. This algorithm accounts for the configuration of the motors, and decides which torque commands are sent into the PWM input of which motor. The motor configuration defined for the ST Drone is presented in figure 3.10

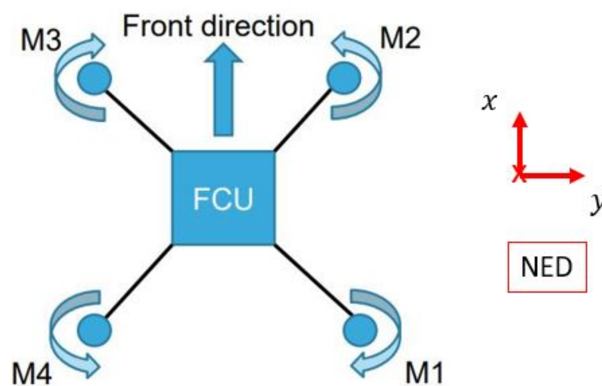

Figure 3.10: ST Drone Motor Configuration

As previously stated, two pairs of diagonally seperated motors spin with different rotation directions. Motors M3 and M1 above spin clockwise, whereas motors M4 and M2 spin counter-clockwise. This is due to the induced yaw torque which eat motor generates simply by spinning. If every motor had the same spin rotation, the quadrotor would yaw out of control. Because two pairs of motors spin opposite to eachother, these yaw torques cancel out so long as they all spin at the same rate. To induce a yaw in the clockwise direction, motors M3 and M1 should speed up by some amount, and motors M4 and M2 should slow down by that same amount. This will induce a yaw torque

while leaving the total thrust the same. This decoupling of the thrust command with the torque commands is important for the controllability of the drone. The roll and pitch torques work in a similar fashion. To induce a right roll torque, motors M3 and M4 would increase by some amount, and motors M1 and M2 would decrease by that same amount. To pitch forward, M2 and M3 would decrease by an amount, and M1 and M4 would increase by that amount.

The previously mentioned roll,pitch,yaw directions of pitch forward, roll right, and yaw counter-clockwise follow the North West Up (NWU) coordinate frame. Onboard the ST Drone, a North East Down (NED) coordinate frame is used. This means that internally, a positive pitch command is interpreted as a commmand to move the drone in the direction opposite the front direction. Similarly, a positive yaw command is interpreted as a command to yaw the drone in a clockwise manner. The positive roll command still means move right.

This relation between thrust and torque commands to physical rotation/motion of the drone can be simplified into a motor mixing algorithm. This algorithm takes in one thrust command and three torque commands (roll,pitch,yaw). The motor mixing algorithm found onboard the ST Drone is the following.

$$\begin{bmatrix} M1 \\ M2 \\ M3 \\ M4 \end{bmatrix} = \begin{bmatrix} T - \tau_{pitch} - \tau_{roll} + \tau_{yaw} \\ T + \tau_{pitch} - \tau_{roll} - \tau_{yaw} \\ T + \tau_{pitch} + \tau_{roll} + \tau_{yaw} \\ T - \tau_{pitch} + \tau_{roll} - \tau_{yaw} \end{bmatrix} \tag{3.3}$$

The code which utilizes this motor mixing algorithm is found at the end of the innerloop controller. This code is provided below:

```
void FlightControlPID_innerLoop(EulerAngleTypeDef *euler_rc, Gyro_Rad *
    gyro_rad, AHRS_State_TypeDef *ahrs, P_PI_PIDControlTypeDef *pid,
    MotorControlTypeDef *motor_pwm)
{
  ...

#ifdef MOTOR_DC
  motor_thr = ((int16_t) (0.05f*(float)gTHR + 633.333f));           //
    Remocon Devo7E >> 630 to 1700
#endif

#ifdef MOTOR_ESC
  motor_thr = 0.28f*gTHR + 750.0f;                          //TGY-i6 remocon and
     external ESC STEVAL-ESC001V1
#endif

  motor_pwm->motor1_pwm = motor_thr - pid->x_s2 - pid->y_s2 + pid->z_s2
     + MOTOR_OFF1;
  motor_pwm->motor2_pwm = motor_thr + pid->x_s2 - pid->y_s2 - pid->z_s2
     + MOTOR_OFF2;
  motor_pwm->motor3_pwm = motor_thr + pid->x_s2 + pid->y_s2 + pid->z_s2
     + MOTOR_OFF3;
  motor_pwm->motor4_pwm = motor_thr - pid->x_s2 + pid->y_s2 - pid->z_s2
     + MOTOR_OFF4;
}
```

Source Code 3.5: FlightControlPID_innerLoop() - Motor Mixer

In this code, a thrust value is calculated based on the originally received $gTHR$ command, and then fed into the motor mixing algorithm along with the torque commands. This algorithm generates PWM signals (saturated between 0 and 1900) for the motors to operate on.

3.3 Drawbacks

The ST Drone product is standard in many ways. For instance, the PID attitude controller is commonly used across many other drone platforms [16] [14] [13],

and the nonlinear complementary filter which governs the AHRS [23] is well-cited and applied to many other autonomous aerospace systems. A few key drawbacks lay in the product assembly, communication protocol, and lack of safety constraints for autonomous flight. Work in Chapter 4 will cover an extension of the software stack described in this section, such that fully autonomous flight with safety constraints is possible. The new flight controller will run efficiently on a low latency line of communication, as well as provide an ability to run more advanced control algorithms by expanding what commands can be sent to the drone. Details regarding this firmware extension are covered next.

Chapter 4

ST Drone Platform Extension

The ST Drone provides a platform aimed at basic quadrotor control via a smartphone app or other external remote control. This drone was not designed with autonomous guidance and control in mind. As such, many improvements need to be made to allow for consistent autonomous flight experimentation. This section will cover a handful of improvements which are made to allow for repeatability of results among experiments, reliable low latency communication, and command structures which aren't limited to commanding typical PID attitude controllers.

4.1 Hardware Accessories

The ST Drone package is delivered with a frame, four motors, four propellers, a battery, and an FCU. There are two crucial components missing from this – namely, a mounting solution for the battery, and a mounting solution for the FCU. Subsection 4.1.1 will describe the mounting solution created for the FCU, and 4.1.2 will describe

the mounting solution for the battery.

4.1.1 Vibration Damping Gel

Care must be taken in mounting any flight controller to the rigid body of the UAS. The primary focus of a mounting solution should be 1) that the mount is rigidly connected to the center of mass, and 2) that the mount is physically distanced from any vibrations. This second point is key because every flight controller will host some suite of sensors. These sensors are always going to be susceptible to process noise. In the case of a quadrotor, the IMU will be susceptible to noise caused by vibration of the frame (usually due to the motors). AHRS estimates are directly effected by noise on the IMU; thus, isolating the IMU from possible frame vibrations is key to a well functioning state observer.

In order to solve this problem, Kyosho Zeal vibration absorbing pads were introduced as the mounting solution. These gel pads (pictured in Figure 4.1 absorb vibrations in the frame, thus isolating the IMU from potential noise.

4.1.2 Mounted Battery Slot

A second hardware based improvement is the mounting solution for the battery. Often times, drones require a trim condition set from the remote controller. This trim condition is essentially an offset for roll and pitch commands such that the drone does not drift when commanded straight upwards. The feedback control at the attitude level can sometimes trim out the quadrotor on it's own if the trim isn't too significant;

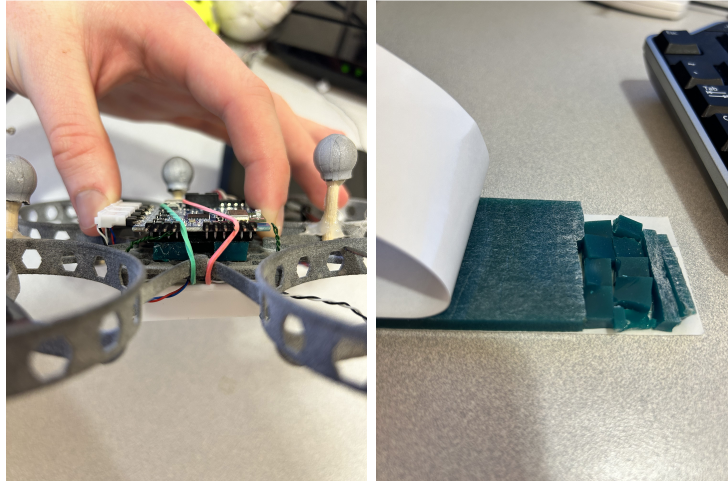
Figure 4.1: Kyosho Vibration Damping Gel Pad

however, if a significant trim is required to keep the drone level during an upward thrust command, then the feedback controller may have a difficult time controlling the drone.

The most common reason for a drone requiring additional trimming is because of a change in the center of mass. On the ST Drone, the battery is one of the heaviest hardware components; thus, based on the placement of the battery, the drone's flight behavior will vary from experiment to experiment. For this reason, a consistent mounting solution is required. This mounting solution will ensure that from experiment to experiment, flight control issues do not stem from a bad initial trimming of the drone. Figure 4.2 pictures the mounting solution for designed for the ST Drone. This mounting solution is 3D printed so is cheap to implement.

4.2 Communication Requirements

The ST Drone is configured by default with a BLE communication module. Because Bluetooth has a lot of overhead, it often times exhibits spikes in it's time-to-

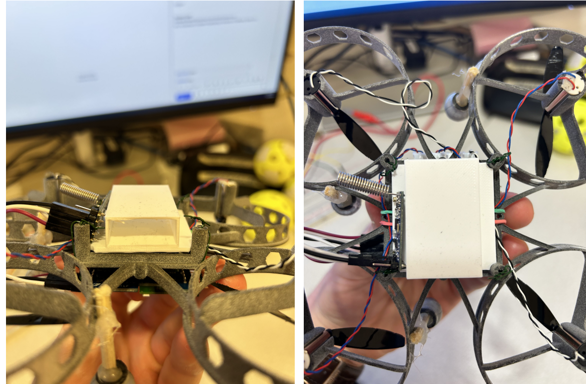
Figure 4.2: Battery Mount Solution

send, which directly induces instability into the system. Additionally, the BLE reception characteristic onboard only accepts attitude commands in the form of 1 byte for thrust, and 3 bytes for attitude. To be able to implement more advanced control strategies, a more flexible option needs to be incorporated which allows any type of commands to be sent to the drone. These commands should be able to do anything from commanding a rotation matrix for a geometric controller, to dynamically tuning PID gains midflight, to switching between different onboard AHRS solutions. In order achieve these goals, a new radio module called the HC12 is implemented.

The HC12 (pictured in Figure 4.3) is an all-in-one radio module which operates on the 433MHz frequency. It utilizes an on-board radio chip (SI4463) and microcontroller (STM8S003FS) to make communication easy. The MCU has already been programmed to handle interacting with the radio chip, and all that's left to do is interface with the MCU via UART. Various *AT* commands can be sent over UART to configure the radio to operate with different baud rates for instance. Lastly, the HC12 operates anywhere from 3.2 to 5 volts. This means that the ST Drone's onboard voltage line

supplied by the UART port is enough to power the module. This means, no extra components are needed to step up or step down the battery voltage. These reasons make the HC12 a simple choice to integrate with the ST Drone. Subsections 4.2.1 through 4.2.3 will describe the various ways in which the HC12 is used onboard the ST Drone to improve performance. A more detailed description of these concepts and the code written onboard is provided in Appendix C

Figure 4.3: HC12 Radio Module

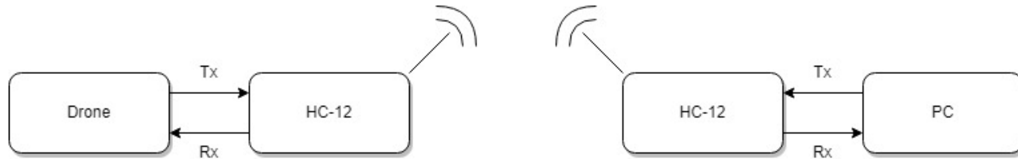

Figure 4.4: HC12 Connection Diagram

4.2.1 Low Latency Communication Link

Addition of the HC12 to the ST Drone provides functionality that the baseline BLE implementation does not allow; one important piece of functionality is the consistent low latency. As previously stated, a consistently fast time-to-send is critical for the performance of many UAS. Typically, BLE time-to-send is around 16ms to send four command bytes worth of data (thrust, roll, pitch, yaw). However, often times the

time-to-send spikes. This spike can range anywhere from 20ms to 200ms or more. When this happens in the middle of flight, the drone becomes unstable. An example of a mild time-to-send spike is shown in Figure 4.5.

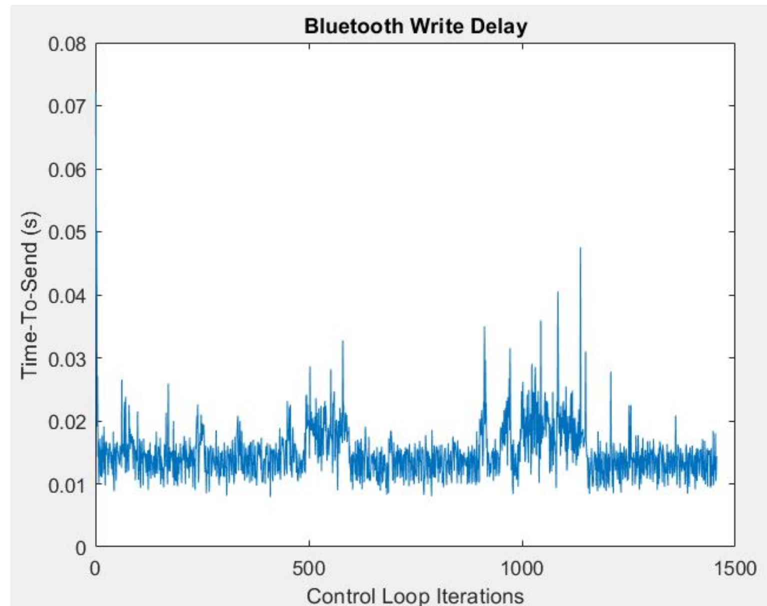

Figure 4.5: Bluetooth Write Time

These spikes tend to occur because of the overhead associated with Bluetooth. Pure radio (without a protocol), however, does not exhibit the same issue. BLE will use a packet acknowledgement protocol which multiplies the amount of messages needed to be passed in order to receive a command. With the HC12, only one packet needs to be sent. The packets the HC12 sends are minimally designed such that only a start/end byte exist to signal to the drone what sort of packet it is receiving, and the inner bytes contain the relevant packet information. Figure 4.6 presents that low time to send for the HC12 operating at a baudrate of 19,200. At this baudrate, data transmission is roughly 1 byte per millisecond. Because 4 bytes were being sent, it took 4ms.

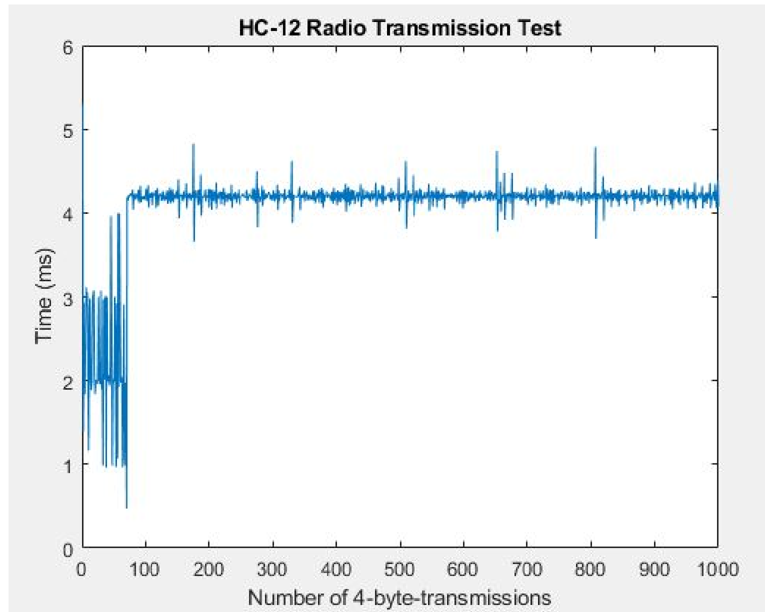

Figure 4.6: HC12 Write Time

4.2.2 Real Time Data Collection

Real time data collection is a necessity for many autonomous control platforms. For one, during the design phase of novel control algorithms, it allows for verification of onboard parameters or estimates in real time. This is especially important when debugging issues. It also allows researchers to replay (in simulation) the events that occurred onboard the drone during the flight.

The improvement of the HC12 only currently lacks in one area; namely, its ability to act as a transciever. Switching back and forth between receiver and transmitter takes a long time, and as such, the HC12 only acts as a receiver for the drone. Future researchers may be able to improve this. As for now, transmission of data off of the drone occurs over Bluetooth. Specifically, this data transmission occurs using the IMU BLE characteristic. The IMU BLE characteristic is designed to send out 20 bytes worth

of data. These 20 bytes worth of data translate to 2 bytes for an onboard timestamp, and 18 bytes for 9 IMU measurements – 3 gyroscope, 3 accelerometer, 3 magnetometer. By default, this BLE characteristic structure looks like the following:

Bytes 1-2	Bytes 3-4	Bytes 5-6	Bytes 7-8	Bytes 9-10
Timestamp	Accel (X-axis)	Accel (Y-axis)	Accel (Z-axis)	Gyro (X-axis)

Bytes 11-12	Bytes 13-14	Bytes 15-16	Bytes 17-18	Bytes 19-20
Gyro (Y-axis)	Gyro (Z-axis)	Mag (X-axis)	Mag (Y-axis)	Mag (Z-axis)

Table 4.1: BLE IMU Characteristic Data Alignment

The data inside these slots may be changed to anything the user wants. Care should be taken when attempting to send floating point values. Because only integers can be sent over Bluetooth, scaling factors need to multiply the floating points values in order to preserve information stored after the decimal place. Similarly, at the ground control level, one needs to know that same scale factor in order to extract the decimal information and receive the data properly.

In most cases, these 20 bytes are enough for any potential data a researcher wants to read off of the drone, however, if desired, there are two other smaller-sized BLE characteristics which may be used as well. These are the characteristics for pressure sensor / battery, as well as for armed/disarmed status. These characteristics are described in below.

Bytes 1-2	Bytes 3-6	Bytes 7-8	Bytes 9-10	Bytes 11-12
Timestamp	Pressure Sensor	Battery Level	Temperature	RSSI

Table 4.2: BLE Environmental Characteristic Data Alignment

Bytes 1-2	Byte 3
Timestamp	Arming Status

Table 4.3: BLE Arming Status Characteristic Data Alignment

4.2.3 Onboard Data Modification

As discussed previously, many quadrotor's are designed based on PID architectures. These architectures are limited due to their linear model assumptions. Currently, only attitude commands can be received, however, what if we want to change gain values or switch between various controllers in real time? The HC12 is a radio module which can send any command type we want since we define the structure. There are currently three types of commands which exist. The first is a PID attitude command, the second is a geometric control attitude command, and the third is a Data Update Request (DR) command. These commands have specific packet structures which logic onboard can detect and determine what the packet is used for. The two command packets are for two different types of control (PID and geometric). The geometric control structure will be discussed later on in Part II of this thesis, however, the PID attitude command structure and DR structure are provided here.

The PID command is structured as 6 bytes. The first and last bytes are signals to the drone that the data being received is a valid packet. These bytes correspond to a start byte (SB) of value 245, and end byte (EB) of value 2. The inner four bytes are the commands themselves. For an attitude command, they are related to the thrust (T), roll (ϕ_d), pitch (θ_d), and yaw (ψ_d). This structure is outline in the table below:

Drone Attitude Commands	
Command	Description
[SB, ψ_d, T, ϕ_d, θ_d, EB]	Attitude command

The DR structure is somewhat similar in that a start byte (SB) and end byte (EB) signal a packet. However, the 2nd and 3rd bytes in this packet indicate that the packet is a DR packet. The 4th and 5th bytes then indicate what type of data update request is being made. This, for instance, could be a command to arm the drone, update onboard gains, or switch onboard controllers. Current functionality for these data update requests are provided in the table below. Further extensions to this can easily be made by following the code provided in Appendix C.

Drone Data Update Requests	
Command	Description
[SB, startByteDR, endByteDR, 1, x, EB]	Arm the drone if x=1, disarm if x=0
[SB, startByteDR, endByteDR, 2, x, EB]	Calibrate if x=1
[SB, startByteDR, endByteDR, 3, x, EB]	Switch onboard control mode to value of x. AOMC=0, MOMC=1, EOMC=2.

4.3 Safety Requirements

Bluetooth uses a complex protocol to verify the sending and receiving of packets, resulting in a relatively inconsistent time-to-send. If this time-to-send consistently passes some threshold, which may be determined either analytically or empirically, then the closed loop system will become unstable. A solution was then presented to solve this issue by utilizing an HC12 radio module to handle receiving command packets. This module was shown to have much lower latency and be more consistent than the onboard BLE module; however, while operating autonomous flight controllers, extra measures should be taken to ensure safety of the vehicle and the vehicles' operators. By moni-

toring the packet count received onboard, it's possible to flag a loss of stability event when the amount of packets received falls below some threshold. When this happens, the drone should subsequently switch to a new on-board controller where the system delay is more reliable and within the defined safety constraint.

Moreover, if the radio is performing well but the autonomous control algorithm isn't, then this loss of stability should also be accounted for to preserve safety. In this case, it is a common solution to allow a safety pilot to take over the drone and revert back to manual flight. As the ST Drone is setup by default, this is not possible. Section 4.3.1 will describe the mode control algorithm necessary to switch between three modes of control – namely, Autonomous Operating Mode Controller (AOMC), Manual Operating Mode Controller (MOMC), and Emergency Operating Mode Controller (EOMC). Switching between these mode controllers ensures a safe flight control setup.

4.3.1 Mode Control Algorithm

The mode control algorithm implemented onboard the ST Drone contains three modes, or equivalently, three states. The first mode is AOMC and is in effect while autonomous control is being performed. If a flight experiment goes well, this will be the only mode the drone operates in. In the case that some loss of stability of the drone occurs due to the outer loop controller, a safety pilot may switch over to MOMC, the manual mode, in order to directly fly based on attitude commands. In the third situation, if a significant loss of packets is observed over a period of 1 second, then EOMC mode is activated, which forces the drone back to level flight and slowly throttles down

the thrust. This mode may be triggered when either AOMC or MOMC is active. If the packets start being received correctly again, the drone will resume back to the previous mode (either AOMC or MOMC). A depiction of this interaction between modes is provided in the state machine below.

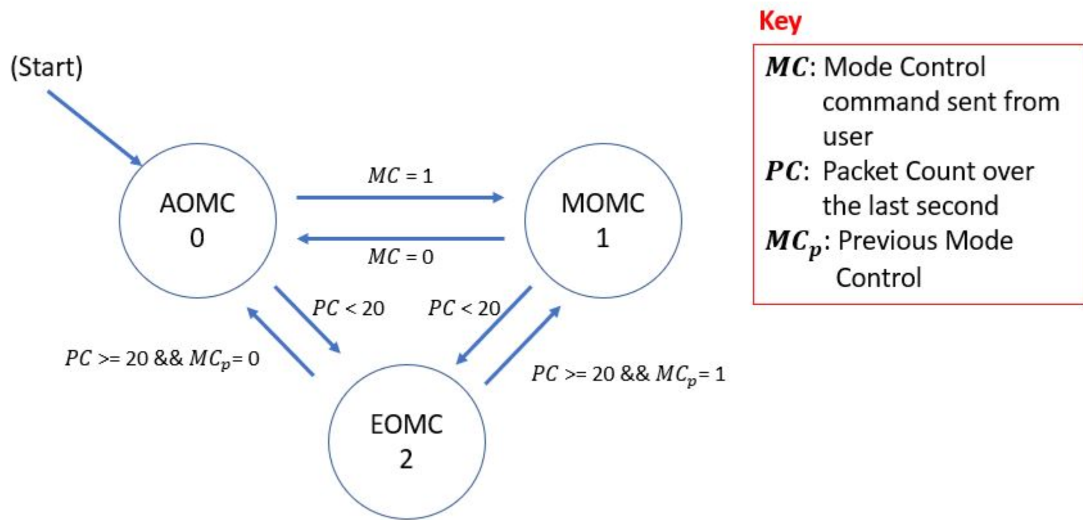

Figure 4.7: Mode Control State Machine

The mode control state machine presented in Figure 4.7 enables safe flight experiments which adhere to a strong set of safety constraints. Further work can be focused on the EOMC mode in this algorithm. Currently, an open-loop controller runs to reduce the throttle slowly over a set amount of time. A more robust solution may be to make use of the onboard pressure sensor, and design a closed loop altitude controller which can hover the drone in place for a few seconds, and then land the drone gracefully. Code for the AOMC, MOMC, and EOMC modes are provided below:

```
1 if(fly_ready){
2     ...
3 
4     // Check packet count
5     if(HAL_GetTick() - startTime >= 1000){
```

```
        // Switch to EOMC if necessary
        if(packetCount < PACKET_RATE_THRESHOLD){
            ControlMode = EOMC;
        } else {
            ControlMode = AOMC;
        }

        // Update start time and reset packet count
        startTime = HAL_GetTick();
        packetCountInASecond = packetCount;
        packetCount = 0;
    }

    ...

    // Run Control Mode
    switch(ControlMode){
        // Manual Operation Mode Control
        case MOMC :
            // BSP_LED_On(LED1);
            // BSP_LED_Off(LED2);

            // Set command values
            gRUD = (attitudeCmd[0]-128)*(-13);
            gTHR = attitudeCmd[1]*13;
            gAIL = (attitudeCmd[2]-128)*(-13);
            gELE = (attitudeCmd[3]-128)*13;
            break;

        // Autonomous Operation Mode Control
        case AOMC :
            //  BSP_LED_Off(LED1);
            //  BSP_LED_Off(LED2);

            // Set command values
            gRUD = (attitudeCmd[0]-128)*(-13);
            gTHR = attitudeCmd[1]*13;
            gAIL = (attitudeCmd[2]-128)*(-13);
            gELE = (attitudeCmd[3]-128)*13;
            break;

        // Emergency Operation Mode Control (on-board control)
        case EOMC :
            //  BSP_LED_On(LED1);
            //  BSP_LED_On(LED2);

            // Set desired attitude to level flight
            gAIL = 0;
            gELE = 0;
            gRUD = 0;

            // Slowly throttle down thrust (open loop for now)
            if(gTHR > AVG_HOVER_THR_CMD){
```

```
59                    gTHR = AVG_HOVER_THR_CMD; // Average hovering state
60                }
61
62                if(mod % 3 == 0){
63                    if(gTHR > 0){
64                        gTHR = gTHR - 1;
65                    }
66                }
67
68                mod++;
69                break;
70        }
71 }
```

Source Code 4.1: Mode Control Algorithm

One may notice here that AOMC and MOMC operate in practically the same manner. This is true, however, the current implementation allows for any further control/logic to be implemented in case the user is controlling the drone vs autonomous algorithms controlling the drone. LED's may also be used to show which mode the drone is currently flying in. This is useful for debugging purposes. The last thing to note here is the use of the open-loop controller for EOMC. As stated previously, this is a naive solution which sometimes results in the drone landing harded than it needs to, or not reacting fast enough to a drone which may be flying at high velocity towards the ceiling. Future work should extend EOMC to use the pressure sensor to close the loop on an onboard altitude controller.

Chapter 5

Ground Control

In Chapter 3, the ST Drone platform was presented as a lightweight and cost-effective quadrotor option to utilize for research. This platform as-is has drawbacks which make it undesirable as an option for experimentation of autonomous control algorithms. These drawbacks include an inability to command more than just euler angles, an inability to operate under consistent and low latency communication, and an inability to maintain safety constraints related to the stability of the drone. These drawbacks were addressed in Chapter 4, posturing the platform for autonomous guidance and control. This chapter now covers theory and a hardware implementation for position control of the quadrotor. Achieving position control is a baseline for most autonomous quadrotor control platforms, and enables research into higher level flight control algorithms such as path planning and obstacle avoidance, to name a few.

Position control of a quadrotor relies on positional feedback. In commercial applications, this feedback may stem from onboard camera systems and/or GPS. In

lab settings, this feedback tends to come from motion capture systems. This thesis implements a position controller based on motion capture feedback being fed into an external Ground Control Station (GCS). This GCS runs software which applies PID control techniques to command attitude trajectories for the drone to follow in order to achieve position tracking. This concept is discussed in detail in Section 5.1. After position control is achieved, a remote controller configuration is developed to facilitate subsequent experiments. This controller configuration is discussed in Section 5.2.

5.1 Position Control

Because a quadrotor is an underactuated system, position control is only possible by controlling a combination of the previously mentioned controllable states - thrust, roll, pitch, yaw. From Chapter 2, it was shown that altitude feedback control is possible by commanding the drones thrust. The final two degrees of freedom (horizontal x, y position), however, are not directly controllable. In order to circumvent this issue, a clever solution is employed to use roll and pitch commands to track desired horizontal positions x, y. Based on the current yaw orientation of the drone, one can always determine a set of roll and pitch commands to steer the drone to any point in space. Now, in the design of an outer loop position controller, x, y, z are controllable states by commanding ϕ, θ, T respectively. ψ is arbitrary in this setup, so can be commanded to anything the user wants.

Designing a position controller for autonomous flight requires two things: 1) An

effective low level attitude controller, and 2) Position based feedback control. Regarding the first point, a position controller will only be as effective as the lower level attitude controller it feeds commands into. The process for tuning and validating the attitude controller is described in Subsection 5.1.1. Once the lower level controller is validated, a position controller can be designed and tuned. Any position controller is going to need position feedback. In this thesis, this feedback comes from a motion capture system called Optitrack. Optitrack and the PID position controller design are discussed in Subsection 5.1.2. Finally, position tracking results are provided in Subsection 5.1.3.

5.1.1 Verifying Attitude Control Performance

Just as a position controller will only be as effective as it's lower level attitude controller, an attitude controller will only be as effective as its attitude estimation algorithm. To begin validating the attitude controller, validation of the estimation algorithm should take place. As previously stated, the position feedback in this control design comes from a motion capture system called Optitrack. While this system can provide position estimates of any object it detects, it can also provide attitude estimates. By utilizing Optitrack, it is possible to store a *true* attitude throughout a flight experiment, and compare it against an estimate of attitudes derived from the onboard AHRS. An experiment was run to rotate the drone plus and minux 90 degrees along both the roll and pitch axes. The onboard AHRS estimates were then stored and compared against MOCAP attitude estimates. The results from this experiment are presented in Figure 5.1.

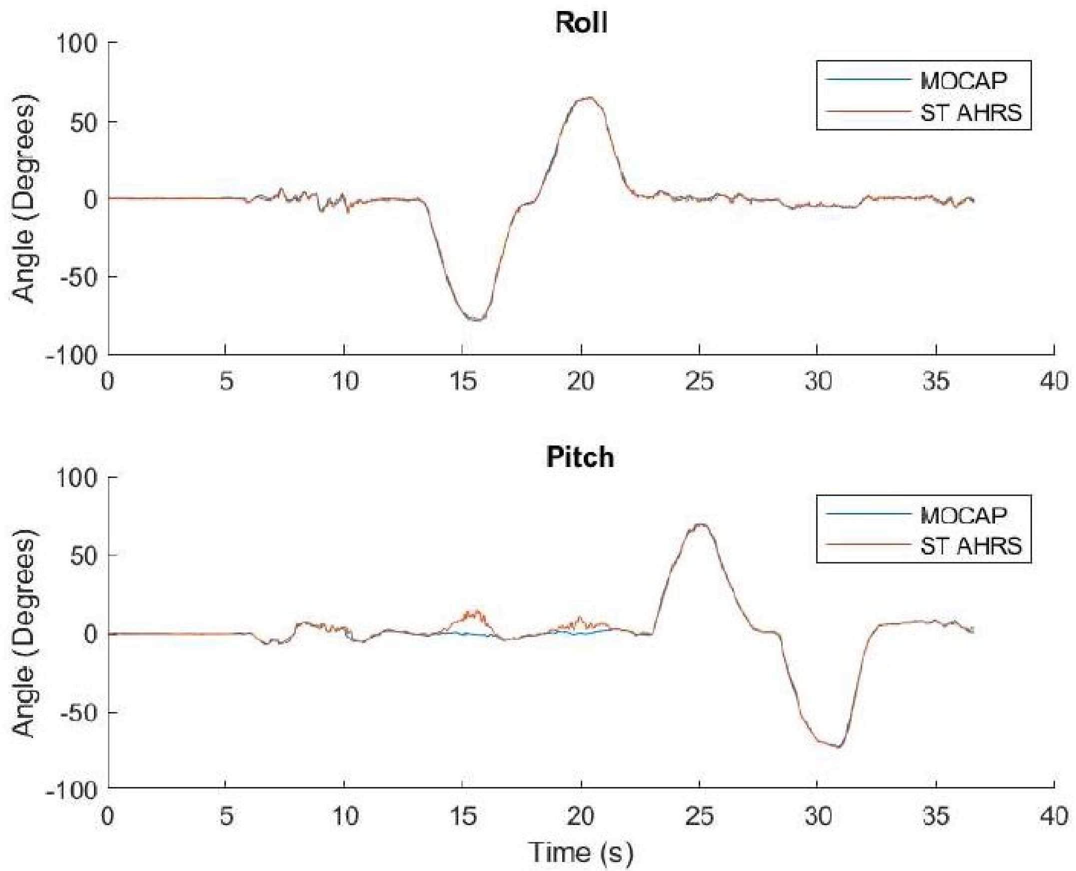

Figure 5.1: Attitude Estimate Comparison - MOCAP vs ST AHRS

The attitude RMSE in this experiment is $\widetilde{\phi} = 0.97^{\circ}$ and $\widetilde{\theta} = 2.66^{\circ}$. In practice, these estimates are close enough to the truth and thus are well suited to run attitude control on. At the 15 and 20 second marks, the pitch estimate of the AHRS seem to deviate from the truth. This is based on the fact that the AHRS solely depends on accelerometer inertial direction measurements to maintain the correct frame of reference, however, while the system accelerates, this inertial measurement becomes useless. To adjust for this, one can place a weighting on the estimator gains which varies with the distance the accelerometer deviates from the expected gravity magnitude. This concept

is explained in detail in Section 6.3.

After validating the accuracy of the attitude estimates, the PID attitude controller can be tuned. A variety of tuning methods exist for PID controllers, the most common of which is called Ziegler–Nichols (as described in Chapter 6 of [11]). When verifying attitude control tracking performance, Optitrack should not be used. Direct comparisons of the AHRS estimate and onboard received attitude commands should be used. If one decides to plot attitude command values which were sent from the GCS, there may be small resolution errors from the true command received onboard due to byte mapping resolution. With a command size of 1 byte per attitude command, as well as a mapping range of -30 to +30 degrees, the max resolution that may be received onboard is roughly 0.24 degrees. Because this resolution is quite small, in these experiments it is acceptable to use the commanded attitude sent from the GCS itself for plotting purposes. Setting the proportional outerloop constant to $K_p = 3$, the innermost loop was then tuned using the previously mentioned Ziegler-Nichols PID tuning method. Figure 5.2 presents final attitude tracking results.

The roll and pitch commands in these results are decently tracked. One thing to note here is that the difference between MOCAP estimates of attitude and AHRS estimates of attitude is noticeable. Once again, this is because of the fact that the drone is accelerating during these roll and pitch maneuvers, causing the AHRS to lose its frame of reference. However, in practice, the estimates are still close enough to reliably control the drone along various attitude trajectories, as seen in Figure 5.2. With the attitude controller tuned and working, a position PID controller can be implemented.

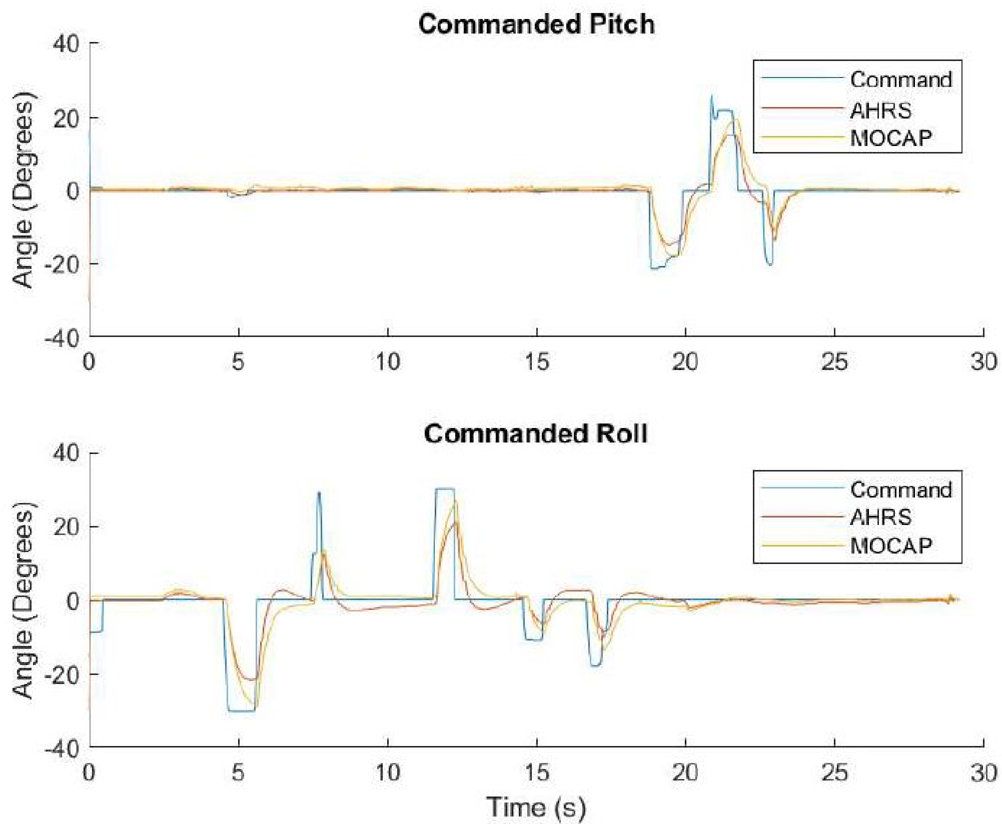

Figure 5.2: PID Attitude Command Tracking

5.1.2 PID Position Feedback Control

The position controller described in this subsection is facilitated by a motion capture system called Optitrack. Optitrack motion capture works by utilizing infrared cameras capable of picking up infrared markers which are physically attached to the object we want to track. An example of these markers attached to the drone are provided in Figure 5.3 below. The corresponding Optitrack software, called Motive, can group these markers into one rigid body, who's pose (position and attitude) is broadcasted to the GCS. This rigid body is presented in Figure 5.4.

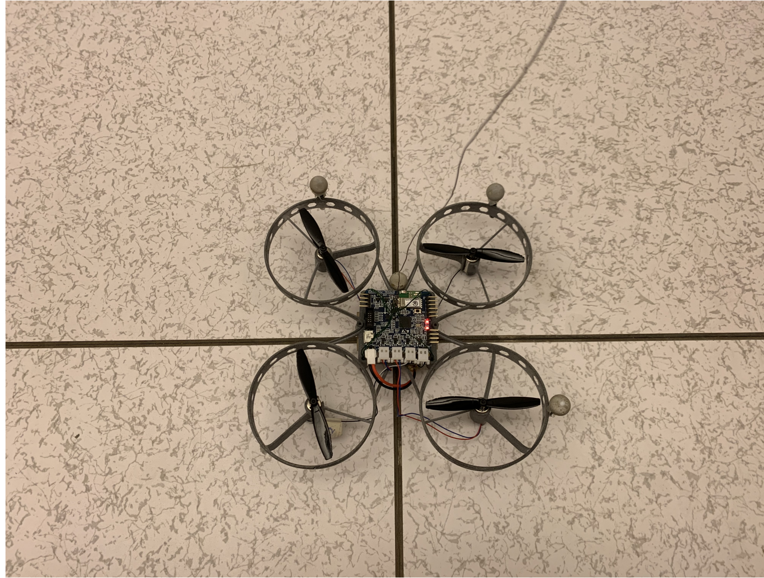
Figure 5.3: Infrared Markers Attached to ST Drone

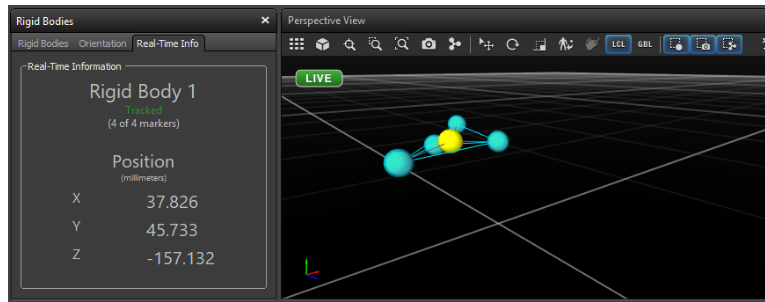

Figure 5.4: Motive Software With Rigid Body Defined

The Optitrack system guarantees millimeter-precise estimates of position, and centidegree precise estimates of attitude. These estimates update at a max update rate of 100Hz. Importantly, this means the max possible position control loop rate is 100Hz. A common rule in designing cascading PID loops is to make the next loop in the sequence run 3-5x faster than the previous loop. Onboard the ST Drone, the inner-most loop operates at 800Hz, and the outer-most loop operates at 160Hz. The position controller in this thesis operates at 40Hz, which is 4x faster then the loop below it. A

flow chart representing the entire feedback loop from MOCAP to attitude control of the drone is presented in figure 5.5.

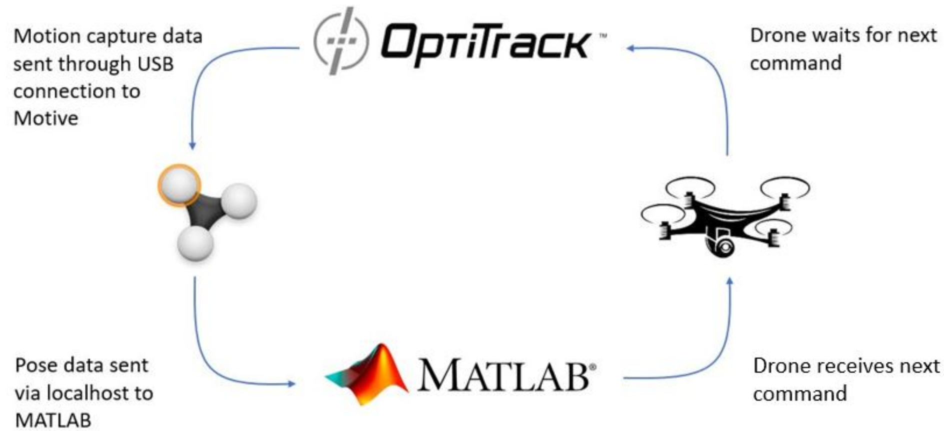

Figure 5.5: Complete Closed Loop System Description

Here, Optitrack is the physical camera system which detects infrared markers, Motive is the software running at the ground control station which runs algorithms to estimate pose, and Matlab represents the position control script running on the ground control station. This Matlab position control script outputs desired attitudes which the ST Drone's attitude control system attempts to track.

These desired attitude commands are generated via a PID controller operating on position commands and position feedback. Position commands may come from a setpoint generator which monitors the drone's current position, and when it is close to a setpoint for a pre-defined amount of time, it generates the next setpoint. In Section 5.2, an Xbox controller is introduced which can also be used to command new setpoints. The PID position block operates at 40Hz and connects with the rest of the closed loop system as described in Figure 5.6

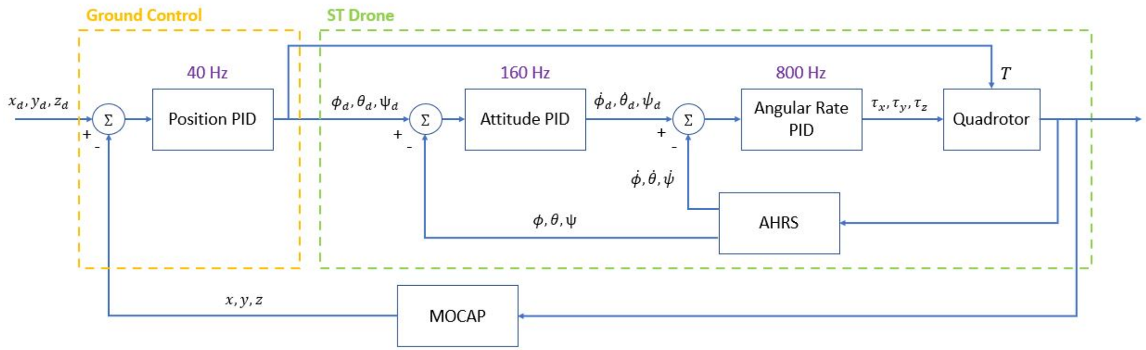

Figure 5.6: Closed Loop Position Control

This block diagram is relatively high level, and simply describes the connections between each component of the system. To reiterate, the position controller works by reading in position commands x_d, y_d, z_d (which may come from some higher level path following loop), comparing those commands to the current position of the system x, y, z as read by MOCAP, and outputting desired attitudes ϕ_d, θ_d, ψ_d and desired thrust T. Horizontal position tracking is achieved by commanding ϕ, θ, vertical position tracking is achieved by commanding T. ψ_d is arbitrary, and in this implementation is set to the constant $\psi_d = 0$. These desired attitudes, ϕ_d, θ_d, ψ_d, and thrust T are fed into the onboard ST Drone control system. PIV control, as described in Section 3.2.3, utilizes AHRS estimates of attitude and filtered gyro estimates of attitude rate in order to command three body torques τ_x, τ_y, τ_z to force attitude tracking. These torque commands are realized through a motor mixing algorithm which commands angular rates for each motor of the quadrotor. Some of the more detailed components of the entire control structure were left out to maintain simplicity of the discussion. The next two block diagrams provided in Figures 5.7 and 5.8 go into greater detail, describing exactly how

a set of desired positions is translated into attitude commands and how these attitude commands are translated into motor PWM signals.

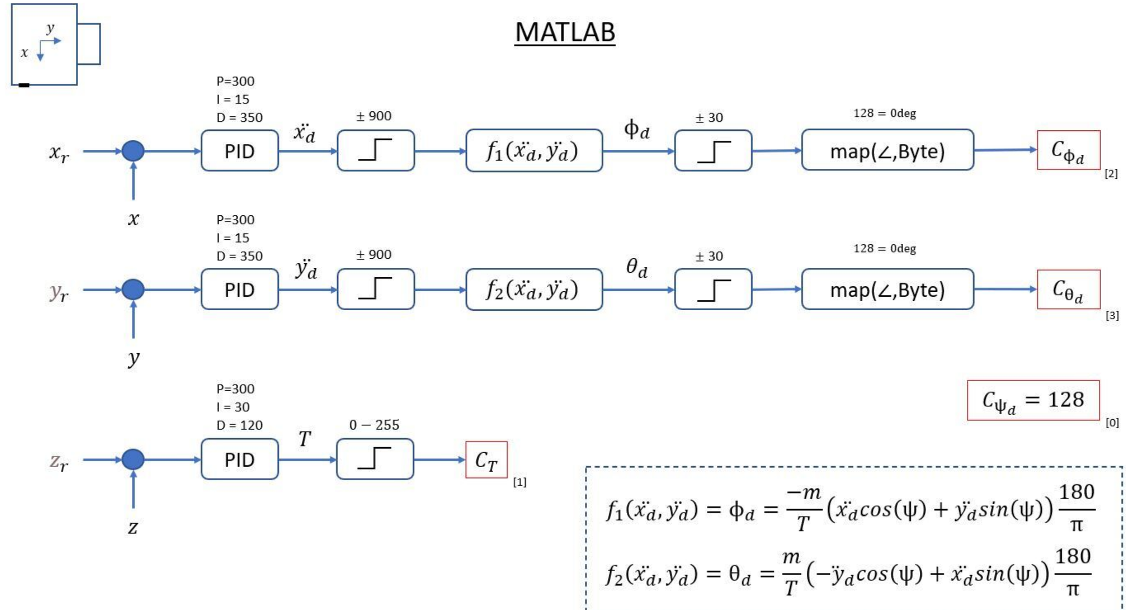

Figure 5.7: GCS Position Control

In Figure 5.7, the control flow which governs the software running on the GCS is presented. Desired positions, denoted x_r, y_r, z_r are passed into the PID controller along with MOCAP position feedback. The output of these PID's are related to desired position accelerations. These position accelerations are then saturated, and fed into a function which uses the current yaw orientation of the drone to determine the roll and pitch commands corresponding to the desired accelerations. This is necessary because if the drone is turned 180°, a right roll command to move the drone right is actually going to be a left roll command, for instance. These desired attitudes are then saturated to within plus or minus 30°, and converted to a byte for transmission to the ST Drone. The byte range is between 0 and 255. With respect to attitude, a byte value of 0

represents -30°, a value of 128 represents 0°, and a value of 255 represents 30°. The thrust command is determined via a PID loop on altitude, and the output of the PID is such that its range is already between 0 and 255, and it saturates between those two values. Finally, the desired yaw command is always set to 0°, which is equivalent to 128 on the byte mapping. These commands $(C_\phi, C_\theta, C_\psi, C_T)$ represent the four byte valued commands which are sent to the ST Drone. The next figure describes what happens when these commands are received onboard the ST Drone.

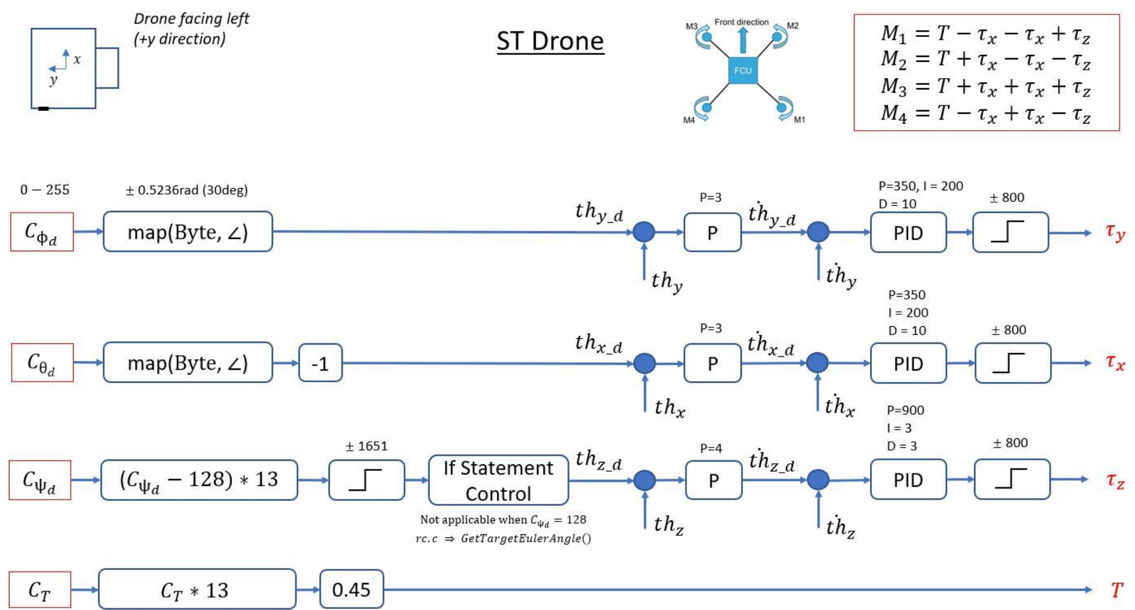

Figure 5.8: ST Drone Attitude Control

In Figure 5.8, the control flow which governs the software running on the ST Drone is presented. The first thing to note here is the coordinate frame (depicted in the upper left corner). Due to differing coordinate frames between the GCS and ST Drone, a rotation of the desired attitude commands is made – In order to translate the Matlab desired attitude into an ST Drone equivalent desired attitude, the pitch

command is inverted. The desired attitudes are then fed into the PIV controller, and saturated to within another limit. These desired torque outputs are fed into the motor mixing algorithm, along with desired thrust, to generate PWM signals for the motors. The motor configuration and motor mixing alogrithm and displayed in the top right corner. In the next subsection, position tracking results are presented for this exact control framework.

5.1.3 Results

In this section, two sets of results are provided for position tracking related experiments. While tuning a position controller, it is important to focus on one axis of control at a time. One should begin by running experiments to tune the altitude controller. Once the altitude controller is tuned, one should move onto roll/pitch PID tuning. Each of these states are independent in terms of the control, but often times pitch and roll react similarly, so can be tuned at the same time.

In the first experiment, a single setpoint is generated at $[x, y, z] = [0m, 0m, 0.7m]$. The PID controller then begins commanding a series of rolls and pitches in order to attempt tracking that point. The results for a fully tuned PID position controller tracking a single setpoint is provided in Figure 5.9.

These point tracking experiments are important in showing the steady state tracking effectiveness of the PID controller. In the next experiment, a circular trajectory is generated for the drone to track. This will provide results related to the frequency response of the drone while using the tuned PID controller. In the following results, a

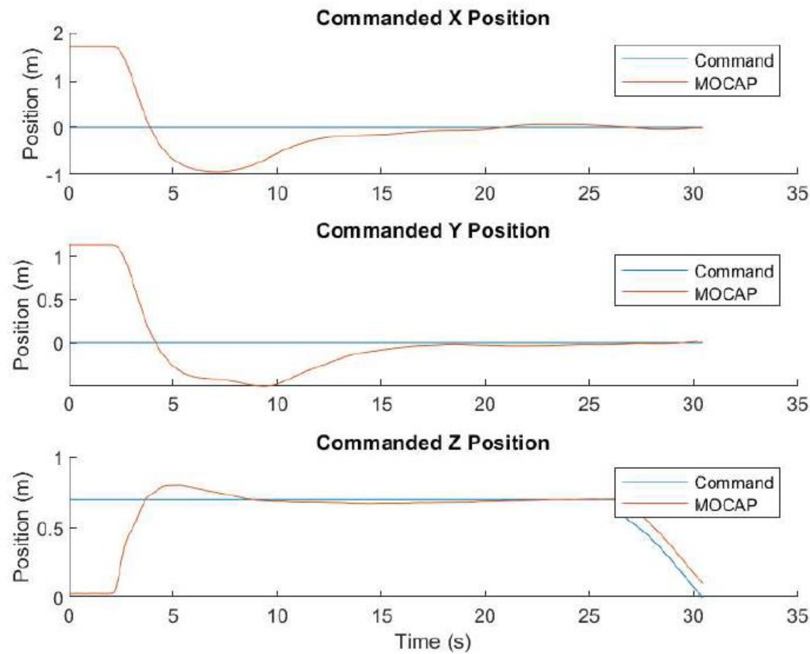

Figure 5.9: Position Point Tracking

circular trajectory with angular rate of 0.2 rad/s with an amplitude of 0.5m is generated. The altitude is again commanded to 0.7m. The tracking results for this experiment are presented below.

Figure 5.10 shows decent tracking performance with relatively low phase lag. These tests represent a PID controller tuned well enought to track points with no steady state error, as well as track time-varying trajectories. During each of these experiments, the setpoints were hardcoded. In the next section an Xbox controller interface is presented to better facilitate running these experiments. It will include features such as setpoint generation, and manual control override.

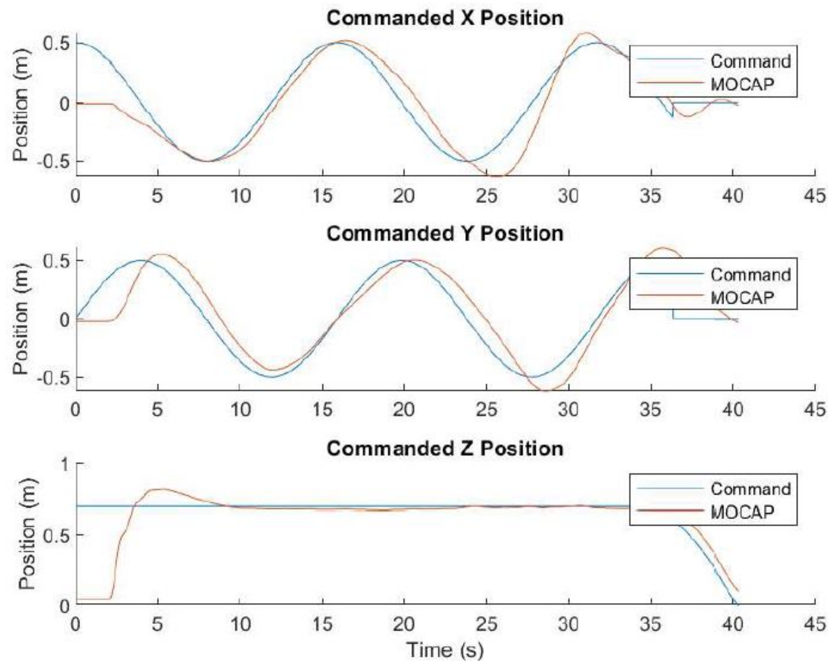

Figure 5.10: Position Circle Tracking

5.2 Xbox Control

In order to maintain better control over the position controller defined in Section 5.1, a remote controller is introduced. Specifically, an Xbox controller is used which allows the user to command various modifiable tasks. The most important reason for incorporating a remote controller is the ability to manually override the drone in case of a loss of stability. During any flight experiment, a safety pilot must be using the Xbox controller. To ensure this, autonomous flight experiments may only begin when the user has calibrated and armed the drone via the left and right bumpers on the Xbox controller.

Once the drone is in flight, the user can toggle through a series of predefined

position setpoints using the left and right toggles on the D-pad. If the user wishes to, or in the case of a loss of stability of the autonomous flight controller, manual mode may be entered by pressing the B button. Once this button is pressed, the user must control the drone manually using the joysticks. The left joystick commands thrust (vertical movement) as well as yaw angle (horizontal movement). The right joystick commands pitch (vertical movement) and roll (horizontal movement). If the user wishes to go back into autonomous flight control mode, he/she may press B again, and it will automatically switch. When ready to finish the flight experiment, the user should press the start button. This will initiate a landing sequence which slowly commands a decrease in height until the drone has reached the ground. A summary of all of these possible Xbox controller commands in provided in Figure 5.11.

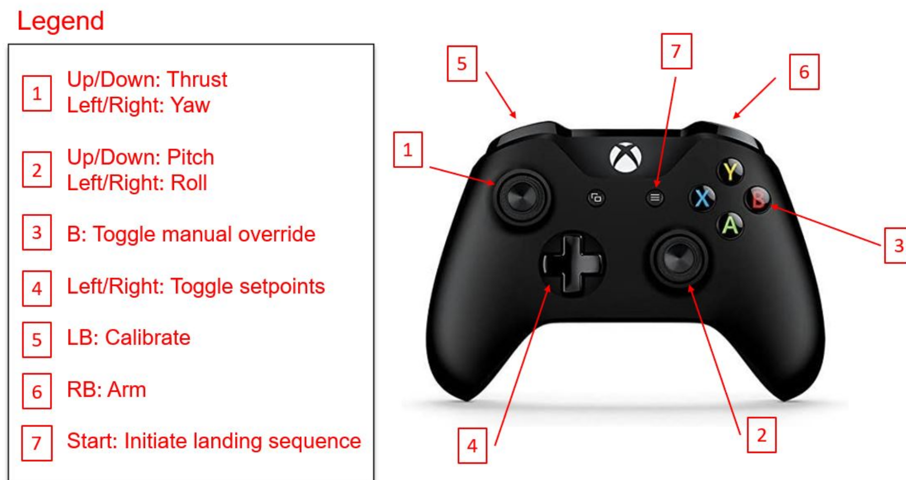

Figure 5.11: XBox Controller Commands

All of these features combine to provide the user with a very pleasant flight control experiment. It is a necessary add-on to the project in terms of safety, but

also allows for easier use of the system at hand. With the manual control method implemented, this concludes the end of Part I of the thesis. Throughout this part, the ST Drone quadrotor platform was presented, and extended for use in autonomous guidance and control experimentation. A position controller was then developed for the drone, and results were presented which showed it's reliability under various scenarios. In Part II of this thesis, a new form of attitude control is investigated. This attitude controller is then implemented on hardware, and compared against the PID attitude controller results presented in this Chapter.

Part II

Geometric Estimation & Control

Chapter 6

Theory & Simulation

The most common attitude control scheme typically runs some form of PID control. With respect to the ST Drone, a so called PIV control scheme [4] is employed. This structure consists of a proportional controller on attitude, which outputs commands to a PID controller on angular velocity. While this scheme is simple to tune, it maintains drawbacks which were previously discussed in Chapter 1. That is, PID schemes inherently require an euler angle attitude representation. There may be tricks possible to operate a PID on quaternions as done in [30], however, by and large PID attitude schemes use euler angle representations. A problem with this euler angle representation is that they are susceptible to gimbal lock, which is a phenomena that leads to a loss of a degree of freedom, causing a rotation to only be visible in two dimensions.

Geometric control schemes, which operate on the Special Orthogonal rotation group, $SO(3)$, avoid gimbal lock. This means there are no singularities, and state estimation will be valid across the entire state space. For this reason, geometric attitude

control is preferred over typical PID euler angle based attitude control. A requirement to operate a geometric controller is a geometric observer. It is not enough to simply build a rotation matrix using the euler angle outputs of an euler angle based estimator. This is because gimbal lock will still occur; it just happens before the generation of the rotation matrix. A widely used geometric attitude estimation algorithm is derived in [23]. This algorithm fuses IMU accelerometer, magnetometer, and gyroscope measurements to accurately estimate a rigid body attitude. It is found in many quadrotor AHRS's, and is even used onboard the ST Drone. In [23], the authors derive euler angle, quaternion, and $SO(3)$ implementations of the filter. The ST Drone uses the quaternion implementation; this thesis will extend it to the $SO(3)$ implementation. Once a geometric observer is implemented, the geometric controller can be written. These types of controllers are nonlinear, and as such, do not suffer from typical linear rigid body controller drawbacks such as predominantly being effective near the point of linearization. This means a geometric attitude controller will likely be more effective at commanding the rigid body along more aggressive desired rotations such as flipping maneuvers. A key drawback of geometric control, however, is a result which states no globally asymptotically stable static continuous controller exists on $SO(3)$ [9]. This issue will be addressed in later sections. Another key issue is the fact that these controllers require computationally expensive linear algebra functions, which may limit the bandwidth of the low-level attitude controller. Some geometric controller even require a matrix log operation, as is the case for the geometric controller used in this thesis. The log operation is a particularly expensive one, and typically requires a different approach to implement it

on hardware; this different approach uses a well known rotation property for matrices on $SO(3)$ to find an equivalent log representation, which reduces the computational effort down to simply running some matrix multiplication operations rather than determining eigenvalues. These approaches and their implementations are discussed in Section 6.4.

In this chapter, theory behind a class of geometric controllers given in [2] is provided and applied to a simulated quadrotor model. Beginning with Section 6.1, a model of a quadrotor is derived and validated through simulations in order to test the geometric controller. After that, Section 6.2 devises a complete position control setup which follows closely the control architecture that flies the ST Drone from Part I. Mimicking the real-world ground control as well as onboard attitude control is an important step in providing a level of confidence that the transition from simulated geometric control to hardware geometric control will work. Next, Section 6.3 will cover the attitude estimation algorithm necessary to estimate the rotation matrix directly on $SO(3)$. Section 6.4 dives into the geometric control theory to be used as a replacement for the ST Drone's PIV scheme. Finally, simulated geometric control results are presented and a discussion on subsequent hardware transition steps is provided.

6.1 Quadrotor Model

A quadrotor is an under-actuated system with respect to its pose. There are four motors, which generate four forces perpendicular to each respective motor. Combinations of these four forces can work together to control 2 rigid body angles (roll and

pitch), as well as 1 body-frame axis of motion (aligned with z-axis). Additionally, the angular speed of each motor sum together to effect yaw. This means the rigid body attitude is fully-actuated, however, the position is under-actuated. There are a variety of ways to model a quadrotor's dynamics; this section will consider a model derivation following [6]. Figure 6.1 below depicts the aforementioned forces/moments operating on the vehicle as setup in [6].

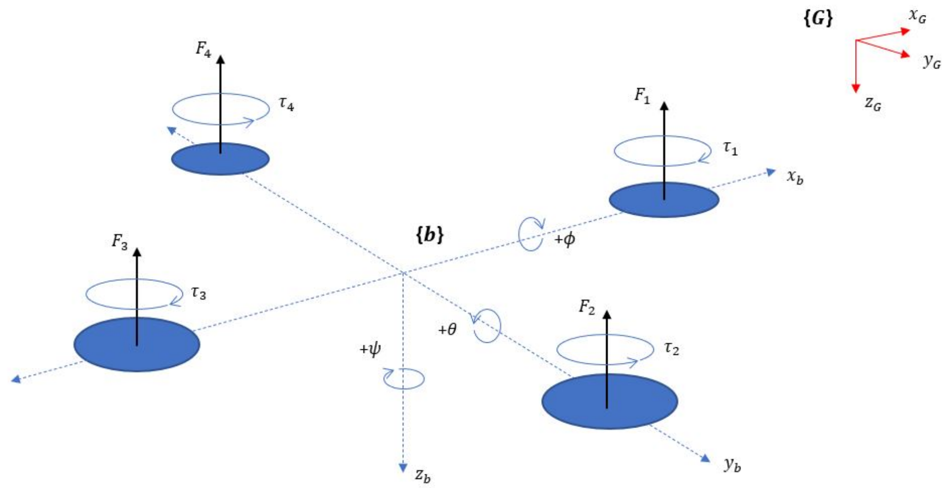

Figure 6.1: Quadrotor Forces and Moments

In this diagram, $[F_1, F_2, F_3, F_4]$ are the four forces generated by the motors onboard, $[\tau_1, \tau_2, \tau_3, \tau_4]$ are the four reaction torques generated by each spinning motor, and the body frame $\{b\}$ is defined as NED, where x_b points North, y_b points East, and z_b points down. Additionally, the global coordinate frame $\{G\}$ is defined as NED from the initialized pose of the quadrotor, where x_G points North, y_G points East, and z_G points down. The roll angle ϕ is defined along x_b, pitch angle θ is defined along y_b,

and yaw angle ψ is defined along z_b. Finally, let $[p, q, r]$ represent the roll, pitch, and yaw rates respectively. Note that these do not directly represent $[\dot{\phi}, \dot{\theta}, \dot{\psi}]$, as these time derivatives of the attitude are defined in intermediate coordinate frames.

Now, let the full state $\mathbf{X} \in \mathbb{R}^{12}$ of the quadrotor be represented by its pose

$$X = \begin{bmatrix} x^G \\ y^G \\ z^G \\ \dot{x^G} \\ \dot{y^G} \\ \dot{z^G} \\ \phi \\ \theta \\ \psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{6.1}$$

where $x^G \in \mathbb{R}$, $y^G \in \mathbb{R}$, $z^G \in \mathbb{R}$ is the position of the quadrotor with respect to the global coordinate frame and $\phi \in \mathbb{R}$, $\theta \in \mathbb{R}$, $\psi \in \mathbb{R}$ is the attitude of the quadrotor with respect to the body frame. Because transitions from $\{G\} \rightarrow \{b\}$ and vice versa are going to be necessary, rotation matrices must be used to facilitate these coordinate frame transfor-

mations. These rotation matrices are defined as follows

$$R(\psi) = \begin{bmatrix} cos(\psi) & sin(\psi) & 0 \\ -sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}, R(\theta) = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{6.2}$$

$$, R(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix}$$

where $R(\psi)$ represents an intermediate yaw rotation, $R(\theta)$ represents an intermediate pitch rotation, and $R(\phi)$ represents an intermediate roll rotation. Multiplying these rotations together, one gets $R^b_G = R(\phi)R(\theta)R(\psi)$, where $R^b_G \in \mathbb{R}^{3\times3}$ is a rotation from the global frame $\{G\}$ to the body frame $\{b\}$. This rotation matrix is commonly referred to as a so called Direction Cosine Matrix (DCM). The fully multiplied DCM is evaluated as

$$R^b_G = \begin{bmatrix} c(\psi)c(\theta) & s(\psi)c(\theta) & -s(\theta) \\ c(\psi)s(\psi)s(\theta) - c(\phi)s(\psi) & s(\phi)s(\psi)s(\theta) + c(\phi)c(\psi) & c(\theta)s(\phi) \\ c(\phi)c(\psi)s(\theta) + s(\phi)s(\psi) & c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) & c(\phi)c(\theta) \end{bmatrix} \tag{6.3}$$

where $c(.)$ is shorthand for cos(), and $s(.)$ is shorthand for sin(). R_G^b can rotate any vector from $\{G\} \rightarrow \{b\}$ using the relation $P_b = R_G^b P^G$, where $P_b \in \mathbb{R}^3$ is a position vector in the body frame, and $P^G \in \mathbb{R}^3$ is a position vector in the global frame. The rotation from $\{b\} \rightarrow \{G\}$ is also possible, and is related by $P^G = (R_G^b)^\top P_b = R_b^G P_b$, where

$$R_b^G = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\phi)s(\theta) - c(\phi)s(\psi) & c(\phi)c(\psi)s(\theta) + s(\phi)s(\psi) \\ s(\psi)c(\theta) & s(\phi)s(\psi)s(\theta) + c(\phi)c(\psi) & c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\phi)c(\theta) \end{bmatrix} \tag{6.4}$$

Now that the DCM's have been defined, a relation between the angular rates and the time derivatives of the euler angles can be defined. Remember, [6] defines the euler angles in an intermediate coordinate frame, so a rotation from the body rate $\omega = [p, q, r]^\top$ to $[\dot{\phi}, \dot{\theta}, \dot{\psi}]$ is necessary. This relation is defined as

$$\omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -sin(\theta) \\ 0 & cos(\phi) & sin(\phi)cos(\theta) \\ 0 & -sin(\phi) & cos(\phi)cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{6.5}$$

Left multiplying both sides by the transpose of this rotation matrix results in

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \frac{sin(\phi)}{cos(\theta)} & \frac{cos(\phi)}{cos(\theta)} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{6.6}$$

At this point, for simplicity sake, this thesis deviates from the derivation in [6]. The model derived hereafter will not consider various electrical effects from the motors, or even the angular velocity of the motors themselves. Drag forces, centripital effects, and various aerodynamic rotational disturbances are also not considered. These are outside of the scope of the thesis, so have been excluded. The remainder of this section will cover the translational dynamics, rotational dynamics, and finally the complete equations of motion for the quadrotor.

6.1.1 Translational Dynamics

Beginning from first principles, the summation of the forces acting on the quadrotor in frame $\{G\}$ is given by

$$m\ddot{X}^G = F_g - F_T^G \tag{6.7}$$

where $m \in \mathbb{R}$ defines the mass of the quadrotor, $\ddot{X}^G \in \mathbb{R}^3$ represents the acceleration vector $[\ddot{x}^G, \ddot{y}^G, \ddot{z}^G]^\top$ in the global frame, $F_T^G \in \mathbb{R}^3$ represents the total thrust force generated by the quadrotor in the global frame, and $F_g \in \mathbb{R}^3$ is represented by

$$F_g = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{6.8}$$

where $g = 9.81$ m/s^2 is the acceleration due to gravity. F_T^G from (6.7) can be computed by rotating the known total thrust force from $\{b\} \rightarrow \{G\}$ via

$$F_T^G = R_b^G F_T^b \tag{6.9}$$

where $F_T^b \in \mathbb{R}^3$ is defined as

$$F_T^b = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} F_i \end{bmatrix} \tag{6.10}$$

and F_i represents the force generated by the ith motor. Now, substituting (6.8) and (6.9) into the translational dynamics equation (6.7), the acceleration of the quadrotor is derived to be

$$\ddot{X}^G = \begin{bmatrix} \ddot{x}^G \\ \ddot{y}^G \\ \ddot{z}^G \end{bmatrix} = \begin{bmatrix} -\frac{1}{m}(cos(\phi)cos(\psi)sin(\theta) + sin(\phi)sin(\psi))\|F_T^b\| \\ -\frac{1}{m}(cos(\phi)sin(\psi)sin(\theta) + cos(\psi)sin(\phi))\|F_T^b\| \\ -\frac{1}{m}(cos(\phi)cos(\theta))\|F_T^b\| + g \end{bmatrix} \tag{6.11}$$

This set of equations describes the motion of the quadrotor with respect to the total thrust generated by all four motors. This total thrust is to be provided as one of the four input commands. The next subsection will describe the quadrotors rotational motion.

6.1.2 Rotational Dynamics

This subsection will describe how a given a set of torque commands on the drone will rotate it. This strategy of directly injecting real rigid body torque values is one step removed from what happens onboard a real drone, as well as what is often setup in other quadrotor model derivations; typically, in simulation, a set of three torque commands are given (along with the thrust command F_T^b), which are then fed into a motor mixing algorithm which outputs motor angular velocities ω_{m_i} for $i = 1, 2, 3, 4$. On hardware, PWM signals replace ω_i. In this setup, the motor mixing algorithm is skipped, and the torque commands are injected directly into the system. This design choice will make geometric control design simpler later on.

Starting with first principles again, the following describes the rotational dynamics of the quadrotor

$$J_b \dot{\omega} = \tau_m - (\omega \times J_b \omega) \tag{6.12}$$

where $\tau_m \in \mathbb{R}^3$ are the body torques $[\tau_x, \tau_y, \tau_z]^\top$ generated from the motors (or in our

case, directly commanded), ω is the angular velocity defined in (6.5), $\omega \times J_b\omega$ are the reaction torques of the system, and $J_b \in \mathbb{R}^{3\times3}$ is the inertia matrix which is defined as

$$J_b = \begin{bmatrix} J_x & 0 & 0 \\ 0 & J_y & 0 \\ 0 & 0 & J_z \end{bmatrix} \quad (6.13)$$

where J_x, J_y, J_z are the inertia values for each body frame axis of the quadrotor. $\dot{\omega} \in \mathbb{R}^3$ from (6.12) is the angular acceleration which is related to the euler angle accelerations by

$$\dot{w} = \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} (\omega \times J_b\omega) = \begin{bmatrix} \dot{\theta}\dot{\psi}(J_z - J_y) \\ \dot{\psi}\dot{\phi}(J_x - J_z) \\ \dot{\theta}\dot{\phi}(J_y - J_z) \end{bmatrix} \quad (6.14)$$

Substituting (6.5), (6.13), and (6.14) into (6.12), the following equations of rotation are derived

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{1}{J_x}((J_y - J_z)qr + \tau_x) \\ \frac{1}{J_y}((J_z - J_x)pr + \tau_y) \\ \frac{1}{J_z}((J_x - J_y)pq + \tau_z) \end{bmatrix} \quad (6.15)$$

6.1.3 Complete Equations of Motion

Combining the equations of motion from (6.16) and (6.15), the final state space model used in subsequent simulations of the quadrotor is defined as

$$\dot{X} = \begin{bmatrix} \dot{x}^G \\ \dot{y}^G \\ \dot{z}^G \\ \ddot{x}^G \\ \ddot{y}^G \\ \ddot{z}^G \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \dot{x}^G \\ \dot{y}^G \\ \dot{z}^G \\ -\frac{1}{m}(cos(\phi)cos(\psi)sin(\theta) + sin(\phi)sin(\psi))\|F_T^b\| \\ -\frac{1}{m}(cos(\phi)sin(\psi)sin(\theta) + cos(\psi)sin(\phi))\|F_T^b\| \\ -\frac{1}{m}(cos(\phi)cos(\theta))\|F_T^b\| + g \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \frac{1}{J_x}((J_y - J_z)qr + \tau_x) \\ \frac{1}{J_y}((J_z - J_x)pr + \tau_y) \\ \frac{1}{J_z}((J_x - J_y)pq + \tau_z) \end{bmatrix} \tag{6.16}$$

6.1.4 Model Validation

Now that a state space model of the quadrotor exists, the model needs to be validated. This subsection will cover a handful of experiments that validate the quadrotor's expected behavior for a set of open loop control inputs. Note that the global NED coordinate frame is used for plotting, so a negative z-axis direction actually

points upward in a NWU frame. Also note that the control input given is a vector $[T, \tau_x, \tau_y, \tau_z]$ where $T \in \mathbb{R}$ is the net upward force control input (same as $\|F_T^b\|$ in 6.16), and $\tau_x, \tau_y, \tau_z \in \mathbb{R}$ are the three torque control inputs which correspond to an instant physical torque on the system.

6.1.4.1 Open Loop Experiment 1: Hover

In this first experiment, the quadrotor is simulated with the input $[T, \tau_x, \tau_y, \tau_z] = [mg, 0, 0, 0]$. This input corresponds to an upward thrust force vector that is equal to the downward force of gravity. Given this, the expected behavior for the quadrotor is for it to hover in place. Figure 6.2 below verifies this expected behavior.

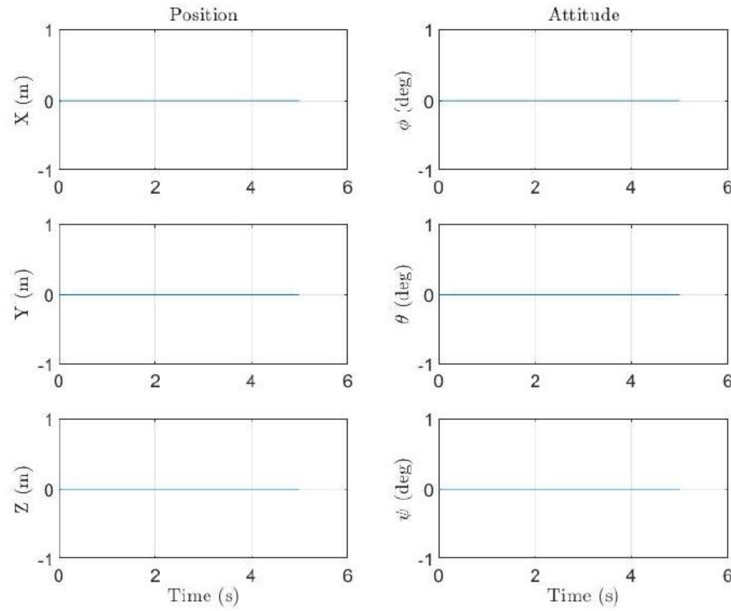

Figure 6.2: Experiment 1: Hover State

6.1.4.2 Open Loop Experiment 2: Positive Roll

In the second experiment, the quadrotor is simulated with the input $[T, \tau_x, \tau_y, \tau_z] = [2mg, 0.1, 0, 0]$. The force from the thrust input is enough to move the drone upwards, and the roll torque command is enough to begin rotating the drone. The expected behavior for control input like this is for the drone to move upwards (negative z-direction), and along the positive y-axis (to the right). Because the drone will continue to roll, it should eventually rotate past 90° and start forcing the drone downwards (positive z-direction). Figure 6.3 below verifies this expected behavior.

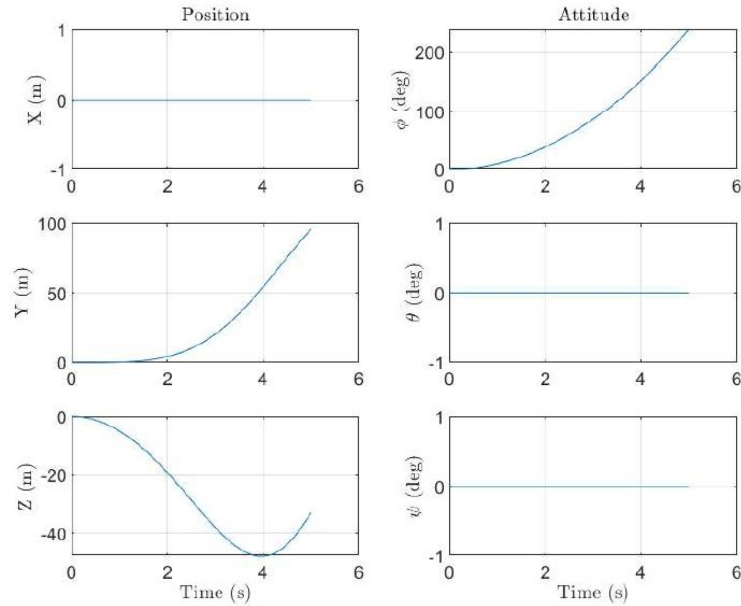

Figure 6.3: Experiment 2: Positive Roll

6.1.4.3 Open Loop Experiment 3: Positive Pitch

In the second experiment, the quadrotor is simulated with the input $[T, \tau_x, \tau_y, \tau_z] = [2mg, 0, 0.1, 0]$. The force from the thrust input is enough to move the drone upwards,

and the pitch torque command is enough to begin rotating the drone. The expected behavior for this control input is for the drone to move upwards (negative z-direction), and along the negative x-axis (backwards). Because the drone will continue to pitch, it should eventually rotate past 90° and start forcing the drone downwards (positive z-direction). Figure 6.4 below verifies this expected behavior.

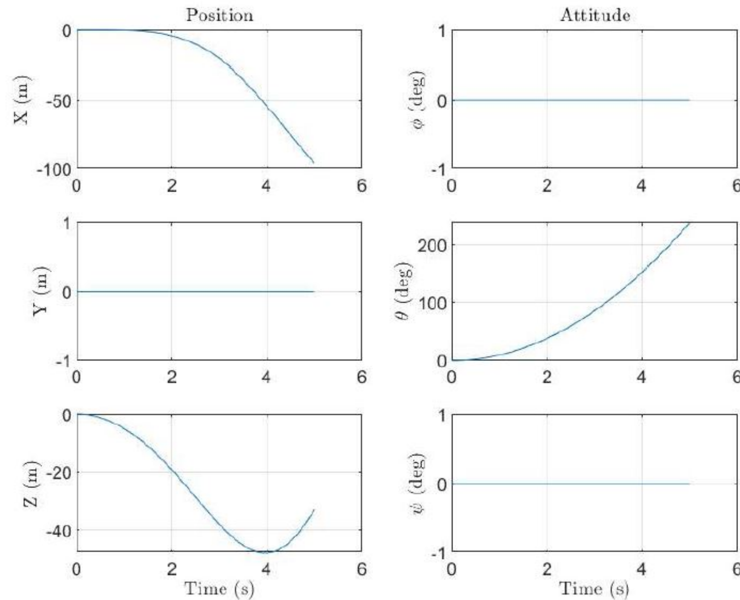

Figure 6.4: Experiment 2: Positive Pitch

6.1.4.4 Open Loop Experiment 4: Positive Yaw

In the second experiment, the quadrotor is simulated with the input $[T, \tau_x, \tau_y, \tau_z] = [2mg, 0, 0, 0.1]$. The force from the thrust input is enough to move the drone upwards, and the yaw torque command should start spinning the drone along the z-axis. The expected behavior for this control input is for the drone to move upwards (negative z-direction) while all other states except for yaw angle remain at 0. Figure 6.5 below

verifies this expected behavior.

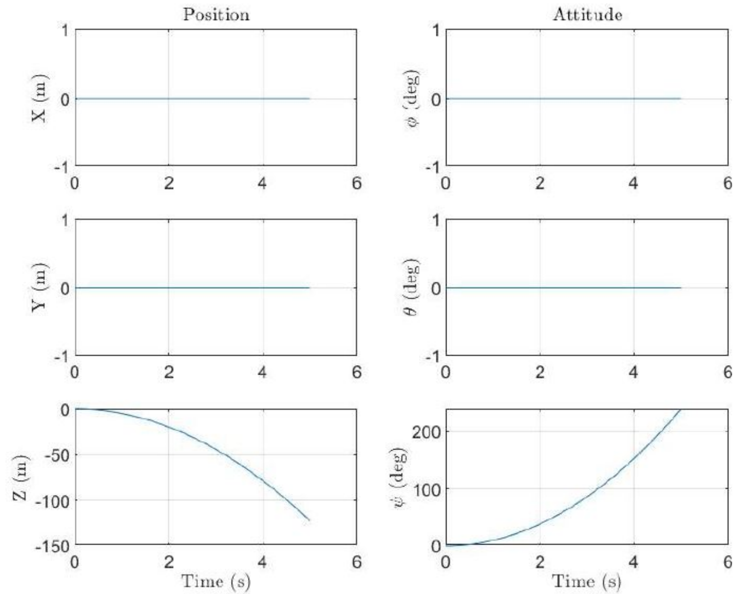

Figure 6.5: Experiment 4: Positive Yaw

6.2 Closed Loop Quadrotor Position Control

Now that the quadrotor model has been sufficiently tested in open loop input scenarios, a more realistic control scheme can be developed. Part I of this thesis describes a control platform setup capable of controlling the position of the physical ST Drone. This same position control setup is achievable in simulation, and is important to the hardware development of a currently theoretical geometric controller. This simulation environment, which relates closely to the hardware, will act as a testbed to run new control code. It is in this way that novel algorithms can be first tested in a safe environment before being deployed on a physical drone.

Subsection 6.2.1 provides a recap of the full closed loop position control setup

which flies the ST Drone. Many details from the hardware solution will be ported over to the simulated solution in order to maintain a level of confidence that the results in simulation will transition to hardware. A complete block diagram of the closed loop structure is provided, and then results are presented which validate the position controller.

6.2.1 Implementation

There are three feedback loops which control the ST quadrotor platform from Part I. Two of these loops govern the attitude control onboard the drone, and one runs at the position control at the ground control station. The attitude controller contains an outerloop which runs at 160Hz, and an inner loop running at 800Hz. The outerloop runs a proportional controller on attitude and outputs angular rate commands. The inner loop receives these rate commands and tracks them via a PID controller which outputs torques commands that feed into the motor mixer. While these two loops run onboard the drone, a postion controller offboard the drone exists to command attitudes which will help to achieve position tracking. This position level PID controller operates at roughly 40Hz. The entire closed loop feedback controller is visualized in Figure 6.6.

This exact loop structure has been implemented in Matlab in order to simulate the system as a whole. With a replica of the hardware implementation running on Matlab, a geometric controller can easily be validated before moving to hardware. Simulations which verify the position controller are provided in Figure 6.7.

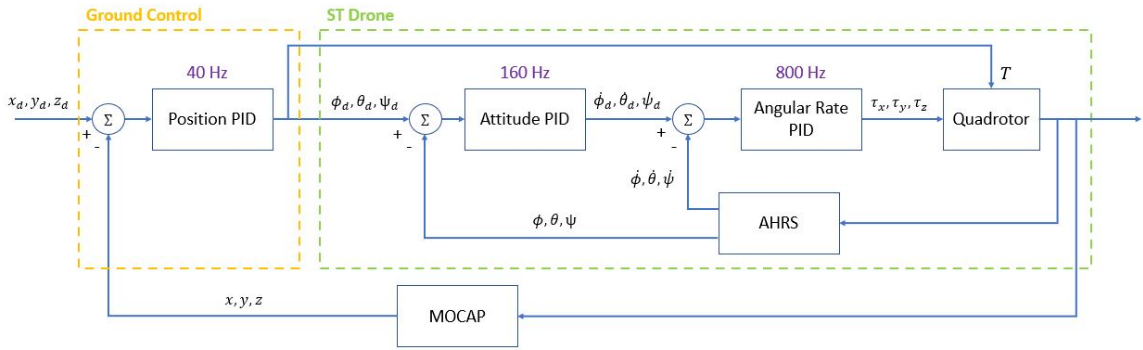

Figure 6.6: Closed Loop Position Control

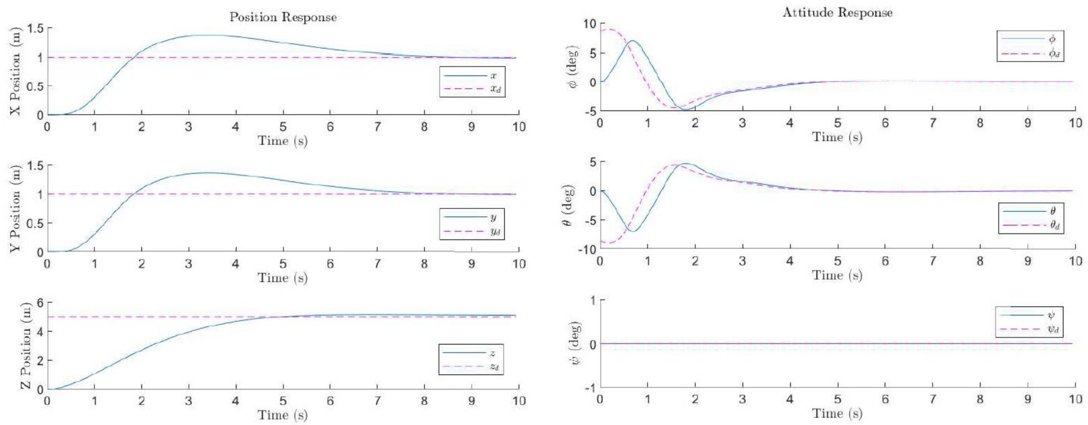

Figure 6.7: Position Control Response

This position controller is effectively responds to and tracks a step input along each axis. Attitude response plots show that attitude tracking is working well, with some lag in the response. As will be shown in sections to come, a geometric attitude controller here will improve attitude tracking performance, and subsequently improve position tracking as well. These plots will be compared and analyzed. Before implementing the geometric controller, a geometric observer is necessary. This observer design follows in Section 6.3.

6.3 Geometric Estimation

An important step in implementing any control scheme developed on $SO(3)$ is to implement an $SO(3)$ based estimation algorithm. Without a corresponding geometric estimator the system will still be susceptible to gimbal lock, which was a large reason for choosing geometric control methods in the first place. For this reason, we need an estimation algorithm that directly estimates the rotation matrix.

There are a variety of attitude estimation options, however, they usually fall into one of two categories – Kalman Filter (KF) or Complementary Filter (CF). A few different examples of Kalman Filters include the nonlinear Extended Kalman Filter (EKF) and the Unscented Kalman Filter (UKF) [17]. Examples of Complementary Filters include TRIAD [7], Madgwick [21], and Mahony [23]. Various publications have analyzed and compared these algorithms with respect to attitude estimation; typically, KF based algorithms are more computationally expensive than CF algorithms [29]. This imposes limitations on feedback loop update rates, and at the attitude level, this is a crucial issue because this is often the lowest level of the feedback loop so the control needs to be reactive to fast dynamics. Another key analysis of these estimation algorithms acting on a quadrotors IMU data shows that Mahony's method achieves the fastest runtime, and outperforms Madgwick and Kalman Filtering in terms of error convergence [20].

The Mahony algorithm is perfect for aerospace applications, as it fuses IMU sensor data together to create a remarkably accurate estimate of attitude. It should

also be noted that Mahony defines three versions of the same filter – An euler angle representation, a quaternion representaion, and an $SO(3)$ representation. Subsection 6.3.1 will loosely cover the theory used to derive the $SO(3)$ algorithm, and will present the final set of equations which estimates the attitude as a rotation matrix. In Subsection 6.3.2, this set of equations is implemented in Matlab, and run against real IMU data which was measured during a flight experiment of the ST Drone. Mahony attitude estimates are generated and compared against sub-degree precise Optitrack MOCAP measurements.

6.3.1 Nonlinear Complementary Filtering on $SO(3)$

This section summarizes key points in the Mahony derivation [23], beginning with a formal definition of the 3D rotation group known as $SO(3)$, the associated Lie Alegbra group $\mathfrak{so}(3)$, and a few properties related to these two groups. Next, measurement models are provided for the IMU, and a pair of solutions using these models are suggested. A dive into the Lyapunov-based solution illuminates the various filters that are built off of eachother. A simple filter is designed assuming some unrealistic assumptions, then a filter based on more realistic assumptions is provided, and finally a filter suitable for use with an IMU is derived.

6.3.1.1 The $SO(3)$ Rotation Group

The Special Orthogonal group $SO(3)$ is a rotation group under which every possible 3D orientation exists. This group of rotations contains no singularities, and is

thus well suited for a robust attitude representation. These rotations are all centered about the origin of three dimensional Euclidean space, and are defined as $R(v) \in SO(3) : \mathbb{R}^3 \to \mathbb{R}^3$, where $v \in \mathbb{R}^3$ is the vector to be rotated. This rotation group preserves four key properties:

1. Isometry - Rotations are distance-preserving transformations and every rotation matrix maintains $det(R) = 1$. This can also be stated as $|R\vec{x}| = |\vec{x}|$.

2. Rotations preserve angles between two vectors: $< R\vec{x}, R\vec{y} >=< \vec{x}, \vec{y} >$.

3. The columns of R are orthonormal.

4. The columns of $R^\top$ (or rows of R) are orthonormal.

5. The inverse of a rotation matrix is equal to its transpose: $R^\top = R^{-1}$. This yields the relation $R^\top R = RR^\top = I$.

A critical final point here is that if a 3x3 matrix R satisfies any one of these properties, the rest of the properties are also satisfied for that same matrix R, and thus R is a rotation matrix on $SO(3)$.

Another key concept related to the rotation group $SO(3)$ is its associated Lie Algebra group $\mathfrak{so}(3)$. This group is defined as the tangent space of $SO(3)$ at the identity matrix I. This group space is important, as it allows for algebraic manipulations such as addition or multiplication while preserving its own group properties. These sorts of operations are key in the design of an estimator on $SO(3)$ because $SO(3)$ itself does not allow for these sorts of operations. Using the associated Lie Algebra, however, one can

apply a transformation $SO(3) \to \mathfrak{so}(3)$, perform algebraic manipulation in $\mathfrak{so}(3)$, and then transform back to $SO(3)$. This essentially enables the use of filter gains to drive rotation related errors to zero. $\mathfrak{so}(3)$ is defined as the set of anti-symmetric matrices

$$\mathfrak{so}(3) = \{A \in \mathbb{R}^{3\times3} | A = -A^\top\}. \tag{6.17}$$

A fundamental operation in Lie Algebra is the Lie Bracket, which states any $A, B \in \mathbb{R}^{3\times3}$, $[A, B] = AB - BA$. We also have that for any $\Omega \in \mathbb{R}^3$, the skew-symmetric operation, $\times$, transforms $\Omega \to \mathfrak{so}(3)$ via

$$\Omega_\times = \begin{bmatrix} 0 & -\Omega_3 & \Omega_2 \\ \Omega_3 & 0 & -\Omega_1 \\ -\Omega_2 & \Omega_1 & 0 \end{bmatrix}. \tag{6.18}$$

A noteworthy property of this transformation is its relation to the cross product of two vectors. Namely, for any $v \in \mathbb{R}^3$, $\Omega_\times v = \Omega \times v$. The vex operator, defined as $\text{vex}(\Omega_\times) = \Omega \in \mathbb{R}^3$, performs a transformation from $\mathfrak{so}(3) \to \mathbb{R}^3$.

Another key point regarding the $SO(3)$ rotation group is that the trace of $R \in SO(3)$, denoted $tr(R)$, is bounded such that

$$3 \geqslant tr(R) \geqslant -1, \tag{6.19}$$

where if $tr(R) = 3$, then $R = I_3$ is geometrically interpreted as a the euler angle

representation of $[\phi, \theta, \psi] = [0, 0, 0]$. If $tr(R) = -1$, then either $\phi = \pm\pi$, $\theta = \pm\pi$, or $\psi = \pm\pi$.

Finally, a common method to denote the "error" between two rotation matrices on $SO(3)$ is to multiply them as $R_a^\top R_b$. When R_a matches R_b, this multiplication equates to the identity matrix. This concept is used in the subsequent observer design.

6.3.1.2 IMU Measurement Models

Three models related to the gyroscope, accelerometer, and magnetometer on an IMU are described in [23]. Each of these models accounts for a measurement bias and measurement noise, and use the following notation for the inertial and body frame – $\{A\}$: Inertial Frame, $\{B\}$: Body Frame. Beginning with the model for a gyroscope, a gyroscope is a measurement device capable of sensing angular rotations in a the body frame with respect to an inertial frame. The following measurement structure is assumed

$$\Omega^y = \Omega + b_g + \mu_g \in \mathbb{R}^3, \tag{6.20}$$

where $\Omega \in \{B\}$ denotes the true angular rate in the body frame, $b_g \in \mathbb{R}^3$ represents a constant gyro measurement bias, and $\mu_g \in \mathbb{R}^3$ represents the measurement noise.

The accelerometer is a measurement device which measures instantaneous linear accelerations of $\{B\}$ relative to $\{A\}$, expressed in $\{A\}$. This linear acceleration is denoted $\dot{v}$. An ideal "motionless" accelerometer should measure the acceleration due to gravity as $\dot{v} = R_A^B g$ where $g = [0, 0, -9.81]^\top \in \{A\}$ and $R_A^B : \{A\} \to \{B\}$. However,

generally, the following model holds

$$a = R_A^B(\dot{v} - g) + b_a + \mu_a \in \mathbb{R}^3, \tag{6.21}$$

where $b_a \in \mathbb{R}^3$ again denotes the measurement bias or offset, and $\mu_a \in \mathbb{R}^3$ denotes measurement noise. The accelerometer is known as an orienting measurement device. This means it is capable of reconstructing an accurate attitude on it's own. Consider the low frequency case where the accelerometer is almost motionless. Because g is so large relative to small perturbations along any axis of motion, the directional vector will still represent a decently accurate representation of the inertial down direction. This can be used to help determine orientation of the rigid body. Under low frequency acceleration, the inertial direction can be estimated via

$$v_a = \frac{a}{|a|} \approx -R_A^B e_3, \tag{6.22}$$

where $e_3 = [0, 0, 1]^\top$, and $v_a \in \mathbb{R}^3$ represents the current unit acceleration vector in the body frame, which at lower frequencies, represents the inertial down direction.

Similarly, the Magnetometer is an orienting measurement device which can reconstruct true orientation. The model for the magnetometer is given by

$$m = R_A^B A_m + B_m + \mu_b, \tag{6.23}$$

where $A_m \in \mathbb{R}^3$ is earths magnetic field in the inertial frame, $B_m \in \mathbb{R}^3$ is the body frame

magnetic field disturbances (from closeby electronic devices such as running motors), and $\mu_b \in \mathbb{R}^3$ represents measurement noise in the body frame. Sensor noise for these sensors is typically small, however, disturbances in the B_m term occur often onboard systems that house many electronic devices. Similar to the accelerometer, only the unit vector of the magnetic field is considered, and under small disturbances present in the B_m term, the following magnetometer direction measurement represents the earths magnetic field at the location of measurement

$$v_m = \frac{m}{|m|}. \tag{6.24}$$

. In the implementation of the subsequent estimation algorithm, the magnetometer and accelerometer do not necessarily need to be used together, however, having both present as a method for reconstructing an inertial unit direction is important for the circumstances where either the system is maneuvering too fast for the accelerometer to be useful, or where the system is experiencing significant magnetic field disturbances.

6.3.1.3 Estimation Approaches

From the previous sensor models, a few approaches can be taken to solve the estimation problem. Given the fact that the accelerometer or magnetometer can be used to reconstruct an attitude under a set of circumstances, it seems obvious to try

and use these two sensors directly. In fact, an optimization problem can be setup as

$$\arg\min_{R\in SO(3)}(\lambda_1|e_3 - Rv_a|^2 + \lambda_2|v_m^* - Rv_m|^2) \approx R_B^A \tag{6.25}$$

where $\lambda_1, \lambda_2 \in \mathbb{R}$ are weights corresponding to confidence levels of the accelerometer and magnetometer inertial unit vectors respectively, and $v_m^* \in \mathbb{R}^3$ represents the inertial magnetic field direction where the system is being operated. By solving this optimization problem, one can find an accurate rotation matrix R_B^A. However, issues arrive in the implementation. For one, optimization problems are costly to solve, and when the implementation is at the attitude level, fast enough loop rates is necessary. Another issue is that in the case the accelerometer and/or magnetometer are not operating close to the inertial vector reconstruction assumptions made for each of them, then the optimal solution to 6.25 is useless. For these reasons, an alternative solution which additionally makes use of the gyroscope is desirable.

The approach taken here on out focuses on a Lyapunov-based solution. To start, a definition of the error criteria needs to be formed. Let $\widehat{R}$ denote the estimation of the true rotation $R = R_B^A$. The coordinate frame associated with $\widehat{R}$ is defined as the estimator coordinate frame $\{E\}$. The associated frame transformation is

$$\widehat{R} = \widehat{R}_E^A : \{E\} \to \{A\}. \tag{6.26}$$

The goal of the estimation problem is to drive $\widehat{R} \to R$. The rotation error $\widetilde{R}$ is defined

as

$$\widetilde{R} := \widehat{R}^{\top} R, \widetilde{R} = \widetilde{R}_E^B : \{B\} \to \{E\}. \tag{6.27}$$

Subsequent observer design will be based on a Lyapunov Stability analysis with the following Lyapunov candidate function

$$E_{tr} = \frac{1}{2}(I_3 - \widetilde{R}). \tag{6.28}$$

.

From previously stated properties, it follows that

$$E_{tr} = \frac{1}{2}(I_3 - \widetilde{R}) = (1 - \cos(\theta)) = 2\sin(\frac{\theta}{2})^2 \tag{6.29}$$

where θ is the angle associated with the rotation from $\{B\} \to \{E\}$. Now, assuming that observer update laws can be derived to force $\dot{E}_{tr} < 0$, θ is necessarily going to be driven to 0, or equivalently $\widetilde{R} \to I_3$.

The following subsections derive nonlinear filters using this Lyapunov function, however, each filter is designed under varying levels of accurate assumptions. These filters build on top of eachother, however, and lead to a filter capable of using direct IMU sensor measurements to estimate attitude.

6.3.1.4 Complementary Filter

In [23], there are four separate filter designs. The first filter, termed Complementary Filter (CF), is built as a baseline CF with the assumption that the true $R(t)$ and $\Omega(t)$ are known. The goal is to derive a dynamical equation for $\widehat{R}(t)$ such that $\widetilde{R} \to I_3$. The true kinematic system model is given as

$$\dot{R} = R\Omega_\times = (R\Omega)_\times R \tag{6.30}$$

where $R \in SO(3)$ is the current rotation from $\{B\} \to \{A\}$, and $\Omega \in \mathbb{R}^3$ is the angular velocity in $\{B\}$. The observer kinematics are designed as

$$\dot{\widehat{R}} = (R\Omega + k_p\widehat{R}\omega)_\times \widehat{R} \tag{6.31}$$

where $k_p > 0$ is a proportional gain, and $\omega \in \mathbb{R}^3$ is some correct term which is to be designed. Note that the form given in 6.31 matches closely to that given in 6.30. The primary difference is that now in the observer design, we add a lumped term which depends on a proportional gain, the current estimate of R, and an innovation term ω, in order to adjust our current estimate of R.

The goal is now to design some ω such that subsequent Lyapunov analysis on 6.28 results in $\widetilde{R} \to I_3$. From [22], [15], a good choice was found to be

$$\omega := \mathrm{vex}(\mathbb{P}_a(\widetilde{R})) = \mathrm{vex}(\mathbb{P}_a(\widehat{R}^\top R_y) \tag{6.32}$$

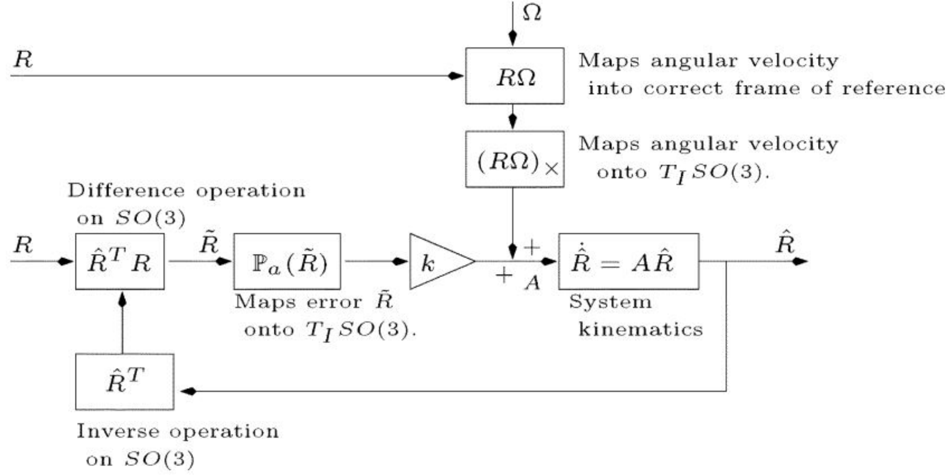

Figure 6.8: Complementary Filter [23]

where R_y is the current measurement of R, and

$$\mathbb{P}_a(\widetilde{R}) := \frac{1}{2}(\widetilde{R} - \widetilde{R}^\top) \quad (6.33)$$

This design choice for ω leads to a Lyapunov analysis (not shown here) which proves asymptotic stability on $\widetilde{R}$. This filter design has an equivalent block diagram structure shown in Fig. 6.8

The blocks related to $\mathbb{P}_a(\widetilde{R})$ and $(R\Omega)_\times$ are both operations to move to the tangent space of $SO(3)$ at the identity. Because algebraic functions are not allowed on $SO(3)$ directly, it is necessary to move to the tangent space in order to add or multiply terms. In this case, the gain term k is multiplied by $\mathbb{P}_a(\widetilde{R})$ in the tangent space, and then added to the $(R\Omega)_\times$ term which resides on $\mathfrak{so}(3)$. These then group together to form some correction matrix A, which aids in updating the current estimate via $\dot{\widehat{R}} = A\widehat{R}$.

This approach works well with a known rotation R, however, in practice, R

is not known exactly. Instead, an estimate $\hat{R}$ or measurement R_y needs to be used. There are two possible filter designs which [23] lays out using these alternatives – these are the *Direct Complementary Filter*, and the *Passive Complementary Filter*. The next subsection will focus on the passive filter, as that is what is used to derive the final *Explicit Complementary Filter*.

6.3.1.5 Passive Complementary Filter

In the passive complementary filter, the current estimate $\hat{R}$ is used in place of R, and a measured instance of angular velocity Ω_y is used in place of the true Ω. Substituting these terms into equation (6.31) yeilds

$$\dot{\hat{R}} = (\hat{R}\Omega_y + k_p\hat{R}\omega)_\times \hat{R} \tag{6.34}$$

A key point here is that the Lyapunov analysis which proves stability for (6.31) is also valid for (6.34). Following details from [23], equation (6.34) can be simplified further into

$$\dot{\hat{R}} = \hat{R}(\Omega_\times + k_p\mathbb{P}_a(\tilde{R})) \tag{6.35}$$

under the same choice for ω as in the first complementary filter design. The benefit of the form in 6.35 is that all operations can now take place in the body frame, and then get rotated after. This makes for a simpler block diagram as shown in Figure 6.9.

Assuming Ω_y can be estimated via the gyroscope model in (6.20), and R_y can be measured at low acceleration frequencies using the accelerometer model (6.22)

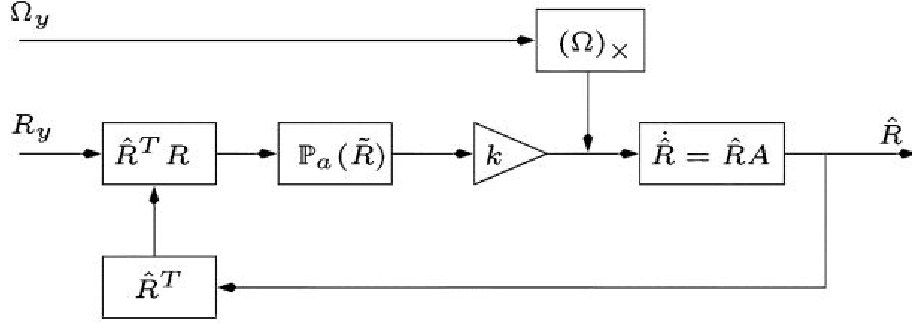

Figure 6.9: Passive Complementary Filter [23]

or measured under low magnetic disturbances via the magnetometer model (6.24), the following approximations are considered,

$$R_y \approx R \tag{6.36}$$

$$\Omega_y \approx \Omega + b \tag{6.37}$$

where R and Ω are the true rotation matrix and angular velocity respectively, and $b \in \mathbb{R}^3$ is a gyroscope bias term which may be estimated. Substituting these assumptions into the previously stated passive complementary filter (6.35), a new complete set of equations governing the final passive complementary filter is given as

$$\dot{\widehat{R}} = \widehat{R}(\Omega_y - \widehat{b} + k_p\omega) \tag{6.38}$$

$$\dot{\widehat{b}} = -k_I\omega \tag{6.39}$$

$$\omega = \text{vex}(\mathbb{P}_a(\widetilde{R})) \tag{6.40}$$

where $k_I > 0$ is an integral term who's purpose is to aid in estimating the bias term. An

analysis on this set of equations proves the filter to be asymptotically stable, however, the drawback is that the R_y term is still assumed to be measured directly. The next iteration on this the passive filter, coined the *Explicit Complementary Filter*, addresses this issue by utilizing accelerometer and magnetometer (inertial direction sensors) readings to replace the R_y assumption.

6.3.1.6 Explicit Complementary Filter

This final form of the Mahony filter will reformulate the previous versions using direct measurements from the accelerometer and/or magnetometer. To begin, let a set of n known inertial directions be represented by $v_{0i} \in \{A\}, i = 1, 2, .., n$. Measurements v_i are considered which are equivalent to body frame representations of these inertial vectors. Namely,

$$v_i = R^\top v_{0i} + \mu_i, \quad v_i \in \{B\} \tag{6.41}$$

where $\mu_i \in \mathbb{R}^3$ is small process noise. For an accelerometer, this process noise can be considered the acceleration which is not due to gravity, and for the magnetometer, this noise can be considered as magnetic field disturbances caused by nearby electronics. Because only the direction is important, all measurements should be normalized before being used. It should also be notes here, that typically only two inertial reference measurements are available ($n = 2$) – the accelerometer and magnetometer. Denote $\widehat{v}_i$ as the estimated inertial direction in the body frame $\{B\}$, thus we have

$$\widehat{v}_i = \widehat{R}^\top v_{0i} \tag{6.42}$$

As an example, if only the accelerometer is being used, this equation is equivalently described as

$$\frac{\vec{a}}{|\vec{a}|} = \widehat{R}^\top e_3 \tag{6.43}$$

where $\vec{a}$ is the acceleration measurement, and $e_3 = [0, 0, 1]^\top$. At zero forced acceleration (excluding gravity), the body frame acceleration measurement should match the inertial frame direction rotated by the current true rotation matrix. Mahony incorporates this idea into the passive complementary filter, turning it into the so-called explicit complementary filter

$$\dot{\widehat{R}} = \widehat{R}((\Omega_y - \widehat{b})_\times + k_p(\omega_{\text{mes}})_\times) \tag{6.44}$$

$$\dot{\widehat{b}} = -k_I \omega_{\text{mes}} \tag{6.45}$$

$$\omega_{\text{mes}} := \sum_{i=1}^{n} k_i(v_{0i} \times \widehat{v}_i) \tag{6.46}$$

where $k_i > 0$ is a confidence term that multiplies the cross product between the true inertial direction and the body frame representation of the inertial direction. For instance, if k_i is small, then the confidence in the usability of the inertial measurement is low. This makes sense if there are a lot of magnetic disturbances onboard for instance. This concept is not described well in Mahony's derivation, however, [32] provides an analysis on the variety of methods used to reject bad inertial estimates.

To summarize, there are two approaches to rejecting bad inertial measurements – hard switching, and soft switching. In hard switching, one checks if a threshold has

been passed, and if so, sets $k_i = 0$. Using the accelerometer as an example, if the maneuvers of the aerial vehicle are too quick, the inertial measurement estimate becomes unreliable and thus should not be used. In soft switching, this k_i term may take on a spectrum of values, starting closer to zero at higher accelerations, and increasing as the vehicle's acceleration becomes dominated by gravity. Model-based compensation of the for the inertial measurement term is also possible, however, is not considered during this thesis. The estimator designed for the quadrotor in this thesis implements hard switching for simplicity.

6.3.2 Implementation and Validation

To validate the observer design for the *explicit complementary filter* derived in Subsection 6.3.1.1, real IMU sensor measurements were taken during a flight experiment of the ST Drone. This data was processed, and piped into Matlab as a simulation of the real system. Loop delays were estimated at 800Hz as is realistic for the update rate of the sensor data. Finally, the explicit complementary filter was run against the sensor data, and the outputted estimates were compared against MOCAP measurements. Figures 6.10 and 6.11 present results for roll and pitch estimates from Mahony compared to estimates from MOCAP.

6.4 Geometric Control

With a geometric estimator in place, geometric control now makes sense. The specific controller implemented in this section comes from a class of rigid body controllers

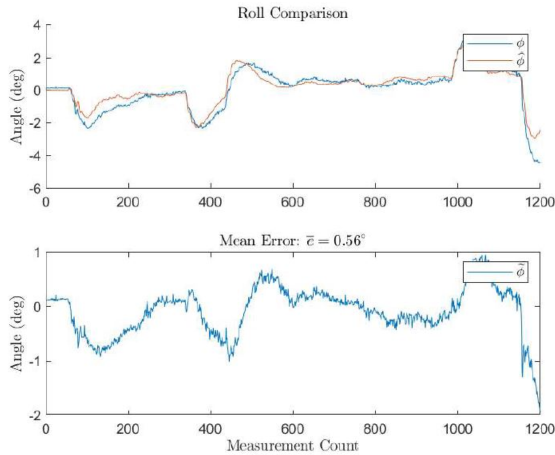

Figure 6.10: Top: Mahony roll estimate (red) compared to MOCAP truth (blue). Bottom: Error between Mahony and MOCAP. Mean error = 0.56°

designed on $SO(3)$ in [2] This section will first provide a brief summary of the attitude control design from [2], then discuss an approximation assumption being made, and finally present tracking results under both a simple dynamical model, as well as the full closed loop system from Section 6.2. These results serve as a basis for the justification of implementing the controller on hardware. This work is unique in that no other prior work has implemented this geometric controller on a physical system.

6.4.1 Controller Design

The controller design in [2] begins with a definition of the rigid body dynamics,

$$\dot{R}(t) = R(t)\Omega_{\times}(t) \tag{6.47}$$

$$J\dot{\Omega}(t) = \tau(t) - (\Omega(t) \times J\Omega(t)) \tag{6.48}$$

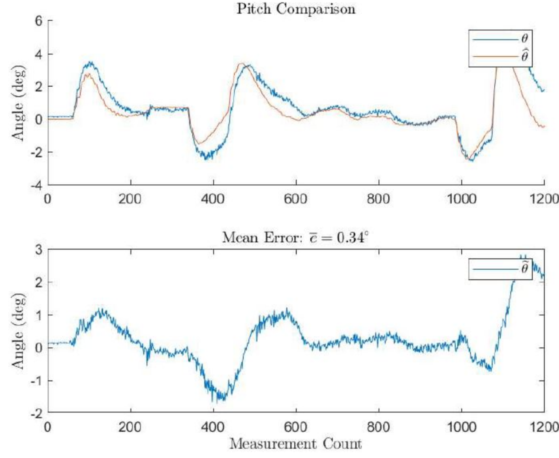

Figure 6.11: Top: Mahony pitch estimate (red) compared to MOCAP truth (blue). Bottom: Error between Mahony and MOCAP. Mean error = 0.34°

where $R \in SO(3)$ represents the current rigid body attitude, $\Omega \in \mathbb{R}^3$ is the current rigid body attitude rate, $\Omega_\times \in \mathfrak{so}(3)$ is the Lie Algebra representation of the current rigid body attitude rate, $J \in \mathbb{R}^{3\times 3}$ is the inertia matrix, and $\tau := [\tau_x, \tau_y, \tau_z]^\top \in \mathbb{R}^3$ are the torques acting on the rigid body. By designing τ as

$$\tau(t) = \Omega(t) \times J\Omega(t) + Ju(t), \tag{6.49}$$

it is clear that $J\dot{\Omega}(t) = Ju(t)$, thus

$$\dot{\Omega}(t) = u(t) \tag{6.50}$$

From this result, the control design in [2] is now derived assuming the following

set of attitude dynamics

$$\dot{R}(t) = R(t)\Omega_\times(t) \tag{6.51}$$

$$\dot{\Omega}(t) = u(t) \tag{6.52}$$

as well as the following exogenous system

$$\dot{R}_d(t) = R_d(t)\Omega_{d\times}(t) \tag{6.53}$$

$$\dot{\Omega}_d(t) = u_d(t) \tag{6.54}$$

In order to force the attitude (R, Ω) to track (R_d, Ω_d), a family of Lie Algebra valued functions, $\mathfrak{F}_R$, is designed which induce a controller class $\mathfrak{C}_R$. [2] describes, in detail, examples of functions in $\mathfrak{F}_R$ as well as the stability properties of the controllers they induce. The stability analysis is purposefully left out here, however, this thesis selects the following Lie Algebra valued function and subsequent geometric controller:

$$f := \mathrm{Log}(R_d^\top R) \tag{6.55}$$

$$\begin{aligned} u_\times :=& (R^\top R_d)\dot{\Omega}_{d\times}(R_d^\top R) + ((R^\top R_d)\Omega_{d\times} - \Omega_\times(R^\top R_d))\Omega_{d\times}(R_d^\top R) \\ &+ (R^\top R_d)\Omega_{d\times}((R_d^\top R)\Omega_\times - \Omega_{d\times}(R_d^\top R)) - (K_1 f^\vee)^\wedge - (K_2 \dot{f}^\vee)^\wedge \end{aligned} \tag{6.56}$$

where $f^\vee$ is equivalent to vex(f), and the $\wedge$ operator is equivalent to a skew symmetric operation. Ie. $(K_1 f^\vee)^\wedge : \mathbb{R}^3 \to \mathfrak{so}(3)$. This controller is almost-global. This is because the function f which induces u contains a matrix log operator that inherently is not

defined for every rotation on $SO(3)$ [3]. Explicitly, $\log(R_d^\top R)$ is not defined when the trace$(R_d^\top R)$ = -1. On $SO(3)$, the trace equals -1 at the following three points:

1. $R_d^\top R$ =diag(-1,-1,1)

2. $R_d^\top R$ =diag(-1,1,-1)

3. $R_d^\top R$ =diag(1,-1,-1)

These points correspond to the following set of euler angles:

1. $[\phi, \theta, \psi] = [\pi, 0, 0]$

2. $[\phi, \theta, \psi] = [0, \pi, 0]$

3. $[\phi, \theta, \psi] = [0, 0, \pi]$

This means that when the attitude error defined by $R_d^\top R$ reaches π on one of the axis, the matrix log operator and subsequently the controller break down. This may be fixed in practice by implementing approximations of the matrix log near these points. Another issue with the matrix log operator is the computational effort required to compute it. Often times attitude controllers need to operate at high frequencies. If the common (yet expensive) approach is taken to calculate the matrix log via eigenvalues, this high frequency bandwidth becomes an issue. This issue and solutions are discussed in the next section.

6.4.2 Matrix Log Approximation

As previously stated, a globally asymptotically stable geometric controller is impossible to design [9]. This fact lies in the nature of the definition of the matrix logarithm. Moreover, the matrix log is typically an expensive operation which involves calculating eigenvalues amongst other things. As this function will be run every single time the geometric controller function is called, this imposes a direct limitation on the loop rate. The current onboard loop rate is 800Hz which is approximately 1.25ms per loop. If the entire geometric controller takes longer than that, then the system begins to break. It is for this reason that a computationally efficient approximation of the matrix log is desirable.

6.4.2.1 Power Series Expansion Approximation

The power series expansion is capable of approximating various types of functions. It turns out that this expansion can also approximate the matrix log. Depending on the length of the power series, this approximation is more or less accurate. The theorem stated in [**?**] provides the following approximation for the matrix log

$$\mathrm{Log}(R) \approx \sum_{k=1}^{N} (-1)^{k+1} \frac{(R-I)^k}{k} = (R-I) - \frac{(R-I)^2}{2} + \frac{(R-I)^3}{3} - ... \tag{6.57}$$

where N is the length of the power series approximation. If R is sufficiently close to the identity (ie. $\|R - I\| < 1$) and $N \to \infty$, then $log(R)$ equals the right hand side exactly. Namely, $e^{log(R)} = R$.

The key factor in this approximation is how close R is to I. From the previously discussed geometric control design, the matrix log operator is going to be used on an error rotation term $\widetilde{R}$. It then suffices to say that so long as the approximated rotation $\widehat{R}$ does not stray too far from the true rotation R, and N is sufficiently large, then the approximation is good. These two assumptions, however, are not always valid. For one, if the error ever becomes too large, the system will likely become unstable in it's attempt to control itself on innacurate log approximations. Secondly, as N becomes larger, the number of computationally expensive exponent operations becomes larger – this again inhibits the low level controller from performing at a fast loop rate. The approximation method provided in the next subsection, however, does not fall short with respect to these two assumptions.

6.4.2.2 Rodriguez Method

Another widely used matrix logarithm method is known as the Rodriguez method [8]. This method defines the matrix log as follows:

$$\theta = \text{acos}(\frac{tr(R) - 1}{2}) \tag{6.58}$$

$$\widehat{u} = \frac{1}{2\text{sin}(\theta)}(R - R^\top) \tag{6.59}$$

$$log(R) = \theta\widehat{u} \tag{6.60}$$

This method is actually exact for the logarithm of matrices in the special orthogonal group. Moreover, it is well defined except for instances where the matrix

log, by definition, is undefined, and at the hover state $[\phi, \theta, \psi] = [0, 0, 0]$. At the hover state, $R = I_3$, the $tr(R) = 3$ causing $\hat{u}$ to blow up to infinity. Therefore, at $tr(R) = 3$, the matrix log is directly defined as

$$\text{Log}(R) := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad tr(R) = 3 \tag{6.61}$$

6.4.2.3 Result Comparison

Not only is the Rodriguez method exact for matrices on $SO(3)$, in practice, it is much faster than both the power series approximation and the traditional eigenvalue-based matrix log method. Both the power series approximation and the Rodriguez method were compared for a realistic set of data. In the following experiments, a constant current attitude is set as $R = I$, corresponding to the euler angles $[\phi, \theta, \psi] = [0, 0, 0]$. The desired roll then shifts from -200 to +200 degrees, in 1 degree increments. This change in desired euler angle is converted to a rotation matrix R_d, and the error rotation is computed as $\widetilde{R} = R^\top R_d$.

To be concise, a desired rotation matrix R_d is generated for every desired attitude from $[\phi_d, \theta_d, \psi_d] = [-200, 0, 0]$ to $[\phi_d, \theta_d, \psi_d] = [+200, 0, 0]$ in 1 degree increments. This desired attitude R_d is multiplied by the transpose of the constant attitude $R = I$. This $\widetilde{R}$ term is then fed into each respective log approximation function, and every element of the resulting 3x3 matrix is compared against a true matrix log which was

computed using Matlab's built in logm() function. The differences between the true matrix log element values and the approximated matrix log element values are plotted. These results are presented below in Figures 6.12 and 6.13.

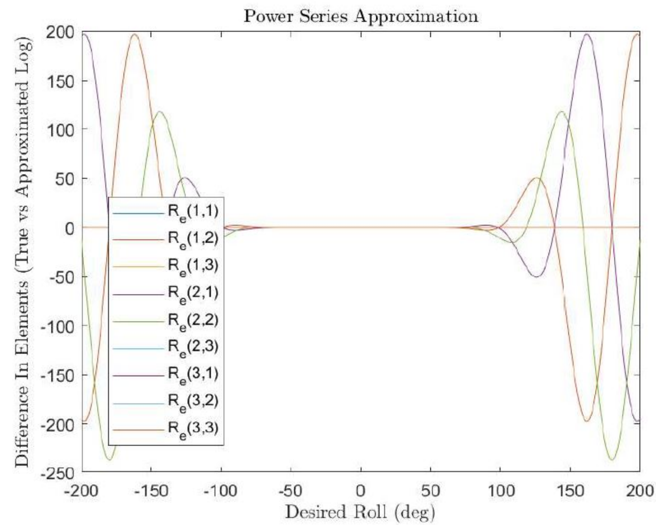

Figure 6.12: Power Series Log Approximation Experiment

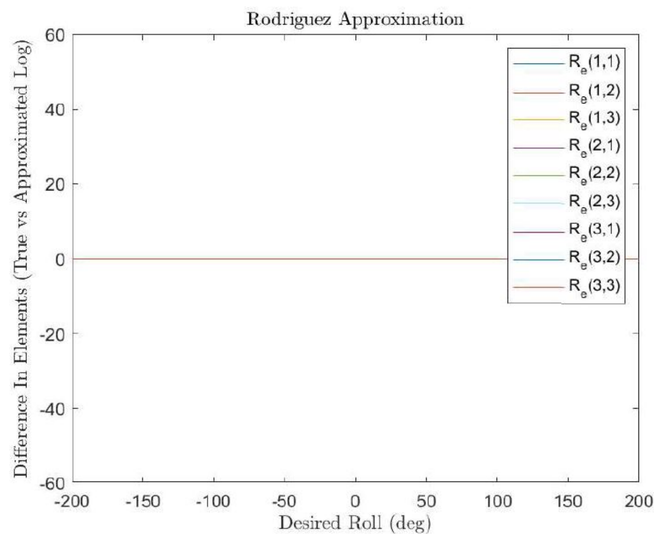

Figure 6.13: Rodriguez Method Experiment

Visually, one can easily tell the Rodriguez method outperforms the Power Series approximation. Again, this is because the Rodriguez method is exact on $SO(3)$.

The mean across the difference of all elements in the rodriguez approximation is on the scale of 10e-17. The power series approximation also has a mean on this scale, but only around the plus or minus 30 degree range. While although the ST Drone has a max attitude command of plus or minus 30°, the computational effort required to run the power series is much more than that of the Rodgriguez method. As such, there is virtually no reason to go with the power series method.

6.5 Results

Prior to a hardware implementation, simulations must validate the control for a set of rigid body dynamics. This subsection will describe two experiments – the first experiment applies the above geometric controller to a simple linear system. The second experiment incorporates the controller design into the quadrotor position controller from Section 6.2. Both of these implementations will make use of the previously discussed Rodriguez method for calculating the matrix logarithm. In Section 6.4 this method was shown to be both much faster to compute than traditional eigenvalue-based methods, and even exact for matrices on $SO(3)$.

6.5.1 Simple Dynamical System Simulation

Consider a system with rotational dynamics governed by the linear state space model

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{I_x} & 0 & 0 \\ 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & \frac{1}{I_z} \end{bmatrix} \tag{6.62}$$

$$, C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \mathrm{D} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

where the state $x = [\phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}]^\top$, control input $u = [\tau_x, \tau_y, \tau_z]^\top$, and inertia matrix $I = diag(I_x, I_y, I_z)$. This is a very simple state space model for rotational dynamics; however, it serves its purpose as the foundation for a proof of concept of the geometric controller. Next, a desired trajectory is generated as the pair R_d, Ω_d where the rotation matrix and rotation rate are computed via

$$R_d = R_z(\psi_d)R_y(\theta_d)R_x(\phi_d) \tag{6.63}$$

$$P_d = \text{vex}(\log(R_d)) \tag{6.64}$$

$$\Omega_d = \frac{P_d - P_{d^{\text{prev}}}}{\Delta T} \tag{6.65}$$

In the above set of equations, a desired rotation matrix is first generated using the desired euler angles and corresponding rotation matrices. Following this, a 3x1 vector of desired parameters, P_d, is extracted from R_d. These desired parameters are related to the desired attitude associated with the rotation matrix. Finally, Ω_d is calculated by differentiating the desired parameters for a small time-step ΔT. This pair (R_d, Ω_d) is then sent as an attitude command to the geometric controller. There are a variety of ways to define the pair (R_d, Ω_d), this is simply one method.

The attitude controller outputs torque commands which are directly injected into the dynamical system. The tracking ability of the controller is presented below for a time-varying desired attitude trajectory $[\phi_d, \theta_d, \psi_d] = [\frac{30\pi}{180}sin(t), \frac{10\pi}{180}sin(t), 0]$, where $t \in [0, 3]$ is measured in seconds.

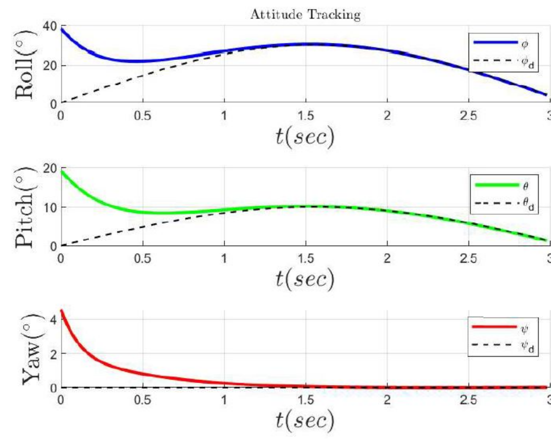

Figure 6.14: Geometric Attitude Tracking

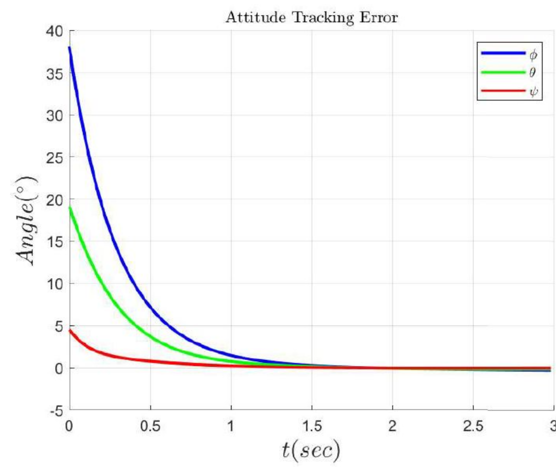

Figure 6.15: Geometric Attitude Tracking Error

The controller tracking results presented here are promising. They exhibit virtually no phase lag, which is common among PID controllers, and they are quick to converge. With the attitude tracking validated for a simple rotational dynamical model, the attitude controller can now replace the PID controller in the quadrotor control setup defined in Section 6.2. This new control design is detailed in the next section.

6.5.2 ST Drone Simulation

In Section 6.1 and 6.2, a model of a quadrotor was derived, and a control architecture similar the architecture described in Chapter 5 was designed. This architecture runs a position controller at 40Hz, which outputs attitude commands to a PID loop running at 160Hz, which finally outputs attitude rate commands to the inner-most PID loop running at 800Hz. This structure of operating an attitude control loop that feeds in attitude rate commands is referred to as PIV control. This PIV controller works by utilizing both attitude and attitude rate feedback. The geometric controller derived in Section 5.4 also operates on attitude and attitude rate feedback. This necessarily means that replacing the PID control scheme for the geometric control scheme means removing both the 160Hz outerloop and the 800Hz innerloop. Only the 800Hz inner loop is to be replaced by the geometric controller. This is a key difference. Rather than two loops operating at different rates, we now have one geometric control loop operating at 800Hz. This loop is effectively doing the job of both the 160Hz and 800Hz PID loops because it operates on command pairs R_d, Ω_d. After changing this structure, the new closed loop system block diagram is pictured in Figure 6.16.

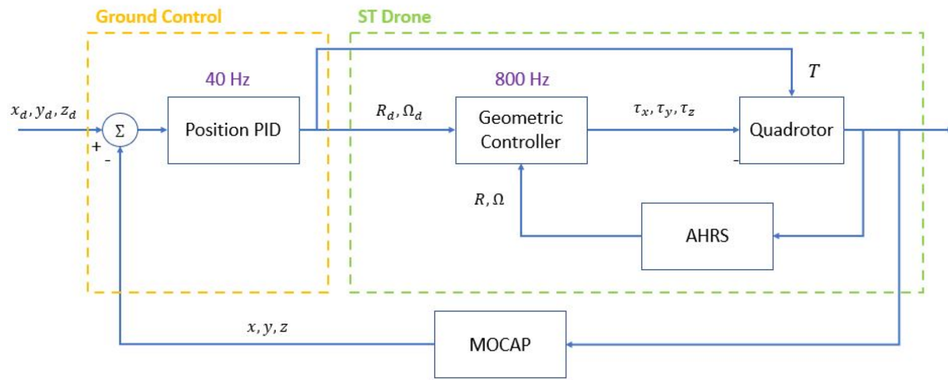

Figure 6.16: Closed Loop System with Geometric Control

With the two inner loop PID controllers removed, the control architecture is actually much simpler. It is also now possible to increase the rate of the position PID loop for performance. However, Optitrack limits the rate to 100Hz in this case.

Now that the geometric controller has replaced the inner loop PIV controller, it can be tested using the same position controller. Because the geometric controller seems to have faster convergence and no phase lag, one can expect this improvement to translate to the results of the position controller as well. In the subsequent results, a simulated experiment was performed in which a position setpoint of $[x_d, y_d, z_d] = [2m, 2m, 5m]$ is tracked. The geometric controller is expected to be more effective at tracking the desired attitudes produced by the position controller.

These results are similar to the simple rotational dynamical system test from the previous section. Convergence is fast with little to no phase lag. This is promising, and is a good starting point for transitioning the controller to hardware. While these results are strong, there is an inherent issue with the design of the control architecture. A large reason for implementing the attitude controller on $SO(3)$ was to remove any

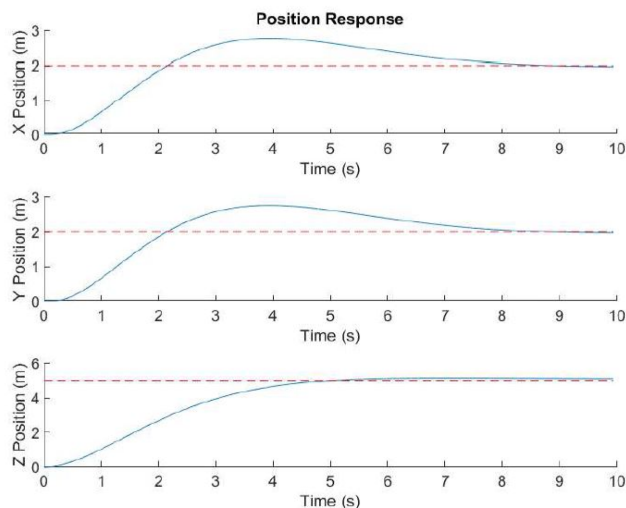

Figure 6.17: Geometric Position Control Results

issues related to the gimbal lock phenomena. While this controller implementaiton is a solid step in the right direction, it is not the entire solution. To completely remove the possibility of gimbal lock from the system, a geometric outerloop position controller needs to be implemented. As is, the PID position controller still outputs euler angles, which are then converted to rotation matrices. These euler angles are susceptible to gimbal lock, and just converting them to a rotation matrix on $SO(3)$ isn't enough; this is similar reasoning as to why we couldn't just use the original euler angle based AHRS and convert to rotation matrices. A geometric outerloop position controller is able to run feedback control on position, and directly output commands in $SO(3)$ (ie. output an R_d term). This thesis does not focus on implementing an outerloop geometric controller, but leaves it to future researchers taking on this project.

In the hardware implementation to come, a few key differences are found which are not present in simulation. These differences primarily arise in the requirement to

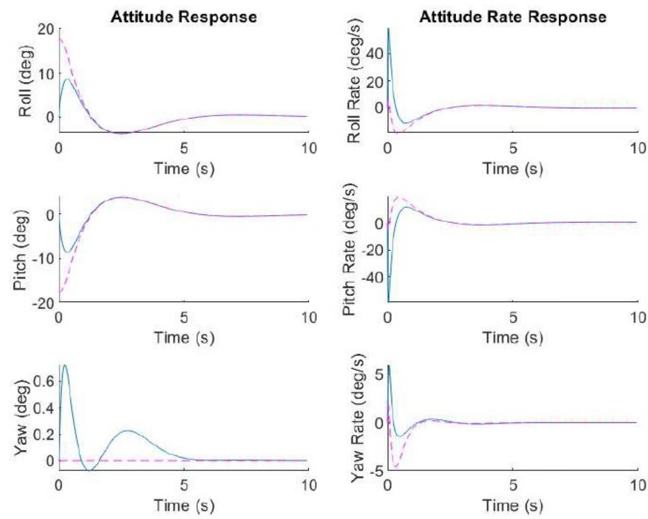

Figure 6.18: Geometric Attitude Control Results

filter the R_d, Ω_d pair, as well as the requirement to add an integral-like term into the control design in order to account for unknown trim conditions. The reason these issues do not appear in simulation is because this is the ideal case. For one, the exact inertia matrix is known, and two, the output of the controller is a true torque on the system. On hardware, the output is a desired torque which is not exactly related to the true torque. Moreover, the mass distribution of the drone is not consistent or symmetric. This means the inertia matrix, which appears in the geometric control law, is not well known. These issues are addressed in detail in the next chapter.

Chapter 7

Hardware Implementation

In Chapter 6, theory behind an attitude controller designed on $SO(3)$ was presented. This form of control is preferable over standard PID control techniques because of its nonlinear design, as well as ability to avoid gimbal lock issues. The specific controller used comes from a class of rigid body stabilizing controllers on $SO(3)$ given in [2]. A prerequisite to using this so-called Geometric controller, is the use of a Geometric (or $SO(3)$ based) estimator. The estimator chosen in this thesis follows [23]'s DCM implementation of the explicit complementary filter. This filter and controller were both tested and validated in simulation in Sections 6.3 and 6.4 respectively. The controller proved an ability to force fast trajectory tracking convergence for a simple rigid body kinematic system, as well as for a dynamical model of a quadrotor under discrete-time control methods similar to what is found onboard the ST Drone. These results are the foundation for a hardware implementation.

In this chapter, the hardware implementation of this controller, as well as

many of the nuances to it's successful deployment, are presented. Similar to the steps taken in Chapter 5, this chapter will begin with a description of the implementation and validation of the estimator in Section 7.1. Once the estimator has been verified, the geometric controller is implemented and tested in Section 7.2.

7.1 Geometric Estimator

In Section 6.3.2, the DCM version of the filter described in [23] was run against real IMU data collected during a flight experiment. The estimator performed well, however, the test results most likely do no represent the optimal performance. This IMU data was sent from the drone via Bluetooth which inherently limits the rate at which the measurements can be stored. While the simulated estimator ran on IMU data with a slower update rate, the hardware implementation will run in real-time. Still, the offline test was a necessary first step in proving the concept. Now, to implement and validate the estimator onboard, the following steps need to be taken:

1. Implement and validate a linear algebra library in C in order to write the algorithm onboard the drone. This library must be capable of performing the following functions:

    - **Matrix Multiplication & Division:** (3x3)*(3x3), (3x3)*(3x1), α*(3x3), α*(3x1).
    - **Matrix Addition & Subtraction:** (3x3), (3x1).
    - **Matrix Transpose:** For (3x3) matrices.

- **Skew Symmetric Operation:** To take $\Omega \in \mathbb{R}^3 \rightarrow \mathfrak{so}(3)$.
- **Cross Product:** For (3x1) vectors.
- **Vector Norm:** For (3x1) vectors.

2. Use the linear algebra library to implement the AHRS algorithm described in equation (6.44).

3. Verify the outputs of the onboard algorithm match with the outputs of the Matlab algorithm for the same set of hardcoded IMU measurements.

4. Tune the AHRS gains for good estimation performance.

A detailed discussion of all the linear algebra functions is provided in Appendix D. A noteworthy point here is that the magnetometer is unused in the current configuration. As such, accurate estimates of yaw may not always be available. More often than not, however, the gyroscope alone is able to estimate yaw over the short duration (1 minute) of flight experiments in the lab.

After completing the previously described steps, a working geometric estimator was implemented ombaord. The final validation step of comparing the original quaternion AHRS, our DCM version of the AHRS, and MOCAP proves the outputted estimates of this new scheme to be just as effective as the quaternion version.

In Figure 7.1, the top row shows attitude estimation for roll, and the bottom row shows attitude estimation for pitch. The RMSE's (as compared to *true* MOCAP measurements) are the following: $\widetilde{\phi} = 0.97°$ and $\widetilde{\theta} = 2.66°$ for ST (quaternion) AHRS

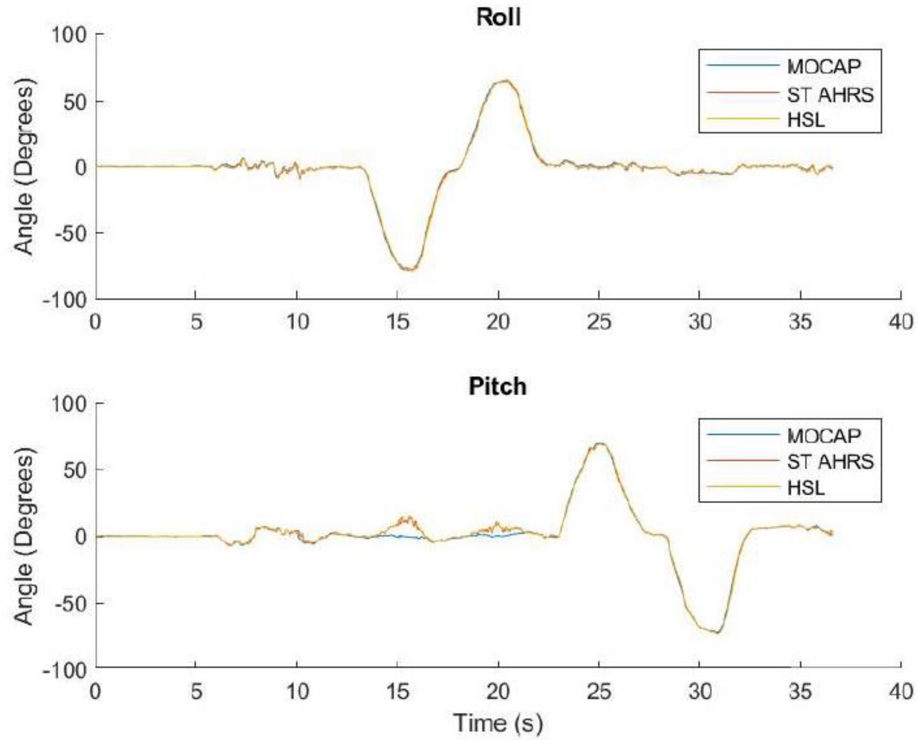

Figure 7.1: Mahony DCM Hardware Estimation Comparison

and $\widetilde{\phi} = 0.96°$ and $\widetilde{\theta} = 2.64°$ for the DCM AHRS version. Again, the larger error in both pitch estimates are due to the spikes in acceleration at the 15 and 20 second marks. The only inertial measurement device being used is the accelerometer, and because of this, the AHRS loses the inertial frame whenever the quadrotor accelerates too fast. This issue can be remedied by using a weighting term on the inertial direction which decreases as the norm of the acceleration measurement moves away from the expected gravitational constant. This effectively tells the AHRS to trust the inertial direction measurement less as the drone accelerates. These estimation results are expected to be similar to the original ST Drone AHRS, as the same filter was implemented, except using a new attitude representation. To reiterate, this step was necessary so that a

geometric controller can run and avoid the issue of gimbal lock. In the next section, this controller and a few nuances to its successful design are presented.

7.2 Geometric Controller

Again, with a geometric estimator implemented, a geometric controller with all its benefits can be utilized. There are a few significant differences in the hardware implementation of this geometric controller vs the theoretical implementation, however. For one, the controller is no longer directly commanding rigid body torques, nor is it going to use the exact inertia matrix of the system. Another key difference is that the physical system is not symmetric and so often times is not trimmed correctly; in practice, this leads to larges biases in attitude tracking. Raising the proportional and derivative gains on this geometric controller does not sufficiently reduce the steady state error, and tends to amplify higher frequency signals before it achieves desirable tracking behavior. To remedy this, an integral action is appended to the geometric controller. An issue related to the integral action, however, is that because a large integrator gain is needed to close down the steady state error quick, it is susceptible to forcing the drone to drift even when the command to the drone is to hover. Lastly, in the generation of a desired trajectory pair R_d, Ω_d, the Ω_d term needs a low pass filter applied to it. Without one, the numerical derivatives often contain frequency components which are too high. By tuning the low pass filter, one can effect how gentle or aggressive the desired trajectories should be.

The approach to implement this controller is similar to the approach taken to implement the estimator. These steps are as follows:

1. Write and validate extra required linear algebra functions. After the implementation of the AHRS, only one function is left to write, which is the matrix logarithm using Rodriguez's method.

2. Use this linear algebra library to implement control law (6.56).

3. Verify the outputs of the onboard controller match the outputs of the Matlab controller for the same set of inputs R_d, Ω_d and gains.

4. Implement R_d, Ω_d generation logic at the Matlab level to convert desired euler angles to a desired rotation matrix.

5. Test and tune the controller for performance.

Again, the linear algebra functions and code are provided in Appendix D. Additionally, the pair R_d, Ω_d are to be generated via equations (6.63, 6.65). Now, unlike in the previous section, the completion of these items did not result in a working controller. There are a few nuances which need to be addressed. These nuances were described earlier in this section; to summarize, here is a list of issues which the hardware version of this controller faces:

1. Because the controller does not directly output rigid body torques anymore, the true system inertia matrix J cannot be used as-is (even if it is known). J is

proportional to the output of the controller, but since the output of the controller is not rigid body torques, a different scaling needs to occur.

2. The physical quadrotor is not symmetrical, and is also likely not trimmed correctly. As such, large tracking biases are observed while running the controller.

3. The numerical derivative used to calculate Ω_d results in noisy command signals. This induces instability into the system.

Solutions to these issues are summarized in Subsections 7.2.1, 7.2.2, and 7.2.3 below. Then, results are presented in Subsection 7.2.4.

7.2.1 Inertia/Torque Solution

Because the output of the controller is no longer a 1:1 mapping for a rigid body torque, the output needs to be scaled. On the real system, the inertia matrix J contains elements on the scale of 10e-3. With an inertia matrix this small, this forces the desired torque output to also be very small. To fix this, we choose an arbitrarily large J matrix in order to generate torque outputs on a scale that matches up with the original PID attitude controller. An effective value for the inertia matrix is $J = \text{diag}(0.3, 0.3, 0.3)$.

7.2.2 Large Steady State Error Solution

The quadrotor not being symmetric nor well trimmed is actually a significant issue for this controller. In practice, the proportional and derivative gain components of the controller can only be increased so much before they begin amplifying high fre-

quency noise rather than reducing the steady state tracking error. For this reason, an integrator term is added which acts on the roll,pitch,yaw state errors. Care needs to be taken here while tuning, as a large integral gain is needed to reduce the steady state error to zero, however, a large integral gain forces situations where after an aggressive banking maneuver occurs and one attempts to level the drone out, leftover integration error causes the system to drift. With this solution implemented, the control law from equation (geometricControlLaw) now transforms into:

$$f := \mathrm{Log}(R_d^\top R) \tag{7.1}$$

$$\begin{aligned} u_\times := {} & (R^\top R_d)\dot{\Omega}_{d\times}(R_d^\top R) + ((R^\top R_d)\Omega_{d\times} - \Omega_\times(R^\top R_d))\Omega_{d\times}(R_d^\top R) \\ & + (R^\top R_d)\Omega_{d\times}((R_d^\top R)\Omega_\times - \Omega_{d\times}(R_d^\top R)) - (K_1 f^\vee)^\wedge - (K_2 \dot{f}^\vee)^\wedge - (K_3 \int f^\vee)^\wedge \end{aligned} \tag{7.2}$$

where $(K_3 \int f^\vee)^\wedge$ with a slight abuse of notation refers to $(K_3 \int (f dt)^\vee)^\wedge$. This is the term related to the integral of roll,pitch,yaw errors, and K_3 is the integrator gain.

7.2.3 Noisy Ω_d Solution

The solution to this problem is elegant in its simplicity, but also in its desirable effect on the system as a whole. Ω_d is noisy in practice because it is obtained via a simple numerical derivative on parameters of the desired rotation matrix, R_d. An obvious solution to this is issue is to run a lowpass filter on Ω_d. The beauty in this solution,

however, is that one may now tune this lowpass filter to effectively decide how gentle or aggressive of a trajectory the drone should take in achieving the desired orientation R_d. This gives the user one more tuning parameter to change the effects of the controller.

7.2.4 Results

With these solutions in effect, a series of tests were performed similar to those done for the PID attitude controller in Section 5.1. These tests show off the ability of the drone to converge quickly to commanded attitudes. In Figure 7.2, the quadrotor is shwon reacting to a roll command generated during manual flight.

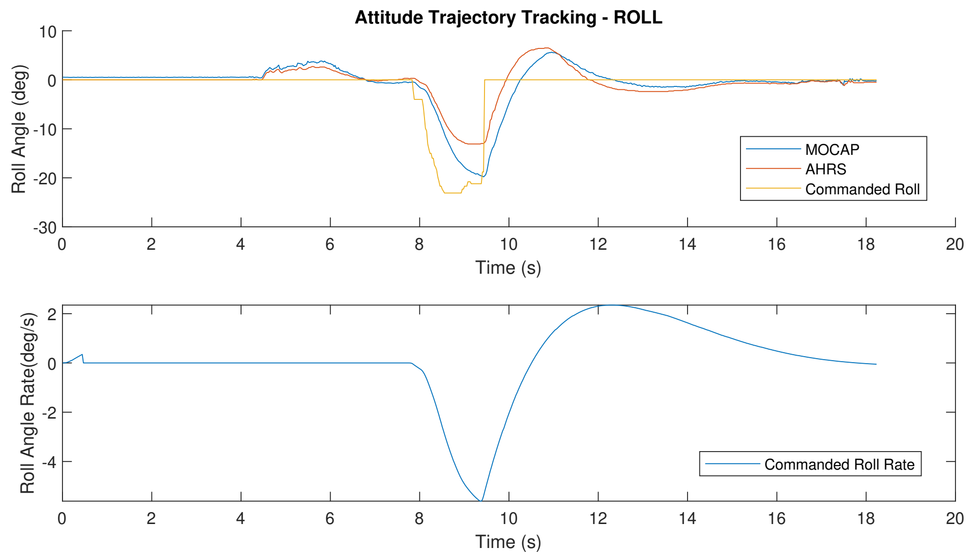

Figure 7.2: Geometric Attitude Control Hardware Roll Results

This geometric controller has a fast rate of convergence to the commanded signal. With this controller performing as-is, tests can now be performed to evaluate whether or not position tracking performance improves. Experiments similar to those run for the PID controller in Section 5.1 are now run for the geometric controller; that

is, a point position tracking test and circular trajectory position test are performed with the geometric attitude controller, using the original PID position controller. Figure 7.3 shows off point tracking results. The drone is placed some distance away from the center of the room, and is told to converge to the point $[x, y, z] = [0, 0, 0.7]$.

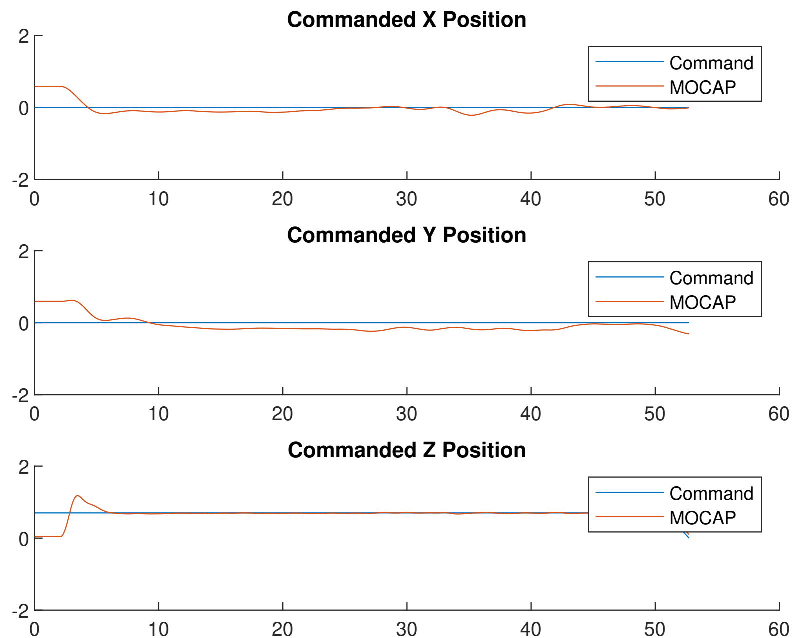

Figure 7.3: Position Tracking Performance with Geometric Attitude Controller

The drone effectively moves towards this desired position and stays there with little deviation from the reference. Next, a circular trajectory is generated with $\omega = 0.1$ rad/s (as done in Section 5.1), and the drone attempts to track the circle. These results are presented in Figure 7.4 below.

In this test, the drone exhibits an improvement in terms of phase lag over the PID attitude control scheme. This is expected as the nonlinear controller seemingly

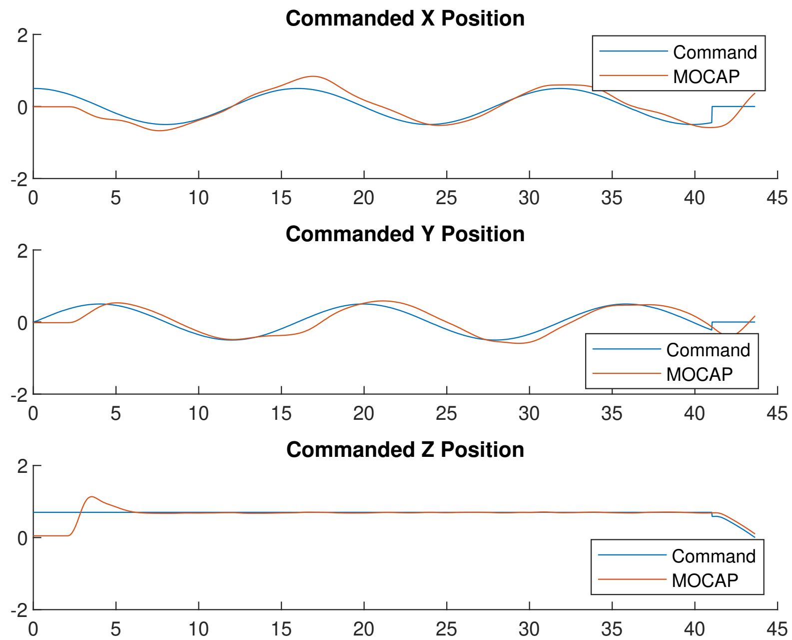

Figure 7.4: Position Tracking Performance with Geometric Attitude Controller

has faster convergence in simulation as well. With these series of tests complete, the geometric attitude controller has proven effective in both a simulation environment and on hardware in a real-world setting.

Chapter 8

Conclusion & Future Work

In this thesis, the ST Drone quadrotor platform was discussed in detail. The standalone version of this drone lacks much functionality required for use in autonomous guidance and control experimentation. In Part I of this thesis, a firmware extension was developed to address some of these issues such as no manual override method and an inconsistent communication channel. Upon completion of extending the platform to be compatible with autonomous flight controllers, a position controller was developed and validated through analyzing experimental results. Next, in Part II, an advanced geometric attitude control scheme was investigated. This control scheme improves upon standard PID attitude controllers by removing the possibility of gimbal lock, as well as not being confined to typical drawbacks of linear controllers. The specific geometric controller which was implemented is almost-globally asymptotically stable, meaning tracking behavior is theoretically much more effective at any point in the state space, as opposed to just near the equilibrium (which is where a linear controller should be

effective). This controller was tested in both simulation and hardware, and shown to improve the overall attitude tracking of the ST Drone.

Future work remains on both the hardware side, and theoretical side. As for hardware, improvements may be made on 1) real-time communication, and 2) utilizing the pressure sensor and magnetometer. The drone platform currently relies on using specific BLE characteristics to send data to the GCS. This is a limiting factor, as there is a pre-defined maximum amount of data which can be transmitted off of the drone. The HC12's transceiver properties should be investigated further to draw a stronger conclusion the feasibility of transmitting data via the HC12 whilst receiving data over the HC12. With respect to the second point, including the pressure sensor and magnetometer would help in the following ways: a) The pressure sensor allows for use of an onboard altitude controller which may replace the open-loop controller when in EOMC, b) The magnetometer may improve AHRS estimates of the drones orientation.

On the theoretical side, improvements may be made to both the position and attitude controllers. The position controller is not geometric, which means gimbal lock is still a problem which the system may face. To truly remove the possibility of gimbal lock, a geometric position controller should be implemented to directly command rotations matrices on $SO(3)$. Furthermore, the geometric attitude controller implemented onboard is limited in it's almost-global nature. Because no geometric attitude controller can provide global stability (as explained in Chapter 6), a hybrid solution should be investigated. Using hybrid feedback control techniques, it should be possible to seamlessly stitch together a series of geometric based controllers to create one

globally stable geometric attitude controller. This work would be novel, and a working hardware implementation of it would benefit the fields of hybrid, geometric, nonlinear controls.

Bibliography

[1] Mohamed Abbas-Turki, Gilles Duc, Benoit Clement, and Spilios Theodoulis. Robust gain scheduled control of a space launcher by introducing lqg/ltr ideas in the ncf robust stabilisation problem. In *2007 46th IEEE Conference on Decision and Control*, pages 2393–2398, 2007.

[2] Adeel Akhtar and Steven L. Waslander. Controller class for rigid body tracking on SO(3). *IEEE Transactions on Automatic Control*, 66(5):2234–2241, 2021.

[3] Akhtar, Adeel. *Nonlinear and Geometric Controllers for Rigid Body Vehicles*. PhD thesis, University of Waterloo, 2018.

[4] Rama Koteswara Rao Ana, Niraj Choudhary, J. S. Lather, and G. L. Pahuja. Piv and lead compensator design using lambert w function for rotary motions of srv02 plant. In *2014 IEEE 10th International Colloquium on Signal Processing and its Applications*, pages 266–270, 2014.

[5] Randal Beard. Quadrotor dynamics and control rev 0.1. 2008.

[6] Randal W. Beard. Quadrotor dynamics and control. 2008.

[7] HAROLD D. BLACK. A passive system for determining the attitude of a satellite. *AIAA Journal*, 2(7):1350–1351, 1964.

[8] José Luis Blanco-Claraco. A tutorial on **SE**(3) transformation parameterizations and on-manifold optimization, 2022.

[9] Nalin A. Chaturvedi, Amit K. Sanyal, and N. Harris McClamroch. Rigid-body attitude control. *IEEE Control Systems Magazine*, 31(3):30–51, 2011.

[10] C.-K. Chu, G.-R. Yu, E.A. Jonckheere, and H.M. Youssef. Gain scheduling for fly-by-throttle flight control using neural networks. In *Proceedings of 35th IEEE Conference on Decision and Control*, volume 2, pages 1557–1562 vol.2, 1996.

[11] George Ellis. Chapter 6 - four types of controllers. In George Ellis, editor, *Control System Design Guide (Fourth Edition)*, pages 97–119. Butterworth-Heinemann, Boston, fourth edition edition, 2012.

[12] Matthias Faessler, Antonio Franchi, and Davide Scaramuzza. Differential flatness of quadrotor dynamics subject to rotor drag for accurate tracking of high-speed trajectories. *IEEE Robotics and Automation Letters*, 3(2):620–626, 2018.

[13] Wojciech Giernacki, Piotr Kozierski, Jacek Michalski, Marek Retinger, Rafal Madonski, and Pascual Campoy. Bebop 2 quadrotor as a platform for research and education in robotics and control engineering. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1733–1741, 2020.

[14] Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweł Wroński,

and Piotr Kozierski. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 37–42, 2017.

[15] T. Hamel and R. Mahony. Attitude estimation on so[3] based on direct inertial measurements. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2170–2175, 2006.

[16] James Jackson, Daniel Koch, Trey Henrichsen, and Tim McLain. Rosflight: A lean open-source research autopilot. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1173–1179, 2020.

[17] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960.

[18] Hamid Saeed Khan and Muhammad Bilal Kadri. Attitude and altitude control of quadrotor by discrete pid control and non-linear model predictive control. In *2015 International Conference on Information and Communication Technologies (ICICT)*, pages 1–11, 2015.

[19] Shida Liu, Zhongsheng Hou, and Jian Zheng. Attitude adjustment of quadrotor aircraft platform via a data-driven model free adaptive control cascaded with intelligent pid. In *2016 Chinese Control and Decision Conference (CCDC)*, pages 4971–4976, 2016.

[20] Simone A. Ludwig and Kaleb D. Burnham. Comparison of euler estimate using

extended kalman filter, madgwick and mahony on quadcopter flight data. In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1236–1241, 2018.

[21] Sebastian Madgwick. An efficient orientation filter for inertial and inertial / magnetic sensor arrays. 2010.

[22] R. Mahony, T. Hamel, and J.-M. Pflimlin. Complementary filter design on the special orthogonal group so(3). In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 1477–1484, 2005.

[23] Robert Mahony, Tarek Hamel, and Jean-Michel Pflimlin. Nonlinear complementary filters on the special orthogonal group. *IEEE Transactions on Automatic Control*, 53(5):1203–1218, 2008.

[24] Vandana Mansur, Srinivasulu Reddy, Sujatha R, and R. Sujatha. Deploying complementary filter to avert gimbal lock in drones using quaternion angles. In *2020 IEEE International Conference on Computing, Power and Communication Technologies (GUCON)*, pages 751–756, 2020.

[25] Christopher Mayhew, Ricardo Sanfelice, and Andrew Teel. On quaternion-based attitude control and the unwinding phenomenon. pages 299 – 304, 08 2011.

[26] Christopher G. Mayhew, Ricardo G. Sanfelice, and Andrew R. Teel. On quaternion-based attitude control and the unwinding phenomenon. In *Proceedings of the 2011 American Control Conference*, pages 299–304, 2011.

[27] D.R. Mix, J.S. Koenig, K.M. Linda, O. Cifdaloz, V.L. Wells, and A.A. Rodriguez. Towards gain-scheduled h/sup /spl infin// control design for a tilt-wing aircraft. In *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, volume 2, pages 1222–1227 Vol.2, 2004.

[28] Fang Nan, Sihao Sun, Philipp Foehn, and Davide Scaramuzza. Nonlinear mpc for quadrotor fault-tolerant control. *IEEE Robotics and Automation Letters*, 7(2):5047–5054, 2022.

[29] Harris Teague. Comparison of attitude estimation techniques for low-cost unmanned aerial vehicles, 2016.

[30] Qiyu Wang, Jianping Yuan, and Zhanxia Zhu. The application of error quaternion and pid control method in earth observation satellite's attitude control system. In *2006 1st International Symposium on Systems and Control in Aerospace and Astronautics*, pages 4 pp.–131, 2006.

[31] Senlin Wang, Shangqiu Shan, Wei Luo, and Xinhong Yu. Robust adaptive nonlinear attitude control for a quadrotor uav. In *2017 36th Chinese Control Conference (CCC)*, pages 3254–3259, 2017.

[32] Yongjun Wang, Zhi Li, and Xiang Li. External disturbances rejection for vector field sensors in attitude and heading reference systems. *Micromachines*, 11:803, 08 2020.

[33] Yong Zeng, Qiang Jiang, Qiang Liu, and Hua Jing. Pid vs. mrac control tech-

niques applied to a quadrotor's attitude. In *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 1086–1089, 2012.

Appendix A

Attitude Heading and Reference System Using Quaternions

```
1  void ahrs_fusion_ag(AxesRaw_TypeDef_Float *acc, AxesRaw_TypeDef_Float *
     gyro, AHRS_State_TypeDef *ahrs)
2  {
3    float axf,ayf,azf,gxf,gyf,gzf;
4    float norm;
5    float vx, vy, vz;
6    float ex, ey, ez;
7    float q0q0, q0q1, q0q2, /*q0q3,*/ q1q1, /*q1q2,*/ q1q3, q2q2, q2q3,
     q3q3;
8    float halfT;
9
10
11   if(gTHR<MIN_THR)
12   {
13     ahrs_kp = AHRS_KP_BIG;
14   }
15   else
16   {
17     ahrs_kp = AHRS_KP_NORM;
18   }
19
20   axf = acc->AXIS_X;
21   ayf = acc->AXIS_Y;
22   azf = acc->AXIS_Z;
23
24   // mdps convert to rad/s
25   gxf = ((float)gyro->AXIS_X) * ((float)COE_MDPS_TO_RADPS);
26   gyf = ((float)gyro->AXIS_Y) * ((float)COE_MDPS_TO_RADPS);
```

```
27  gzf = ((float)gyro->AXIS_Z) * ((float)COE_MDPS_TO_RADPS);
28
29
30  // auxiliary variables to reduce number of repeated operations
31  q0q0 = q0*q0;
32  q0q1 = q0*q1;
33  q0q2 = q0*q2;
34  //q0q3 = q0*q3;
35  q1q1 = q1*q1;
36  //q1q2 = q1*q2;
37  q1q3 = q1*q3;
38  q2q2 = q2*q2;
39  q2q3 = q2*q3;
40  q3q3 = q3*q3;
41
42  // normalise the accelerometer measurement
43  norm = invSqrt(axf*axf+ayf*ayf+azf*azf);
44
45  axf = axf * norm;
46  ayf = ayf * norm;
47  azf = azf * norm;
48
49  // estimated direction of gravity and flux (v and w)
50  vx = 2*(q1q3 - q0q2);
51  vy = 2*(q0q1 + q2q3);
52  vz = q0q0 - q1q1 - q2q2 + q3q3;
53
54  ex = (ayf*vz - azf*vy);
55  ey = (azf*vx - axf*vz);
56  ez = (axf*vy - ayf*vx);
57
58  // integral error scaled integral gain
59  exInt = exInt + ex*AHRS_KI*SENSOR_SAMPLING_TIME;
60  eyInt = eyInt + ey*AHRS_KI*SENSOR_SAMPLING_TIME;
61  ezInt = ezInt + ez*AHRS_KI*SENSOR_SAMPLING_TIME;
62
63  // adjusted gyroscope measurements
64  gxf = gxf + ahrs_kp*ex + exInt;
65  gyf = gyf + ahrs_kp*ey + eyInt;
66  gzf = gzf + ahrs_kp*ez + ezInt;
67
68  // integrate quaternion rate and normalise
69  halfT = 0.5f*SENSOR_SAMPLING_TIME;
70  q0 = q0 + (-q1*gxf - q2*gyf - q3*gzf)*halfT;
71  q1 = q1 + (q0*gxf + q2*gzf - q3*gyf)*halfT;
72  q2 = q2 + (q0*gyf - q1*gzf + q3*gxf)*halfT;
73  q3 = q3 + (q0*gzf + q1*gyf - q2*gxf)*halfT;
74
75  // normalise quaternion
76  norm = invSqrt(q0q0 + q1q1 + q2q2 + q3q3);
77  q0 *= norm;
78  q1 *= norm;
79  q2 *= norm;
```

```
80   q3 *= norm;
81
82   ahrs->q.q0 = q0;
83   ahrs->q.q1 = q1;
84   ahrs->q.q2 = q2;
85   ahrs->q.q3 = q3;
86
87 }
```

Source Code A.1: ahrs_fusion_ag() - Filtered accelerometer and gyroscope measurements are fed into this function. Based on this sensor feedback, a nonlinear complementary filter [23] is used to estimate the drones current orientation in quaternions.

Appendix B

Attitude Heading and Reference System Using DCM

```
1
2 void ahrs_fusion_ag_2(AxesRaw_TypeDef_Float *acc, AxesRaw_TypeDef_Float
      *gyro, AHRS_State_TypeDef *ahrs){
3   // Get latest acc measurements
4   float accMes[3] = { acc->AXIS_X, acc->AXIS_Y, acc->AXIS_Z };
5     float g[3] = { 0.0, 0.0, 1.0 };
6
7   // Get latest gyro measurement (mdps convert to rad/s)
8   float gyrMes[3] = { ((float)gyro->AXIS_X) * ((float)COE_MDPS_TO_RADPS
    ), ((float)gyro->AXIS_Y) * ((float)COE_MDPS_TO_RADPS), ((float)gyro
    ->AXIS_Z) * ((float)COE_MDPS_TO_RADPS) };
9
10  // Store norm of acceleration
11  float accNorm = norm3(accMes);
12
13  if (accNorm > 0) {
14    // Get current inertial vector estimate
15    float accInerHat[3] = { 0 }; float Rtrans[3][3] = { 0 };
16    transpose3(ahrs->R, Rtrans);
17    mult33_31(Rtrans, g, accInerHat);
18
19    // Get inertial vector estimate error
20    float wMes[3] = { 0 }; float x1[3] = { 0 }; float x2[3] = { 0 };
21    scalarMult31(accMes, 1 / accNorm, x1);
22    cross31(x1, accInerHat, x2);
23    scalarMult31(x2, kW2, wMes);
24
25    // Update gyro bias estimate
```

```
26      float biasDot[3] = { 0 }; float x3[3] = { 0 };
27      scalarMult31(wMes, -AHRS_KI2, biasDot); // biasDot
28      scalarMult31(biasDot, SENSOR_SAMPLING_TIME, x3);
29      add31(ahrs->b, x3, ahrs->b); // bias+
30
31      // Update attitude estimate
32      float RhatDot[3][3] = { 0 }; float x4[3] = { 0 }; float x5[3][3] =
        { 0 }; float x6[3][3] = { 0 }; float x7[3][3] = { 0 }; float x8
        [3][3] = { 0 }; float x9[3][3] = { 0 };
33      sub31(gyrMes, ahrs->b, x4);
34      skewSym3(x4, x5);
35      skewSym3(wMes, x6);
36      scalarMult33(x6, AHRS_KP2, x7);
37      add33(x5, x7, x8);
38      mult33_33(ahrs->R, x8, RhatDot); // RhatDot
39      scalarMult33(RhatDot, SENSOR_SAMPLING_TIME, x9);
40      add33(ahrs->R, x9, ahrs->R); // Rhat+
41    }
42  }
43
44  void initAHRS(AHRS_State_TypeDef *ahrs)
45  {
46    ahrs->R[0][0] = 1.0f; ahrs->R[0][1] = 0.0f; ahrs->R[0][2] = 0.0f;
47    ahrs->R[1][0] = 0.0f; ahrs->R[1][1] = 1.0f; ahrs->R[1][2] = 0.0f;
48    ahrs->R[2][0] = 0.0f; ahrs->R[2][1] = 0.0f; ahrs->R[2][2] = 1.0f;
49
50    ahrs->b[0] = 0; ahrs->b[1] = 0; ahrs->b[2] = 0;
51
52    ahrs->Omega[0] = 0.0f; ahrs->Omega[1] = 0.0f; ahrs->Omega[2] = 0.0f;
53  }
```

Source Code B.1: ahrs_fusion_ag_2() - Filtered accelerometer and gyroscope measurements are fed into this function. Based on this sensor feedback, a nonlinear complementary filter [23] is used to estimate the drones current orientation on $SO(3)$.

Appendix C

HC12 Code

The HC12 radio module connects to the STM32F401 via the UART interface. This module has the capability of running in transceiver mode, however, making it work has proven difficult; this task is left up to future researchers. The radio is used in receiver mode onboard the drone. To do this, the DMA UART line must be used. Without using the DMA, the UART overrun error flag tends to get raised often, which causes the entire UART to hault. Once haulted, the UART must be restart, which takes somewhere on the scale of hundreds of milliseconds.

To configure DMA based UART, the following things need to happen: 1) Initialize DMA, 2) Initialize UART.

```
1
2 /* DMA2 Init Function */
3 void MX_DMA2_DMA_Init(void)
4 {
5   // DMA2_Stream2_IRQn
6   __HAL_RCC_DMA2_CLK_ENABLE();
7
8   hdma2_tx.Instance                    = DMA2_Stream7;
9   hdma2_tx.Init.Channel                = DMA_CHANNEL_4;
10  hdma2_tx.Init.Direction              = DMA_MEMORY_TO_PERIPH;
```

```
11   hdma2_tx.Init.PeriphInc           = DMA_PINC_DISABLE;
12   hdma2_tx.Init.MemInc              = DMA_MINC_ENABLE;
13   hdma2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
14   hdma2_tx.Init.MemDataAlignment    = DMA_MDATAALIGN_BYTE;
15   hdma2_tx.Init.Mode                = DMA_NORMAL;
16   hdma2_tx.Init.Priority            = DMA_PRIORITY_LOW;
17   hdma2_tx.Init.FIFOMode            = DMA_FIFOMODE_DISABLE;
18   hdma2_tx.Init.FIFOThreshold       = DMA_FIFO_THRESHOLD_FULL;
19   hdma2_tx.Init.MemBurst            = DMA_MBURST_SINGLE;
20   hdma2_tx.Init.PeriphBurst         = DMA_PBURST_SINGLE;
21
22   HAL_DMA_Init(&hdma2_tx);
23
24   /* Associate the initialized DMA handle to the the UART handle */
25   __HAL_LINKDMA(&huart1, hdmatx, hdma2_tx);
26
27
28    // Configure DMA here
29   hdma2.Instance                    = DMA2_Stream2;
30   hdma2.Init.Channel                = DMA_CHANNEL_4;
31   hdma2.Init.Direction              = DMA_PERIPH_TO_MEMORY;
32   hdma2.Init.PeriphInc              = DMA_PINC_DISABLE; // Stationary
       UART FIFO address
33   hdma2.Init.MemInc                 = DMA_MINC_ENABLE; // Increment
       memory
34   hdma2.Init.PeriphDataAlignment    = DMA_PDATAALIGN_BYTE; // Check on
        this in datasheet ----
35   hdma2.Init.MemDataAlignment       = DMA_MDATAALIGN_BYTE; // Check on
        this in datasheet ---- (MSIZE)
36   hdma2.Init.Mode                   = DMA_NORMAL;//DMA_CIRCULAR;//
       circ buf
37   hdma2.Init.Priority               = DMA_PRIORITY_MEDIUM; //VERY_HIGH
38   hdma2.Init.FIFOMode               = DMA_FIFOMODE_DISABLE; // Check
       on this in datasheet ---- FIFO threshold?=>DMA_FIFO_THRESHOLD_FULL
39   hdma2.Init.FIFOThreshold          = DMA_FIFO_THRESHOLD_FULL;
40   hdma2.Init.MemBurst               = DMA_MBURST_SINGLE;
41   hdma2.Init.PeriphBurst            = DMA_PBURST_SINGLE;
42
43   HAL_DMA_Init(&hdma2);
44   __HAL_LINKDMA(&huart1, hdmarx, hdma2);
45
46   // Enabling interrupt for tx
47   HAL_NVIC_SetPriority(DMA2_Stream7_IRQn, 0, 1);
48   HAL_NVIC_EnableIRQ(DMA2_Stream7_IRQn);
49
50   // Enable TF interrupt for RX
51   HAL_NVIC_SetPriority(DMA2_Stream2_IRQn, 0, 0);
52   HAL_NVIC_EnableIRQ(DMA2_Stream2_IRQn);
53
54 }
55
56
57 /* USART1 init function */
```

```
58 void MX_USART1_UART_Init(void)
59 {
60 
61   huart1.Instance = USART1;
62   huart1.Init.BaudRate = 38400; //19200,      115200 originally ..
63   huart1.Init.WordLength = UART_WORDLENGTH_8B;
64   huart1.Init.StopBits = UART_STOPBITS_1;
65   huart1.Init.Parity = UART_PARITY_NONE;
66   huart1.Init.Mode = UART_MODE_TX_RX;
67   huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
68   huart1.Init.OverSampling = UART_OVERSAMPLING_16;
69   huart1.Instance->CR1 |= USART_CR1_RXNEIE;
70   huart1.Instance->CR3 |= USART_CR3_EIE;
71   huart1.Instance->CR3 |= (0x1U << (6U)); // ENABLE DMA (DMAR bit) (TRY
       PUTTING AFTER THE HAL_UART_INIT())
72   HAL_UART_Init(&huart1);
73 
74 
75   HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
76   HAL_NVIC_EnableIRQ(USART1_IRQn);
77 
78 }
```

Source Code C.1: DMA UART Init

Once DMA and UART have been initialized to work together in this fashion, one can start receiving or transmitting. Only the receiving function is used onboard currently. To receive a packet onboard, we use the following line: *HAL_UART_Receive_DMA(&huart1, circBuf, 6)*. This will receive the 6 bytes of the packet. In the case that some bytes are lost in transit from the ground controller to the drone, the drone stores the bytes received in a circular buffer. This circular buffer is then checked for the latest valid packet command, and is acted upon appropriately. Circular buffer related code is found in the *HC12.c* source file. The function *processRadioBuffer()* is called every cycle of the main 160Hz loop, and is able to scan the circular buffer for the latest valid packet command. This can either be a data update request, or an attitude command. A status value is returned which signifies the type of packet. This code is found below:

```
1 char processRadioBuffer(uint8_t *circBuf, uint8_t circBufIndex, uint8_t
```

```
    *cmd)
2 {
3   uint8_t index = circularIndex(circBufIndex, 1);
4
5   int i;
6   for(i=0; i<96; i++){
7     if(circBuf[index] == PACKET_END_BYTE && circBuf[circularIndex(index
      , 5)] == PACKET_START_BYTE){
8       // Store the 4 bytes of data
9       cmd[0] = circBuf[circularIndex(index, 4)];
10      cmd[1] = circBuf[circularIndex(index, 3)];
11      cmd[2] = circBuf[circularIndex(index, 2)];
12      cmd[3] = circBuf[circularIndex(index, 1)];
13
14      // Reset the 6 bytes to zeros so as to not read this command from
        the buffer again
15      circBuf[index] = 0;
16      circBuf[circularIndex(index, 5)] = 0;
17      circBuf[circularIndex(index, 4)] = 0;
18      circBuf[circularIndex(index, 3)] = 0;
19      circBuf[circularIndex(index, 2)] = 0;
20      circBuf[circularIndex(index, 1)] = 0;
21
22      // Return status
23      if(isDataUpdate(cmd)){
24        return DATA_UPDATE;
25      } else {
26        return ATTITUDE_COMMAND;
27      }
28    }
29
30    index = circularIndex(index, 1);
31  }
32
33  return INVALID_COMMAND;
34 }
```

Source Code C.2: Process Circular Buffer Code

If the status returned is "ATTITUDE_COMMAND", the 4 bytes of the command will be used as described in Chapter 4.2. If "DATA_UPDATE" is returned, the command sequence will be handled via the *processDataUpdate()* function given below:

```
1 void processDataUpdate(uint8_t *cmd)
2 {
3   uint8_t request = cmd[2]; // request - Data Update Request Num (
      associated with the common dictionary)
4   uint8_t value = cmd[3];
5   switch(request){
6     case DR_UPDATE_ARM:
```

```
7         rc_enable_motor = value;
8         fly_ready = value;
9         armingStatusGlobal = value;
10        break;
11      case DR_UPDATE_CAL:
12        rc_cal_flag = value;
13        break;
14      case DR_UPDATE_CM:
15        ControlMode = value;
16        break;
17    }
18  }
```

Source Code C.3: Process Data Update Request Code

This function is easily extendable by adding cases to the switch statement, defining global 'extern' variables at the top of the source file, and using those to update any sort of data you may need to update onboard. This could be controller gains or even a switching flag meant to be used for hybrid control techniques.

Appendix D

Linear Algebra Functions in C

The following are a series of linear algebra functions required for implementing the geometric estimator and controller described in this thesis.

```
1
2 /* Function Definitions */
3 char logm(float a[3][3], float result[3][3])
4 {
5   // Rodgriguez Method for approximation matrix log
6   float traceA = a[0][0] + a[1][1] + a[2][2];
7   if(traceA >= 2.99999f){ // >2.99
8     logTrace = traceA;
9     for(int i=0; i<3; i++){
10      for(int j=0; j<3; j++){
11        result[i][j] = 0.0f;
12      }
13    }
14    return 1;
15  }
16  if(traceA < -0.9999f){
17    logTrace = traceA;
18    traceA = -0.9999f;
19
20    return 1;
21  }
22  if(traceA <= 3.0f && traceA >= -1.0f){
23    logTrace = 0.0f;
24    float theta = acos((traceA-1.0f)/2.0f);
```

```
25     float temp = 1.0f/(2.0f*sin(theta)); // This is a little off but is
        basically right
26
27     float A_trans[3][3] = { 0 }; float U[3][3] = { 0 }; float temp2
       [3][3] = { 0 };
28     transpose3(a, A_trans);
29     sub33(a,A_trans,temp2);
30     scalarMult33(temp2, temp, U);
31     scalarMult33(U, theta, result);
32     return 1;
33   }
34   return 0;
35 }
36
37
38 void transpose3(float a[3][3], float result[3][3])
39 {
40   result[0][0] = a[0][0];   result[0][1] = a[1][0];   result[0][2] = a
       [2][0];
41   result[1][0] = a[0][1];   result[1][1] = a[1][1];   result[1][2] = a
       [2][1];
42   result[2][0] = a[0][2];   result[2][1] = a[1][2];   result[2][2] = a
       [2][2];
43 }
44
45 void mult33_33(float a[3][3], float b[3][3], float result[3][3])
46 {
47   // Create a matrix here, store that results as it goes, then memcpy
       to result passed as reference
48   float temp_result[3][3] = { 0 };
49   for (int i = 0; i < 3; ++i) {
50     for (int j = 0; j < 3; ++j) {
51       float sum = 0;
52       for (int k = 0; k < 3; ++k) {
53         sum += a[i][k] * b[k][j];
54       }
55       temp_result[i][j] = sum;
56     }
57   }
58
59   memcpy(result, temp_result, sizeof(temp_result));
60 }
61
62 void scalarMult33(float a[3][3], float b, float result[3][3])
63 {
64   for (int i = 0; i < 3; i++) {
65     for (int j = 0; j < 3; j++){
66       result[i][j] = a[i][j] * b;
67     }
68   }
69 }
70
71 void scalarMult31(float a[3], float b, float result[3])
```

```
72 {
73   for (int i = 0; i < 3; i++) {
74     result[i] = a[i] * b;
75   }
76 }
77
78 void skewSym3(float a[3], float result[3][3])
79 {
80   result[0][0] = 0;      result[0][1] = -a[2];   result[0][2] = a[1];
81   result[1][0] = a[2];   result[1][1] = 0;    result[1][2] = -a[0];
82   result[2][0] = -a[1];  result[2][1] = a[0];  result[2][2] = 0;
83 }
84
85 void vex(float a[3][3], float result[3])
86 {
87   // May want to check to make sure all diag elements are zero
88   result[0] = a[2][1];
89   result[1] = a[0][2];
90   result[2] = a[1][0];
91 }
92
93 void mult33_31(float a[3][3], float b[3], float result[3])
94 {
95   for (int i = 0; i < 3; i++) {
96     result[i] = a[i][0] * b[0] + a[i][1] * b[1] + a[i][2] * b[2];
97   }
98 }
99
100 void cross31(float a[3], float b[3], float result[3])
101 {
102   float aCross[3][3] = { 0 };
103   skewSym3(a, aCross);
104   mult33_31(aCross, b, result);
105 }
106
107 void add33(float a[3][3], float b[3][3], float result[3][3])
108 {
109   for (int i = 0; i < 3; i++) {
110     for (int j = 0; j < 3; j++) {
111       result[i][j] = a[i][j] + b[i][j];
112     }
113   }
114
115 }
116
117 void sub33(float a[3][3], float b[3][3], float result[3][3])
118 {
119   for (int i = 0; i < 3; i++) {
120     for (int j = 0; j < 3; j++) {
121       result[i][j] = a[i][j] - b[i][j];
122     }
123   }
124 }
```

```
125
126 void add31(float a[3], float b[3], float result[3])
127 {
128   for (int i = 0; i < 3; i++) {
129     result[i] = a[i] + b[i];
130   }
131 }
132
133 void sub31(float a[3], float b[3], float result[3])
134 {
135   for (int i = 0; i < 3; i++) {
136     result[i] = a[i] - b[i];
137   }
138 }
139
140 float norm3(float a[3])
141 {
142   return (float)sqrt(a[0] * a[0] + a[1] * a[1] + a[2] * a[2]);
143 }
```

Source Code D.1: Linear Algebra Library - A series of linear algebra functions necessary for geometric estimation and control.