William Neeley

Candidate

Electrical and Computer Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Rafael Fierro , Chairperson

Dr. Chaouki Abdallah

Dr. Svetlana Poroseva

Design and Development of a High-Performance Quadrotor Control Architecture Based on Feedback Linearization

by

William Neeley

B.S., Electrical Engineer, University of Alaska Fairbanks, 2013

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

December 2015

Dedication

I would like to dedicate my thesis to my wonderful parents, who have always been there for me and helped me when I needed it.

I would also like to dedicate my thesis to my wonderful girlfriend, Chio. Her love and support has been an invaluable aid to my studies.

Acknowledgments

I would like to thank my advisor, Dr. Rafael Fierro, for enabling me to work on such a challenging problem.

I would also like to thank my friends in the MARHES lab (Rafael Figueroa, Paul Groves, Corbin Wilhelmi, and Patricio Cruz) for contributing to my understanding of the project, helping me work through places where I got stuck, and for providing moral support. I would like to extend a special thank you to Paul for configuring the vast majority of the settings on the Autopilot's high level processor.

I would also like to thank my friends from University of Alaska Fairbanks, especially Robert Barnett and Benjamin Montz, for acting as sounding boards throughout my thesis.

I would also like to thank my high school Physics teacher, Mr. Bill Ennis. His excellent teaching style encouraged me to pursue a degree and a career in electrical engineering, for which I am eternally grateful.

Design and Development of a High-Performance Quadrotor Control Architecture Based on Feedback Linearization

by

William Neeley

B.S., Electrical Engineer, University of Alaska Fairbanks, 2013

M.S., Electrical Engineering, University of New Mexico, 2015

Abstract

The purpose of this thesis is to outline the development of a high-performance quadrotor control system for an AscTec Hummingbird quadrotor using direct motor speed control within a Vicon motion capture system environment. A Ground Control Station (GCS) acts as a user interface for selecting flight patterns and displaying sensor values. An on-board Intel Edison embedded Linux computer acts as the quadrotor's controller. The Vicon system measures the quadrotor's position and orientation, while the Hummingbird's stock AscTec Autopilot board provides inertial measurements and receives motor speed commands.

Based on the flight pattern set by the GCS, smooth and differentiable trajectories are generated. A control program was written for the Edison to obtain measurements, receive flight pattern commands, perform state estimation, calculate control laws, send motor speed commands to the Autopilot board, and log values. The program was written as a multithreaded C++ program for increased performance.

A feedback linearization of the quadrotor's dynamics was performed to account for its nonlinearities. A controller structure designed to ensure exponential Lyapunov stability was applied to the input-output linearized dynamics. The simplex method was used to aid the controller in pushing the Hummingbird's actuators for aggressive maneuvers within set input limitations.

The Edison's Wi-Fi capabilities enable it to contact the Vicon server directly for position and orientation measurements. Accelerations and angular velocities are measured by the Autopilot's inertial measurement unit (IMU). A quick state estimation process was implemented to filter the measured states, and state prediction was used to compensate for latency in the system.

A custom circuit board and communication framework was designed and assembled for interfacing the Edison with the Autopilot. The custom communication framework allowed for a 16 times speed improvement over the default settings while bypassing the stock wireless communication's inherently unreliable timing.

The Hummingbird's physical properties, such as propeller performance and rotational inertias, were characterized via static and step response experiments. The control system's flight performance was evaluated through simulation and experimental tests.

Contents

List of Figures

List of Figures

List of Tables

Glossary

Math Notation

$\mathbf{A}$	A sample matrix.
$\mathbf{A}^T$	Transpose of matrix $\mathbf{A}$.
$\mathbf{R}_i(\theta)$	Rotation matrix around axis $i = x$, y, z in $\{W\}$ by an angle θ.
$\vec{x}$	A sample vector.
$\hat{x}$	Estimate of vector $\vec{x}$.
$\tilde{x}$	Measurement of vector $\vec{x}$.
$\mathring{q}$	A sample quaternion.
$\mathring{q}^{-1}$	Quaternion inverse of quaternion $\mathring{q}$.
$\dot{x}$	Derivative of state x.
$x^{(n)}$	nth derivative of state x.
$\{A\}$	Reference frame A.
$\vec{x}^{\,A}$	Vector in reference frame A.
${}^2\mathbf{R}_1$	Rotation matrix that rotates an object from frame $\{1\}$ to frame $\{2\}$.

Reference Frames

$\{W\}$ World frame; origin is fixed to a calibrated point within the Vicon system.

$\{B\}$ Body frame; origin is fixed to the quadrotor's center of mass.

$\{H\}$ Heading frame; has the same origin and Z axis as $\{B\}$, but is one rotation around an axis defined in the XY plane of $\{W\}$ from being aligned with $\{W\}$. Using Euler XYZ terminology to convert from $\{B\}$ to $\{W\}$, $\{H\}$ would be the frame between the XY and Z rotation matrices.

Matrix Variables

$\mathbf{A} \in \mathbb{R}^{n \times n}$ Linear state matrix for state space equations with n states.

$\mathbf{B} \in \mathbb{R}^{n \times p}$ Linear input matrix for state space equations with n states and p inputs.

$\mathbf{F} \in \mathbb{R}^{m \times 1}$ Lie derivative matrix used for MIMO feedback linearization; comprised of $L_f^\rho h$ terms for m outputs.

$\mathbf{G} \in \mathbb{R}^{m \times m}$ Lie derivative matrix used for MIMO feedback linearization; comprised of $L_g L_f^{\rho-1} h$ terms for m outputs.

$\mathbf{H} \in \mathbb{R}^{m \times n}$ Kalman filtering matrix used to map n states to m measurements.

$\mathbf{I} \in \mathbb{R}^{n \times n}$ Identity matrix of dimension n.

$\mathbf{J} \in \mathbb{R}^{3 \times 3}$ Rotational inertia matrix of the quadrotor (assumed diagonal).

$\mathbf{K}$	Quick filter: Meshing matrix $\in \mathbb{R}^{n\times n}$ for n states. Kalman filters: Kalman gain matrix $\in \mathbb{R}^{n\times m}$ for n states and m measurements.
$\mathbf{M} \in \mathbb{R}^{4\times 4}$	Conversion matrix between squared motor speeds and resulting outputs of thrust and torques.
$\mathbf{P} \in \mathbb{R}^{n\times n}$	Positive definite matrix for generating the Lyapunov function for n states.
$\mathbf{P}_{i\|j} \in \mathbb{R}^{n\times n}$	Kalman filter estimate covariance matrix for n states. For $i = j+1$, represents a predicted estimate covariance; for $i = j$, represents an updated estimate covariance.
$\mathbf{Q} \in \mathbb{R}^{n\times n}$	Kalman filter prediction covariance matrix for n states.
$\mathbf{R} \in \mathbb{R}^{m\times m}$	Kalman filter measurement covariance matrix for m measurements.
$\mathbf{S} \in \mathbb{R}^{m\times m}$	Kalman filter innovation/residual covariance matrix for m measurements.
$\mathbf{W} \in \mathbb{R}^{3\times 3}$	Converts Euler angle rates to angular velocities.

Vector Variables

$\vec{x} \in \mathbb{R}^{13}$	State vector comprised of $\vec{r}$, $\dot{\vec{r}}$, $\mathring{q}$, and $\vec{\omega}$.
$\vec{u} \in \mathbb{R}^4$	Input vector comprised of T and $\vec{\Gamma}$.
$\vec{r} \in \mathbb{R}$	Position vector comprised of x, y, and z (m).
$\dot{\vec{r}} \in \mathbb{R}^3$	Velocity vector comprised of $\dot{x}$, $\dot{y}$, and $\dot{z}$ (m/s).
$\mathring{q} \in \mathbb{R}^4$	Quaternion vector comprised of q_0, q_i, q_j, and q_k.

$\vec{\Theta}$ Euler angle vector comprised of ϕ, θ, and ψ (rad).

$\vec{\omega} \in \mathbb{R}^3$ Angular velocity vector comprised of $\omega_{\mathbf{x}}$, $\omega_{\mathbf{y}}$, and $\omega_{\mathbf{z}}$ (rad/s).

$\vec{\Gamma} \in \mathbb{R}^3$ Torque vector comprised of τ_x, τ_y, and τ_z (N m).

State Variables

$x,\ y,\ z \in \mathbb{R}$ Quadrotor position along X^W, Y^W, and Z^W, respectively (m).

$\dot{x},\ \dot{y},\ \dot{z} \in \mathbb{R}$ Quadrotor velocity along X^W, Y^W, and Z^W, respectively (m/s).

$q_0,\ q_i,\ q_j,\ q_k \in \mathbb{R}$ Unit quaternion values representing the quadrotor's orientation in $\{W\}$.

$\phi,\ \theta,\ \psi \in \mathbb{R}$ Roll (X rotation), pitch (Y rotation), and Yaw/Heading (Z rotation) angles in an Euler XYZ representation (rad). The angles are restricted between $\pm\pi$.

$\omega_{\mathbf{x}},\ \omega_{\mathbf{y}},\ \omega_{\mathbf{z}} \in \mathbb{R}$ Angular velocity around X, Y, and Z axes, respectively (rad/s).

Input Variables

$T \in \mathbb{R}$ Net thrust generated by the propellers (N).

$\tau_x,\ \tau_y,\ \tau_z \in \mathbb{R}$ Net torque generated by the propellers around X^B, Y^B, and Z^B, respectively (N m).

$\Omega_n \in \mathbb{R}$ Angular velocity of motor/propeller $n = 1, 2, 3, 4$ (rad/s).

Model Variables

$M \in \mathbb{R}_+$	Mass of the quadrotor, protective frame, and Edison board.
$\mathbf{J} \in \mathbb{R}^{3\times3}$	Rotational inertia matrix for the quadrotor (kg m^2).
$\mathbf{J}_{xx}, \mathbf{J}_{yy}, \mathbf{J}_{zz} \in \mathbb{R}_+$	Diagonal entries of $\mathbf{J}$ associated with rotations around the X^B, Y^B, and Z^B axes, respectively (kg m^2).
$b \in \mathbb{R}_+$	Thrust coefficient for a set of propellers (N/rpm^2 or N/$(rad/s)^2$).
$k \in \mathbb{R}_+$	Drag coefficient for a set of propellers (N m/rpm^2 or N m/$(rad/s)^2$).
$l \in \mathbb{R}_+$	Length of quadrotor arm (m).

Chapter 1

Introduction

This chapter provides an introduction to the quadrotor control system developed within this thesis. Section 1.1 discusses the motivation behind developing the control system. Section 1.2 provides a problem statement for declaring the scope of work. Section 1.3 examines related works and discusses their applicability to the system design. Section 1.4 lays out the organization of the thesis.

1.1 Motivation

The University of New Mexico's Multi-Agent, Robotics, Hybrid, and Embedded Systems (MARHES) Laboratory has three AscTec Hummingbird quadrotors. Previous work in the MARHES Lab [1, 2] examined treating the quadrotor's default Autopilot controller as a black-box model, characterizing its behavior, and developing a means of generating paths such that the quadrotor flew in a desired pattern. The closed source and unchangeable default controller constricted maneuverability, limited the control scheme to waypoint navigation, and fixed the response time.

Aggressive maneuvarability is key when performing tasks where the quadrotor must react quickly. The MARHES Lab has been researching control of quadrotors with suspended loads [3, 4, 5, 6], which alters the dynamics of the quadrotor. For such applications, being able to react quickly with aggressive, yet small, compensations prevents the need to use slower and large compensations later on. While current results for these tests are quite positive, having a capacity for quicker responses can only help.

Some preliminary research went into examining a means of swinging up a single-axis rigid pendulum cantilevered from the side of a quadrotor. While changes in position could be used to perform the swing-up operation, initial work looked at doing so using yaw operations as in [7]. The waypoint control framework does not offer sufficient freedom to perform abrupt changes in yaw torque, so a new quadrotor control scheme is needed.

The current quadrotor control scheme handles all of the path generation on a Ground Control Station computer. The path commands are sent to the quadrotor serially over a wireless link, which opens up the command communication link to issues such as packet collisions and spectrum interference. In the event of high wireless interference, communication with the quadrotor could be broken and control of its flight behavior could be lost.

To overcome the default controller's limitations, this thesis designs a system that takes direct control of the quadrotor's motor speeds. Rather than relying on an unknown and unchangeable control algorithm, the quadrotor's performance will only be limited by the motor controllers' responsiveness. While the primary control scheme developed in this thesis will focus on following position and heading trajectories, the direct motor control framework enables the development of any number of desired controllers, such as a pendulum swing-up controller.

The architecture developed for this thesis has all control and state estimation performed on the quadrotor by an embedded Linux computer. This allows for hard-wired access to the Inertial Measurement Unit (IMU) data and motor command procedures, which is both faster and more reliable. The fast sample and control rate will allow the controller to react quickly, and the direct motor control permits rapid convergence to a desired trajectory. With the controller mounted in the quadrotor, mechanisms can be developed to fly the quadrotor safely using the on-board IMU even in the event of a failure or severe delay in external communications.

1.2 Problem Statement

This thesis develops a control framework capable of safely exercising the Hummingbird's full range of inputs by using direct motor speed control for aggressive maneuvers. A control structure using feedback linearization and exponential Lyapunov stability formulations accounts for the system's nonlinearities and converges to desired trajectories. Linear programming methods keep the inputs within the realm of feasibility. State estimation and latency compensation algorithms process incoming measurements and refine the quadrotor's state to improve controller performance.

A new on-board hardware configuration is developed using an embedded Linux computer for increased performance. Custom circuitry for the computer was designed and fabricated for providing power and interfacing with the quadrotor. Parameters related to the quadrotor's model, such as propeller performance coefficients and the quadrotor's rotational inertias, are characterized through experiments to improve model accuracy.

1.3 Related Work

Both linear and nonlinear methods for controlling quadrotors have been examined. Bouabdallah, Noth, and Siegwart examined the use of PID and LQ controllers on a quadrotor's attitude [8]. Nonlinear methods such as feedback linearization [9, 10] and backstepping [11, 12, 13] have been successfully applied to the quadrotor's postion and attitude. Different papers have used Euler angles [12, 14] and quaternions [9, 10, 13] for expressing the quadrotor's orientation. The Euler angle representations allow for more intuitive understandings of the quadrotor's orientation, while quaternions offer a computationally efficient, singularity-free alternative.

Fritsch, De Monte, Buhl, and Lohmann propose a feedback linearization of the quadrotor's dynamics using quaternions and apply a controller designed for exponential Lyapunov stability to the linearized system [9, 15]. The exponential Lyapunov controller enabled the quadrotor's performance to be tuned using one parameter, and the feedback linearization process accounted for the quadrotor's nonlinear dynamics. Their simplified controller tuning and computationally efficient nonlinear control scheme served as the foundation for the controller used in this thesis.

Quadrotor control facilities using the AscTec Hummingbird and motion capture system feedback, such as the Flying Machine Arena at ETH Zurich [16] and the quadrotor testbed at University of Pennsylvania [17], demonstrate that high degrees of performance can be achieved when implementing custom control procedures. A paper published on the Flying Machine Arena [16] proposes many fast and efficient algorithms for performing state estimation and latency compensation for improved flight performance.

1.4 Thesis Organization

This section provides an overview of the thesis' organization structure. Chapter 2 provides an overview of the overall quadrotor control system and describes each of the important components. Chapter 3 contains schematics, board layouts, and design decisions for the Intel Edison's custom Quadrotor Block, which is essential for getting the Edison to communicate with the AscTec Autopilot. Chapter 4 covers the UART link between the Edison and the Autopilot, particularly the baud rate configurations and data frames for passing information back and forth.

Chapter 5 provides an overview of using Euler angles and quaternions to represent an object's orientation in 3D space and lists equations that are referenced throughout the thesis. Chapter 6 begins discussing the mathematical model for a quadrotor's flight dynamics. Chapter 7 steps through the process of linearizing the flight dynamics via feedback linearization and places a linear controller on the linearized dynamics. Chapter 8 discusses methods for constraining the inputs to the realm of feasibility, which allows control laws to push the limits of the quadrotor's performance.

Chapter 9 discusses filtering methods used to estimate the state of the quadrotor and the difficulties in their real-time execution. Chapter 10 discusses the Intel Edison's full control program and how it ties into everything. Chapter 11 evaluates the physical model of the AscTec Hummingbird to provide reliable parameters in the main control program. Chapter 12 analyzes the flight performance results, and Chapter 13 provides a conclusion and mentions future work.

Chapter 2

The System

Clearly defined descriptions of a system's components are essential before in-depth analysis of component configurations can begin. Section 2.1 provides an overview of the quadrotor control system and describes how the components fit together. Section 2.2 covers the AscTec Hummingbird quadrotor and its physical parameters. Section 2.3 talks about the AscTec Autopilot board that comes on the Hummingbird, its high and low level processors, and their respective responsibilities. Section 2.4 talks about the Vicon motion capture system that gives position and orientation feedback. Section 2.5 discusses the Ground Control Station (GCS) and its role in sending flight pattern commands to the quadrotor. Section 2.6 examines the Intel Edison System on a Chip (SoC), its various capabilities, and its role as the system's main controller.

2.1 System Overview

Figure 2.1 shows a simplified block diagram of the system. The AscTec Autopilot board, which is directly incorporated into the AscTec Hummingbird quadrotor, is responsible for measuring various parameters at upwards of 1 kHz [18] and for relay-

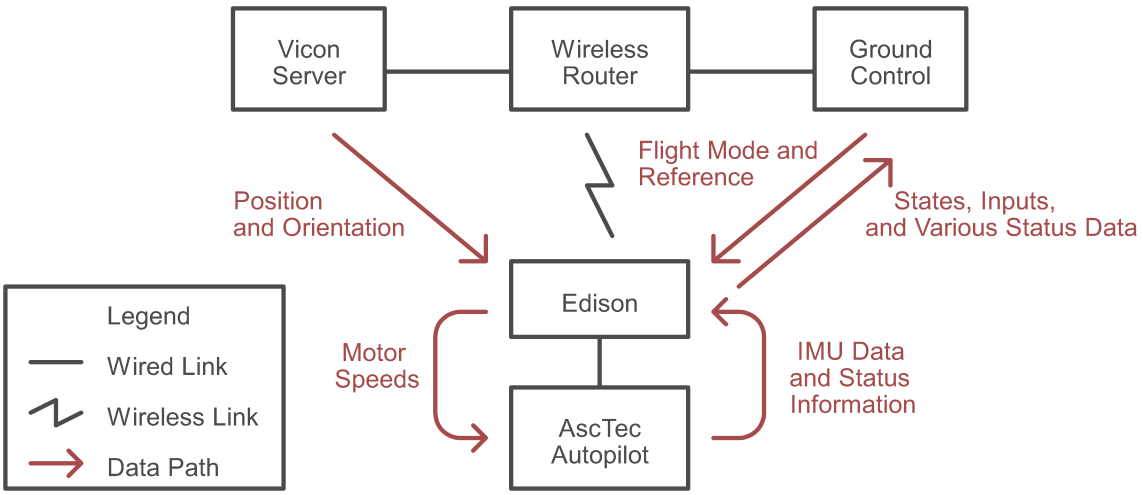


Figure 2.1: A simplified block diagram of the system, complete with a representation of the information passing between each component.

ing the quadrotor's motor speeds from the Intel Edison on-board computer to the proprietary motor controllers. The Vicon motion capture system uses cameras and reflective markers to track the quadrotor's position and orientation at upwards of 250 Hz [19]. The Ground Control Station (GCS) provides several flight mode controls and displays parameters for the person controlling the quadrotor. All three devices interact with the on-board Edison, which is tasked with receiving various measurements, communicating with the GCS, and generating the desired motor speeds.

Figure 2.2 shows a detailed block diagram of the communication links within the system. Many communication links, especially those related to Vicon and the Autopilot, are proprietary and were not changed. While many communication links are hard-wired and highly reliable from a signal strength and timing perspective, the weak link comes in the form of the Wi-Fi connection between the Edison and the wireless router. Due to issues like spectrum interference and packet collision, the Edison's communication with the Vicon server and the GCS are not guaranteed from a timing perspective. Special state estimation procedures were necessary to mitigate measurement delay/loss issues.

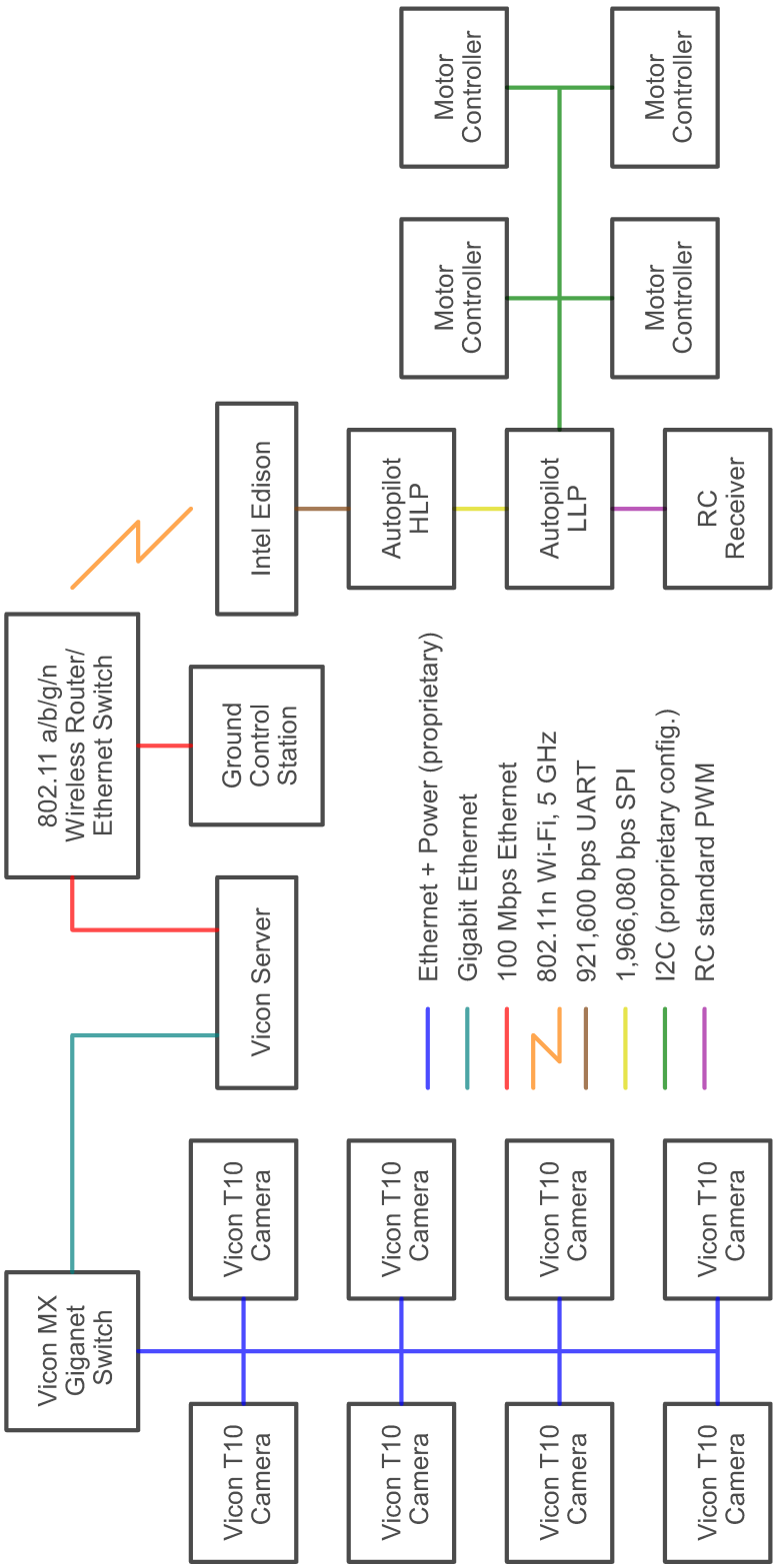


Figure 2.2: A diagram of every communication link used in the full system.

Figure 2.3: AscTec Hummingbird quadrotor with a large protection frame, Intel Edison control board, and Wi-Fi antenna. The orange tape indicates the "front" quadrotor arm.

Figure 2.3 shows a picture of the AscTec Hummingbird quadrotor with a protective frame, reflective markers for use with the Vicon system, a custom Intel Edison-based control board (described in detail in Chapter 3), and a Wi-Fi antenna for the Edison.

2.2 AscTec Hummingbird Quadrotor

The Hummingbird is a high-performance quadrotor designed by Ascending Technologies, GmbH (AscTec) [20]. The Hummingbird can be used straight out of the box as a remotely piloted UAV through a radio controller. Its Autopilot board (discussed in Section 2.3) was customized to allow direct motor control for this thesis. The Hummingbird's arms are composed of balsa wood sandwiched between carbon fiber sheets, which provide strength, rigidity, and robustness while staying light. The

central core of the quadrotor uses magnesium struts to hold the top and bottom carbon fiber panels, the AscTec Autopilot board, and the battery. Plastic screws and nuts hold everything together, which have the advantages of being light and, in the unfortunate event of a crash, providing failure points that absorb the crash energy to reduce the chances of breaking the arms. Proprietary brushless DC motors and their I^2C-based motor controller boards are attached to each of the quadrotor's four arms. Table 2.1 lists various properties for the Hummingbird.

The Hummingbird quadrotor can be configured with multiple propeller options. By default, AscTec supplies flexible propellers for durability at the cost of a lower thrust coefficient. The MARHES lab has acquired aftermarket propellers with higher thrust and drag coefficients, but they are brittle and sharp. As the flexible propellers provide adequate flight performance and are safer to operate, they were used for this thesis. Chapter 11 examines the measured thrust and drag coefficients for a set of flexible propellers. Table 2.2 lists the flexible propellers' default specifications.

Table 2.1: Specifications for the AscTec Hummingbird[1] [20]

Parameter	Value
Mass (M)	0.500 kg
Arm length (l)	0.17 m
$\mathbf{J}_{xx}$	0.00365 kg m^2
$\mathbf{J}_{yy}$	0.00368 kg m^2
$\mathbf{J}_{zz}$	0.00703 kg m^2
Battery Type	2100 mAh 3 cell LiPo

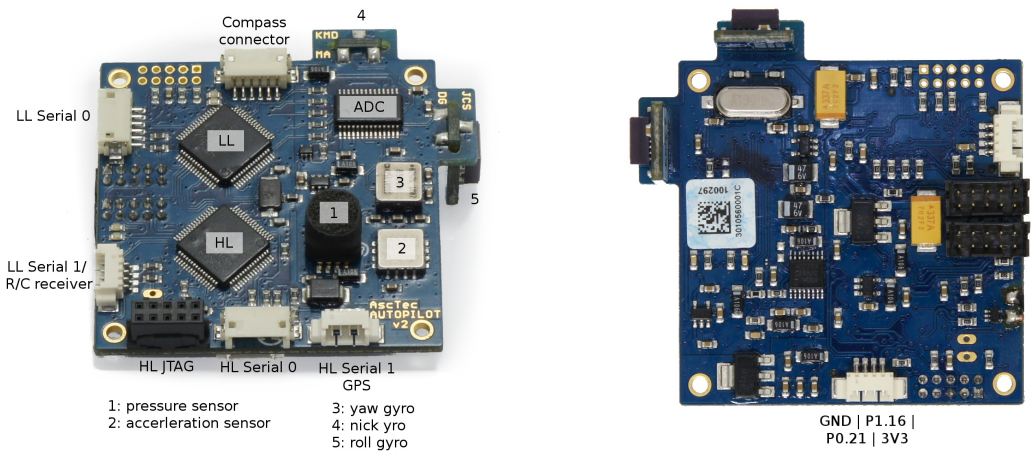
Table 2.2: Default Specifications for the AscTec Hummingbird's Flexible Propellers

Parameter	Value
Propeller Radius [20]	10 cm
Thrust Coefficient (b) [21]	6.11×10^{-8} N/(rpm)^2
Drag Coefficient (k)[21]	1.5×10^{-9} N m/(rpm)^2

[1]Mass/inertia specifications include the battery, but not the protective frame the Edison. See Chapter 11 for measured values.

2.3 AscTec Autopilot

The AscTec Autopilot comes installed as the default control board for the AscTec Hummingbird quadrotor [18]. Figure 2.4 shows the top and bottom of the Autopilot board with labeled parts, ports, and pins. The two main components of the Autopilot board are the High Level Processor (HLP, or HL in Figure 2.4a) and the Low Level Processor (LLP, or LL in Figure 2.4a). Figure 2.5 shows a block diagram of the two on-board processors, the sensors, and optional attachments such as an external GPS sensor and possible XBee or cable-based UART connections.



(a) Top (b) Bottom

Figure 2.4: The labeled pinout of the AscTec Autopilot board. Source: [22]

2.3.1 Low Level Processor (LLP)

The Autopilot's LLP is responsible for obtaining measurements from the Inertial Measurement Unit (IMU), battery voltage, and motor speeds on the Autopilot board. The IMU measures the quadrotor's accelerations and angular velocities at a rate of 1 kHz. The LLP is capable of performing data fusion on these values, performing

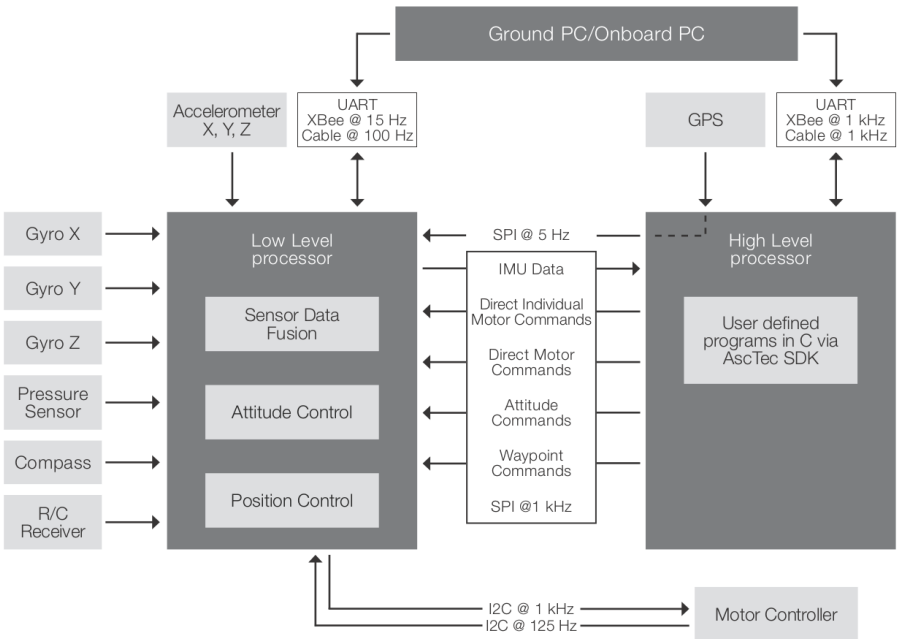


Figure 2.5: Block Diagram for the AscTec Autopilot and Possible Configurations. Source: [18]

attitude and position control algorithms, and sending motor speed commands to the motor controllers. As the purpose of this thesis is to develop a high-performance controller free of unchangeable response times and maneuverability restrictions, the LLP's default controllers and data fusion will not be used. The HLP can be used to augment and/or override the LLP's controllers, which will be discussed in Section 2.3.2.

An RC controller serves as the default method of interacting with the quadrotor. It handles turning on and off the motors, selecting the flight mode (Manual Mode, Height Mode, or GPS Mode), sending input commands (based on the selected flight mode), and enabling/disabling control via serial interfaces (whether through the LLP's serial port or from the HLP). Even though the HLP can override all these features, the RC controller acts as a safety mechanism for turning the motors on and off, so it is left in the system for this sole purpose.

The LLP's IMU is configured to follow an X-forward, Y-right, Z-down coordinate system in compliance with the DIN 9300 air norm [23]. The Euler ZYX convention is used on-board to represent the quadrotor's orientation in roll, pitch, and yaw. The LLP has the following sets of raw/calibrated measurements at it disposal [23] [24], all of which are measured within its frame of reference. Of the measurements available, the angular velocities, linear accelerations, motor speeds, and battery voltage will be used.

- Angular Velocities ($\vec{\omega}$)
  - Quantized at 0.0154 °/s, or 2.69 $\times 10^{-4}$ rad/s
  - Empirically observed to saturate at $\sim \pm$ 7.25 rad/s, or $\sim \pm$415.4 °/s
- Linear Accelerations ($\ddot{\vec{r}}^B$)
  - Calibrated around gravitational acceleration
  - Quantized at G/10,000
  - Stored as signed 16 bit integer, so theoretically saturated at $\sim \pm$3.27 Gs
- Motor Speeds (Ω_1 through Ω_4)
  - Quantized at 64 rpm
- Battery Voltage
  - Reported with 1 mV precision; ADC accuracy unknown
- Magnetometer Readings
- Temperature
- Air Pressure

With input values from a GPS system, the LLP can perform data fusion to obtain orientation angles, linear velocities, altitude, and magnetic heading. As the MARHES lab's location blocks GPS signals, the GPS data would need to be spoofed, so the data fusion utilities are not applicable.

The AscTec Communication Interface (ACI) allows for communicating with the LLP directly over its serial port (labeled "LL Serial 0" in Figure 2.4a) at 57,600 bps. The HLP UART configuration (Chapter 4) was 16 times faster with reduced communication overhead, so the ACI utility is neglected.

2.3.2 High Level Processor (HLP)

The AscTec Autopilot's High Level Processor (HLP) allows researchers to customize the quadrotor's behavior and leverage the on-board sensors for their own needs. Table 2.3 lists the processor's specifications. While the HLP is disabled by default, AscTec provides a Software Development Kit (SDK) for writing custom C code onto the HLP while providing a framework for obtaining data from/sending data to the LLP. All of the source code is provided, so any aspect of the processor's programming can be customized, from the program loop rate (set to 1,000 Hz by default) to the individual GPIO pins. This is in stark contrast with the LLP, for which the only reprogramming options consist of firmware updates and calibration procedures.

Table 2.3: Specifications for the AscTec Autopilot's High Level Processor

Parameter	Value
Manufacturer	Philips (now NXP Semiconductors)
Model	LPC2146
Architecture	32-bit ARM7
Crystal Oscillator	14.7456 MHz
Clock Speed	58.9824 MHz
On-chip SRAM	32 kB + 8 kB
On-chip FLASH	256 kB

The HLP can send commands to the LLP in four different ways:

- GPS Waypoint Navigation: the LLP takes care of all flight control and flies to commanded waypoints.
- Attitude/Thrust Control: the LLP's attitude controller receives commands as if they were coming from the remote control sticks. These values translate into a desired pitch angle, a desired roll angle, a desired yaw rate, and net thrust.
- Standard Output Control: the LLP's attitude controller is disabled, and commanded values for thrust (0 to 200 $\rightarrow$ 0% to 100%) and pitch, roll, and yaw torques (0 to 200 $\rightarrow$ -100% to +100%) are executed.
- Direct Motor Control (abbreviated as DMC): the LLP's attitude controller is disabled, and the motor speeds can be set directly via motor command values (0 = off, 1 to 200 $\rightarrow \Omega_{\min}$ to $\Omega_{\max}$).

The first three options are undesirable, as the GPS and Attitude/Thrust methods use the on-board controller and the Standard Output method doesn't provide units for its thrust/torque commands. As the DMC method has a specific equation for relating motor commands and motor speeds (See Chapter 11) and it provides complete control freedom, this method is used to control the quadrotor.

The HLP is programmed to communicate with the Intel Edison via a high-speed serial UART connection. The HLP reports IMU frames (acceleration and angular velocity measurements) at 200 Hz and Status frames (battery voltage, HLP load, LLP status, and measured motor speeds) at 100 Hz. The Edison will attempt to send direct motor speed commands at approximately 200 Hz. Chapter 3 addresses the physical circuitry necessary for the UART link, while Chapter 4 covers the various settings, the fixed-size frame structures used to pass data back and forth, and the performance of the link.

2.4 Vicon MX Motion Capture System

The Vicon motion capture system provides a means of capturing an object's position and orientation at a rate of up to 250 Hz using cameras in tandem with reflective markers [25]. Figure 2.6 shows a diagram of the Vicon system configuration in the MARHES lab. Eight Vicon T10 cameras are positioned around a capture volume of 16 feet wide, 22 feet long, and 9 feet tall. The cameras are oriented such that they cover as much of the volume as possible while providing sufficient overlap for accurate triangulation of any reflective markers they see. The T10 cameras are attached to a Vicon MX Giganet Ethernet switch via proprietary cables that provide Gigabit Ethernet, power, and camera synchronization and identification. The Vicon MX Giganet is attached via Gigabit Ethernet to a host computer with a Vicon Ethernet card, a generic Ethernet port, and the Vicon Tracker v1.3 application installed.

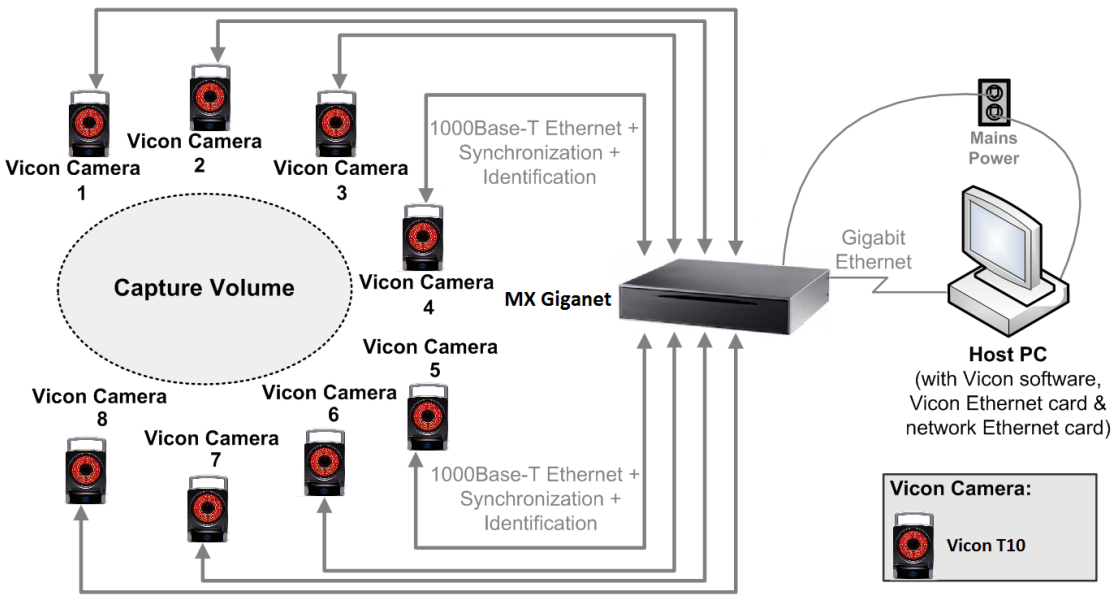


Figure 2.6: Configuration of the Vicon motion capture system in the MARHES lab. The image was edited from its original source to reflect the updated hardware. Original source: [25].

Table 2.4: Vicon T10 Camera Specifications [19]

Resolution (pixels)	1120 × 896 (1 MP)
Sensor Size (mm)	7.84 H x 6.27 V
Full Resolution Max Frame Rate	250 fps
Strobe Wavelength	623nm (Visible Red)

The motion capture process starts with the Vicon T10 cameras [19]. Table 2.4 lists the specifications for the T10 cameras used in the MARHES lab. At any given moment, one camera is selected by the host computer to capture data. The selected camera illuminates its LEDs with a configurable intensity to shine a particular wavelength of light into the capture volume. Any reflective markers within the capture volume will efficiently reflect the light back to the camera. An optical filter in front of the camera filters out all but the desired wavelength of light. The grayscale sensor within the camera then calculates the size and centroid of groupings of light that pass a brightness threshold (configurable via Vicon Tracker). When properly configured, the threshold will only permit reflective markers to be detected. Once the camera has calculated the centroid and radius of any reflective markers it detects, this information is relayed through the MX Giganet switch to the host computer.

Once all eight cameras have relayed their information at a configurable frequency, the host computer will use calibrated settings to triangulate the position of various markers relative to a calibrated origin. The Vicon Tracker software allows particular configurations of markers to be defined as a single rigid body object, so once all the markers have been detected and positioned, Tracker can determine where a prespecified object (such as a quadrotor) is located and how it is oriented. Figure 2.7 shows a screenshot of the Vicon Tracker software generating several quadrotor models based on marker locations. Using the the Vicon Datastream SDK v1.2, multiple client computers can access the position/orientation information generated by the host computer. The Vicon Datastream SDK is available for use in C++, MATLAB, and .NET programming environments on x86 and x64 devices [26].

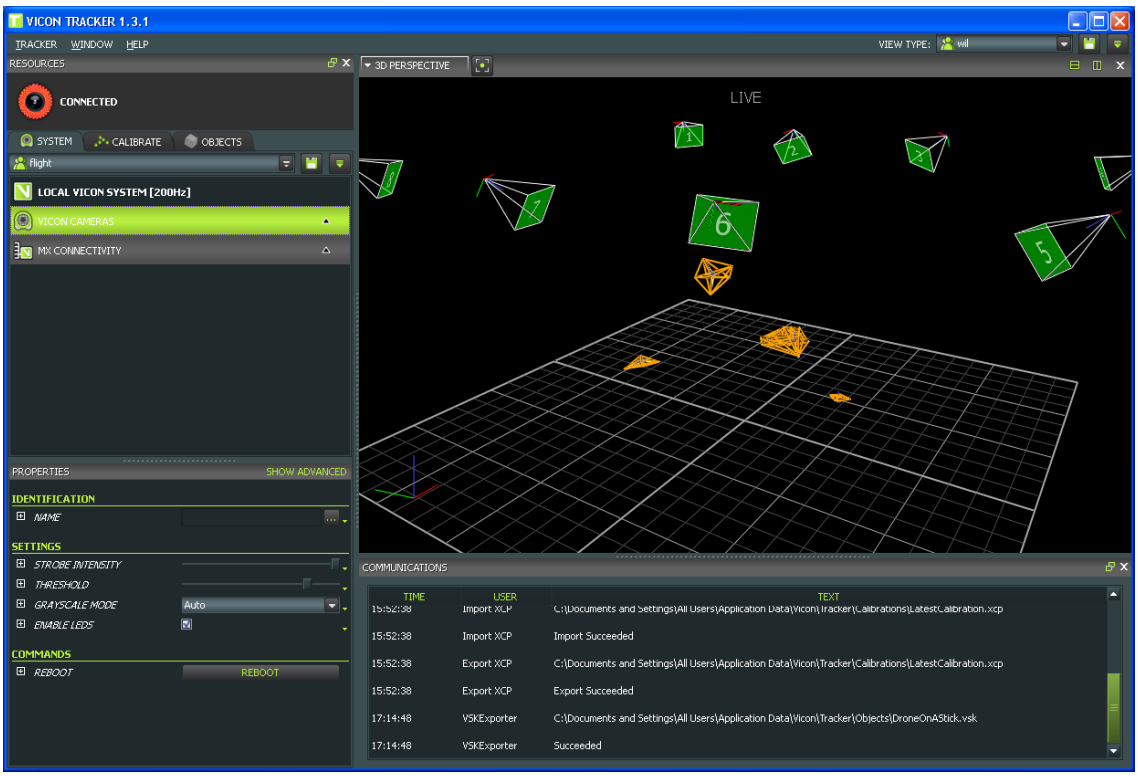


Figure 2.7: Screenshot of Vicon Tracker v1.3 capturing the locations of four quadrotors.

When properly calibrated, the Vicon system is capable of calculating an object's position within fractions of a millimeter [16]. The orientation of the object is also fairly precise, assuming the markers are configured in a sufficiently asymmetrical pattern. Symmetric patterns can periodically cause the calculated orientation to jump by large amounts within the span of a few measurements. Via the Vicon Datastream SDK, positions are reported in millimeters from the calibrated origin, and orientations can be returned in helical coordinates, as a rotation matrix, as a unit quaternion, or as a set of Euler XYZ angles [26]. A comparison of Euler angles and quaternions, as well as an explanation of how they work, is examined in Chapter 5. For the actual quadrotor control program, quaternions are the orientation expression of choice.

2.5 Ground Control Station (GCS)

The Ground Control Station (GCS) is a computer with a custom program installed to allow a quadrotor operator/pilot to control the flight behaviors of the AscTec Hummingbird with the full control system in place. Figure 2.8 shows a screenshot of the GCS program running on a Linux computer. The program allows the user to easily choose a desired flight pattern (Off, Idle, Takeoff, Land, etc.) or testing mode (Direct Motor Control, Cycle Motors, etc.). The program also displays

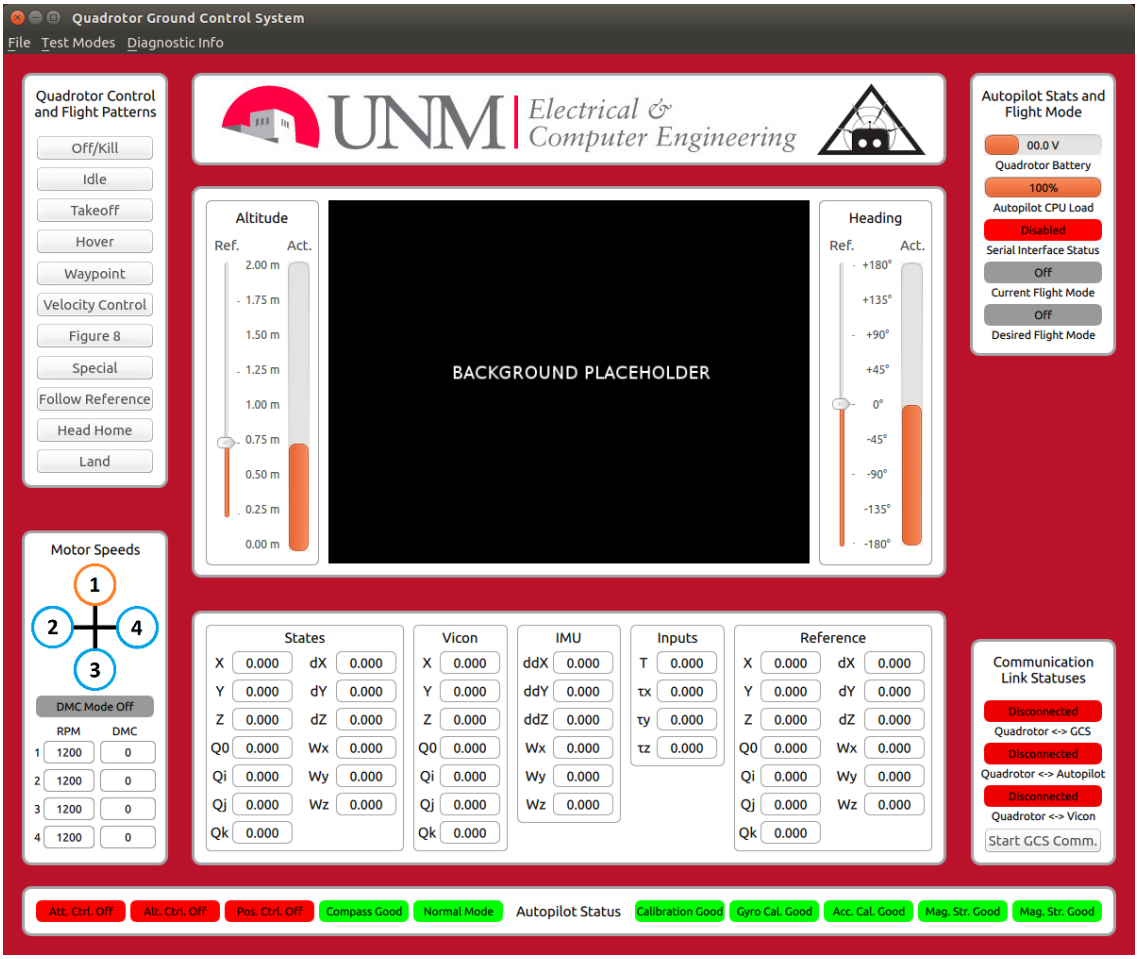


Figure 2.8: the Graphical User Interface (GUI) for the Ground Control Station (GCS) program.

status information (communication link health, battery voltage, and Autopilot status values) and quadrotor state values (position, velocity, orientation, and angular velocities) in real time. Chapter 10, Section 10.2 discusses the particulars of flight pattern selection, while Section 10.6 discusses the information coordinated between the GCS and the quadrotor.

The GCS program makes use of a combination of Qt 5.4.0 and C++ to handle everything. Qt Creator 3.3.0 was used to simplify creation and development of the Qt components. C++ classes handle the IP socket communication with the on-board Intel Edison (discussed in Section 2.6), while all of the Graphical User Interface (GUI) elements use Qt Quick for simplified development. While the program was primarily developed for use in Ubuntu Linux (and only tested on Ubuntu 14.04), the C++ classes should only need minimal modifications to make them cross-platform compatible with Windows, and Qt advertises that it makes cross-platform development incredibly simple [27]. Due to time constraints and lack of experience with cross-platform development, compatibility with Windows will be left to the efforts of any future, ambitious MARHES students.

Below is a list of features whose development was started but not finished:

- GUI-based waypoint placement.
- Live 3-D plots of various state/status information.
- Sending trajectory information to the quadrotor, whether generated by the GCS or generated by another computer and passing through the GCS.
- Velocity control so the quadrotor could fly remotely from the GCS. The arrow keys/WASD could control flight direction, while other keys could control altitude and heading.

2.6 Intel Edison System on a Chip (SOC)

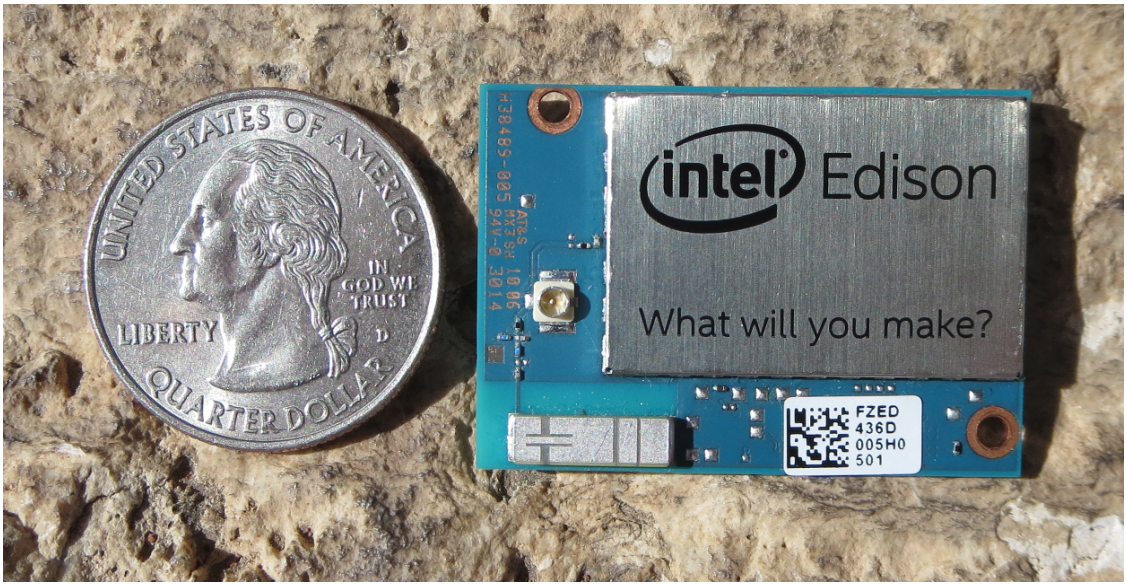
The Intel Edison System on a Chip (SoC) is an embedded x86-architecture computer capable of running compact, GUI-less distributions of Linux [28]. Figure 2.9a shows a picture of the Edison next to a quarter for scale, and Figure 2.9b shows the Edison mounted on a custom Quadrotor Block 2 board specifically designed for interfacing the Edison with the AscTec Autopilot (described in depth in Chapter 3). Table 2.5 lists the Edison's specifications, while Table 2.6 lists the Edison's available I/O ports.

Table 2.5: Specifications for the Intel Edison System on a Chip (SOC) [28]

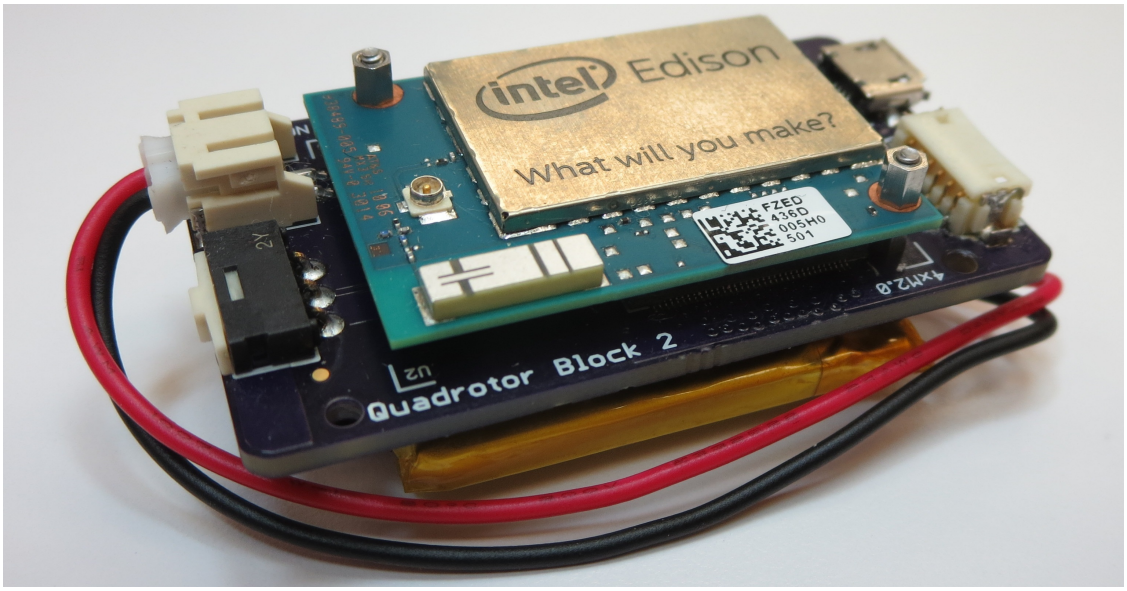
Component	Description
Size	35.5 × 25.0 × 3.9 mm
Weight	5 g
Power input	3.15 V to 4.5 V
Processor	22 nm Intel SoC with a dual-core, dual-threaded Intel Atom CPU at 500 MHz and a 32-bit Intel Quark microcontroller at 100 MHz
Architecture	32-bit x86
RAM	1 GB LPDDR3 POP memory
Internal Storage	4 GB eMMC (v4.51 spec)
Wireless	Dual-band (2.4 and 5 GHz) IEEE 802.11 a/b/g/n
Bluetooth	BT 4.0 + 2.1 EDR
Antenna	Dual-band onboard chip antenna or u.FL for external antenna

Table 2.6: Available I/O Ports on the Intel Edison [28]

Port Type	Description
SD card	1 interface
UART	2 controllers (one configured as a serial console by default)
I^2C	2 controllers
SPI	1 controller with 2 chip selects
I2S	1 controller
GPIO	14 pins (4 with PWM capabilities)
USB 2.0	1 OTG controller



(a) Intel Edison next to a quarter.



(b) Edison on custom Quadrotor Block

Figure 2.9: The Intel Edison

The Edison has several distributions of Linux available for it, with the two most prominent options being Yocto Linux and Ubilinux. Yocto Linux focuses on being able to heavily customize the Linux kernel and image to keep useful components, remove dead weight, and add in any desired features [29]. While this is useful for creating an efficient operating system, it has a high learning curve. The lackluster performance of the "opkg" package manager and periodic crashes of the DNS lookup routine prompted the examination of Ubilinux, which is a pared-down version of Debian [30, 31]. The better package manager and highly similar behavior to Ubuntu (also Debian-based) led to Ubilinux being used on the Edison.

The Edison is tasked with performing the following operations:

- Interfacing directly with the Vicon system over Wi-Fi to obtain the quadrotor's position and orientation.
- Communicating with the AscTec Autopilot's HLP over a high-speed UART link to obtain the quadrotor's angular velocities, linear accelerations, measured motor speeds, and status parameters.
- Speaking with the GCS using C++-based IP socketing over Wi-Fi to receive flight mode information and to relay information to the user.
- Filtering the various measurement parameters and compensating for latency to generate an adequate estimate of the quadrotor's current state.
- Executing control laws to find the desired quadrotor inputs to follow a desired trajectory.
- Calculating the motor speeds required for executing the desired inputs and relaying the appropriate motor commands to the Autopilot's HLP.
- Logging data pertaining to each step of the quadrotor control program.

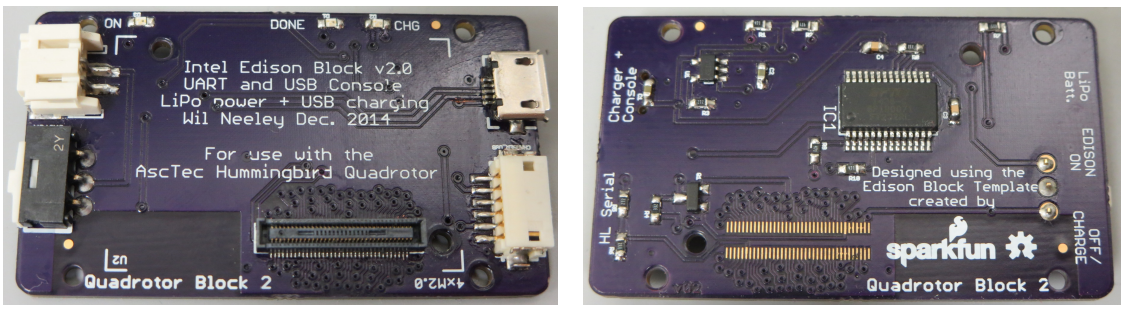
All of these responsibilities are tied together and handled by one custom C++ program. Chapter 10 covers the full extent of the program and discusses each of its individual components.

The Edison is unique in that it is one of the few, if not the only, embedded Linux computer with an x86 architecture. The Vicon Datastream SDK only offers libraries for x86 and x64 architecture machines; there currently are no libraries for ARM processors [26]. Many other compact embedded computers, such as the Raspberry Pi compute module [32], the Gumstix line [33], and the Odroid line [34] operate under Linux operating systems, but they all use ARM architecture processors. Until Vicon generates an ARM library for the Datastream SDK, the ARM-based computers cannot get information directly from the Vicon server. Fortunately, the Edison bypasses this pitfall.

Chapter 3

Intel Edison Quadrotor Block

A custom circuit board called the Intel Edison Quadrotor Block was designed for integrating the Intel Edison with the existing AscTec Hummingbird's Autopilot board. While the first revision was fraught with issues, the second revision (Quadrotor Block 2) fixed all of the previous problems and added a serial console. Figure 3.1 shows top and bottom pictures of the final Quadrotor Block 2. The design and construction of a custom board was necessary for several reasons: the serial UART link to the Autopilot's High Level Processor (HLP) required level-shifting circuitry, battery power was needed, everything had to pass through the Edison's tiny 70-pin Hirose DF40 connector, and no boards were commercially available that offered these features. Section 3.1 details the design process for the UART link. Section 3.2 covers the power and serial console design for the board. Section 3.3 provides the final schematic, layout, and parts list for the board. Section 3.4 provides a list of potential revisions to the board.



(a) Top (b) Bottom

Figure 3.1: Pictures of the custom made Quadrotor Block 2 for the Intel Edison

3.1 UART Circuitry Design

Both the Edison and the HLP have UART ports operate on CMOS TTL logic levels. The Edison's two UART ports, UART1 and UART2, operate at 1.8V levels and are only tolerant of 1.8V signals [28]. By default, Linux distributions for the Edison make use of UART2 as a serial console port, so this leaves UART1 open for use. The HLP's available UART port, HL Serial 0 (labeled on the Autopilot board in Section 2.3, Figure 2.4a), outputs 3.3V signals and is tolerant of 5V inputs [22]. The HLP's UART port was designed to work with a Digi XBee [35] for wirelessly transmitting serial data to/from a ground station, so in addition to the standard Tx, Rx, and Ground lines for basic UART communication, the port also has 5V and 3.3V power outputs and a CTS (Clear To Send) line.

Figure 3.2 shows a schematic of the level shifting circuitry used between the Edison and the HLP. The voltage divider safely drops the Autopilot's 3.3V signal down to 1.8V, and the NMOS/resistor combination leverages both the Edison's 1.8V reference output and the Autopilot's 5V output to boost the Edison's $0-1.8$V signal to $0-5$V. For further details about the UART's configuration and performance, see Chapter 4.

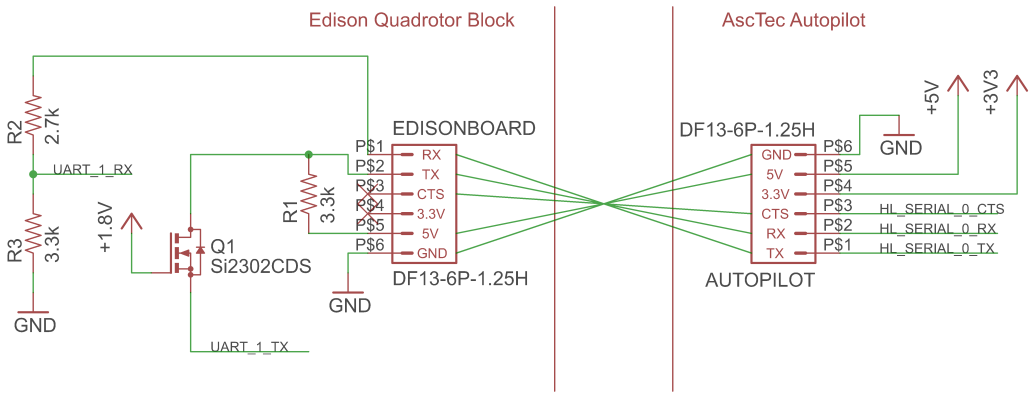


Figure 3.2: Schematic of the UART link and its level shifting circuitry.

3.2 Power and Serial Console Design

The Intel Edison can be powered by an input voltage between 3.15V and 4.5V [28]. The selected voltage range is ideally geared towards powering the Edison with a single cell lithium battery. Depending on CPU and WiFi load, the Edison's average current needs range between 50mA and 200mA with periodic spikes up to 300mA [36]. A single cell 3.7V 2,000mAh lithium polymer battery was selected to provide power to the Edison. Assuming an unrealistic worst-case scenario where the system is perpetually drawing 300mA, the Edison will stay on for six hours and 40 minutes, which is long enough to provide power through almost a full work day's worth of flights without recharging. By using its own battery instead of drawing power from the Hummingbird, the Edison can be continuously powered on even when swapping quadrotor batteries or the quadrotor is off.

A Microchip MCP73831 Li-Polymer Charge Management IC [37] was incorporated to charge the battery when the Edison was powered down without having to remove the battery. The MCP73831 receives its power from a micro-B USB connector. As computer USB 2.0 ports typically have a limited current output of 500 mA,

the MCP73831's maximum charging current was selected to be $370mA$[1] to be well under this threshold while still allowing for moderately quick charging. To prevent complications where the charger is trying to supply constant current to the battery while the Edison is powered on, the power switch was implemented such that the battery is either connected to the Edison (powered on) or connected to the charger (powered off/charging). The power switch was selected for its small footprint and its 12V, 500mA rating.[2]

As mentioned in Section 3.1, Linux distributions are configured by default to use the Edison's UART2 port as a serial console. While other, custom distributions may use different settings, Yocto Linux and Ubilinux both use a UART configuration of 115,200 Bd, 8 bits, no parity, one stop bit, and no flow control. An FTDI FT232RL [38] was used to convert the serial console's UART signal to USB. The USB signal uses the same micro-B connector used by the charging circuit. The FT232RL supports a diverse range of baud rates and configurations, so it should work with the Edison regardless of the Linux distribution used. The UART $\leftrightarrow$ USB conversion allows a computer to connect to the micro-B USB port, see the serial port (after installing the appropriate FTDI drivers), and open it to communicate with the Edison. The FTDI IC is powered by the 5V supply coming from the USB port, so it does not consume battery power.

[1]The charging current is programmed through the selection of a "programming" resistor. The regulated current is programmed via the equation $I_{REG} = 1000V/R_{PROG}$, where I_{REG} is in milliamperes and R_{PROG} is in $k\Omega$s. By selecting a 2.7 $k\Omega$ resistor to minimize the number of unique parts used, the charging current works out to be around 370mA.

[2]When the board was created, documentation was incredibly sparse. A few posts in the Intel Support Forum suggested the Edison's current draw spiked upwards of 500mA. A cheaper switch with a smaller current rating could be used in future iterations.

3.3 Final Board Schematic and Layout

The Quadrotor Block 2 circuit board was designed using a template board provided by Sparkfun [39] using EagleCAD 7.0.0 Light (the free version). Figure 3.3 shows the schematic of the final board. Figure 3.4 shows the physical board layout of the circuitry specified in the schematic. Table 3.1 provides a parts list of the components necessary to build the custom Quadrotor Block 2 board.

Board assembly can potentially be problematic. While the first group of boards were soldered by hand, soldering the Hirose DF40 connector requires moderate skill and practice. Applying lots of rosin flux to the pins, blanketing the pins in solder, and removing the excess (including solder bridges) with solder wick works fairly well. An economic alternative would involve creating a stencil for the board, applying solder paste, placing the components, and simulating reflow soldering through use of a toaster oven or an electric skillet.

3.4 Areas for Improvement

Below is a list of ways in which the current circuit board can be improved.

- Currently, there is no circuitry in place to protect the battery from over-discharging. A protection circuit would be beneficial.
- Currently, there is no way to measure the voltage of the battery to check its charge level. Some means of measuring battery voltage on the Edison would be helpful.
- Now that the Edison's current consumption behaviors have been better characterized [36], the power switch can be dropped from a 500mA rating to around a 300mA rating.

- A means of charging the LiPo battery while the Edison is powered up would be helpful, although it isn't critical.
- The FT232RL IC has the functionality to drive transmit and receive LEDs to indicate when data is on the UART link. While it would be helpful for troubleshooting purposes, it isn't necessary under typical operation.
- The micro-B USB connectors were very tricky to solder properly by hand. Over-exposure to the soldering iron seemed to warp the connectors and made it difficult to get a good electrical connection when plugging in a cable. Either special care should be taken when attaching the micro-B USB connectors, or they should be changed out for mini-B USB connectors. The added size and thermal mass should reduce over-exposure problems.
- The board design was started with little to no experience designing printed circuit boards (PCBs). As such, the layout could easily be improved.

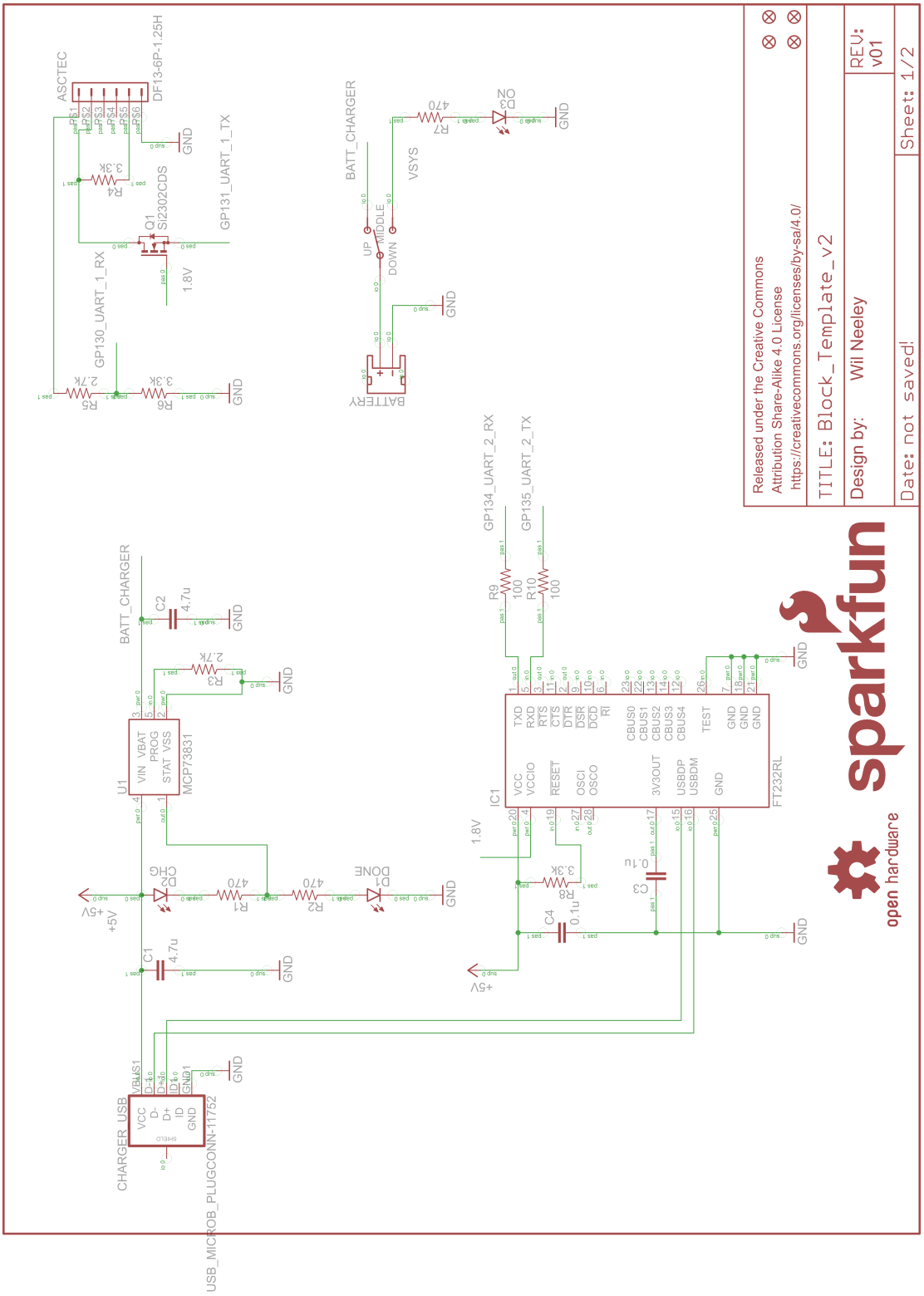
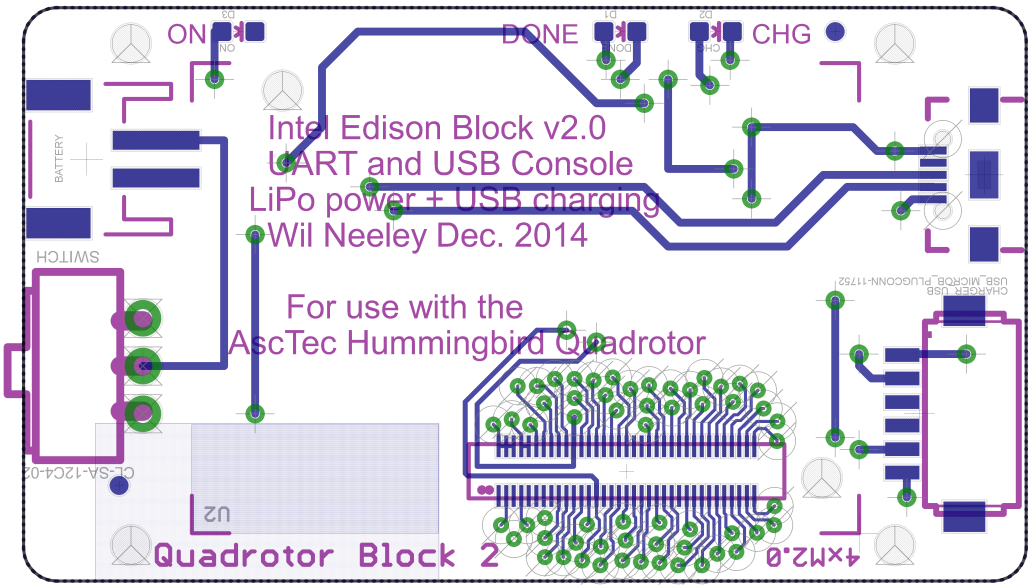
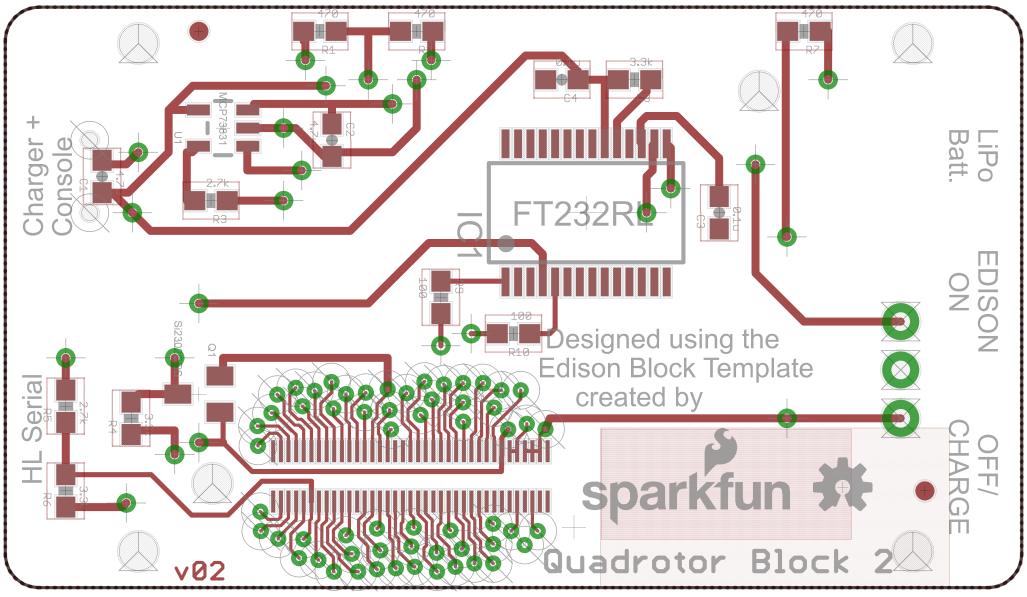


Figure 3.3: Schematic for the Intel Edison Quadrotor Block



(a) Top



(b) Bottom

Figure 3.4: Layout for the Intel Edison Quadrotor Block. The ground planes on both sides are not shown.

Table 3.1: Parts List for the Intel Edison Quadrotor Block

Part	Qty.	Source	Source Part Number
Quadrotor Block 2 board	1	OSH Park	N/A (custom schematic)
Slide Switch, SPDT, 12V 500 mA	1	Digikey	563-1386-ND
JST Right Angle Connector	1	Sparkfun	PRT-08612
Micro USB SMD Connector	1	Sparkfun	PRT-08533
Hirose DF13-6P-1.25H(20) Right Angle Connector	1	Mouser	798-DF136P125H20
Hirose DF40HC(3.0)-70DS-0.4V(51) Mezzanine Connector	1	Mouser	798-DF40HC3070DS4V51
FTDI FT232RL USB to Serial UART IC, SSOP-28 SMD	1	Mouser	895-FT232RL
Microchip MCP73831 Battery Management Charger IC	1	Mouser	579-MCP73831T-2ATIOT
NMOS Transistor, SOT-23-3 SMD, $V_{GS(th)} \approx 0.6V$	1	Mouser	781-SI2302CDS-E3
Green LED, 0603 SMD, 1.9V	2	Mouser	859-LTSTC193KGKT5A
Red LED, 0603 SMD, 2.3V	1	Mouser	859-LTSTC193KRKT5A
470 Ω Resistor, 0603 SMD	5	Mouser	71-CRCW0603J-470-E3
2.7 kΩ Resistor, 0603 SMD	2	Mouser	71-CRCW06032K70JNEB
3.3 kΩ Resistor, 0603 SMD	2	Mouser	71-CRCW06033K30JNEB
4.7 μF Capacitor, 0603 SMD, 6.3V	2	Mouser	963-JMK107BJ475KA-T
0.1 μF Capacitor, 0603 SMD, 16V	2	Mouser	77-VJ0603Y104JXJPBC

Chapter 4

Edison and Autopilot UART Link

The reliability and consistency of information sent over the Autopilot/Edison UART link is critical to the Edison's effectiveness in performing state observation and executing control laws. Without a robust link between the Edison and the Autopilot, IMU information will be worthless, and the Edison will have little to no true control over the quadrotor. Fully characterizing the UART link's latency and data loss is essential to developing algorithms capable of mitigating any delays or irregularities.

In this chapter, the UART connection between the Intel Edison and the AscTec Autopilot's High Level Processor (HLP) will be covered. For a discussion of the UART's physical layer, see Chapter 3, Section 3.1. Section 4.1 discusses the UART configuration, as well as constraints and limitations associated with baud rates and buffer sizes. Section 4.2 details the frame structures used to reliably relay information back and forth over the UART link. Section 4.3 discusses the method used to evaluate the communication link's reliability and the results. Section 4.4 looks into the regularity with which the program running on the Edison receives IMU data from the Autopilot. Section 4.5 discusses the process for setting up the Edison's UART port each time before being used.

4.1 UART Configuration

This section looks at the configurations and limitations of each device's UART port. As the configuration of the communication link can have effects on transmission time, data throughput, and reliability, configuration options need to be discussed.

Buffer sizes dictate how much data can be sent at once, so understanding the UART's buffer limitations is essential to formulating frames to send data back and forth. Both the Edison and the HLP possess 16550 compliant UART ports, enabling both devices to have hardware FIFO (First In First Out) buffers to reduce interrupt loads on the processors and to prevent overflow issues. The Edison has 64-byte buffers on the receive and transmit lines [28], while the HLP has 16-byte buffers [40]. While the various Linux distributions for the Edison effectively handle any interrupt routines required to read data from the UART, the HLP can be configured for interrupts when 1, 4, 8, or 14 bytes have been received. The 14 byte interrupt configuration was used, and information frames were scaled to avoid triggering the interrupt routine. In doing so, the HLP can read the UART port at a predictable time, and potential complications from interrupt routines are avoided.

Maximizing the data transmission rate is key to minimizing transmission delays, so the viable baud rates on each system need to be discussed. The HLP has a serial UART port that can be configured for baud rates up to 1/16th of the processor's clock rate [40]. As the processor's clock rate is set at 58.9824 MHz (based on examining the source code from the AscTec SDK [23]), the HLP's UART port can be set to a maximum baud rate of 3.6864 MBd; as the UART operates on binary logic levels, this translates to a maximum transmission speed of 3.6864 Mbps. While the HLP can be configured for nonstandard baud rates (non-integer fractions of the maximum clock rate), this is not necessary, as the standard baud rates match perfectly with the Edison.

Table 4.1: Relevant Baud Rates Supported in Hardware for the Intel Edison and the AscTec Autopilot's High Level Processor (LPC2146)

Baud Rate	57.6 K	115.2 K	230.4 K	921.6 K	1.8432 M	3.6864 M
Edison	Yes	Yes	Yes	Yes	No	Yes
HLP	Yes	Yes	Yes	Yes	Yes	Yes

Table 4.1 lists some of the Edison's and the HLP's baud rates that are supported in hardware. On both of the Edison's tested Linux installations (Yocto Linux and Ubilinux), the *stty* command for configuring serial ports would not accept baud rates faster than 921,600 bps. Alternative methods of configuring serial ports may potentially allow for higher baud rates. However, as 921,600 bps is more than sufficiently fast for transmitting small data frames, and as both devices support this speed, no extra effort was put into speeding up the baud rate. As for the remaining settings, the UART link was configured for 8 bits, no parity, one stop bit, and no flow control.

4.2 UART Frame Structures

Once the Edison and the HLP have been configured to talk to each other, an appropriate means of reliably sending data from one device to another needed to be developed. Both devices need to know what kind of data they are receiving (e.g. IMU vs. status information), where the data starts, where the data ends, and whether or not they received an entire block of data. A custom frame structure was developed to handle this kind of information coordination.

Fundamental to any data transmission framework is knowing the kind of data that will be sent back and forth. Table 4.2 gives a list of the types of values to be sent to the Autopilot's HLP from the Intel Edison. The primary values needed by the quadrotor are the motor commands for the motor speeds the Autopilot should execute, each of which ranges between 0 (off) and 200 (full speed). For UART

Table 4.2: Data Values and Types to be Sent to the Autopilot's High Level Processor

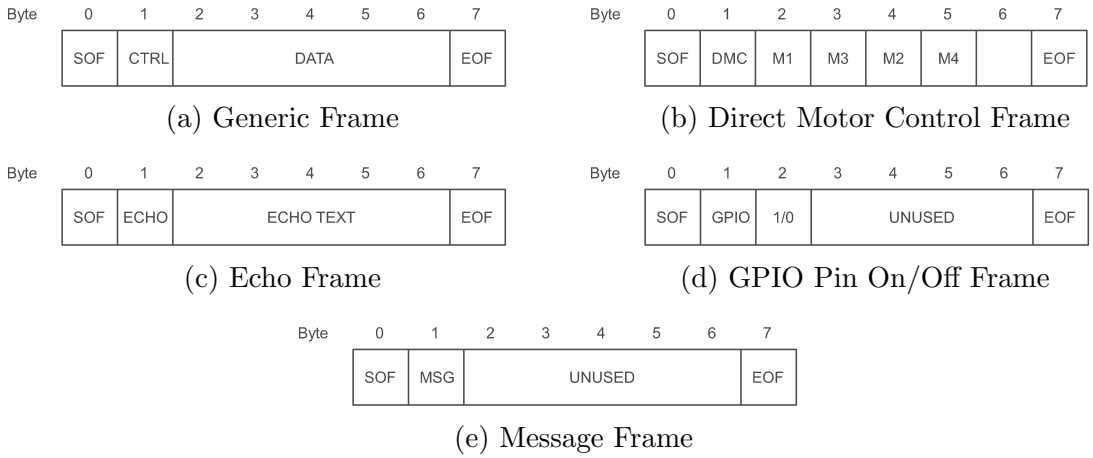
Data Description	C Data Type	Byte Size	# Values
Motor Commands	unsigned char	1	4
Echo Text	char	1	5
GPIO Pin On/Off	char	1	1

Table 4.3: Data Values and Types to be Sent to the Intel Edison

Data Description	C Data Type	Byte Size	# Values
Angular Velocities	signed short	2	3
Linear Accelerations	signed short	2	3
HLP Status	unsigned short	2	1
Battery Voltage	unsigned short	2	1
CPU Load	unsigned short	2	1
Motor Speeds	unsigned char	1	4
Echo Text	char	1	5
HLP Version Text	char	1	12

integrity evaluation purposes, an echo command was developed to send five bytes of text at a time to the HLP and have it echo the text back. The link quality was evaluated based on the number of lost or incorrect characters. As an additional diagnostic tool, a command was created for setting the high/low state of one of the Autopilot's GPIO pins (general purpose input-output). One byte conveyed enough information to turn the pin on or off.

Table 4.3 gives a list of the parameters to be sent from the HLP to the Edison. The angular velocities and the linear accelerations are needed for state estimation purposes. The HLP status, battery voltage, and CPU load values provide valuable information about the present state of the quadrotor, while the measured motor speeds are useful for post-flight data analysis. The version text allows a user to verify the HLP's programming is up to date, and the echo text sends back the five characters received from the echo command sent from the Edison.



(a) Generic Frame

(b) Direct Motor Control Frame

(c) Echo Frame

(d) GPIO Pin On/Off Frame

(e) Message Frame

Figure 4.1: Diagrams of custom frame structures sent over the UART link from the Intel Edison to the AscTec Autopilot's High Level Processor.

The HLP's transmit and receive buffers are the largest limiting factor for passing data over the UART link. Only 16 bytes can be queued in the transmit buffer at any given moment. At 14 bytes, the receive buffer will trigger an undesirable interrupt. The frames should be as small as possible to minimize communication delays as much as possible. To simplify frame processing techniques, all of the frames going in a particular direction over the UART link should be the same size.

For sending data from the Edison to the HLP, an eight-byte frame structure was selected. Figure 4.1 shows the byte organization for each frame type. Looking at the generic frame structure (Figure 4.1a), each frame has one start of frame (SOF) byte, one control (CTRL) byte to indicate the frame type, five bytes of data, and one end of the frame (EOF) byte. The five bytes of data are more than enough for sending direct motor control data (Figure 4.1b), echo text (Fig. 4.1c), GPIO state information (Fig. 4.1d), and message/request frames (Fig. 4.1e). The message/request frame allows for requesting the HLP program version or toggling the GPIO pin. With a transmission speed of $921,600$ bps and the UART configuration given in Section 4.1, it takes 86.81 μs to transmit one frame of data from the Edison to the HLP.

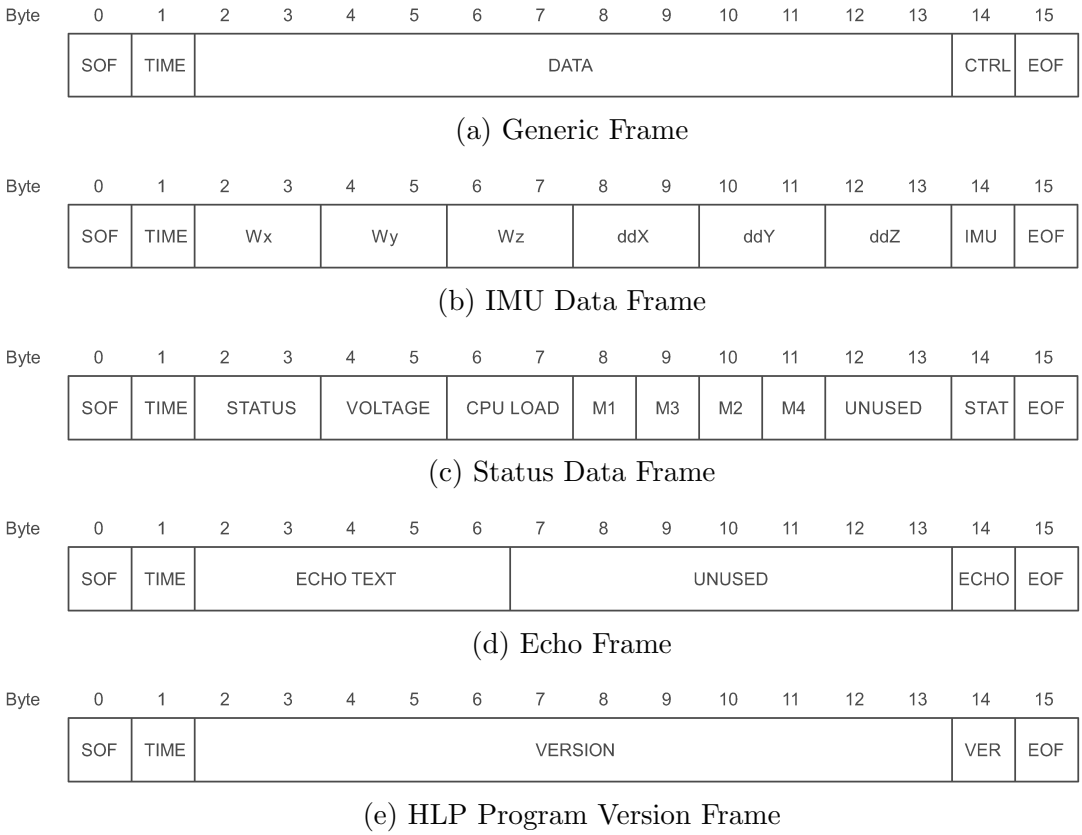


Figure 4.2: Diagrams of custom frame structures sent over the UART link from the AscTec Autopilot's High Level Processor to the Intel Edison.

For sending data from the HLP to the Edison, a sixteen-byte frame structure was selected. Figure 4.2 shows each frame type's byte organization. Looking at the generic frame (Figure 4.2a), each frame has a start of frame (SOF) byte, an incrementing/overflowing (TIME) byte for the HLP loop in which the frame was sent, twelve bytes of data, one frame type (CTRL) byte, and one end of frame (EOF) byte. The twelve bytes of data allows all the IMU's accelerations and angular velocities to be sent at once (Fig. 4.2b) while leaving room for status information (Fig. 4.2c), returning echo text (Fig. 4.2d), or reporting the HLP's program version (Fig. 4.2e). With a transmission speed of $921,600$ bps and the UART configuration given in Section 4.1, it takes 173.61 μs to transmit one sixteen-byte frame.

Both frame structures have a means of checking for character loss with the SOF and EOF bytes, but they lack a proper means of error detection (like a parity check or a cyclic redundancy check) or error correction (such as using Hamming distances). Since the UART link was shown to be highly reliable and character loss was the only real concern (see Section 4.3), error detection and correction were unnecessary. Character loss was handled by dumping any partial frames and starting over with the next whole frame. As IMU information is leaving the HLP at a rate of 200 Hz and motor commands are leaving the Edison at a rate of 100 Hz, minor setbacks due to occasional frame loss are not a significant problem.

NOTE: For the two frame structures involving motors (Figures 4.1b and 4.2c), the motor number labels follow the convention established in Chapter 6, Section 6.3 (1 = front, 2 = left, 3 = back, 4 = right). However, to match the Autopilot's motor labeling convention (1 = front, 2 = back, 3 = left, 4 = right), information for motors 2 and 3 need to be swapped before being transmitted and after being received.

4.3 Reliability Analysis

The reliability of the UART link was tested by sending a stream of known text from the Edison to the Autopilot's HLP via echo commands, recording the time required to receive an echo response, saving the text echoed back, tallying the number of received and dropped echoes, and post-processing the results. See Algorithm 1 for a pseudocode representation of the test.

A Plain Text UFT-8 version of "War of the Worlds" by H. G. Wells was obtained from Project Gutenberg [41] and used as the test file. The "War of the Worlds" text file had two main advantages: it was in the public domain, and it had $365,445$ characters, which evenly divided into $73,089$ five-byte frames. $73,089$ frames provide a statistically significant number of tests for verifying the UART link's integrity.

Algorithm 1 UART Echo Test Algorithm

```
Open Test File, Result File, and Timing File;
received = 0; dropped = 0;
Initialize response timer;
while Test File ! = empty do
    Read five characters from Test File;
    Send characters to HLP via echo command;
    while !response & time ≤ 10 ms do
        Check for response;
    if reponse then
        Save time since last response to Timing File;
        Save response to Result File;
        received += 1;
    else
        dropped += 1;
    Reset response timer;
     return received and dropped;
```

Table 4.4: UART Echo Test Frame Loss Results

Trial	Frames Sent	Frames Received	Frames Dropped	Percent Dropped
1	73,090	73,078	12	0.0164%
2	73,090	73,081	9	0.0123%
3	73,090	73,085	5	0.0068%
4	73,090	73,081	9	0.0123%
5	73,090	73,081	9	0.0123%
6	73,090	73,083	7	0.0096%
7	73,090	73,084	6	0.0082%
8	73,090	73,080	10	0.0137%
9	73,090	73,083	7	0.0096%
10	73,090	73,077	13	0.0178%

Ten different trials were run using Algorithm 1. Table 4.4 lists the results reported at run time. The C++ program responsible for running the algorithm generated one extra frame (73,090 instead of 73,089) due to a bug in checking whether or not characters were actually received when reading from the file. Regardless, the number of dropped frames across all ten trials was very small. At less than 0.02%, the percentage of frames dropped, whether at the transmission or reception end, is sufficiently low that data loss is not an important concern.

The result files were later post-processed in MATLAB. A script was written to compare the test file to the result file; in the event that a particular result file character didn't match the test file, the test file character was flagged as an error and the next test file character was compared with the same result file character. The program will keep scanning through the test file and flagging characters as errors until it finds a match to the result file. This simple algorithm made it easy to check for character loss in the result file, but it produces absurdly large numbers of incorrect characters in the event a character's value gets changed. The algorithm will continue scanning the test file and marking errors until it stumbles upon a match to the changed character, at which point the two files are completely out of sync, which will cause many more errors.

Figure 4.3 shows the results of this algorithm for all ten trials, as well as a sample case where one character was changed from "H" to "h" partway through the document. The sample case registered the majority of characters after the change as being incorrect, whereas the other cases only have isolated groups of errors. Figure 4.4a shows a histogram of the number of error streaks seen for the sample case. An error streak as long as 16,079 characters was seen. In a UART echo test where the UART is assumed to be error-free yet capable of dropping data, a single character error could imply that 3,215 frames in a row, plus most of another frame, were not successfully echoed back. The UART link would be completely unusable.

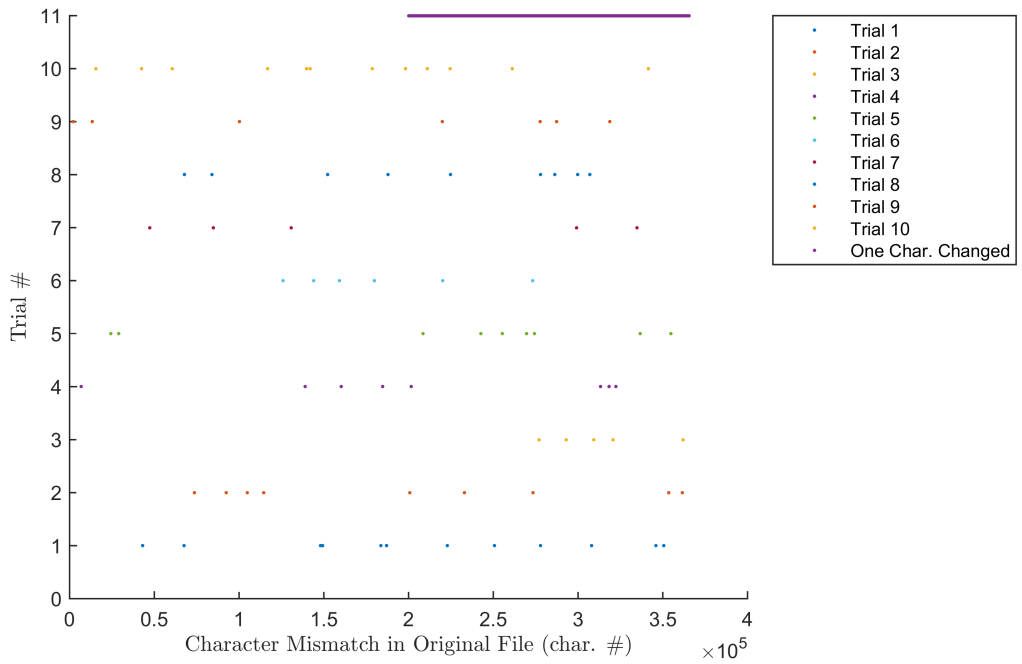
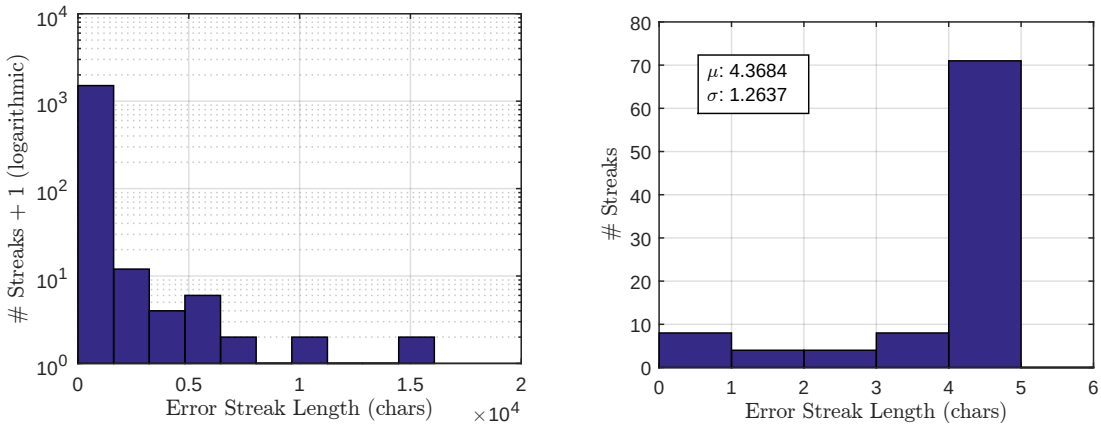


Figure 4.3: Plot of UART Echo Test Character Mismatches for Each Trial



(a) (b)

Figure 4.4: Histograms of Error Streaks for a Single Character Change (4.4a) and for the Ten Trials (4.4b).

By comparison, Figure 4.4b shows a histogram of the number of error streaks for all ten trials. No error streaks exceeded five characters, the majority had exactly five characters, and some were seen with less characters. The one-error streaks were always adjacent to four-error streaks, and the two-error streaks were always adjacent to three-error streaks. After visually comparing the test and result files side by side, characters were seen to always be lost in groups of five; the non-five error streaks were caused by one of the lost characters appearing right after the lost frame. The character loss occurring in groups of five is explained by dropped frames.

While the echo test provides no answers as to what exactly caused the frames to be lost, it shows three important things: data is occasionally lost but never changed, data is lost in frame-sized chunks, and the losses occur sparingly. Given the results of this test, error checking schemes are not necessary, as frames will never be corrupted. Rarely dropped frames are not a problem, as frame transmissions happen frequently enough that a replacement will come along shortly.

4.4 Timing Analysis

A test was constructed to verify both the HLP's timing and the Edison's timing were correct for IMU frames. The HLP was configured to report IMU frames once every five HLP loops, which corresponds to frames being transmitted at 200 Hz. A program was written for the Edison to receive IMU frame timestamps, calculate the program time since the last frame, and record both values. Figure 4.5 shows both the timestamp differences and the program time differences on one plot, while Figure 4.6 shows a histogram of the program's recorded times. The HLP consistently reported frames 5 ms apart. The Edison's program timing had a mean of 4.9999 ms and a standard deviation of 0.048 ms, which is really good. The Edison's non-real time OS resulted in some fluctuations, but the variations were not significantly bad.

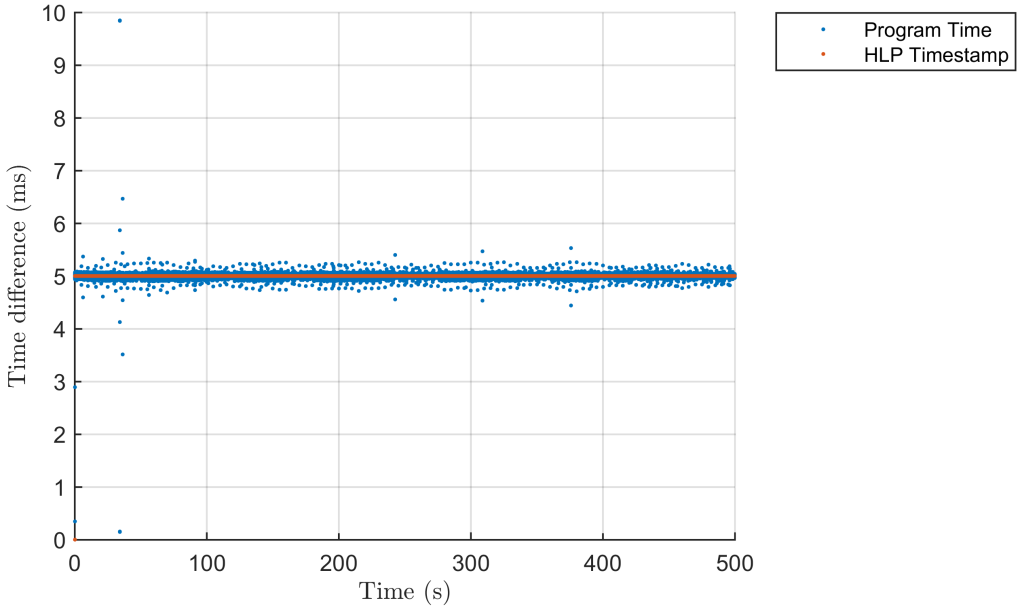


Figure 4.5: Plot of time between program log times and time between IMU timestamps

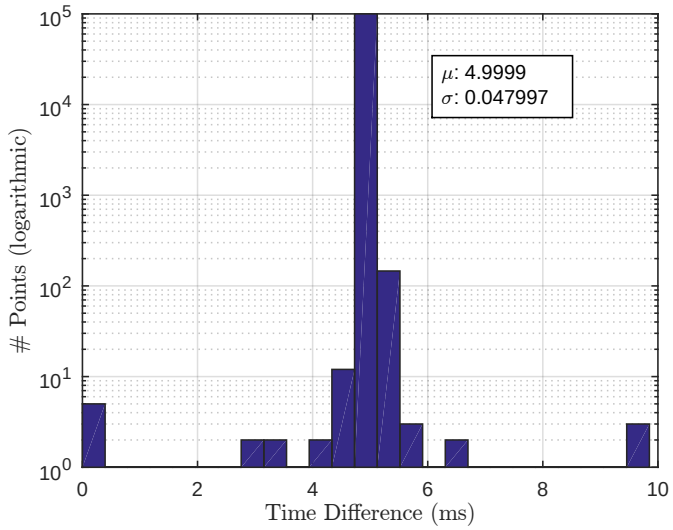


Figure 4.6: Histogram of program time between received IMU messages

4.5 Edison UART Port Configuration Commands

Documentation for configuring the Edison's UART port was incredibly sparse when the UART configuration process began, so this section explains the process for setting up communication on UART1 within Linux. The information provided here applies to both Yocto Linux and Ubilinux.

Linux offers several commands to configure serial/UART ports, such as *setserial* and *stty*. For this project, the *stty* command was used, as it was shown to work on both Yocto Linux and Ubilinux. Within the Linux operating system, UART1 shows up as file labeled */dev/ttyMFD*1.[1] To configure UART1 to have a data rate of 921,600 bps and a standard raw configuration (8 bits, no parity, one stop bit, no flow control), the following command should be entered into the Linux terminal or used as a system call within a given program:

```
stty -F /dev/ttyMFD1 921600 raw
```

While this works properly on most system, the Edison's UART1 port was found to be configured by default to echo received characters. This caused numerous bugs that were quite difficult to track down. To remove the echo problems, several additional flags need to be set. With the new flags in place, the command becomes the following:

```
stty -F /dev/ttyMFD1 921600 raw -iexten -echo
        -echoe -echok -echoctl -echoke
```

[1] It took many hours of browsing through forums to find this.

Chapter 5

Representations of Orientation

Multiple methods can be used to represent an object's orientation in three dimensions, such as rotation matrices, Euler angles, quaternions. While this chapter assumes that the concept of rotation matrices is understood, Euler angles and quaternions are explained in detail. Section 5.1 talks about the two main frames of reference (World frame and Body frame) for representing values. Section 5.2 discusses the use of Euler angles, while Section 5.3 discusses the use of quaternions. Section 5.4 discusses methods for converting between quaternions and Euler angles. Section 5.5 compares Euler angles and quaternions to understand their strengths and weaknesses. Section 5.6 introduces the Heading frame, its motivation, and the split quaternion used to create it.

5.1 Frames of Reference

Before discussing converting orientations from one reference frame to another, it is important to establish a set of reference frames. There are two main references frames used to represent an object: a static World frame and moving Body frame.

The World frame, represented by $\{W\}$ and occasionally referred to as Earth frame, Inertial frame, or Land frame, intuitively remains static relative to a fixed point on planet Earth [42]. The Vicon motion capture system is static relative to Earth and retains a fixed coordinate system, so its measurements are in $\{W\}$. The Vicon system measures in right-handed coordinate systems with configurable axis directions [26]. For the sake of intuition, a standard X-right, Y-forward, Z-up coordinate system was selected. The origin of $\{W\}$ is calibrated to be in the center of the Vicon capture volume at ground level. Coordinates in $\{W\}$ are denoted (when necessary) with a W superscript, such as $\vec{x}^W$.

The Body frame, represented by $\{B\}$, intuitively remains static relative to the body of the quadrotor [42]. While the AscTec Autopilot defines its internal coordinate system as X-forward, Y-right, Z-down relative to the "forward" arm (as indicated with orange tape) and measures accordingly [23], $\{B\}$ is defined as having an X-forward, Y-left, Z-up orientation to better match $\{W\}$. To change the linear accelerations and angular velocities measured by the Autopilot's inertial measurement unit (IMU) to match $\{B\}$, the Y and Z values (for both linear accelerations and angular velocities) need to have their signs changed. The origin of $\{B\}$ is centered between the quadrotor arms. Coordinates in $\{B\}$ are denoted with a B superscript, such as $\vec{x}^B$.

To transition from one frame of reference to another, two actions need to take place: a translation and a rotation [42]. If $\{B\}$'s origin is defined at a particular orientation (${}^W\mathbf{R}_B$) and position (${}^W\vec{r}_B$) from $\{W\}$'s origin, transitioning an object $\vec{r}^B$ from $\{B\}$ to $\{W\}$ starts by rotating the object using ${}^W\mathbf{R}_B$, followed by translating the object using ${}^W\vec{r}_B$. Equation 5.1 represents this in equation form. This chapter emphasizes the rotation transition and the various methods for representing ${}^W\mathbf{R}_B$.

$$\vec{r}^W = {}^W\mathbf{R}_B\vec{r}^B + {}^W\vec{r}_B. \tag{5.1}$$

5.2 Euler Angles

Euler angles operate by defining an object's orientation as a series of three rotations around non-consecutive axes of a coordinate frame [43]. Euler angle sequences are named after the axes around which the rotations are performed, such as XYZ or XZX. The non-consecutive axes requirement means that sequences like XYZ, ZYX, ZXY, ZXZ, and YXY are all valid Euler angle combinations, but YYZ is not[1], nor is YZZ or XXX. There are a total of 12 possible Euler angle combinations, it is very easy to mix them up, and they produce radically different equations, so it is always important to specify and understand which set of Euler angles is being used in any given context. For the rest of this chapter, the Euler XYZ system will be used unless noted otherwise. While Euler ZYX is the typical standard used for discussing aircraft [44], Euler XYZ helps with explaining the Heading frame introduced in Section 5.6.

Ultimately, the name for an Euler angle combination (ZXZ, XYZ, ZYX, etc.) stems from the order in which various axial rotation matrices need to be put together to execute the desired rotation [42]. The Euler XYZ's rotation matrix for going from Body frame $\{B\}$ to World frame $\{W\}$ is represented as in Equation 5.2

$$ {}^{W}\mathbf{R}_{B} = \mathbf{R}_{x}(\phi)\mathbf{R}_{y}(\theta)\mathbf{R}_{z}(\psi), \tag{5.2} $$

where

$\phi \in \mathbb{R}$ is the object's roll (X rotation) angle,

$\theta \in \mathbb{R}$ is the object's pitch (Y rotation) angle,

$\psi \in \mathbb{R}$ is the object's yaw (Z rotation) angle, and

$\vec{\Theta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \in \mathbb{R}^3$ is the Euler angle vector that groups these terms together.

[1] Sorry, Neil Peart.

Before determining the rotation matrix order for a given rotation order (first around the Z axis, then around the Y axis, and finally around the X axis), the question must be asked: around which set of X/Y/Z axes are the rotations being performed? Those in a fixed frame of reference (extrinsic), or those in the frame of reference being rotated (intrinsic)? If performing intrinsic rotations, a Z $\rightarrow$ Y $\rightarrow$ X axis rotation order yields the Euler ZYX rotation matrix order. If performing extrinsic rotations, a Z $\rightarrow$ Y $\rightarrow$ X axis rotation order yields the Euler XYZ rotation matrix order. An intrinsic X $\rightarrow$ Y $\rightarrow$ Z rotation order yields the same result as an extrinsic Z $\rightarrow$ Y $\rightarrow$ X rotation order with the same angles. Adding in the appropriate rotation matrices for X, Y, and Z axis rotations [43], Equation 5.2 becomes

$$\begin{aligned} {}^{W}\mathbf{R}_{B} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} c\theta c\psi & -c\theta s\psi & s\theta \\ c\phi s\psi + s\phi s\theta c\psi & c\phi c\psi - s\phi s\theta s\psi & -s\phi c\theta \\ s\phi s\psi - c\phi s\theta c\psi & s\phi c\psi + c\phi s\theta s\psi & c\phi c\theta \end{bmatrix}, \; c = \cos(), \; s = \sin(). \end{aligned} \tag{5.3}$$

The operation can be reversed by inverting the rotation matrix (${}^{B}\mathbf{R}_{W} = ({}^{W}\mathbf{R}_{B})^{-1}$). If an object has been rotated and its rotation matrix is known, its corresponding Euler angles can be calculated from the rotation matrix's entries via Equations 5.4 as shown by their derivations.

$$\begin{aligned} \phi &= \tan^{-1}\left(\frac{-\mathbf{R}_{23}}{\mathbf{R}_{33}}\right) = \tan^{-1}\left(\frac{\sin(\phi)\cos(\theta)}{\cos(\phi)\cos(\theta)}\right) = \tan^{-1}\left(\tan(\phi)\right), \\ \theta &= \tan^{-1}\left(\frac{-\mathbf{R}_{13}}{\sqrt{1-\mathbf{R}_{13}^{2}}}\right) = \tan^{-1}\left(\frac{\sin(\theta)}{\sqrt{1-\sin(\theta)^{2}}}\right) = \tan^{-1}\left(\frac{\sin(\theta)}{\cos(\theta)}\right), \\ \psi &= \tan^{-1}\left(\frac{-\mathbf{R}_{12}}{\mathbf{R}_{11}}\right) = \tan^{-1}\left(\frac{\cos(\theta)\sin(\psi)}{\cos(\theta)\cos(\psi)}\right) = \tan^{-1}\left(\tan(\psi)\right). \end{aligned} \tag{5.4}$$

Of note is the dependence on several terms involving $\cos(\theta)$. When $\theta = \pm\frac{\pi}{2}$ radians, these terms become zero, so the calculations become singular and a unique set of solutions no longer exists. For an airplane, this is associated with it pitching up/down to the point where it becomes perfectly vertical, and roll and yaw become the same thing.

It's important to discuss the relationship between changes in Euler angles and the angular velocities of the object being represented with Euler angles. As such, let there be an angular velocity vector

$$\vec{\omega} = \begin{bmatrix} \omega_{\mathbf{x}} & \omega_{\mathbf{y}} & \omega_{\mathbf{z}} \end{bmatrix}^T, \tag{5.5}$$

where

$$\begin{aligned} \omega_{\mathbf{x}} \in \mathbb{R} \quad & \text{is the angular velocity around the } \mathrm{X}^B \text{ axis,} \\ \omega_{\mathbf{y}} \in \mathbb{R} \quad & \text{is the angular velocity around the } \mathrm{Y}^B \text{ axis, and} \\ \omega_{\mathbf{z}} \in \mathbb{R} \quad & \text{is the angular velocity around the } \mathrm{Z}^B \text{ axis.} \end{aligned}$$

The Euler angle most closely associated with the object in question (for Euler XYZ, the Z rotation angle, or ψ) is changing directly in line with the object's Body frame, so its rate of change $\dot{\psi}$ directly coincides with the object's Z^B angular velocity $\omega_{\mathbf{z}}$ [44]. Now, a change in the next Euler angle (the Y rotation angle, or θ) is not in line with Body frame; to find the set of angular velocities associated with $\dot{\theta}$, the ${}^1\mathbf{R}_B = \mathbf{R}_z(\psi)$ rotation must be undone, leading to $\dot{\theta}$ being rotated by $\mathbf{R}_z^{-1}(\psi) = \mathbf{R}_z(-\psi)$. The last Euler angle (the X rotation angle, or ϕ) is even further from Body frame, so $\dot{\phi}$ must undergo even more rotations to find the corresponding angular velocities. Putting everything together, the net equation for finding angular velocities from changes in Euler XYZ angles becomes

$$\begin{aligned}\begin{bmatrix}\omega_{\mathbf{x}}\\ \omega_{\mathbf{y}}\\ \omega_{\mathbf{z}}\end{bmatrix} &= \mathbf{I}_3\begin{bmatrix}0\\0\\\dot{\psi}\end{bmatrix} + \mathbf{R}_z(-\psi)\begin{bmatrix}0\\\dot{\theta}\\0\end{bmatrix} + \mathbf{R}_z(-\psi)\mathbf{R}_y(-\theta)\begin{bmatrix}\dot{\phi}\\0\\0\end{bmatrix};\\ &= \begin{bmatrix}\cos(\theta)\cos(\psi) & -\sin(\psi) & 0\\ \cos(\theta)\sin(\psi) & \cos(\psi) & 0\\ -\sin(\theta) & 0 & 1\end{bmatrix}\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix} = \mathbf{W}\dot{\vec{\Theta}}.\end{aligned} \tag{5.6}$$

Inverting this process, the changes in Euler ZYX angles can be found from a set of angular velocities via

$$\begin{aligned}\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix} &= \mathbf{W}^{-1}\begin{bmatrix}\omega_{\mathbf{x}}\\ \omega_{\mathbf{y}}\\ \omega_{\mathbf{z}}\end{bmatrix} = \begin{bmatrix}\sec(\theta)\cos(\psi) & \sec(\psi)\sin(\psi) & 0\\ -\sin(\psi) & \cos(\psi) & 0\\ \tan(\psi)\cos(\psi) & \cos(\psi)\sin(\psi) & 1\end{bmatrix}\vec{\omega}\\ \dot{\vec{\Theta}} &= \frac{1}{\cos(\theta)}\begin{bmatrix}\cos(\psi) & \sin(\psi) & 0\\ -\cos(\theta)\sin(\psi) & \cos(\theta)\cos(\psi) & 0\\ \sin(\theta)\cos(\psi) & \sin(\theta)\sin(\psi) & \cos(\theta)\end{bmatrix}\vec{\omega}.\end{aligned} \tag{5.7}$$

Interestingly enough, a division by zero problem occurs at $\theta = \pm\frac{\pi}{2}$ radians. This is linked with the singularity involved with the Euler XYZ coordinate system at $\theta = \pm\frac{\pi}{2}$ radians.

5.3 Quaternions

Unit quaternions are an alternative way of expressing an object's orientation in 3-D space by using hyper-complex numbers [45]. While at first glance this sounds intimidating, it isn't too bad after stepping through it appropriately. To get a feel for how unit quaternions ultimately work: while Euler angles involve using three rotations around pre-defined axes to encode orientation, the unit quaternion involves defining an axis in 3-D space, then executing a single rotation around said axis. This section attempts to step through how unit quaternions achieve that goal. For additional resources, [45] offers an excellent explanation, while [42] and [46] offer supplementary information.

Invented by William Hamilton in 1843, the quaternion consists of four elements: one real element and three hypercomplex elements i, j, and k. The hypercomplex elements are usually grouped together as a vector; as such, referring to the "scalar" part of a quaternion means the real component, while referring to the "vector" part of a quaternion means the hypercomplex component. Equation 5.8 shows typical representations of the quaternion. Equation 5.9 shows how each of the hypercomplex values interacts multiplicitavely with each other. It is important to note that i, j, and k are not commutative; that is, $ij = -ji \neq ji$. All of the other relationships between the hypercomplex values can be derived by manipulating Equation 5.9. Source [45] notes that the hypercomplex multiplications are similar to the cross products of unit vectors $\hat{i}$, $\hat{j}$, and $\hat{k}$ with each other, except that multiplying an element by itself yields -1, not 0.

$$\mathring{q} = q_0 + q_i i + q_j j + q_k k = \begin{bmatrix} q_0 & q_i & q_j & q_k \end{bmatrix}^T = \begin{bmatrix} q_0 & \vec{q}^T \end{bmatrix}^T, \tag{5.8}$$

$$i^2 = j^2 = k^2 = ijk = -1. \tag{5.9}$$

Quaternions have three main mathematical operations: quaternion addition, scalar multiplication, and quaternion multiplication. Using example scalar a and example quaternions $\mathring{p}$ and $\mathring{q}$, Equations 5.10, 5.11, and 5.12 show quaternion addition, scalar multiplication, and quaternion multiplication, respectively [46]. Since the hypercomplex elements within quaternions are not commutative, neither are quaternions in quaternion multiplication.

$$\begin{aligned}\mathring{p} + \mathring{q} &= (p_0 + p_i i + p_j j + p_k k) + (q_0 + q_i i + q_j j + q_k k) \\ &= \begin{bmatrix} p_0 \\ \vec{p} \end{bmatrix} + \begin{bmatrix} q_0 \\ \vec{q} \end{bmatrix} = \begin{bmatrix} p_0 + q_0 \\ \vec{p} + \vec{q} \end{bmatrix},\end{aligned} \tag{5.10}$$

$$a\mathring{q} \ = aq_0 + aq_i i + aq_j j + aq_k k = \begin{bmatrix} aq_0 \\ a\vec{q} \end{bmatrix}, \tag{5.11}$$

$$\begin{aligned}\mathring{p} * \mathring{q} &= (p_0 + p_i i + p_j j + p_k k)(q_0 + q_i i + q_j j + q_k k) \\ &= p_0 q_0 + p_0 q_i i + p_0 q_j j + p_0 q_k k + p_i q_0 i + p_i q_i i^2 + p_i q_j ij + \dots \\ &= \begin{bmatrix} p_0 & -p_i & -p_j & -p_k \\ p_i & p_0 & -p_k & p_j \\ p_j & p_k & p_0 & -p_i \\ p_k & -p_j & p_i & p_0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_i \\ q_j \\ q_k \end{bmatrix} = \begin{bmatrix} p_0 q_0 - \vec{p} \cdot \vec{q} \\ p_0 \vec{q} + q_0 \vec{p} + \vec{p} \times \vec{q} \end{bmatrix}.\end{aligned} \tag{5.12}$$

Two important additional properties of quaternions are the quaternion norm and the quaternion inverse. Equations 5.13 and 5.14 show the process for calculating the norm and the inverse of a given quaternion, respectively [46]. A quaternion with a norm of 1 is said to be a unit quaternion. Calculating the inverse of a unit quaternion is easy, as the fraction is canceled out.

$$\|\mathring{q}\| = q_0^2 + q_i^2 + q_j^2 + q_k^2, \tag{5.13}$$

$$\mathring{q}^{-1} = \begin{bmatrix} q_0 \\ \vec{q} \end{bmatrix}^{-1} = \frac{1}{\|\mathring{q}\|} \begin{bmatrix} q_0 \\ -\vec{q} \end{bmatrix}. \tag{5.14}$$

While complex numbers of unit magnitude can be used to express a rotation of angle θ in a 2-D plane around an axis perpendicular to the plane (as in Euler's formula, shown in Equation 5.15) [45], unit quaternions can be used to express a rotation of angle θ in a 3-D space around an axis specified by a unit vector $\hat{n}$ [46]. Equation 5.16 specifies the unit vector $\hat{n}$ in terms of direction angles given by α, β, and γ, each of which describes the unit vector's angle with the X, Y, and Z axes, respectively. Equation 5.17 shows how the quaternion is composed of the rotation angle and unit vector/direction angles for the desired rotation.

$$e^{i\theta} = \cos(\theta) + i\sin(\theta), \tag{5.15}$$

$$\hat{n} = \begin{bmatrix}\cos(\alpha) & \cos(\beta) & \cos(\gamma)\end{bmatrix}^T, \tag{5.16}$$

$$\begin{aligned}\mathring{q} &= \begin{bmatrix}\cos(\theta/2) \\ \sin(\theta/2)\hat{n}\end{bmatrix} \\ &= \cos(\theta/2) + \sin(\theta/2)\cos(\alpha)i + \sin(\theta/2)\cos(\beta)j + \sin(\theta/2)\cos(\gamma).\end{aligned} \tag{5.17}$$

To rotate a 3-D Euclidean position vector (such as $\vec{r}^{B}$) using a given quaternion $\mathring{q}$, a new quaternion (for example, $\mathring{v}$) is created with a scalar component equal to 0 and a vector component equal to the vector to be rotated. The new quaternion is pre-multiplied by $\mathring{q}$ and post-multiplied by $\mathring{q}^{-1}$ to give a newer quaternion $\mathring{u}$. The vector component of $\mathring{u}$ holds the rotated position vector (such as $\vec{r}^{W}$), while the scalar component will be 0. Equation 5.18 illustrates this process, while [46] provides additional explanation. Performing this process with $\mathring{q}^{-1}$ replacing $\mathring{q}$ and vice versa undoes the rotation.

$$\mathring{u} = \begin{bmatrix}0 \\ \vec{r}^{W}\end{bmatrix} = \mathring{q} * \mathring{v} * \mathring{q}^{-1} = \mathring{q} * \begin{bmatrix}0 \\ \vec{r}^{B}\end{bmatrix} * \mathring{q}^{-1}. \tag{5.18}$$

Applying Equation 5.18 to a generic vector, a generalized rotation matrix can be generated from the unit quaternion's values as in Equation 5.19.

$$
{}^{W}\mathbf{R}_{B} = \begin{bmatrix} q_0^2 + q_i^2 - q_j^2 - q_k^2 & 2(q_i q_j - q_0 q_k) & 2(q_i q_k + q_0 q_j) \\ 2(q_i q_j + q_0 q_k) & q_0^2 - q_i^2 + q_j^2 - q_k^2 & 2(q_j q_k - q_0 q_i) \\ 2(q_i q_k - q_0 q_j) & 2(q_j q_k + q_0 q_i) & q_0^2 - q_i^2 - q_j^2 + q_k^2 \end{bmatrix}. \tag{5.19}
$$

Due to the nature of two quaternion terms always being multiplied together at a time, a quaternion $\mathring{q}$ and its negative $-\mathring{q}$ will both execute the same rotation. Conversely, for any given rotation, there are only two possible, real quaternion entry solutions: a particular quaternion and its negative. By constraining the quaternion's scalar component to be ≥ 0, only one solution becomes possible, except when $q_0 = 0$. Fortunately, this means quaternions do not suffer the singularity problems faced by Euler angles.

As with Euler angles, it is important to discuss the relationship between quaternion rates of change and the angular velocities of the object whose orientation is represented by the quaternion. Using the same angular velocity information as in Section 5.2, Equation 5.20 gives the full change in the quaternion values based on angular velocities [46].

$$
\begin{bmatrix} \dot{q_0} \\ \dot{q_i} \\ \dot{q_j} \\ \dot{q_k} \end{bmatrix} = \dot{\mathring{q}} = \frac{1}{2}\mathring{q} * \begin{bmatrix} 0 \\ \vec{\omega} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -\omega_{\mathbf{x}} & -\omega_{\mathbf{y}} & -\omega_{\mathbf{z}} \\ \omega_{\mathbf{x}} & 0 & \omega_{\mathbf{z}} & -\omega_{\mathbf{y}} \\ \omega_{\mathbf{y}} & -\omega_{\mathbf{z}} & 0 & \omega_{\mathbf{x}} \\ \omega_{\mathbf{z}} & \omega_{\mathbf{y}} & -\omega_{\mathbf{x}} & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_i \\ q_j \\ q_k \end{bmatrix}. \tag{5.20}
$$

The quaternion rate calculations do not suffer any singularity problems, which is quite nice. In practice, however, updating the quaternion based on quaternion rates causes it to slightly use its norm of 1. Over time, if left unchecked, the unit quaternion will blow up and become a non-unit quaternion and wreak havoc on rotation calculations. To prevent this problem after updates, the quaternion needs to be renormalized back to a norm of 1. This can be done by dividing the quaternion by the square root of its quaternion norm.

5.4 Euler Angle/Quaternion Conversions

Should the need arise to move from one orientation coordinate system to the other, methods exist to convert between Euler angles and quaternions [43]. Equation 5.21 shows the process for converting Euler ZYX angles to a unit quaternion, while Equation 5.22 shows the process for converting a unit quaternion into Euler ZYX angles.

$$\begin{aligned} q_0 &= -\sin(\phi/2)\sin(\theta/2)\sin(\psi/2) + \cos(\phi/2)\cos(\theta/2)\cos(\psi/2), \\ q_i &= +\sin(\phi/2)\cos(\theta/2)\cos(\psi/2) + \sin(\phi/2)\sin(\theta/2)\cos(\psi/2), \\ q_j &= -\sin(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\cos(\psi/2), \\ q_k &= +\sin(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\cos(\psi/2). \end{aligned} \tag{5.21}$$

$$\begin{aligned} \phi &= \tan^{-1}\left(\frac{2(q_0 q_i - q_j q_k)}{1 - 2(q_i^2 + qj^2)}\right), \\ \theta &= \tan^{-1}\left(\frac{-2(q_0 q_j + q_i q_k)}{\sqrt{1 - (2(q_0 q_j + q_i q_k))^2}}\right), \\ \psi &= \tan^{-1}\left(\frac{2(q_0 q_k - q_i q_j)}{1 - 2(q_j^2 + q_k^2)}\right). \end{aligned} \tag{5.22}$$

5.5 Comparison of Euler Angles and Quaternions

While Euler angles and quaternions each have their benefits, they also have their drawbacks. Section 5.5.1 discusses the strengths and weaknesses of Euler angles, while Section 5.5.2 discusses the strengths and weaknesses of quaternions.

5.5.1 Euler Angles

Euler angles have the benefit of being intuitive to understand and relatively easy to visualize, especially when expressed in degrees. While it may take someone a little bit of thinking to figure out what an Euler ZYX roll of 10°, pitch of 15°, and yaw of 73° looks like, it can be done. It's also possible to visually estimate an object's Euler angles based on visual inspection. Making decisions about unsafe orientations is also easier, such as choosing to turn off the quadrotor's motors when the pitch and/or roll angle exceeds 60°. When dealing with robotic arms and manipulators with one or two degrees of rotational freedom, Euler angles make a lot of sense, as they easily tie directly to the joint angles of the device.

Unfortunately, these benefits come with several downsides. Singularity issues occur when the middle Euler axis is rotated by $90°/\frac{\pi}{2}$ radians, which makes it impossible to uniquely identify a set of angles for a given rotation. The singularity problem also causes the Euler angle rates to blow up near $90°/\frac{\pi}{2}$ radians, which does bad things.

When working with Euler angles, trigonometric functions become a necessity. Calculating a rotation matrix for a given set of angles requires at least six trigonometric function evaluations (cosine and sine for each angle), each of which is relatively computationally expensive. While modern processors perform these calculations quickly, they add up over time and slow down the program. The trig-based functions also tend to gradually grow in size when differentiated, which causes problems when performing operations like feedback linearization.

From a practicality standpoint, Euler angles are quite easy to confuse. Intrinsic vs. extrinsic rotations is a large source of confusion when determining the rotation matrix ordering. When writing Euler angle equations, the large number of sines and cosines with three different angle values makes it easy to write an equation wrong

and makes it difficult to find errors. When referencing various academic papers that use Euler angles, different papers use different Euler angle combinations like XYZ, ZYX, and ZXY while still using similar/the same variables, which can quickly kill any direct applicability to a project without realizing it.

5.5.2 Unit Quaternions

When performing computations or theoretical derivations related to orientation, unit quaternions have many benefits. They have no inherent singularities, so a given orientation only has one unit quaternion associated with it (it's assumed that the negative unit quaternion will be disregarded). There's only one way to use unit quaternions to represent orientation (not twelve), so there are never any issues associated with confusing one set of unit quaternions with another set.

When differentiated, the unit quaternion terms seem to either start vanishing or remain constant in number. They don't get progressively larger over time, except in complicated equations. For computations, unit quaternions largely only depend on multiplications and additions with the occasional division, all of which are computationally cheaper than trigonometric functions.

As for the downsides, unit quaternions are not very intuitive to work with. While rotations around single axes produces somewhat intuitive results (q_i steadily gets larger and approaches 1 as the object is rotated to $+180°$ around the X axis, and so on for other values and axes), a quaternion of $\begin{bmatrix}0.9745 & 0.1024 & -0.1990 & 0.0167\end{bmatrix}^T$ has no intuitive meaning, other than it's rotated a little bit positively around the X axis, a little more negatively around the Y axis, and hardly at all around the Z axis. The lack of intuitiveness makes it a poor coordinate system for choosing problematic orientations. Fortunately, the unit quaternion can be converted to a set of Euler angles for such decisions.

When updating unit quaternion values from angular velocities, numerical errors and calculation errors will steadily compound over time and cause the norm to deviate away from 1. To prevent this from being a problem, the quaternion periodically needs to be renormalized to 1. Fortunately, this process is more efficient than trying to renormalize something like a rotation matrix, as a rotation matrix has 9 entries that need normalization while the quaternion has 4 entries.

5.6 Heading Frame and The Split Quaternion

While on the topic of reference frames and rotations, it is time to introduce a new frame of reference. The Heading frame, represented by $\{H\}$ and introduced as the Auxiliary frame in [9], comes into existence as part of the feedback linearization process covered in Chapter 7. Effectively, it s a frame used to help re-express the quadrotor's heading/yaw angle (briefly mentioned in Section 5.2) in a manner that has no effect on the thrust vector (discussed in Chapter 6). As the quadrotor's thrust vector lies along the Z^B axis, performing a rotation around the Z^B axis first has no effect on its orientation.

Coordinates in $\{H\}$ are denoted with an H superscript, such as $\vec{x}^H$. Frame $\{H\}$ has the same origin and Z axis as $\{B\}$, except it has undergone a rotation around the Z^B axis such that a single rotation around a XY^W axis will bring it in line with $\{W\}$. Using Euler XYZ angles as an analogy, $\{H\}$ is the frame that exists between the XY and Z matrices. Figure 5.1 shows the rotational coordinate system change from $\{W\}$ through $\{H\}$ to $\{B\}$.

Using unit quaternions, as is done in the feedback linearization process, the $\{B\}$ to $\{W\}$ quaternion is broken into two parts: $\mathring{q}_{xy}$ and $\mathring{q}_z$, as shown in Equation 5.23. Equation 5.24 shows the resulting quaternion rotation. $\{H\}$ exists between the two new quaternions.

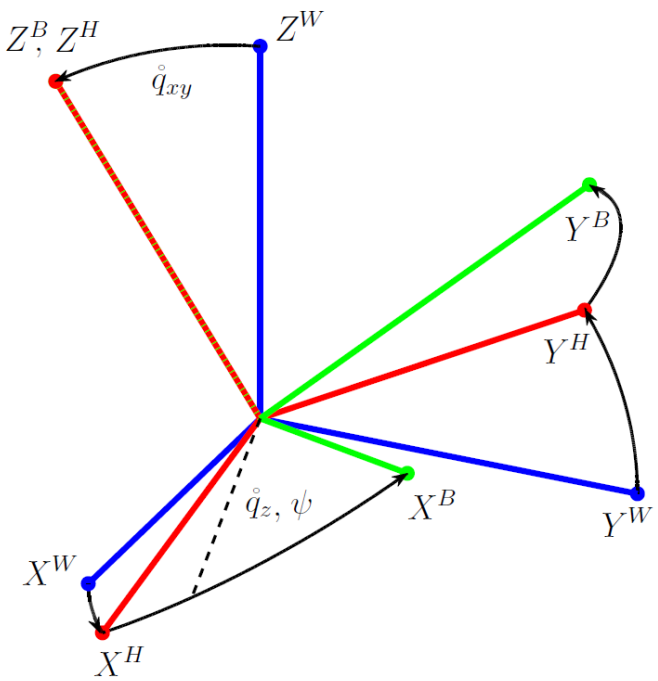


Figure 5.1: Depiction of relative orientations of World frame $\{W\}$, Heading frame $\{H\}$, and Body frame $\{B\}$. Original image source: [9]

$$\mathring{q} = \mathring{q_{xy}} * \mathring{q}_z, \tag{5.23}$$

$$\begin{bmatrix} 0 \\ \vec{r}^W \end{bmatrix} = \mathring{q}_{xy} * \mathring{q}_z * \begin{bmatrix} 0 \\ \vec{r}^B \end{bmatrix} * \mathring{q}_z^{-1} * \mathring{q}_{xy}^{-1}, \tag{5.24}$$

$$\mathring{q_{xy}} = \begin{bmatrix} q_p & q_x & q_y & 0 \end{bmatrix}^T, \tag{5.25}$$

$$\mathring{q}_z = \begin{bmatrix} q_w & 0 & 0 & q_z \end{bmatrix}^T, \tag{5.26}$$

$$1 = q_p^2 + q_x^2 + q_y^2, \tag{5.27}$$

$$1 = q_w^2 + q_z^2, \tag{5.28}$$

$$\begin{bmatrix} q_0 \\ q_i \\ q_j \\ q_k \end{bmatrix} = \begin{bmatrix} q_p \\ q_x \\ q_y \\ 0 \end{bmatrix} * \begin{bmatrix} q_w \\ 0 \\ 0 \\ q_z \end{bmatrix} = \begin{bmatrix} q_p q_w \\ q_w q_x + q_z q_y \\ -q_z q_x + q_w q_y \\ q_p q_z \end{bmatrix}. \tag{5.29}$$

Equations 5.30 through 5.33 demonstrate how to decompose a normal quaternion $\mathring{q}$ into $\mathring{q}_{xy}$ and $\mathring{q}_z$ based on Equation 5.29. As a note, the decomposition process only holds if $q_p \neq 0$. As $q_p = 0$ would correspond to a full 180° rotation around an XY axis (looking at Equation 5.17), this only happens when the quadrotor is perfectly upside down. As the position controller will largely keep the quadrotor upright so the thrust can counteract gravity, this potential singularity issue is largely negligible.

$$q_p = \sqrt{q_0^2 + q_k^2}, \tag{5.30}$$

$$q_w = \frac{q_0}{q_p}, \tag{5.31}$$

$$q_z = \frac{q_k}{q_p}, \tag{5.32}$$

$$\begin{bmatrix} q_x \\ q_y \end{bmatrix} = \begin{bmatrix} q_w & -q_z \\ q_z & q_w \end{bmatrix} \begin{bmatrix} q_i \\ q_j \end{bmatrix}. \tag{5.33}$$

Even though $\mathring{q}_{xy}$ has three variables (q_p, q_x, and q_y), its unit norm requirement (Equation 5.27) effectively restricts q_p based on q_x and q_y, leaving two degrees of freedom (DOF). A similar process with $\mathring{q}_z$ and Equation 5.28 shows that $\mathring{q}_z$ only has one DOF. While Euler XYZ could be used to represent this process, the use of quaternions simplifies the derivation for the feedback linearization process and cuts down on calculation times.

Chapter 6

The Quadrotor Model

The focus of this chapter is a relationship between motor speeds and physical phenomena like thrust and torques, the development of state and input definitions, and a working model of the quadrotor. Section 6.1 talks about the position coordinate system, while Section 6.2 talks about the orientation coordinate system. Section 6.3 examines the conversion from motor speeds to a net thrust and torques applied to the quadrotor. Section 6.4 presents the formal definition of the quadrotor's states and inputs and develops a working model from translational and rotational mechanics.

6.1 Position Representation

As the Vicon system measures object position relative to $\{W\}$, any expression of the quadrotor's position is stated relative to $\{W\}$'s origin. As mentioned in Section 5.1, the Vicon system is set to have an X-right, Y-forward, Z-up coordinate system. The position vector, shown in Equation 6.1, is comprised of x, y, and z. The velocity vector, shown in Equation 6.2, is simply the time derivative of the position vector. While the Vicon system can measure the quadrotor's position, there are no means

to directly measure the quadrotor's velocities.

$$\vec{r} = \begin{bmatrix} x & y & z \end{bmatrix}^T, \tag{6.1}$$

$$\dot{\vec{r}} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T. \tag{6.2}$$

6.2 Orientation Representation

Chapter 5 thoroughly examines the use of Euler angles and quaternions in representing an object's orientation. Quaternions were selected for modeling the quadrotor's orientation due to their lack of trigonometric function evaluations, their simplified algebra, and the nonlinear controller's inherently quaternion-based derivation (discussed in Chapter 7). Euler angles do have a place, however, in creating intuitive orientation decisions, such as for setting unsafe flight condition thresholds. The quaternion can be converted to a set of Euler angles for any particular application that requires them (see Chapter 5, Section 5.4), so using quaternions as the main coordinate system does not eliminate the benefits of using Euler angles.

The quaternion for representing the quadrotor's orientation in $\{W\}$ is expressed in Equation 6.3. Equation 6.4 gives the angular velocity vector used to express the rotation speeds of the quadrotor in $\{B\}$. The Vicon system measures the quaternion, while the Autopilot's IMU measures the angular velocities.As mentioned in Chapter 5, Section 5.6, the quaternion can be broken apart into $\mathring{q}_{xy}$ and $\mathring{q}_z$ components at any time, provided the quadrotor is not upside down.

$$\mathring{q} = \begin{bmatrix} q_0 & q_i & q_j & q_k \end{bmatrix}^T, \tag{6.3}$$

$$\vec{\omega} = \begin{bmatrix} \omega_{\mathbf{x}} & \omega_{\mathbf{y}} & \omega_{\mathbf{z}} \end{bmatrix}^T. \tag{6.4}$$

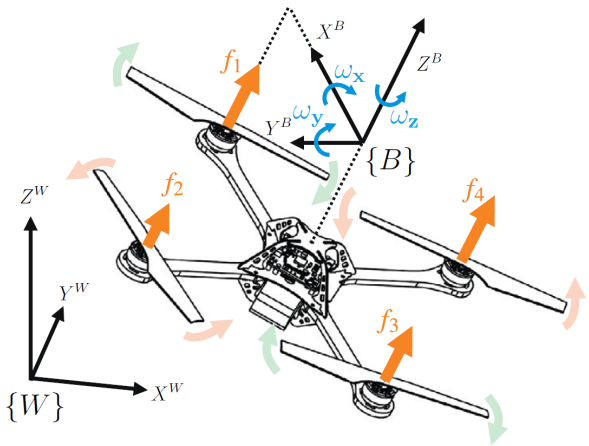


Figure 6.1: AscTec Hummingbird quadrotor model with labeled axes and motors. Source: [16]

6.3 Input Transformation

While a quadrotor's motor speeds are ultimately what controls its movement, their effects on the quadrotor's dynamics aren't as direct as parameters like thrust and torques. This section establishes a relationship between motor/propeller speeds and the net thrust/torques applied to the quadrotor, which simplifies future physical modeling.

Before discussing the physics associated with various propellers rotating at different speeds, it is important to establish a motor/propeller labeling convention to avoid confusion. Figure 6.1 shows a picture of the AscTec Hummingbird model with its axes and motors labeled. The motor labeling convention shown here is the one used throughout the thesis and in the flight control program. As an important note, the AscTec Autopilot uses a motor labeling convention where motors 2 and 3 are switched from Figure 6.1 [24]. To compensate for this discrepancy, motor commands sent to the Autopilot are switched before being sent, and measured motor speeds are swapped once received from the Autopilot. Chapter 4, Section 4.2 touches on this.

When working with the physics associated with the quadrotor's propellers, the three most important parameters are

$b \in \mathbb{R}$ the propeller's thrust coefficient,
$k \in \mathbb{R}$ the propeller's drag coefficient, and
$l \in \mathbb{R}$ the length of the quadrotor's arm.

For a given propeller, the thrust f produced when spinning spinning at an angular velocity Ω is $f = b\Omega^2$ [21]. As the propellers have a fixed pitch, the thrust coefficient b cannot be changed at will to generate more or less thrust, so it remains more or less constant. The motors are fixed to spin around the quadrotor's Z^B axis and to push air downwards, so each propeller's thrust creates a force on the quadrotor along the positive Z^B axis. Adding up the thrusts from all four propellers, the net upward force felt by the quadrotor becomes

$$T = f_1 + f_2 + f_3 + f_4 = b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2). \tag{6.5}$$

All of the propellers create a thrust at a distance l from the quadrotor's center of mass, which generates a torque on the system. Looking back to the coordinate system and motor labeling convention shown in Figure 6.1, propeller 2's thrust creates a positive torque around X^B, while propeller 4's thrust creates a negative torque around X^B. Examining the torques around Y^B, propeller 3's thrust creates a positive torque, while propeller 1's thrust creates a negative torque. Expressing this in equation form,

$$\tau_{\mathbf{x}} = f_2 l - f_4 l = bl(\Omega_2^2 - \Omega_4^2), \tag{6.6}$$

$$\tau_{\mathbf{y}} = f_3 l - f_1 l = bl(\Omega_3^2 - \Omega_1^2). \tag{6.7}$$

As the quadrotor's motors are imparting torque on the propellers to counteract the air drag created by propelling air, the propellers are imparting a torque in the opposite direction on the body of the quadrotor. The drag torque τ generated by an individual propeller is related to its rotational speed Ω by $\tau = k\Omega^2$ [21].

Re-examining Figure 6.1 and recalling the Right-Hand Rule for rotations, motors 1 and 3 are spinning in the negative direction relative to Z^B and motors 2 and 4 are spinning in the positive direction. Bearing in mind the change in direction due to the reactionary forces on the propellers, propellers 1 and 3 are generating a positive torque around Z^B, while propellers 2 and 4 are generating a negative torque. Putting this in equation form,

$$\tau_{\mathbf{z}} = \tau_1 + \tau_2 + \tau_3 + \tau_4 = k(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2). \tag{6.8}$$

Combining Equations 6.5 through 6.8, the thrust and torques generated by a given set of motor speeds can be found via Equation 6.9.

$$\begin{bmatrix} T \\ \tau_{\mathbf{x}} \\ \tau_{\mathbf{y}} \\ \tau_{\mathbf{z}} \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & bl & 0 & -bl \\ -bl & 0 & bl & 0 \\ k & -k & k & -k \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} = \mathbf{M} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix}. \tag{6.9}$$

As b, l, and k will all be greater than zero for any real quadrotor, $\mathbf{M}$ is full rank; as it's a square matrix, $\mathbf{M}$ is invertible, so a unique set of squared motor speeds can be found for a given set of inputs as in Equation 6.10. As the motors can only spin one direction, only one solution will exist for each motor speed.

$$\begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} T \\ \tau_{\mathbf{x}} \\ \tau_{\mathbf{y}} \\ \tau_{\mathbf{z}} \end{bmatrix}. \tag{6.10}$$

While the Autopilot's HLP is configured to send Direct Motor Commands to the LLP (Section 2.3.2), models and control laws can be designed around the more intuitive inputs of thrust and torques. When it's time to generate a given input, the conversion found in Equation 6.10 can be used to get the right motor speeds. From there, Equation 11.2 can be used to get the right motor command values for the HLP/LLP system.

The motors have upper and lower bounds on their rotation speeds. While this creates a rather simple set of motor speed constraints shown as $\Omega_{\min} \leq \Omega_1, \Omega_2, \Omega_3, \Omega_4 \leq \Omega_{\max}$, the constraints on the thrust and torques are not so obvious. Chapter 8 addresses the heuristics and linear programming methods used to ensure the control laws do not violate these constraints.

6.4 Quadrotor Model

Combining the coordinate system definitions from Sections 6.1 and 6.2, the states for the quadrotor become those seen in Equation 6.11. While 13 states are listed, the norm requirement for unit quaternions effectively restricts q_0 in terms of q_i, q_j, and q_k, leaving 12 free states. To simplify calculations and prevent errors resulting from sign changes as the states evolve, q_0 is kept in the state vector.

$$\begin{aligned} \vec{x} &= \begin{bmatrix} \vec{r}^T & \dot{\vec{r}}^T & \mathring{q}^T & \vec{\omega}^T \end{bmatrix}^T \\ &= \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & q_0 & q_i & q_j & q_k & \omega_{\mathbf{x}} & \omega_{\mathbf{y}} & \omega_{\mathbf{z}} \end{bmatrix}^T. \end{aligned} \tag{6.11}$$

Using the mapping established in Section 6.3, the inputs to the quadrotor change from the motor speeds to the net thrust and torques as seen in Equation 6.12.

$$\vec{u} = \begin{bmatrix} T & \tau_{\mathbf{x}} & \tau_{\mathbf{y}} & \tau_{\mathbf{z}} \end{bmatrix}^T. \tag{6.12}$$

For the translational dynamics, only two forces are deemed to be working on the quadrotor: gravity and the net thrust coming from the propellers. While drag is technically affecting the quadrotor, it is deemed to be negligible for the purposes of controller generation and state estimation. Gravity provides a constant acceleration downwards, while the quadrotor's net thrust provides an acceleration inversely proportional to the quadrotor's mass in a direction determined by the quadrotor's orientation. These linear dynamics are represented by Equation 6.13 [21].

$$\ddot{\vec{r}} = {}^{W}\mathbf{R}_{B} \begin{bmatrix} 0 \\ 0 \\ T/M \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix}. \tag{6.13}$$

Using the split quaternion mentioned in Section 5.6, the translational dynamics of the quadrotor collapses to those seen in Equation 6.14 through judicious application of Equations 5.27, 5.28, and 5.29. This removes the heading angle $\mathring{q}_z$ from the dynamics, which helps in the feedback linearization process discussed in Chapter 7. For state prediction purposes, Equation 6.13 is used instead.

$$\begin{aligned} \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{T}{M} \begin{bmatrix} 2((q_p q_w)(q_w q_y - q_z q_x) + (q_w q_x + q_z q_y)(q_p q_z)) \\ 2((q_w q_y - q_z q_x)(q_p q_z) - (q_w q_x + q_z q_y)(q_p q_w)) \\ (q_p q_w)^2 - (q_w q_x + q_z q_y)^2 - (q_w q_y - q_z q_x)^2 + (q_p q_z)^2 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{T}{M} \begin{bmatrix} 2(q_p q_y(q_w^2 + q_z^2)) \\ 2(-q_p q_x(q_w^2 + q_z^2)) \\ (q_w^2 + q_z^2)(q_p^2 - q_x^2 - q_y^2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{T}{M} \begin{bmatrix} 2q_p q_y \\ -2q_p q_x \\ q_p^2 - q_x^2 - q_y^2 \end{bmatrix}. \end{aligned} \tag{6.14}$$

As for the rotational dynamics of the quadrotor, Equation 5.20 already provides the relationship between the quaternion rates and the quadrotor's angular velocities. Euler's equation of motion [42] gives the angular accelerations for a system experiencing net torques, as shown in Equation 6.15. While several effects are left out of Equation 6.15 (gyroscopic effects due to the spinning propellers, torques caused by accelerating/decelerating motors/propellers, and rotational drag), they are deemed to be negligible for the purposes of control law generation.

$$\mathbf{J}\dot{\vec{\omega}} = \vec{\Gamma} - \vec{\omega} \times \mathbf{J}\vec{\omega}, \tag{6.15}$$

where

$\mathbf{J} \in \mathbb{R}^{3x3}$	is the quadrotor's rotational inertia matrix (assumed to be diagonal) and
$\vec{\Gamma} = \begin{bmatrix} \tau_{\mathbf{x}} & \tau_{\mathbf{y}} & \tau_{\mathbf{z}} \end{bmatrix}^T$	represents the roll, pitch, and yaw torque inputs.

Combining Equations 6.13, 5.20, and 6.15, the dynamics of the system become

$$\begin{aligned}
\ddot{x} &= \frac{2T}{M}(q_0 q_j + q_i q_k), \\
\ddot{y} &= \frac{2T}{M}(q_j q_k - q_i q_0), \\
\ddot{z} &= \frac{T}{M}(q_0^2 - q_i^2 - q_j^2 + q_k^2) - g, \\
\dot{q_0} &= -\frac{1}{2}(q_i \omega_{\mathbf{x}} + q_j \omega_{\mathbf{y}} + q_k \omega_{\mathbf{z}}), \\
\dot{q_i} &= \frac{1}{2}(q_0 \omega_{\mathbf{x}} + q_j \omega_{\mathbf{z}} - q_k \omega_{\mathbf{y}}), \\
\dot{q_j} &= \frac{1}{2}(q_0 \omega_{\mathbf{y}} - q_i \omega_{\mathbf{z}} + q_k \omega_{\mathbf{x}}), \\
\dot{q_k} &= \frac{1}{2}(q_0 \omega_{\mathbf{z}} + q_i \omega_{\mathbf{y}} - q_j \omega_{\mathbf{x}}), \\
\dot{\omega}_{\mathbf{x}} &= (\tau_{\mathbf{x}} + \omega_{\mathbf{y}} \omega_{\mathbf{z}} (\mathbf{J}_{yy} - \mathbf{J}_{zz}))/\mathbf{J}_{xx}, \\
\dot{\omega}_{\mathbf{y}} &= (\tau_{\mathbf{y}} + \omega_{\mathbf{x}} \omega_{\mathbf{z}} (\mathbf{J}_{zz} - \mathbf{J}_{xx}))/\mathbf{J}_{yy}, \\
\dot{\omega}_{\mathbf{z}} &= (\tau_{\mathbf{z}} + \omega_{\mathbf{x}} \omega_{\mathbf{y}} (\mathbf{J}_{xx} - \mathbf{J}_{yy}))/\mathbf{J}_{zz}.
\end{aligned} \tag{6.16}$$

Chapter 7

Quadrotor Feedback Linearization and Controller Design

This chapter steps through the lengthy process of performing feedback linearization on the dynamics of the quadrotor using the methods described in [9]. Once the inputs have been formulated such that they linearize the outputs with respect to artificial inputs, an appropriate controller can be applied to the artificial inputs. Section 7.1 explains how feedback linearization works to provide context for some derivation decisions. Section 7.2 discusses the outputs used for the linearization process. Section 7.3 discusses a remapping of the system's inputs such that they more readily appear in the system's dynamics during the linearization process. Section 7.4 examines the linearization process on the quadrotor's altitude, while Section 7.5 examines the linearization process on the quadrotor's X^W/Y^W position. Section 7.6 discusses the leftover heading dynamics. Section 7.7 examines the feasibility of the feedback linearization process. Section 7.8 briefly discusses requirements for flight trajectories. Sections 7.9 through 7.12 apply controllers to the linearized dynamics for exponential Lyapunov stability. Forewarning: this chapter is quite dense.

7.1 Feedback Linearization

This section provides a quick summary of feedback linearization and how it applies to developing nonlinear controllers. For a more rigorous explanation of the whole process, consult [47], [48], and [49]. For a single input, single output (SISO) example system, feedback linearization works by taking a nonlinear system of the form

$$\dot{\vec{x}} = f(\vec{x}) + g(\vec{x})u,$$
$$y = h(\vec{x}),$$

and steadily differentiating the output $y \in \mathbb{R}$ with Lie derivatives [47] with respect to the states $\vec{x} \in \mathbb{R}^n$ until the input $u \in \mathbb{R}$ appears in the form shown in Equation 7.1.

$$y^{(\rho)} = L_f^\rho h + L_g L_f^{\rho-1} h \, u. \tag{7.1}$$

The system is said to have a *relative degree* ρ equal to the number of derivatives taken as long as $L_g L_f^{\rho-1} h \neq 0$ and $L_g L_f^i h = 0$ for $i = 0, 1, \ldots, \rho - 2$. If $\rho = n$, the system will not have uncontrollable internal dynamics [47]. If there are uncontrolled internal dynamics and they're stable, the system can still be controlled to some extent. if they aren't stable, the system cannot be controlled.

Once the inputs appear, equations for the inputs are generated such that they cancel out the dynamics and add an artificial input $v \in \mathbb{R}$ as in Equation 7.2

$$u = \left(L_g L_f^{\rho-1} h\right)^{-1} \left(-L_f^\rho h + v\right), \tag{7.2}$$

which holds as long as $L_g L_f^{\rho-1} h$ is invertible.

With the input equation established in Equation 7.2, Equation 7.1 simplifies to $y^{(\rho)} = v$, which is completely linear with respect to y. A choice for v in the form of

$$v = y_d^{(\rho)} + a_{\rho-1}(y_d^{(\rho-1)} - y^{(\rho-1)}) + \ldots + a_1(\dot{y}_d - \dot{y}) + a_0(y_d - y) \tag{7.3}$$

will cause the system to track a trajectory [47], where a_i, $i = 0, \ldots, \rho - 1$ are adjustable gains for setting the eigenvalues/poles of the now-linear system, and $y_d, \dot{y}_d, \ldots, y_d^{(\rho)}$ represents the desired trajectory and its derivatives.

From there, any kind of linear control law can be used to set a_i, $i = 0, 1, \ldots, \rho - 1$ in Equation 7.3, such as PID-based controllers [50], LQR-based controllers [51], or controllers based on exponentially decaying Lyapunov functions [15]. Once v has been generated, it can be substituted back into Equation 7.2 to get the input.

To expand the feedback linearization process to a multiple input, multiple output (MIMO) system, the same effective process is performed on all of the outputs: keep taking Lie derivatives until an input appears, then formulate inputs such that they invert the system's nonlinearities and add artificial inputs [48]. For an example two-input, two-output system of the form

$$\dot{\vec{x}} = f(\vec{x}) + g_1(\vec{x})u_1 + g_2(\vec{x})u_2, \qquad y_1 = h_1(\vec{x}), \quad y_2 = h_2(\vec{x}),$$

the derivative process yields

$$\begin{aligned} y_1^{\rho_1} &= L_f^{\rho_1} h_1 + L_{g1} L_f^{\rho_1 - 1} h_1 \, u_1 + L_{g2} L_f^{\rho_1 - 1} h_1 \, u_2, \\ y_2^{\rho_2} &= L_f^{\rho_2} h_2 + L_{g1} L_f^{\rho_2 - 1} h_2 \, u_1 + L_{g2} L_f^{\rho_2 - 1} h_2 \, u_2, \end{aligned}$$

which, after collecting terms, becomes

$$\begin{aligned} \begin{bmatrix} y_1^{\rho_1} \\ y_2^{\rho_2} \end{bmatrix} &= \begin{bmatrix} L_f^{\rho_1} h_1 \\ L_f^{\rho_2} h_2 \end{bmatrix} + \begin{bmatrix} L_{g1} L_f^{\rho_1 - 1} h_1 & L_{g2} L_f^{\rho_1 - 1} h_1 \\ L_{g1} L_f^{\rho_2 - 1} h_2 & L_{g2} L_f^{\rho_2 - 1} h_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \\ &= \mathbf{F}(\vec{x}) + \mathbf{G}(\vec{x}).\vec{u} \end{aligned} \tag{7.4}$$

Provided the cumulative relative degree $\sum_{i=1}^{2} \rho_i = n$, internal dynamics will not be a problem. The input can then be constructed with the form

$$\vec{u} = \mathbf{G}^{-1}(\vec{x}) \left(-\mathbf{F}(\vec{x}) + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \right) \tag{7.5}$$

which works as long as $\mathbf{G}(\vec{x})$ is invertible. As $\mathbf{G}(\vec{x})$'s invertibility necessitates a square matrix, this implies via Equation 7.4 that y and u must have the same dimensionality for the feedback linearization process to work.

The feedback linearization process allows the system's full nonlinearities to be properly taken into account without making linear approximations. However, this process is only feasible as long as there are no unstable internal dynamics and $L_g L_f^{\rho-1} h$ or $\mathbf{G}(\vec{x})$ is invertible. As the derivation process is highly model-dependent, this method is potentially highly sensitive to minor variations in the system's parameters (e.g. mass and rotational inertias). An accurate estimate of system parameters can make all the difference in the controller's performance.

7.2 Feedback Linearization Output Definitions

When attempting to control the quadrotor's position output for states x, y, and z, the number of outputs to be linearized is 3. However, as established in Chapter 6, Section 6.3, the total number of inputs is 4. As mentioned in the previous section, the number of outputs must match the number of inputs. Fortunately, Chapter 5, Section 5.6 managed to create a fourth output in the form of the heading/yaw as represented by $\mathring{q}_z$. This output falls out of the dynamics associated with x, y, and z as demonstrated by Equation 6.14 in Chapter 6, Section 6.4.

7.3 Input Redefinition

The torques will never appear in the position feedback linearization, so this section seeks to remap the input vector $\vec{u}$ to values that will appear. As the position feedback linearization will be working with the dynamics using $\{W\}$ and $\{H\}$, the quadro-

tor's states and inputs in $\{B\}$ need to be mapped to $\{H\}$. the word "auxiliary"[9] will be used to denote new states/inputs in $\{H\}$ to distinguish them from original states/inputs in $\{B\}$. The angular velocities $\vec{\omega}$ in $\{B\}$ map to the auxiliary angular velocities $\widetilde{\vec{\omega}}$ in $\{H\}$ via Equation 7.6. As $\{B\}$ and $\{H\}$ share the same Z axis, $\omega_{\mathbf{z}}$ and $\widetilde{\omega_{\mathbf{z}}}$ end up being the same.

$$\begin{bmatrix} 0 \\ \widetilde{\vec{\omega}} \end{bmatrix} = \mathring{q}_z * \begin{bmatrix} 0 \\ \vec{\omega} \end{bmatrix} * \mathring{q}_z^{-1} \quad \Rightarrow \quad \begin{bmatrix} \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{y}}} \\ \widetilde{\omega_{\mathbf{z}}} \end{bmatrix} = \begin{bmatrix} q_w^2 - q_z^2 & -2q_w q_z & 0 \\ 2q_w q_z & q_w^2 - q_z^2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \omega_{\mathbf{x}} \\ \omega_{\mathbf{y}} \\ \omega_{\mathbf{z}} \end{bmatrix}. \tag{7.6}$$

Applying Equation 5.20 to $\mathring{q}_{xy}$ and $\{H\}$, $\mathring{q}_{xy}$ updates via Equation 7.7.

$$\begin{bmatrix} \dot{q_p} \\ \dot{q_x} \\ \dot{q_y} \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -\widetilde{\omega_{\mathbf{x}}} & -\widetilde{\omega_{\mathbf{y}}} & -\widetilde{\omega_{\mathbf{z}}} \\ \widetilde{\omega_{\mathbf{x}}} & 0 & \widetilde{\omega_{\mathbf{z}}} & -\widetilde{\omega_{\mathbf{y}}} \\ \widetilde{\omega_{\mathbf{y}}} & -\widetilde{\omega_{\mathbf{z}}} & 0 & \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{z}}} & \widetilde{\omega_{\mathbf{y}}} & -\widetilde{\omega_{\mathbf{x}}} & 0 \end{bmatrix} \begin{bmatrix} q_p \\ q_x \\ q_y \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -q_x & -q_y & 0 \\ q_p & 0 & q_y \\ 0 & q_p & -q_x \\ -q_y & q_x & q_p \end{bmatrix} \begin{bmatrix} \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{y}}} \\ \widetilde{\omega_{\mathbf{z}}} \end{bmatrix}. \tag{7.7}$$

As $\mathring{q}_{xy}$'s $q_k = \dot{q_k} = 0$, the lowest line of 7.7 can be used to remove $\widetilde{\omega_{\mathbf{z}}}$ via the equality expressed in Equation 7.8.

$$\widetilde{\omega_{\mathbf{z}}} = \frac{q_y \widetilde{\omega_{\mathbf{x}}} - q_x \widetilde{\omega_{\mathbf{y}}}}{q_p}. \tag{7.8}$$

After combining Equations 7.7 and 7.8 and removing $\dot{q_p}$ (as q_p and its derivatives are effectively defined by q_x and q_y via Equation 5.27), the update process for $\mathring{q}_{xy}$ reduces to Equation 7.9. Equation 7.10 defines a matrix $\mathbf{Z}$ that can be used to simplify the representation of Equation 7.9. As Equation 7.11 indicates, $\mathbf{Z}$ can be inverted as long as $q_p \neq 0$.

$$\begin{bmatrix} \dot{q_x} \\ \dot{q_y} \end{bmatrix} = \frac{1}{2q_p} \begin{bmatrix} q_p^2\widetilde{\omega_{\mathbf{x}}} + q_y\left(q_y\widetilde{\omega_{\mathbf{x}}} - q_x\widetilde{\omega_{\mathbf{y}}}\right) \\ q_p^2\widetilde{\omega_{\mathbf{y}}} + q_x\left(-q_y\widetilde{\omega_{\mathbf{x}}} + q_x\widetilde{\omega_{\mathbf{y}}}\right) \end{bmatrix} = \frac{1}{2q_p} \begin{bmatrix} 1-q_x^2 & -q_xq_y \\ -q_xq_y & 1-q_y^2 \end{bmatrix} \begin{bmatrix} \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{y}}} \end{bmatrix}, \quad (7.9)$$

$$\mathbf{Z} = \frac{1}{2} \begin{bmatrix} 1-q_x^2 & -q_xq_y \\ -q_xq_y & 1-q_y^2 \end{bmatrix}, \quad (7.10)$$

$$\mathbf{Z}^{-1} = \frac{2}{q_p^2} \begin{bmatrix} 1-q_y^2 & q_xq_y \\ q_xq_y & 1-q_x^2 \end{bmatrix}, \quad (7.11)$$

$$\begin{aligned} \dot{\mathbf{Z}} &= \frac{1}{2} \begin{bmatrix} -2q_x\dot{q_x} & -q_y\dot{q_x} - q_x\dot{q_y} \\ -q_y\dot{q_x} - q_x\dot{q_y} & -2q_y\dot{q_y} \end{bmatrix} \quad (7.12) \\ &= \frac{1}{4q_p} \begin{bmatrix} 2q_x(q_x^2-1)\widetilde{\omega_{\mathbf{x}}} + 2q_x^2q_y\widetilde{\omega_{\mathbf{y}}} & q_y(2q_x^2-1)\widetilde{\omega_{\mathbf{x}}} + q_x(2q_y^2-1)\widetilde{\omega_{\mathbf{y}}} \\ q_y(2q_x^2-1)\widetilde{\omega_{\mathbf{x}}} + q_x(2q_y^2-1)\widetilde{\omega_{\mathbf{y}}} & 2q_xq_y^2\widetilde{\omega_{\mathbf{x}}} + 2q_y(q_y^2-1)\widetilde{\omega_{\mathbf{y}}} \end{bmatrix}. \end{aligned}$$

By deriving Equation 7.9 with respect to time while using Equation 7.10's definition of $\mathbf{Z}$, a relationship between the $\mathring{q}_{xy}$ quaternion's acceleration and auxiliary angular accelerations can be found in Equation 7.13. Equation 7.14, the time derivative of Equation 7.6 determines the relationship between auxiliary angular accelerations, angular accelerations, angular velocities, and $\mathring{q}_z$.

$$\begin{bmatrix} \ddot{q_x} \\ \ddot{q_y} \end{bmatrix} = \frac{1}{q_p}\mathbf{Z} \begin{bmatrix} \dot{\widetilde{\omega_{\mathbf{x}}}} \\ \dot{\widetilde{\omega_{\mathbf{y}}}} \end{bmatrix} + \frac{1}{q_p}\dot{\mathbf{Z}} \begin{bmatrix} \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{y}}} \end{bmatrix} - \frac{\dot{q_p}}{q_p^2}\mathbf{Z} \begin{bmatrix} \widetilde{\omega_{\mathbf{x}}} \\ \widetilde{\omega_{\mathbf{y}}} \end{bmatrix}, \quad (7.13)$$

$$\begin{bmatrix} \dot{\widetilde{\omega_{\mathbf{x}}}} \\ \dot{\widetilde{\omega_{\mathbf{y}}}} \end{bmatrix} = \omega_{\mathbf{z}} \begin{bmatrix} -2q_wq_z & q_z^2-q_w^2 \\ q_w^2-q_z^2 & -2q_wq_z \end{bmatrix} \begin{bmatrix} \omega_{\mathbf{x}} \\ \omega_{\mathbf{y}} \end{bmatrix} + \begin{bmatrix} q_w^2-q_z^2 & -2q_wq_z \\ 2q_wq_z & q_w^2-q_z^2 \end{bmatrix} \begin{bmatrix} \dot{\omega}_{\mathbf{x}} \\ \dot{\omega}_{\mathbf{y}} \end{bmatrix}. \quad (7.14)$$

When Equation 7.13 is combined with Equation 7.14, $\ddot{q_x}$ and $\ddot{q_y}$ can be translated to $\dot{\omega}_{\mathbf{x}}$ and $\dot{\omega}_{\mathbf{y}}$. Euler's equation of motion (Equation 6.15) establishes a relationship between $\dot{\vec{\omega}}$ and the current torque inputs $\vec{\Gamma}$ based on the system's current angular velocities. By choosing to use Equation 6.15 to remap the inputs to

$$\vec{u}_{temp} = \begin{bmatrix} T & \dot{\omega}_{\mathbf{x}} & \dot{\omega}_{\mathbf{y}} & \dot{\omega}_{\mathbf{z}} \end{bmatrix}^T.$$

The inputs can then be remapped again via Equations 7.14 and Equation 7.13, at which point the new inputs to the system become

$$\vec{u}_{new} = \begin{bmatrix} T & \ddot{q}_x & \ddot{q}_y & \dot{\omega}_{\mathbf{z}} \end{bmatrix}^T.$$

Section 7.4 will linearize the z state dynamics by developing an equation for thrust (T). Section 7.5 will linearize the x and y state dynamics by developing equations for $\mathring{q}_{xy}$'s accelerations ($\ddot{q}_x$ and $\ddot{q}_y$). Section 7.6 will linearize the quadrotor's heading ($\mathring{q}_z$) through a simple coordinate system change, and the new coordinate system can easily be controlled through $\dot{\omega}_{\mathbf{z}}$.

7.4 Altitude Linearization

This section truly starts the feedback linearization process of the quadrotor's dynamics, encounters a snag, and develops an equation for the quadrotor's thrust to proceed with the linearization process. After splitting the quaternion in two (Section 5.6), remapping the angular velocities to quaternion rates (Equations 7.6 and 7.9), and ejecting the $\mathring{q}_z$ quaternion and its associated terms from the translational dynamics in Equation 6.14, the effective state space for the position feedback linearization collapses down to 10 degrees of freedom (DOF), as q_p and $\dot{q}_p$ are restricted by the other terms of $\mathring{q}_{xy}$ through Equation 5.27. Below is a list of the effective degrees of freedom for the feedback linearization process.

$$\vec{z} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & q_x & q_y & \dot{q_x} & \dot{q_y} \end{bmatrix}^T. \tag{7.15}$$

When starting the feedback linearization process on $h(x) = \begin{bmatrix} x & y & z \end{bmatrix}^T$, no inputs appear until each state's second derivative, as seen in Equation 6.14. Referencing this equation, $\ddot{x}$ is affected by T as long as $q_p q_y \neq 0$, or as long as the quadrotor is partially rotated around Y^W. A similar relationship exists between $\ddot{y}$ and T for

X^W. $\ddot{z}$ is affected by T as long as $q_x^2 + q_y^2 \neq \frac{1}{2}$, which means the quadrotor cannot be horizontal for vertical thrust. This results in up to six states ($\vec{r}$ and $\dot{\vec{r}}$) being directly affected by one input, which sets the current relative degree $\rho = 6$ and leaves four states as internal dynamics. Applying Equation 7.4 and 7.5 here won't work, as there are three outputs ($\ddot{x}$, $\ddot{y}$, $\ddot{z}$) and one input (T).

To circumvent this issue, [9] proposes performing an input-output linearization for z and T, which converts T from an input to an expression comprised of states and desired trajectory values. Analyzing purely the altitude dynamics of the system, the reduced system has states, input, and output as defined in Equation 7.16.

$$\begin{aligned} \vec{z}_z &= \begin{bmatrix} z & \dot{z} & q_x & q_y \end{bmatrix}^T = \begin{bmatrix} z_{z1} & z_{z2} & z_{z3} & z_{z4} \end{bmatrix}^T, \\ u_z &= T, \\ y_z &= z_{z1}. \end{aligned} \tag{7.16}$$

Relating this to the feedback linearization process discussed in Section 7.1, performing the derivatives on the system's output generates the system equations found in Equation 7.17 and Lie derivatives found in Equation 7.18. The relative degree for the the altitude linearization, provided $L_g L_f h(z_z) \neq 0$ (i.e. the quadrotor doesn't become perfectly horizontal), is $\rho_z = 2$.

$$\begin{aligned} \dot{y_z} &= \dot{z_{z1}} = z_{z2}, \\ \ddot{y_z} &= \dot{z_{z2}} = -g + \frac{1 - 2z_{z3}^2 - 2z_{z4}^2}{M} u_z, \end{aligned} \tag{7.17}$$

$$\begin{aligned} L_f h(\vec{z}_z) &= z_{z2}, & L_g h(\vec{z}_z) &= 0, \\ L_f^2 h(\vec{z}_z) &= -g, & L_g L_f h(\vec{z}_z) &= \frac{1 - 2z_{z3}^2 - 2z_{z4}^2}{M}. \end{aligned} \tag{7.18}$$

Using Equations 7.2 and 7.18, Equation 7.20 gives the resulting value for T, where γ represents $\mathring{q}_{xy}$'s effect on thrust along Z^W. Notably, $T \to \infty$ as $\gamma \to 0$, which happens as $q_x^2 + q_y^2 \to \frac{1}{2}$, or as the quadrotor approaches a horizontal orientation. Equation 7.22 shows the appropriate solution for the artificial input v_z based on Equation 7.3 with the single addition of an integral term.

$$u_z = \frac{1}{L_g L_f h(\vec{z}_z)} \left(-L_f^2 h(\vec{z}_z) + v_z \right), \tag{7.19}$$

$$T = \frac{M}{\gamma} \left(g + v_z \right), \tag{7.20}$$

$$\gamma = 1 - 2q_x^2 - 2q_y^2, \tag{7.21}$$

$$v_z = \ddot{z}_d + a_{2z}(\dot{z}_d - \dot{z}) + a_{1z}(z_d - z) + a_{0z} \int_0^t (z_d - z) dt. \tag{7.22}$$

Equation 7.20 collapses the Z^W dynamics down to $\ddot{z} = v_z$, rendering the altitude control completely linear from v_z to z. With the knowledge that T is now completely in terms of state variables and desired trajectory values, it can be treated as though it's not an input when performing input-output linearization for x and y. Section 7.10 will cover the method used to generate appropriate values for a_{iz}, $i = 0, 1, 2$.

7.5 Horizontal Position Linearization

Now that Section 7.4 has defined the thrust input T in terms of states and desired trajectory terms, it can be differentiated for the purposes of input-output linearization. Performing input-output linearization for x and y requires taking more derivatives on Equation 6.14 to make the inputs $\ddot{q}_x$ and $\ddot{q}_y$ appear. Equations 7.23 and 7.24 show the third and fourth derivatives of the outputs, respectively. The inputs appear after the fourth derivative.

$$\begin{bmatrix} x^{(3)} \\ y^{(3)} \end{bmatrix} = \frac{\dot{T}}{M} \begin{bmatrix} 2q_p q_y \\ -2q_p q_x \end{bmatrix} + \frac{T}{M} \begin{bmatrix} 2q_y & 0 & 2q_p \\ -2q_x & -2q_p & 0 \end{bmatrix} \begin{bmatrix} \dot{q_p} \\ \dot{q_x} \\ \dot{q_y} \end{bmatrix}, \tag{7.23}$$

$$\begin{aligned} \begin{bmatrix} x^{(4)} \\ y^{(4)} \end{bmatrix} = \frac{\ddot{T}}{M} \begin{bmatrix} 2q_p q_y \\ -2q_p q_x \end{bmatrix} + \left(2\frac{\dot{T}}{M} \begin{bmatrix} 2q_y & 0 & 2q_p \\ -2q_x & -2q_p & 0 \end{bmatrix} \cdots \right. \\ \left. + \frac{T}{M} \begin{bmatrix} 2\dot{q_y} & 0 & 2\dot{q_p} \\ -2\dot{q_x} & -2\dot{q_p} & 0 \end{bmatrix} \right) \begin{bmatrix} \dot{q_p} \\ \dot{q_x} \\ \dot{q_y} \end{bmatrix} + \frac{T}{M} \begin{bmatrix} 2q_y & 0 & 2q_p \\ -2q_x & -2q_p & 0 \end{bmatrix} \begin{bmatrix} \ddot{q_p} \\ \ddot{q_x} \\ \ddot{q_y} \end{bmatrix}. \end{aligned} \tag{7.24}$$

Equations 7.25 through 7.28 provide the first and second derivatives of T and q_p, as they appear throughout Equations 7.23 and 7.24. Derivatives of q_p come from the definition established in Equation 5.27. To make things more difficult, $\ddot{q_x}$ and $\ddot{q_y}$ appear in $\ddot{T}$ and $\ddot{q_p}$.

$$\dot{T} = \frac{M\dot{v_z} + 4T\left(q_x\dot{q_x} + q_y\dot{q_y}\right)}{\gamma}, \tag{7.25}$$

$$\ddot{T} = \frac{1}{\gamma}\left(M\ddot{v_z} + 8\dot{T}(q_x\dot{q_x} + q_y\dot{q_y}) + 4T(\dot{q_x}^2 + \dot{q_y}^2) + 4T \begin{bmatrix} q_x & \dot{q_y} \end{bmatrix} \begin{bmatrix} \ddot{q_x} \\ \ddot{q_y} \end{bmatrix} \right), \tag{7.26}$$

$$\dot{q_p} = -\frac{q_x\dot{q_x} + q_y\dot{q_y}}{q_p}, \tag{7.27}$$

$$\ddot{q_p} = -\frac{1}{q_p}\left(\dot{q_p}^2 + \dot{q_x}^2 + \dot{q_y}^2 + \begin{bmatrix} q_x & q_y \end{bmatrix} \begin{bmatrix} \dot{q_x} \\ \dot{q_y} \end{bmatrix} \right). \tag{7.28}$$

For the MIMO input-output linearization to work, the equation to be linearized must have the affine form found in Equation 7.4. This unfortunately means having to substitute Equations 7.25 through 7.28 into 7.24 and refactoring everything until the inputs are shown to be affine with the states.

After a very large amount of tedious refactoring, combining Equations 7.24, 7.26, and 7.28 collapses down to the result below, which is grouped to match Equation 7.4 in an attempt to make the monstrous equation more palatable. While this solution is almost exactly the same as the one presented in [9], their use of a Z-down coordinate system differs with this thesis's Z-up coordinate system, so the whole derivation process had to be repeated to find the small sign changes.

$$\begin{bmatrix} x^{(4)} \\ y^{(4)} \end{bmatrix} = \mathbf{F} + \mathbf{G} \begin{bmatrix} \ddot{q}_x \\ \ddot{q}_y \end{bmatrix}, \tag{7.29}$$

$$\mathbf{F} = \frac{\dddot{v}_z}{\gamma} \begin{bmatrix} 2q_p q_y \\ -2q_p q_x \end{bmatrix} + \left(\frac{4\dot{T}}{M}\mathbf{N}_1 + \frac{2T}{M}\mathbf{N}_2 \right) \begin{bmatrix} \dot{q}_p \\ \dot{q}_y \\ \dot{q}_y \end{bmatrix}, \tag{7.30}$$

$$\mathbf{G} = \frac{2T}{m}\mathbf{N}_3, \tag{7.31}$$

$$\mathbf{N}_1 = \begin{bmatrix} q_y & \dfrac{4q_p q_x q_y}{\gamma} & q_p\left(\dfrac{4q_y^2}{\gamma} + 1\right) \\ -q_x & -q_p\left(\dfrac{4q_x^2}{\gamma} + 1\right) & \dfrac{-4q_p q_x q_y}{\gamma} \end{bmatrix},$$

$$\mathbf{N}_2 = \begin{bmatrix} \dot{q_y} - \dfrac{q_y \dot{q_p}}{q_p} & \dfrac{q_y \dot{q_x}(4q_p^2 - \gamma)}{q_p \gamma} & \dfrac{q_y \dot{q_y}(4q_p^2 - \gamma)}{q_p \gamma} + \dot{q_p} \\ -\dot{q_x} + \dfrac{q_x \dot{q_p}}{q_p} & \dfrac{q_x \dot{q_x}(\gamma - 4q_p^2)}{q_p \gamma} - \dot{q_p} & \dfrac{q_x \dot{q_y}(\gamma - 4q_p^2)}{q_p \gamma} \end{bmatrix},$$

$$\mathbf{N}_3 = \begin{bmatrix} \dfrac{4q_p q_x q_y}{\gamma} - \dfrac{q_x q_y}{q_p} & \dfrac{4q_p q_y^2}{\gamma} + q_p - \dfrac{q_y^2}{q_p} \\ -\dfrac{4q_p q_x^2}{\gamma} - q_p + \dfrac{q_x^2}{q_p} & -\dfrac{4q_p q_x q_y}{\gamma} + \dfrac{q_x q_y}{q_p} \end{bmatrix}.$$

After arriving at the fourth derivatives of x and y, the relative degrees for each output are $\rho_x = \rho_y = 4$ as long as $\gamma \neq 0$ and $q_p \neq 0$. When added together, $\rho_x + \rho_y + \rho_z = 10$, which matches the total degrees of freedom in the position control system. As far as position control is concerned, the system is therefore completely input-output linearizable. Section 7.6 will wrap up linearizing the heading portion of the dynamics.

The system is linearized by choosing $\ddot{q_x}$ and $\ddot{q_y}$ based on Equations 7.30 and 7.31 to be equal to

$$\begin{bmatrix} \ddot{q_x} \\ \ddot{q_y} \end{bmatrix} = (\mathbf{G})^{-1} \left(-\mathbf{F} + \begin{bmatrix} v_x \\ v_y \end{bmatrix} \right), \tag{7.32}$$

$$\mathbf{G}^{-1} = \frac{1}{2q_p(g+v_z)} \begin{bmatrix} \gamma q_x q_y(-4q_p^2+\gamma) & \gamma(\gamma q_y^2 - 4q_p^2 q_y^2 - \gamma q_p^2) \\ \gamma(4q_p^2 q_x^2 + \gamma q_p^2 - \gamma q_x^2) & \gamma q_x q_y(4q_p - \gamma) \end{bmatrix}, \tag{7.33}$$

which, according to Equation 7.33, is viable as long as $q_p \neq 0$ (not perfectly upside down) and $v_z \neq -g$. By setting v_x and v_y based on the formats presented in Equations 7.3 and 7.22, the system will track desired trajectories for states x and y. Section 7.11 will cover the method used to generate appropriate values for a_{ix} and a_{iy}, $i = 0, 1, 2, 3, 4$.

$$\begin{aligned} v_x = {} & x_d^{(4)} + a_{4x}(x_d^{(3)} - x^{(3)}) + a_{3x}(\ddot{x}_d - \ddot{x}) + a_{2x}(\dot{x}_d - \dot{x}) \dots \\ & + a_{1x}(x_d - x) + a_{0x} \int_0^t (x_d - x) dt, \end{aligned} \tag{7.34}$$

$$\begin{aligned} v_y = {} & y_d^{(4)} + a_{4y}(y_d^{(3)} - y^{(3)}) + a_{3y}(\ddot{y}_d - \ddot{y}) + a_{2y}(\dot{y}_d - \dot{y}) \dots \\ & + a_{1y}(y_d - y) + a_{0y} \int_0^t (y_d - y) dt. \end{aligned} \tag{7.35}$$

As a final note, $\dot{v}_z$ and $\ddot{v}_z$ both appear in the x and y input-output linearizations. While they are simply straightforward derivatives of Equation 7.22, they are included below in Equations 7.36 and 7.37 for the sake of being thorough.

$$\dot{v}_z = z_d^{(3)} + a_{2z}(\ddot{z}_d - \ddot{z}) + a_{1z}(\dot{z}_d - \dot{z}) + a_{0z}(z_d - z), \tag{7.36}$$

$$\ddot{v}_z = z_d^{(4)} + a_{2z}(z_d^{(3)} - z^{(3)}) + a_{1z}(\ddot{z}_d - \ddot{z}) + a_{0z}(\dot{z}_d - \dot{z}). \tag{7.37}$$

7.6 Heading Linearization

This section handles the quadrotor's heading "linearization". As the position linearization processes handled 10 free states ($\vec{r}$, $\dot{\vec{r}}$, q_x, $\dot{q_x}$, q_y, and $\dot{q_y}$) and the quadrotor possesses 12 in total ($\vec{r}$, $\dot{\vec{r}}$, $\mathring{q}$, and $\vec{\omega}$), two remain (q_z and $\dot{q_z}$, or q_z and $\omega_{\mathbf{z}}$).

While quaternions are convenient for modeling and control derivations, they are unintuitive when trying to create reference trajectories. As such, $\mathring{q}_z$ is converted to an Euler angle ψ to represent a heading/yaw rotation angle. As $\mathring{q}_z$ is associated with an immediate rotation from $\{B\}$ around Z^W, an Euler XYZ or Euler YXZ convention is comparable. Using [43] and Equation 5.19, Equation 7.38 arises for both Euler XYZ and YXZ conventions.

$$\psi = \tan^{-1}\left(\frac{2q_w q_z}{q_w^2 - q_z^2}\right). \tag{7.38}$$

As the Z axes for both $\{B\}$ and $\{H\}$ are aligned, ψ and $\omega_{\mathbf{z}}$ are also aligned, meaning $\dot{\psi} = \omega_{\mathbf{z}}$ (Chapter 5, Section 5.2). Differentiating ψ one more time, $\ddot{\psi} = \dot{\omega}_{\mathbf{z}}$, which is the remaining input. By simply changing coordinate systems for $\mathring{q}_z$, it becomes linear with the input $\dot{\omega}_{\mathbf{z}}$. The two differentiations to make the final input appear sets the relative degree $\rho_\psi = 2$, bringing the system's relative degree up to 12. The system is therefore shown to be fully input-output linearizable.

A controller of the form shown in Equation 7.39 will cause the quadrotor to track a given heading trajectory and, due to the integral term, overcome any steady state errors. To prevent bizarre behaviors, the difference between ψ_d and ψ should always be restricted between $\pm\pi$. Section 7.12 covers the calculation of $a_{i\psi}, i = 0, 1, 2$.

$$\begin{aligned}\dot{\omega}_{\mathbf{z}} &= \dot{\omega}_{\mathbf{z}\,d} + a_{2\psi}(\omega_{\mathbf{z}\,d} - \omega_{\mathbf{z}}) + a_{1\psi}(\psi_d - \psi) + a_{0\psi}\int_0^t (\psi_d - \psi)dt \\ &= \dot{\omega}_{\mathbf{z}\,d} + a_{2\psi}(\omega_{\mathbf{z}\,d} - \omega_{\mathbf{z}}) + a_{1\psi}(\psi_d - \tan^{-1}\left(\frac{2q_w q_z}{q_w^2 - q_z^2}\right)) \\ &\quad + a_{0\psi}\int_0^t (\psi_d - \tan^{-1}\left(\frac{2q_w q_z}{q_w^2 - q_z^2}\right))dt.\end{aligned} \tag{7.39}$$

7.7 Linearization Feasibility

The linearization technique is susceptible to two issues: $q_p = 0$ and $\gamma = 0$. q_p is extracted from $\mathring{q}$ by manipulating Equations 5.28 and 5.29 to get

$$q_p = \sqrt{q_p^2\left(q_w^2 + q_z^2\right)} = \sqrt{q_0^2 + q_k^2}. \tag{7.40}$$

The value q_p can only be 0 if $q_0 = q_k = 0$, which is associated with a full 180° revolution around an XY^W axis (i.e. when it's fully upside down). As the position controller practically requires the quadrotor to be upright to counteract gravity, this isn't a concern.

Re-examining Equations 6.14 and 7.21, $\gamma = 0$ is associated with the thrust vector being rotated into the XY^W plane where it has no Z^W component (in other words, when the quadrotor's horizontal). This presents a far more real concern, as the quadrotor could potentially attempt to execute an overly aggressive maneuver and approach becoming horizontal, which would cause the thrust (and several other terms) to blow up. As such, special care is required to ensure the controller isn't overly aggressive in its tracking of reference trajectories. At the cost of proper linearization, additional safety measures can be added to keep the thrust value within safe ranges. Chapter 8 addresses this concern.

Addressing the feasibility of inverting $\mathbf{G}$, a singularity only occurs when $q_p = 0$ (addressed above) and $v_z = -g$. As long as the quadrotor is not upside down and is not commanding a large negative acceleration along Z^W, the inversion process remains feasible.

7.8 Trajectory Generation

This section briefly covers the types of trajectory values the feedback linearization process requires to be effective. Equations 7.22, 7.34, 7.35, 7.36, 7.37, and 7.39 all reference setting artificial input values based on the quadrotor's current state and a desired trajectory. Accumulating all trajectory terms in one place, each term and its derivatives really start adding up, with Table 7.1 providing a full list.

Table 7.1: List of Trajectory Values to Generate for Desired Flight Path

x states:	x_d	$\dot{x}_d$	$\ddot{x}_d$	$x_d^{(3)}$	$x_d^{(4)}$
y states:	y_d	$\dot{y}_d$	$\ddot{y}_d$	$y_d^{(3)}$	$y_d^{(4)}$
z states:	z_d	$\dot{z}_d$	$\ddot{z}_d$	$z_d^{(3)}$	$z_d^{(4)}$
ψ states:	ψ_d	$\omega_{\mathbf{z}\,d}$	$\dot{\omega}_{\mathbf{z}\,d}$		

Totaling at 18 different values for effectively four different trajectories, the list is not insubstantial for a system attempting to run quickly in real time. All three position terms require not only position references, but velocity, acceleration, jerk, and snap. Depending on the aggressiveness of the designed controller, discontinuities in given trajectories can lead to violent responses. Sinusoid-based trajectories have the benefit of being smooth and continuously differentiable down to an infinite number of derivatives, though that only works well for continuous flight patterns. The problem of creating a gradual transition into the sinusoidal trajectory still exists. As the focus of this thesis is more on controller/estimator design and hardware configuration, optimal trajectory generation is a problem left for others to solve.

7.9 Exponential Lyapunov Controller

Now that the system's dynamics have effectively been linearized, it's time to apply a controller to the linearized dynamics. For a linear set of system dynamics of the form

$$\dot{\vec{x}} = \mathbf{A}\vec{x} + \mathbf{B}\vec{u}, \tag{7.41}$$

paper [15] discusses the design of controllers that guarantee exponential Lyapunov stability [52]. The controller starts with a desired Lyapunov solution of the form

$$\dot{\mathbf{V}} = -\alpha\mathbf{V}, \tag{7.42}$$

where α is a single tunable convergence parameter. Through the selection of a quadratic Lyapunov function $\mathbf{V} = \vec{x}^{\,T}\mathbf{P}\vec{x}$, Equation 7.42 works out to be

$$\begin{aligned} \dot{\mathbf{V}} &= 2\vec{x}\mathbf{P}\left(\mathbf{A}\vec{x} + \mathbf{B}\vec{u}\right) \\ &= \vec{x}^{\,T}\left(\mathbf{A}^{T}\mathbf{P} + \mathbf{P}\mathbf{A}\right)\vec{x} + 2\vec{x}^{\,T}\mathbf{P}\mathbf{B}\vec{u} = -\alpha\vec{x}^{\,T}\mathbf{P}\vec{x}. \end{aligned} \tag{7.43}$$

By choosing

$$\vec{u} = -\mathbf{B}^{T}\mathbf{P}\vec{x}, \tag{7.44}$$

the equation reduces to

$$\vec{x}^{\,T}\left(\mathbf{A}^{T}\mathbf{P} + \mathbf{P}\mathbf{A}\right)\vec{x} - 2\vec{x}^{\,T}\mathbf{P}\mathbf{B}\mathbf{B}^{T}\mathbf{P}\vec{x} = -\alpha\vec{x}^{\,T}\mathbf{P}\vec{x}, \tag{7.45}$$

for which the Algebraic Ricatti Equation below can be solved [51] for $\mathbf{P}$

$$\left(\mathbf{A} + \frac{\alpha}{2}\mathbf{I}\right)^{T}\mathbf{P} + \mathbf{P}\left(\mathbf{A} + \frac{\alpha}{2}\mathbf{I}\right) - 2\mathbf{P}\mathbf{B}\mathbf{B}^{T}\mathbf{P} = 0. \tag{7.46}$$

After solving Equation 7.46 for $\mathbf{P}$, substituting $\mathbf{P}$ into Equation 7.44 generates an expression for the input based on the states that ensures exponential stability. In essence, [15] creates a linear controller capable of being tuned with only one parameter α.

7.10 Altitude Controller

The linearized altitude dynamics possess a total of two states: z and $\dot{z}$. Generating the linear equation for the system with an added integral term,

$$\begin{bmatrix} z \\ \dot{z} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \int z \\ z \\ \dot{z} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} v_z = \mathbf{A}\vec{z}_z + \mathbf{B}v_z. \tag{7.47}$$

Applying the exponential Lyapunov controller to Equation 7.47 with $\alpha = 3$, the input v_z becomes

$$v_z = -\begin{bmatrix} 27 & 27 & 9 \end{bmatrix} \vec{z}_z. \tag{7.48}$$

Putting this into the form found in Equation 7.22, the expression with the added integral term becomes

$$v_z = \ddot{z}_d + 9(\dot{z}_d - \dot{z}) + 27(z_d - z) - 27\int_0^t (z_d - z)dt. \tag{7.49}$$

7.11 Horizontal Position Controller

Applying the method shown in Section 7.10, the $\mathbf{A}$ and $\mathbf{B}$ matrices for both the x and y linearized dynamics (with integral terms) collapse down to Equation 7.41 with $\mathbf{A}$ and $\mathbf{B}$ being

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Running through the same calculation process with $\alpha = 4$, v_x and v_y from Equations 7.34 and 7.35 become

$$\begin{aligned} v_x &= x_d^{(4)} + 20(x_d^{(3)} - x^{(3)}) + 160(\ddot{x}_d - \ddot{x}) + 640(\dot{x}_d - \dot{x}) \cdots \\ &\quad + 1280(x_d - x) + 1024 \int_0^t (x_d - x)dt, \end{aligned} \tag{7.50}$$

$$\begin{aligned} v_y &= y_d^{(4)} + 20(y_d^{(3)} - y^{(3)}) + 160(\ddot{y}_d - \ddot{y}) + 640(\dot{y}_d - \dot{y}) \cdots \\ &\quad + 1280(y_d - y) + 1024 \int_0^t (y_d - y)dt. \end{aligned} \tag{7.51}$$

The gains seem high at first glance, but for appropriately generated trajectories, they should not create a problem.

7.12 Heading Controller

For the heading controller, the process from Section 7.10 was repeated with $\alpha = 3$ and

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

to modify Equation 7.39 and generate the control input

$$\begin{aligned} \dot{\omega}_{\mathbf{z}} = \dot{\omega}_{\mathbf{z}\,d} + 9(\omega_{\mathbf{z}\,d} - \omega_{\mathbf{z}}) + 27(\psi_d - \tan^{-1}\left(\frac{2q_w q_z}{q_w^2 - q_z^2}\right)) \\ + 27 \int_0^t (\psi_d - \tan^{-1}\left(\frac{2q_w q_z}{q_w^2 - q_z^2}\right))dt. \end{aligned} \tag{7.52}$$

Chapter 8

Input Constraints

This chapter analyzes the feasibility of various inputs and the methods used to ensure commanded inputs remain feasible. Section 8.1 explores the range of feasible outputs given the quadrotor's upper and lower motor speed limits. Section 8.2 examines the hierarchy for determining how inputs are limited in the event of an infeasible set of inputs. Section 8.3 quickly discusses the heuristic decision used to limit the thrust in safe ranges. Section 8.4 examines the optimization-based method for limiting the yaw torque in the event that thrust limitation did not produce a feasible set of inputs. Section 8.5 discusses the method for limiting the pitch and roll torques in the event that the inputs still remain infeasible.

8.1 Input Feasibility

Repeated from Chapter 6, the relationship between the quadrotor's commanded motor speeds and its resulting net thrust and torques is shown in Equation 8.1.

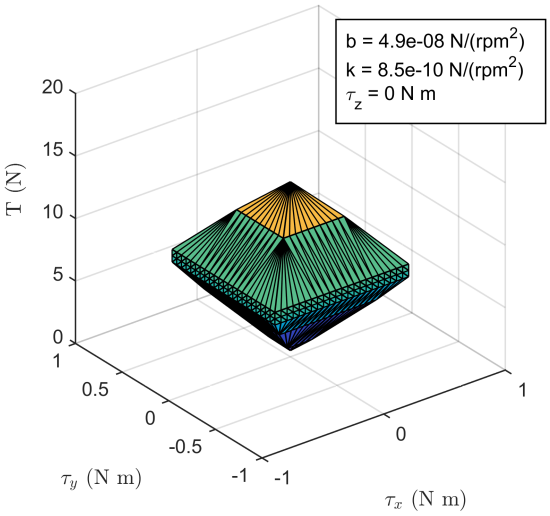
$$\begin{bmatrix} T \\ \tau_{\mathbf{x}} \\ \tau_{\mathbf{y}} \\ \tau_{\mathbf{z}} \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & bl & 0 & -bl \\ -bl & 0 & bl & 0 \\ k & -k & k & -k \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} = \mathbf{M} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix}. \tag{8.1}$$

The Hummingbird quadrotor's motor speeds have upper and lower limits that are characterized in Chapter 11. Putting them into equation form,

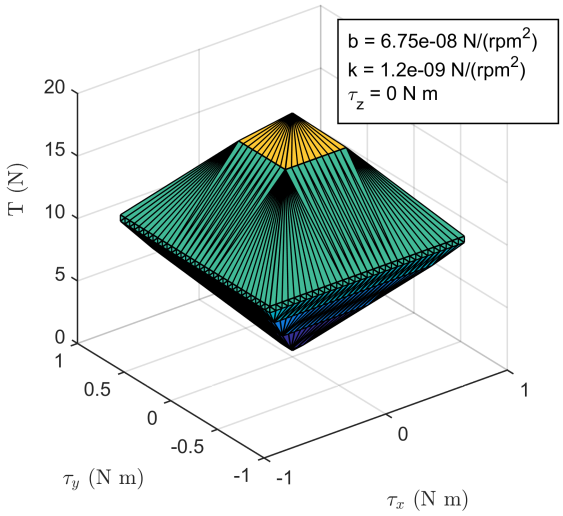
$$\Omega_{\text{min}} \approx 1{,}100 \text{ rpm} \leq \Omega_1, \Omega_2, \Omega_3, \Omega_4 \leq \Omega_{\text{max}} = 8{,}600 \text{ rpm}. \tag{8.2}$$

While the motor speed limitations are fairly clear, the limitations on the thrust and torques are not as clear. Figure 8.1 shows several polytopes that encompass ranges of feasible inputs satisfying Equation 8.2 for varying yaw torques ($\tau_{\mathbf{z}}$) and propeller coefficients.

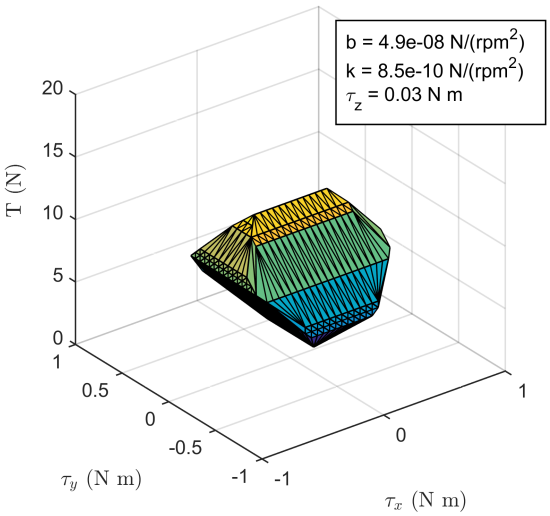
Figures 8.1a and 8.1b show the difference between the flexible and higher performance propellers. The higher performance propellers offer a larger thrust and pitch/roll torque feasible set, which would allow the quadrotor to be more agile. The transition from Figure 8.1a to Figure 8.1c to Figure 8.1d shows the progressive constriction of the feasible set as the yaw torque's magnitude increases. Similar effects can be seen when fixing a different input (like thrust).



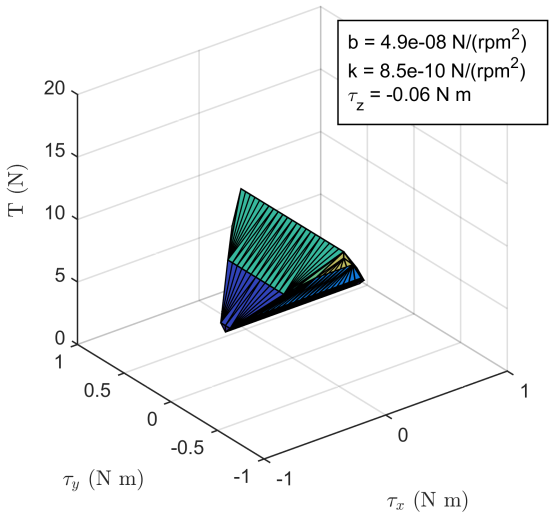
(a) Flexible Propellers, $\tau_\mathbf{z} = 0$



(b) High Performance Propellers, $\tau_\mathbf{z} = 0$



(c) Slightly positive $\tau_\mathbf{z}$



(d) Larger negative $\tau_\mathbf{z}$

Figure 8.1: Polytopes of feasible input combinations for a set $\tau_\mathbf{z}$ value.

8.2 Input Limitation Hierarchy

If any of the controller's commanded inputs falls outside the feasible set, Equation 6.10 will calculate motor speeds that the motor controllers cannot perform. For example, if a large positive yaw torque is commanded, it can potentially send all four calculated motor speeds outside of the feasible range. Simply moving the motor speeds to their nearest feasible value causes problems, as motors 1 and 3 will be at full speed while motors 2 and 4 will be at the minimum speed. Examining Equation 8.1, this results in zero pitch/roll torques and a constant, unchangeable thrust. A process needs to be in place to rein in problematic inputs.

Algorithm 2 shows the process for limiting the inputs in the event the control law generates infeasible inputs. the first parameter to be checked is the thrust, which is heuristically pulled a set distance in from its upper and lower bounds as mentioned in Section 8.3. If the inputs are still infeasible, the yaw torque is set to zero and its feasibility is checked. If the inputs are now feasible, an optimal yaw torque of the same sign is found as described in Section 8.4. If setting the yaw torque to zero didn't fix the problem, the pitch and roll torques are adjusted as mentioned in Section 8.5 until a feasible solution is achieved.

8.3 Thrust Limitation

The thrust plays a large role in controlling altitude and changing position, so it is left with a fairly strong command authority. However, in the event the control law requests infeasibly high or low thrusts, the value is heuristically cut back to be between approximately 150% of the minimum thrust and 85% of the maximum thrust, which is enough to leave room for torque commands. If the input combination is still infeasible, the yaw torque is the next input to be scaled.

Algorithm 2 Input Limitation Process

if $\vec{u}$! = feasible **then**
  if $T > 0.85\,T_{\max}$ **then**
    $T = 0.85\,T_{\max}$
  if $T < 1.5\,T_{\min}$ **then**
    $T = 1.5\,T_{\min}$
  if $\vec{u}$! = feasible **then**
    min/max $\tau_{\mathbf{z}}$ based on sign while preserving other inputs
    if min/max process != feasible **then**
      Calculate $\tau_{\mathbf{x}}$:$\tau_{\mathbf{y}}$ ratio
      min/max $\tau_{\mathbf{x}}$ while preserving T and $\tau_{\mathbf{x}}$:$\tau_{\mathbf{y}}$ ratio
return $\vec{u}$;

8.4 Yaw Torque Limitation

The yaw torque has no particular role in the quadrotor's position control, so it can potentially be scaled back to 0 in an emergency situation. This becomes an optimization problem of minimizing or maximizing (depending on its original sign) the yaw torque $\tau_{\mathbf{z}}$ while preserving the other inputs (T, $\tau_{\mathbf{x}}$, $\tau_{\mathbf{y}}$) subject to constraints on the squares of the motor speeds (represented by $\vec{\Omega}_{sq}$). Putting this into a non-standard optimization form,

$$\max_{\Omega} \text{ or } \min_{\Omega} \quad \tau_{\mathbf{z}} = \vec{c}\,\vec{\Omega}_{sq} = \vec{c} \begin{bmatrix} \Omega_1^2 & \Omega_2^2 & \Omega_3^2 & \Omega_4^2 \end{bmatrix}^T,$$

subject to

$$\mathbf{C}\vec{\Omega}_{sq} = \vec{d}, \quad \Omega_{\min}^2 \leq \Omega_n^2 \leq \Omega_{\max}^2, \quad n = 1, 2, 3, 4,$$

where

$$\vec{c} = \begin{bmatrix} k & -k & k & -k \end{bmatrix},$$
$$\mathbf{C} = \begin{bmatrix} b & b & b & b \\ 0 & bl & 0 & -bl \\ -bl & 0 & bl & 0 \end{bmatrix},$$
$$\vec{d} = \begin{bmatrix} T & \tau_{\mathbf{x}} & \tau_{\mathbf{y}} \end{bmatrix}^T,$$

Manipulating this into standard form with slack variables $\vec{v} \in \mathbb{R}^4$ and $\vec{w} \in \mathbb{R}^4$, the problem becomes one of two optimization problems of the form

$$\min_{\Omega} \quad \tau_{\mathbf{z}} = \begin{bmatrix} \vec{c} & 0_{1x4} & 0_{1x4} \end{bmatrix} \begin{bmatrix} \vec{\Omega}_{sq}^T & \vec{v}^T & \vec{w}^T \end{bmatrix}^T, \tag{8.3}$$

$$\min_{\Omega} -\tau_{\mathbf{z}} = \begin{bmatrix} -\vec{c} & 0_{1x4} & 0_{1x4} \end{bmatrix} \begin{bmatrix} \vec{\Omega}_{sq}^T & \vec{v}^T & \vec{w}^T \end{bmatrix}^T, \tag{8.4}$$

with both problems subject to

$$\begin{bmatrix} \mathbf{C} & 0_{3x4} & 0_{3x4} \\ I_{4x4} & -I_{4x4} & 0_{4x4} \\ I_{4x4} & 0_{4x4} & I_{4x4} \end{bmatrix} \begin{bmatrix} \vec{\Omega}_{sq} \\ \vec{v} \\ \vec{w} \end{bmatrix} = \begin{bmatrix} \vec{d} \\ \Omega_{\min}^2 1_{4x1} \\ \Omega_{\max}^2 1_{4x1} \end{bmatrix}, \tag{8.5}$$

$$\vec{\Omega}_{sq} \geq 0, \quad \vec{v} \geq 0, \quad \vec{w} \geq 0.$$

This creates a set of 12 variables with 11 constraints. The low number of free variables makes it well suited for using the simplex method [53], which optimizes a problem in standard form by creating a feasibility polytope, starting at a vertex (corner), and moving along the edges until the optimal solution is found. While this process is normally computationally cumbersome, the high number of vertices allows the simplex method to effectively converge to the optimal solution in one iteration. If the optimization algorithm of choice cannot find a feasible solution after the thrust has already been limited, the pitch and roll torques need to be limited. Chapter 10, Section 10.9 discusses the C library used to perform the simplex method calculations.

8.5 Roll/Pitch Torque Limitation

In the event the input limitation procedure has reached this state, one or both of the remaining torques ($\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$) is the culprit for generating infeasible motor commands. If both torques are nonzero, rather than scale back the larger of the two, the idea is to attempt to preserve the intent of the controller's torque values (*e.g.*, generate a larger, positive $\tau_{\mathbf{x}}$ and a smaller, negative $\tau_{\mathbf{y}}$) by establishing a ratio between them and maximizing/minimizing one of the two. The thrust retains its equality constraint, while the yaw torque has no constraint. Using β_x and β_y to represent the inverses of the original $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$, respectively, one of the constraints for the optimization problem becomes

$$\beta_x \tau_{\mathbf{x}} = \beta_y \tau_{\mathbf{y}} \qquad \Rightarrow \qquad \beta_x \tau_{\mathbf{x}} - \beta_y \tau_{\mathbf{y}} = 0. \tag{8.6}$$

From there, $\tau_{\mathbf{x}}$ is optimized in the direction associated with its initial sign. Putting this into nonstandard form, the optimization problem becomes

$$\max_{\Omega} \text{ or } \min_{\Omega} \ \tau_{\mathbf{x}} = \vec{c}\,\vec{\Omega}_{sq},$$

subject to

$$\mathbf{C}\vec{\Omega}_{sq} = \vec{d}, \quad \Omega_{\min}^2 \leq \Omega_n^2 \leq \Omega_{\max}^2, \ \ n = 1, 2, 3, 4,$$

where

$$\vec{c} = \begin{bmatrix} 0 & bl & 0 & -bl \end{bmatrix},$$
$$\mathbf{C} = \begin{bmatrix} b & b & b & b \\ \beta_y bl & \beta_x bl & -\beta_y bl & -\beta_x bl \end{bmatrix},$$
$$\vec{d} = \begin{bmatrix} T & 0 \end{bmatrix}^T.$$

In the event that one of the two torques is already set to 0, the problem devolves into minimizing/maximizing the other torque while preserving the first torque's equality with 0. Putting this in nonstandard form,

$$\max_{\Omega} \text{ or } \min_{\Omega} \quad \tau_1 = \vec{c}\,\vec{\Omega}_{sq},$$

subject to

$$\mathbf{C}\vec{\Omega}_{sq} = \vec{d}, \quad \Omega^2_{\min} \leq \Omega^2_n \leq \Omega^2_{\max}, \quad n = 1, 2, 3, 4,$$

where

condition:	$\tau_{\mathbf{x}} = 0,$	$\tau_{\mathbf{y}} = 0,$
$\tau_1 =$	$\tau_{\mathbf{y}},$	$\tau_{\mathbf{x}},$
$\vec{c} =$	$\begin{bmatrix}-bl & 0 & bl & 0\end{bmatrix},$	$\begin{bmatrix}0 & bl & 0 & -bl\end{bmatrix},$
$\mathbf{C} =$	$\begin{bmatrix}b & b & b & b\\ 0 & bl & 0 & -bl\end{bmatrix},$	$\begin{bmatrix}b & b & b & b\\ -bl & 0 & bl & 0\end{bmatrix},$
$\vec{d} =$	$\begin{bmatrix}T & 0\end{bmatrix}^T,$	$\begin{bmatrix}T & 0\end{bmatrix}^T,$

Expanding these problems into standard form as in Section 8.4 yields 12 variables and 10 constraints. The system is sufficiently well constrained that the simplex method converges within one or two iterations and outperforms interior point methods.

Chapter 9

Filtering

This chapter covers the filtering techniques used to process the incoming measurements before using them in the control law. Section 9.1 discusses the principles of the Kalman filter and how it applies to linear systems. Section 9.2 covers the Extended Kalman Filter, how it applies to nonlinear systems, and difficulties with its implementation. Section 9.3 covers the quick filter used in the Extended Kalman Filter's place to bypass its issues with run time. Section 9.4 talks about velocity estimation methods and ultimately settles on a compromise between numerical differentiation and numerical integration. Section 9.5 talks about the state prediction method used to overcome latency within the system.

9.1 Kalman Filter

The Kalman filter is a linear, optimization-based filter that meshes together state predictions and state measurements using their respective covariance values [54]. The prediction covariance matrix $\mathbf{Q}$ and the measurement covariance matrix $\mathbf{R}$ approximate the covariances of the system's predictions and sensor noise statistics,

respectively. Their values govern the behavior of the Kalman filter over time. The estimate covariance matrix $\mathbf{P}_k$ keeps track of filter's confidence in the current states at discrete time step k. The larger the values in $\mathbf{P}_k$, the more the filter relies on measurements; the smaller the values in $\mathbf{P}_k$, the more the filter relies on estimates.

The Kalman filter has two phases: Predict and Update [54]. The Predict phase starts by predicting how the system will evolve using the discrete linear equation found in Equation 9.1, followed by a prediction of how the system's estimate covariance evolves using Equation 9.2. The vector $\hat{x}_{k-1|k-1}$ is the previous state estimate, $\hat{x}_{k|k-1}$ is the predicted state estimate, and $\vec{u}_{k-1}$ is the previous inputs sent to the system. $\mathbf{P}_{k-1|k-1}$ denotes the estimate covariance matrix from the previous iteration, while $\mathbf{P}_{k|k-1}$ denotes the predicted evolution of the estimate covariance.

$$\hat{x}_{k|k-1} = \mathbf{A}_d\hat{x}_{k-1|k-1} + \mathbf{B}_d\vec{u}_{k-1}, \tag{9.1}$$

$$\mathbf{P}_{k|k-1} = \mathbf{A}_d\mathbf{P}_{k-1|k-1}\mathbf{A}_d + \mathbf{Q}. \tag{9.2}$$

During the Update phase, the difference is found between the current measured state $\tilde{z}_k$ and the current predicted state estimate $\hat{x}_{k|k-1}$. Note that $\tilde{z}_k$ can have less states than $\vec{x}$. The matrix $\mathbf{H}$ handles relating the states in $\vec{x}$ to the states in $\tilde{z}_k$. The difference between the two is stored in the residual vector $\vec{v}$. A residual covariance matrix $\mathbf{S}_k$ is calculated, followed by a Kalman gain matrix $\mathbf{K}_k$. The current predicted state $\hat{x}_{k|k}$ is updated using the Kalman gain matrix and the residual vector, while the estimate covariance $\mathbf{P}_{k|k}$ is updated through application of the Kalman gain matrix. Equations 9.3 through 9.7 show this process in equation form [54].

$$\vec{v} = \tilde{z}_k - \mathbf{H}\hat{x}_{k|k-1}, \tag{9.3}$$

$$\mathbf{S}_k = \mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R}, \tag{9.4}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^T\mathbf{S}_k^{-1}, \tag{9.5}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + \mathbf{K}_k\vec{v}, \tag{9.6}$$

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}. \tag{9.7}$$

As the Kalman filtering process relies on knowing the linear $\mathbf{A}_d$ matrix for the system, and as nonlinear systems don't have set $\mathbf{A}_d$ matrices through their very nature, the Kalman filter doesn't really work. While a linear estimate could be generated around a fixed point, deviations in the system's dynamics from this point would cause horrible problems and potentially cause the filter to become unstable.

9.2 Extended Kalman Filter

To circumvent the Kalman filter's dependence on linear systems, the Extended Kalman Filter makes two modifications to the Prediction phase [54]: the state prediction is relaxed to be a nonlinear discrete time equation as in Equation 9.8, and a new $\mathbf{A}_d$ matrix is generated at each time step by linearizing the system [52], followed by discretizing it [51]. The Update phase is not changed in any way.

The discrete nonlinear prediction can quickly be performed via Euler approximation: calculate the state derivatives using the system's continuous time dynamics, multiply them by the discrete time step size, and add them to the state values. The linearization process involves taking the continuous time dynamics of the system and calculating their Jacobian around the previous estimated states as in Equation 9.9. With the continuous time $\mathbf{A}_c$ matrix, the discrete time equivalent $\mathbf{A}_d$ can be calculated via a matrix exponential operation shown in Equation 9.10.

$$\hat{x}_{k|k-1} = f_d(\hat{x}_{k-1|k-1}, \vec{u}_{k-1}), \tag{9.8}$$

$$\mathbf{A}_c = \mathcal{J}(\, f_c(\vec{x}, \vec{u})\,)|_{\hat{x}_{k-1|k-1}\,,\,\vec{u}_{k-1}}, \tag{9.9}$$

$$\mathbf{A}_d = e^{T_s \mathbf{A}_c} = \sum_{i=0}^{\infty} \frac{1}{i!}(T_s \mathbf{A}_c)^i, \tag{9.10}$$

Unfortunately, the matrix exponential calculation process was found to take too long in practice. MATLAB was used to generate a C library for performing matrix exponential calculations using its C Coder utility. While it gave the right answer, it couldn't run fast enough on the Edison. As a workaround, an approximation of Equation 9.10 using only 30 iterations and OpenCV was attempted as in Equation 9.12, but it still ran way too slow. The Extended Kalman Filter was eventually deemed impractical, and alternative solutions were examined.

$$f_d(\hat{x}_{k-1|k-1}, \vec{u}_{k-1}) \approx \hat{x}_{k-1|k-1} + T_s f_c(\hat{x}_{k-1|k-1}, \vec{u}_{k-1}), \tag{9.11}$$

$$e^{T_s \mathbf{A}_c} \approx \sum_{i=0}^{30} \frac{1}{i!} (T_s \mathbf{A}_c)^i. \tag{9.12}$$

9.3 Quick Filter

Members from ETH Zurich proposed a quick, Kalman-esque filter for meshing together measurements and estimates [16]. The proposed filter used a fixed, diagonal tuning factor matrix $\mathbf{K}$ with entries between 0 and 1 to mesh the values together as in Equation 9.13. The entries effectively choose the fraction of the predicted state to use, while the remaining fraction uses the measured state.

$$\begin{aligned} \hat{x}_{k|k-1} &= f_d(\hat{x}_{k-1|k-1}, \vec{u}_{k-1}), \\ \hat{x}_{k|k} &= \mathbf{K}\hat{x}_{k|k-1} + (\mathbf{I} - \mathbf{K})\tilde{x}_k. \end{aligned} \tag{9.13}$$

This algorithm has the benefit of not calculating estimate covariance matrices, matrix inverses, needing to perform matrix exponential calculations, or needing to know statistical information about the quadrotor's behaviors. On the downside, the only practical way to adjust the tuning factor matrix is through trial and error. As another downside, the meshing process requires a measured velocity, and there are no means within the system to directly measure the quadrotor's translational velocity. A means of estimating the quadrotor's velocity will need to be developed.

9.4 Velocity Estimation

Multiple methods exist for numerically calculating a derivative, such as:

$$\text{Two-Point Derivative:} \qquad \dot{\tilde{r}}_k = \frac{\tilde{r}_k - \tilde{r}_{k-1}}{T_S}.$$

$$\text{Three-Point Derivative:} \qquad \dot{\tilde{r}}_k = \frac{\tilde{r}_{k+1} - \tilde{r}_{k-1}}{2T_S}.$$

$$\text{Al-Alaoui Derivative [1, 55]:} \quad \dot{\tilde{r}}_k = -\frac{1}{7}\dot{\tilde{r}}_{k-1} + \frac{8(\tilde{r}_k - \tilde{r}_{k-1})}{7T_S}.$$

The two-point derivative works with a present and past value, but any noise in the position values gets amplified in the velocity. The noise amplification becomes especially problematic as the time step size gets smaller. The three-point derivative isn't as sensitive to immediate changes between points, but it relies upon knowing a future measurement value, which isn't feasible in real time applications. The Al-Alaoui derivative appears to be a modified version of the two-point derivative, but it only exacerbates the noise problem from the two-point derivative.

Keeping the idea in mind of meshing measurements together, not only are the quadrotor's position values available, but as are the measured accelerations. While numerical differentiation is noisy, numerical integration steadily accumulates measurement errors and drifts over time. By using the two together, a solution can be found with less noise and drift than either method alone.

Equation 9.14 presents an original method for numerically calculating an object's velocity using position and acceleration measurements. The first part of the equation takes the measured acceleration, integrates it over a time step, and adds it to the previous velocity estimate. The second part of the equation performs a two-point derivative using the past and present position measurements. A tune-able meshing parameter α combines the two together.

$$\dot{\tilde{r}}_k = \alpha(\dot{\hat{r}}_{k-1} + T_S\ddot{\tilde{r}}_k) + (1-\alpha)\frac{\tilde{r}_k - \tilde{r}_{k-1}}{T_S}. \tag{9.14}$$

Figure 9.1 shows a plot of the various derivative algorithms applied to raw IMU and Vicon measurements. The two-point derivative and Al-Alaoui derivative were very noisy and spiked very rapidly, with the Al-Alaoui derivative creating slightly larger spikes. The three-point derivative resulted in smoother derivatives, though it still makes use of future, non-causal points for its calculation scheme. The custom derivative estimation algorithm presented in Equation 9.14 remained causal and wasn't anywhere near as susceptible to quick spikes when using $\alpha = 0.65$. As a tradeoff, it appeared to have a mild amount of latency at some points. Still, the smoother velocity profiles result in less input fluctuation, so it is a worthwhile tradeoff.

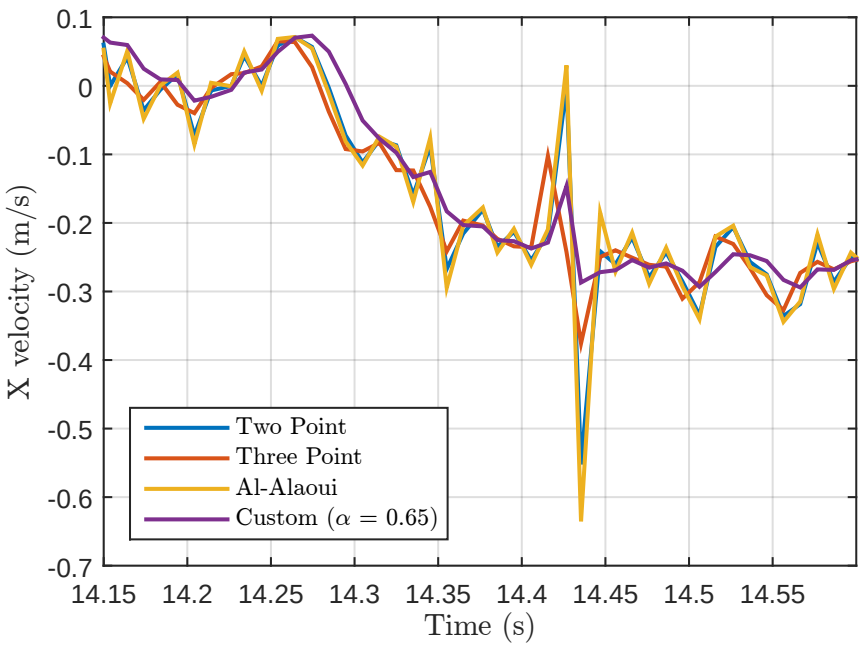


Figure 9.1: Plot of the various velocity estimation algorithms applied to raw IMU/Vicon sensor data.

In practice, the measurements from Vicon would occasionally drop out for a short period of time. This resulted in the same position measurement appearing at both the current and the last time step, resulting in a numerical differentiation value of 0 m/s. Over several iterations, this kills the quadrotor's velocity estimate. To avoid this problem, α was set to 1 for cases where the Vicon measurement has not changed from the previous iteration.

The Vicon dropouts also caused problems when the measurements came back. After holding the same measurement value for several hundred milliseconds, the quadrotor's estimation process works as well as it can, but it drifts over time. As such, the quadrotor drifts as well. When the measurements resume, a large discontinuity in position is created. If the differentiation process immediately starts using this new value with the old one, a large spike in velocity is created. To prevent this from being an issue, α is left at 1 for the first three position measurements after undergoing a lengthy Vicon dropout period.

9.5 Latency Compensation

The Flying Machine Arena paper [16] also proposes a method for compensating for latency. After characterizing a system and finding a latency of N discrete time steps between measurements and input execution, state-based prediction of the form presented in Equation 9.8 is used to predict where the quadrotor will be. Once the measured values have been filtered to provide $\hat{x}_k$, the prediction process is run for N iterations to generate a "future" set of states $\hat{x}_{k+N}$. The control law is applied to the "future" states, and the inputs are sent to the quadrotor. After the system latency, the inputs will arrive and be ready for execution when the quadrotor is at the "future" state $\hat{x}_{k+N}$. Algorithm 3 shows the process used to perform the latency compensation process.

Algorithm 3 Latency Compensation

Use filter to obtain lagged state $\hat{x}_k$.

Predict evolution of system as below:

for i = 1 to N **do**

$\hat{x}_{k+i} = f_d(\hat{x}_{k+i-1}, \vec{u}_{k+i-1})$;

Perform control law on $\hat{x}_{k+N}$.

Chapter 10

Control Program Implementation

This chapter discusses the program designed to run on the Intel Edison for executing all of the desired operations. Section 10.1 provides information about obtaining the source code used in this thesis. Section 10.2 covers the various flight modes used to set trajectories, control laws, and state estimation parameters. Section 10.3 discusses the general program structure. Sections 10.4, 10.5, and 10.6 cover the program implementations used to communicate with the Autopilot, the GCS, and the Vicon server, respectively. Section 10.7 talks about the state estimation procedures. Section 10.8 talks about the control law implementation. Section 10.9 discusses the optimization library used for finding optimal feasible inputs. Section 10.10 quickly talks about the simple trajectory generation scheme. Section 10.11 discusses the multithreaded class used for logging data.

10.1 Source Code

The source code for the GCS, the Edison control program, and the Autopilot's HLP are all available on GitHub. The GCS and Edison source code is currently in pri-

vate repositories, but it will be migrated to the MARHES GitHub account. In the meantime, send an email to `wjneeley@unm.edu` to request access to the code or to receive the updated code location. The HLP code is in a public repository run by Paul Groves. The code can be found at `https://github.com/PaulGrovesAtUNM/QuadrotorHLCode`.

10.2 Flight Modes

The logic behind the quadrotor control program is largely structured around the quadrotor's discrete flight mode. Before discussing how flight modes affect each aspect of the program, a general overview of the different modes should be provided. Table 10.1 provides a comprehensive list of the available flight modes and their purposes. Additional flight modes can be added as needed.

The flight modes can largely be grouped into three sets: General, In-Air, and Test. The General modes are used for implementing general logic related to motor speeds, such as shutting them off or setting them to idle. The In-Air modes are focused on the quadrotor actually flying in the air. What makes one mode different from another is largely the method used to generate each mode's trajectory. The Test modes are used for evaluating various parameters of the quadrotor, whether for checking motor controller configuration/health, trying simple control loops, or sending direct motor commands to the motor controllers.

Most of the flight modes are designed to be selected by the GCS through the custom program, though logic is implemented to prevent unsafe transitions (*i.e.*, trying to hover before taking off, or trying to take off before the propellers have been turned on). See the source code for a thorough breakdown of how the mode logic is implemented. An Excel spreadsheet (included with the source code) details the transitions allowed by the GCS.

Table 10.1: Table of Flight Modes

Name	Type	Description
Off	General	Shuts the quadrotor's motors off. Can be set to this state regardless of the current flight mode.
Idle	General	Idles the motors at the minimum speed possible. Used prior to entering a takeoff procedure or performing a test mode.
Ramp Up	General	Ramps up the motors from idle speed to approximately half speed. Used prior to takeoff.
Takeoff	Flight	Commands the quadrotor to ascend straight up from its location on the ground.
Hover	Flight	Commands the quadrotor to hover in one location.
Waypoint	Flight	Commands the quadrotor to fly to a waypoint. Designed to work in tandem with the GCS for flying to custom waypoints; currently only flies to a hard-coded waypoint.
Velocity	Flight	Intended for doing velocity-based control. Never implemented, but left in the code.
Figure 8	Flight	Commands the quadrotor to fly in a diagonal figure 8 pattern. See Chapter 12 for an example.
Special	Flight	Used to implement special flight patterns for application-specific purposes.
Reference	Flight	Intended to allow the GCS to send reference trajectories to the quadrotor. Not implemented.
Head Home	Flight	Commands the quadrotor to hover above the point where it first took off.
Land	Flight	Commands the quadrotor to land directly below where it is.
Unsafe	Flight	Triggers when an unsafe flight condition occurs (goes past a hard-coded boundary or exceeds a hard-coded Euler angle).
Cycle Motors	Test	Cycles each of the motors in turn from idle to a medium speed, then cycles them in turn down to idle. Good for ensuring motors are working correctly. Quadrotor should be secured.
Latency Test	Test	Used for evaluating the latency of the system. Motors will be off. Not implemented.
DMC Test	Test	Allows direct motor control via motor command values. Quadrotor should be secured.
Attitude Control	Test	Activates a prototype attitude/orientation control loop. Quadrotor should be suspended for this test.
Position Control	Test	Adds a prototype position controller to the Attitude Control test. Quadrotor should be suspended for this test.
Height Control	Test	Adds a prototype altitude controller to the Position Control test. Quadrotor should be suspended for this test.

10.3 Program Structure

As the Vicon Datastream SDK provided a library for use in C++, the program run on the Edison was programmed using C++. Libraries like the Robot Operating System (ROS) [56] were not used due to conflicting library versions when using the Vicon Datastream SDK (discussed later). Classes were created to handle specific tasks whenever possible for clarity. When necessary, tasks were pushed to separate threads, and classes were created to handle interacting with the threads. Multithreading was performed using POSIX threads [57]. A single set of files, called "defs.cpp" and "defs.h", were largely used to define all of the necessary constants (port names, IP addresses, array sizes, latency compensation time steps, etc.) used across all aspects of the program. Figure 10.1 shows a block diagram of the general program structure.

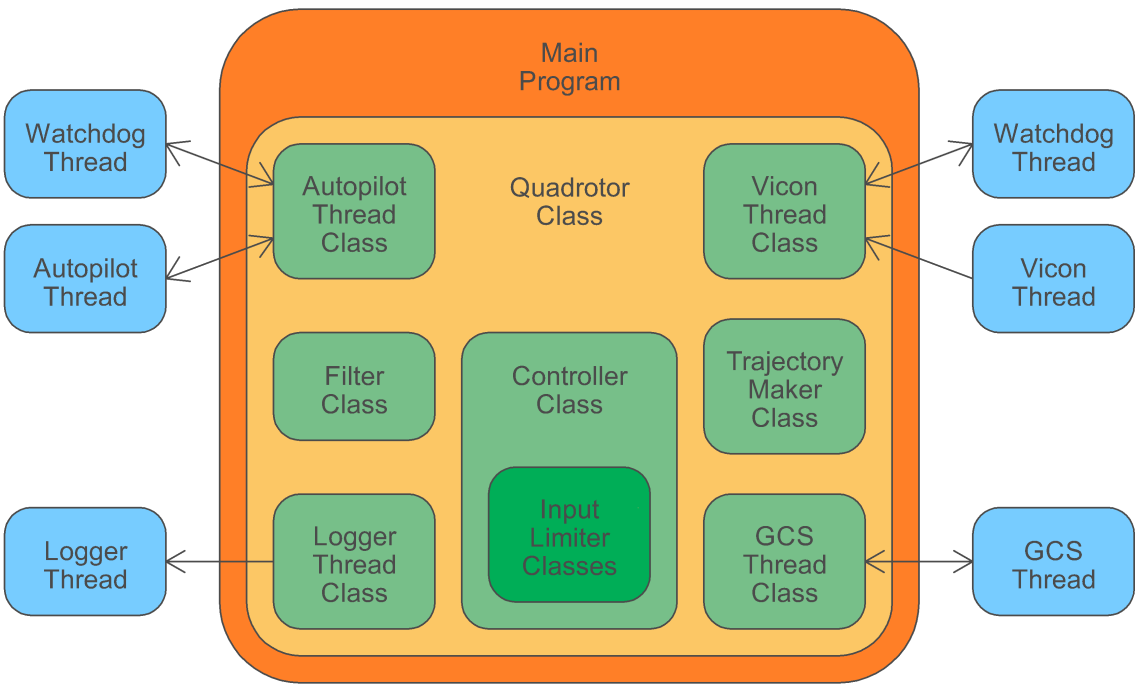


Figure 10.1: Block diagram of the control program's structure.

A Quadrotor class object was created to contain all of the necessary data structures, classes, and function calls to perform various operations (such as obtaining measurements, filtering data, generating control laws, handling flight mode control, sending motor commands, and communicating with the GCS). The Quadrotor class was also designed to load the parameters for a given quadrotor (found in "parameters.cpp") based on an index number passed into the class when the class is created. A main program was written to create the Quadrotor class, pass in an appropriate index number, and handle the timing associated with calling each of the Quadrotor class's functions. The main program was designed to try to run the filter and control loop at 200 Hz, send GCS updates at 10 Hz, and process measurements as they are received. The following sections describe each of the classes within the Quadrotor class, any applicable subclasses, and any applicable threads they managed.

10.4 Autopilot Communication

The Autopilot Thread class was programmed to handle communication with the Autopilot's HLP. This involved configuring the serial port for transmitting and receiving at the same time, sending structured data frames to the HLP, and creating a dedicated thread for processing incoming data frames from the HLP. The separate thread prevents read operations on the serial port from blocking (stopping) the whole program until data is read from the port. See Chapter 4 for detailed information about the configurations and data frames used in this class.

A second thread was created as a sort of watchdog thread. The watchdog thread periodically checks to see if the first thread has read anything recently from the serial port. If a certain amount of time passes without receiving anything, the watchdog thread sets a Boolean variable indicating the port is inactive. Once new data is received, the variable is cleared and the watchdog's timer is reset.

10.5 Vicon Communication

The Vicon Thread class was programmed to handle connecting to the Vicon server at its static IP address over Wi-Fi, receiving frames of data when available, and extracting measurement information out of each frame. The class makes use of the Vicon Datastream SDK and its associated C++ Linux x86 library for these operations. The task of checking for measurement updates was placed onto a dedicated thread so the read operations didn't block the main program. A secondary watchdog thread was created to warn whenever measurements had not been received recently, just like in the Autopilot Thread class.

As a note, the Vicon Datastream SDK is available in three main programming languages: C++, MATLAB (great for troubleshooting and visualizing measurements), and .NET (meant for use with LabVIEW). For the C++ libraries, four versions are available: Windows x86, Windows x64, Linux x86, and Linux x64 (added in v1.2). The addition of Linux x64 libraries in v1.2 made program development and testing on 64-bit computers a lot easier (32-bit compilation on 64-bit machines is a pain), so versions older than v1.2 were excluded from use.

The C++ Datastream SDK makes use of Boost libraries for its operations. The Boost libraries in v1.2 and older were found to conflict with ROS Groovy, as the Datastream SDK's libraries were out of date. These older versions are compatible with Vicon Tracker v1.0 through v1.3. In Datastream SDK v1.3 and onward, Vicon changed their library implementation. As the current Vicon server in the MARHES lab makes use of Vicon Tracker v1.3, and as the Datastream SDK's backwards compatibility was not verified for v1.3 onward, the C++ Datastream SDK v1.2 was used for this thesis. As making ROS work with the outdated Datastream SDK would have required extensive workarounds, ROS was effectively infeasible.

10.6 GCS Communication

The Quad Client Thread class was created to handle communicating with the GCS program at a static IP address over an IP socket [58]. A matching class, called GCS Server Thread, was created in C++ for incorporation into the Qt-based GCS program to handle communicating with the Quad Client Thread class. The GCS Server Thread class created a dedicated thread for waiting for connections, receiving incoming data, and sending information to the quadrotor. The Quad Client Thread class was designed to connect to the server, push operations for receiving data onto a separate thread, and periodically send information to the server.

Predefined C-based structs were created for structuring data being sent in each direction. One struct was created for sending information from the quadrotor to the GCS, while another struct was created for sending information from the GCS to the quadrotor. The structs were designed to hold fixed point, fixed byte integer numbers. Integers longer than one byte were converted from host ordering to network ordering [58] before being transmitted and were converted back after being received. This avoided issues with byte ordering used in different architectures (Big Endian vs. Little Endian).

At transmit time, the appropriate struct was populated and passed to the transmission function as raw binary using a pointer. When receiving, a pointer to the receiving struct was given to the receiving function, the data was received as raw binary, and information was unpackaged from the struct. By using the TCP protocol instead of UDP, data was guaranteed to be delivered and in order, so this method of transmitting data back and forth had no issues with potential data loss. Table 10.2 lists the data sent from the quadrotor the GCS, while Table 10.3 lists the data sent from the GCS to the quadrotor.

Table 10.2: Quadrotor to GCS Frame Contents

Parameter	Bytes/ea	Values	Description
States	4	13	Current quadrotor states $\times 10^6$.
Reference	4	13	Reference quadrotor state $\times 10^6$.
Inputs	4	4	Quadrotor thrust/torque inputs $\times 10^6$.
Vicon	4	7	Raw measurements from Vicon $\times 10^6$.
IMU	4	6	Raw measurements from the Autopilot's IMU $\times 10^6$.
Motors	1	4	Motor speed commands sent to the quadrotor.
AP Status	2	1	Raw binary representation of the Autopilot's LLP status variable [23].
Battery	4	1	Autopilot's measured battery voltage $\times$ 1,000.
CPU	4	1	HLP CPU load percentage $\times$ 1,000.
UART	1	1	Status of UART connection.
Vic Status	1	1	Status of Vicon connection.
Mode	1	1	Current flight mode on the quadrotor.
Total	**189**		

Table 10.3: GCS to Quadrotor Frame Contents

Parameter	Bytes/ea	Values	Description
Trajectory	4	18	Desired flight trajectory values $\times 10^6$.
Reference	4	13	Desired reference quadrotor state $\times 10^6$.
Motors	1	4	Desired motor speed commands.
Mode	1	1	Desired flight mode on the quadrotor.
Total	**129**		

10.7 Filter Calculation

A dedicated Filter class was created for processing the incoming measurements. Legacy code related to implementing the Extended Kalman Filter remains in the class, which creates a dependence on using OpenCV [59] for matrix calculations. Still, the Filter class's main utility in the Quadrotor class is for implementing the quick filter described in Section 9.3. This includes performing the derivative calculations discussed in Section 9.4. The latency compensation process was largely left as a separate function call in the Quadrotor class, though it makes use of the Filter class's nonlinear state prediction function. The latency compensation functionality could easily be shifted into the Filter class.

The Filter class's state prediction method works differently depending on the quadrotor's selected flight mode. If the quadrotor is in a flight mode where it is expected to be on the ground (Off, Idle, or Ramp Up), the prediction algorithm assumes the quadrotor won't move, which means predicting the position/orientation values won't change and the velocities/angular velocities will be equal to 0. Otherwise, the state prediction uses the dynamics covered in Chapter 6 and Euler approximation to predict the quadrotor's flight evolution over time. As mentioned in Section 9.4, special methods are also used to defend against discontinuities in Vicon measurements.

The quick filter's tuning matrix $\mathbf{K}$ was configured such that the diagonal entries associated with position and quaternion states had values of 0.25, while those associated with velocities and angular velocities had values of 0.50. As for latency compensation, $N = 4$ time steps of latency compensation were used. These values were selected through trial and error. Further testing may yield values that provide better results.

10.8 Control Law Calculation

A generic Controller class was created for implementing various control law schemes. Legacy functions associated with performing PD-based cascading position and attitude control loops are left in the class. The nonlinear feedback linearization controller was tied into the class as well. The feedback linearization-related function calls were coded as a separate set of files using as much C-based syntax as possible. The heavy C syntax should make the code easier to move to the Autopilot's HLP should future control architecture iterations choose to do so.

Depending on the flight mode, the Controller class calculates inputs based on the system's filtered states. All of the In-Air flight modes use the nonlinear controller, while some Test modes make use of the PD-based cascading controllers. For flight modes that don't use controls, the Controller class is not called.

10.9 Input Limitation

Two classes were created for performing optimization problems: Yaw Optimizer for performing the yaw optimization calculation, and Pitch Roll Limit for limiting the pitch and roll. Section 10.9.1 discusses the library used for the optimization algorithms, and Section 10.9.2 shows the run times for each of the optimization problems.

10.9.1 GNU Linear Programming Kit

The GNU Linear Programming Kit (GLPK) is an open source, ANSI C-based library for performing linear programming and mixed integer programming algorithms [60]. The two main linear programming methods at its disposal are the simplex method

and an interior point method. More information on these methods and how they work can be found at [53, 60]. Of note is that both algorithms have configurable parameters for attempting to speed up the optimization process. The simplex method can be calculated in either its primal or dual form, while the interior point method provides several ordering techniques. Problems can be generated using nonstandard formulations as well as standard formulations.

10.9.2 Execution Time

A test procedure was created to determine which optimization process ran fastest for each problem. Algorithm 4 lists the test procedure for a given optimization process. Both the simplex method and the interior point method were evaluated. Standard and nonstandard problem formulations were evaluated. For the simplex method, both primal and dual methods were evaluated. For the interior points method, all of the ordering methods were tested. All timing tests were done on the Intel Edison.

Algorithm 4 Optimization Algorithm Timing Test

```
Create optimization model
for T = 0; T ≤ 22; T += 1 do
    for τ_x = -1.5; τ_x ≤ 1.5; τ_x += 0.1 do
        for τ_y = -1.5; τ_y ≤ 1.5; τ_y += 0.1 do
            Update the problem with new objective values
            Run method for maximization
            Record iterations, run time, and solution
            Run method for minimization
            Record iterations, run time, and solution
Analyze results
```

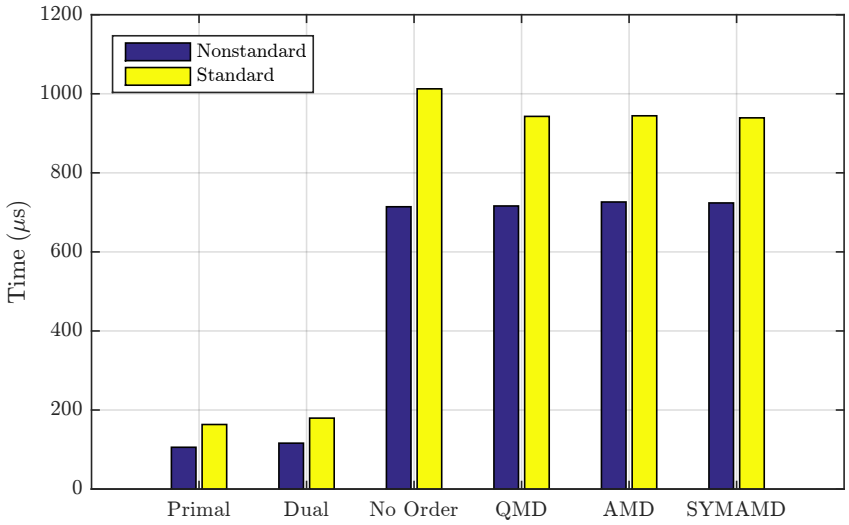


Figure 10.2: Mean yaw optimization algorithm run times on the Intel Edison for various optimization algorithms in both standard and nonstandard form. Only simulations that resulted in feasible solutions are represented.

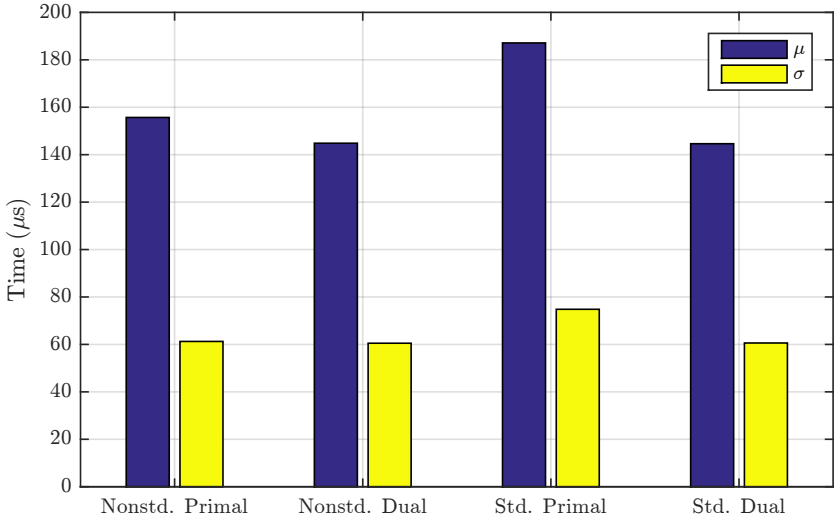


Figure 10.3: Mean and standard deviations of pitch/roll optimization run times on the Intel Edison for various simplex algorithm implementations in both standard and nonstandard form. Only simulations that resulted in feasible solutions are represented.

Figure 10.2 shows a bar graph of the Yaw Optimizer's mean run times for each algorithm when there was a feasible solution. The interior point method took significantly longer than the simplex method, regardless of being in standard/nonstandard form or the ordering method used. For the simplex method, nonstandard problem formulations took less time. While Figure 10.2 doesn't show it very well, the nonstandard primal simplex method converged to a solution in an average of 106 μs. The nonstandard dual simplex method converged to a solution in 116 μs on average. While the difference is not substantial, the primal simplex performed slightly better. The simplex method always converged in one iteration.

Figure 10.3 shows a bar graph of the Pitch Roll Limiter's mean run times for the simplex method when there was a feasible solution. The standar deviations are included as well. The interior point methods kept crashing at run time, so data is not available for their performance. The dual simplex ended up outperforming the primal simplex, regardless of the standard/nonstandard problem formulation. Both the standard and nonstandard dual simplex methods converged in an average time of 145 μs. The simplex method always converged within one iteration.

10.10 Trajectory Generation

The Trajectory Maker class was created to handle generating trajectories based on the currently selected flight mode. All of the trajectories were formulated based on either maintaining a constant position/heading or using a sinusoidal sweep from one point to another. This generated computationally simple trajectories that were easily differentiable down to their fourth derivative. The figure 8 trajectory is the only exception, as it continuously kept executing a sinusoidal trajectory. While the sinusoids were constructed to provide continuous position reference, no effort was made for velocity, acceleration, jerk, or snap continuity.

10.11 Logging Thread

The Logger Thread class was programmed to handle writing lines of log information to a Comma Separated Value (CSV) file. Data was accumulated in a string until a full line was obtained, at which point the string was passed to a thread. The thread handled grabbing the string from the main process and saving it to a log file. The multithreaded nature of this process stemmed from the OS timing issues caused by file writes. Before the multithreaded version was implemented, the control program would periodically see loop times jump from 5 ms to 50 ms. The delays sometimes rose as high as 200 ms, which caused problems for the control loop. By implementing the logging with multithreading, the problem vanished.

Chapter 11

AscTec Hummingbird Physical Model Evaluation

As the feedback linearization process effectively involves inverting the mechanics of the quadrotor to create artificial linear inputs, mismatches between the expected quadrotor model and the actual quadrotor model can produce poor linearization performance, which leads to poor controller performance. This chapter looks into measuring the physical parameters of the quadrotor rather than relying upon ideal specifications. Section 11.1 examines the expected motor speeds for various commands and compares them to the measured motor speeds. Section 11.2 examines the step response of the motor's speed when rapidly increasing and decreasing the motor command value. Section 11.3 looks into measuring the propellers' thrust and drag coefficients to obtain a more accurate average number. Section 11.4 uses the results from the previous three sections to attempt to measure the rotational inertias of the quadrotor with the attached protective frame and Intel Edison Quadrotor Block.

11.1 Motor Commands vs. Motor Speeds

This section examines the process for generating desired motor speeds using motor command values. The AscTec Autopilot's motor speed command system consists of sending a direct motor command value between 0 and 200 to each motor. 0 turns a motor off, and 1 through 200 scale linearly from the lowest speed to the highest speed. This is the extent of the documentation presented on the AscTec Wiki [24], so other sources are needed for more information.

Examining the source code [61] provided in the AscTec SDK, the file "sdk.h" provides the relationship between a direct motor command value DMC_i and a motor's rotation speed ω_i (rpm) as in Equation 11.1. Equation 11.2 provides the corresponding conversion factor from a given motor speed ω_i (rpm) to the desired motor command character DMC_i. Extrapolating from Equation 11.1, the minimum motor speed is ∼1,100 rpm, while the maximum motor speed is 8,600 rpm.

$$\omega_i = (25 + (DMC_i * 175/200)) * 43, \tag{11.1}$$

$$DMC_i = ((\omega_i/43) - 25) * 200/175. \tag{11.2}$$

To evaluate the accuracy of Equation 11.1, a set of data where the quadrotor's motors were set to constant DMC values for an extended period of time (i.e. data collected for the thrust test done in Section 11.3.1) was analyzed and the measured motor speeds were processed. Data for each motor was processed as follows:

1. Separate the data into portions where the DMC was held at a constant value.
2. Remove the first 0.8 seconds of data from each portion to eliminate any transient effects in motor speed after a DMC change. If the portion of data is shorter than 0.8 seconds, eliminate it from the data set.
3. Calculate the mean and standard deviation of the measured motor speed for each portion and plot the results.

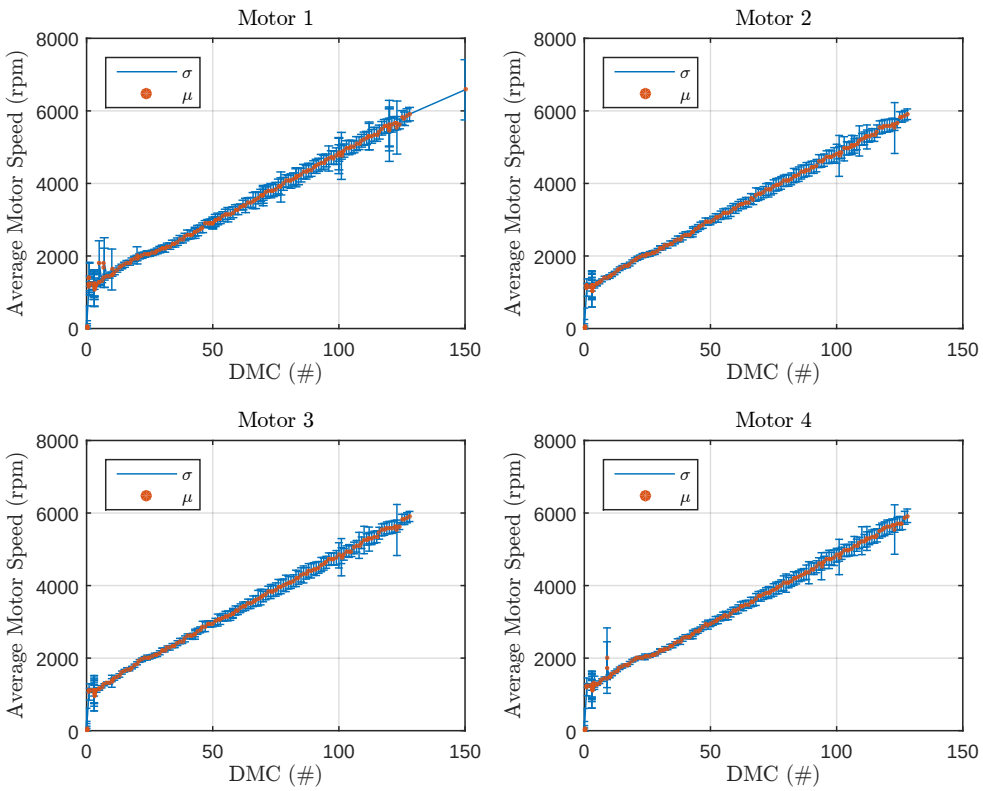


Figure 11.1: Plots of mean measured motor speeds with standard deviation brackets for each motor.

Figure 11.1 shows plots for each motor of the mean and standard deviation in the measured motor speeds for a given DMC value. Examining the reults for each motor, all four motors appear to be predominantly linear. A minor nonlinearity appears near a DMC value of 25, though it largely remains linear on either side of this area. Some minor deviations appear near the lower DMC values, but these outlying points are residual transient effects from the process used to set the DMCs (through the GCS software) and aren't a problem. The standard deviation gradually increased as the motor speeds increased with periodic spikes in size (likely from a low amount of data for that value).

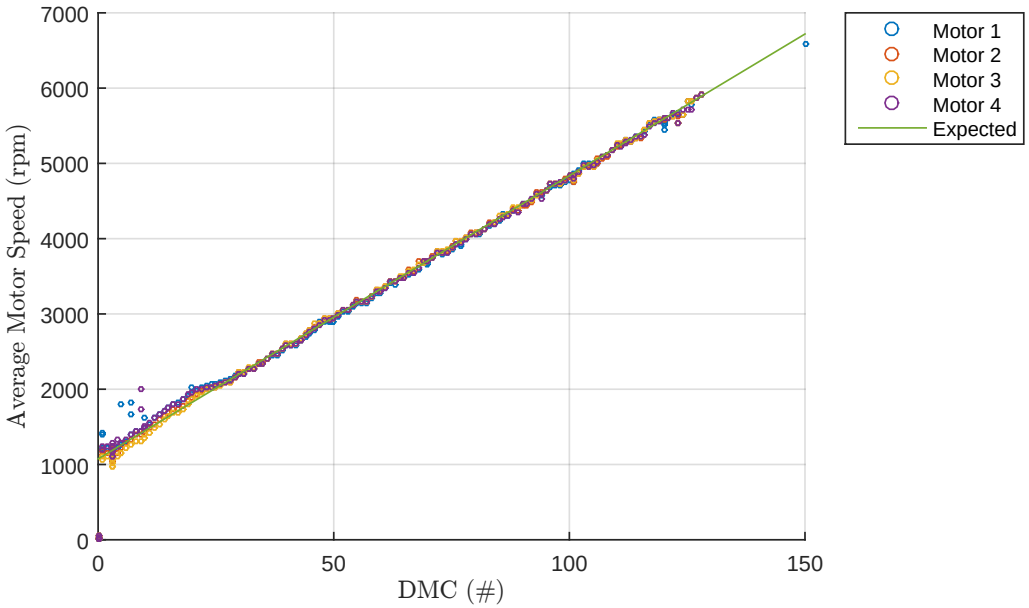
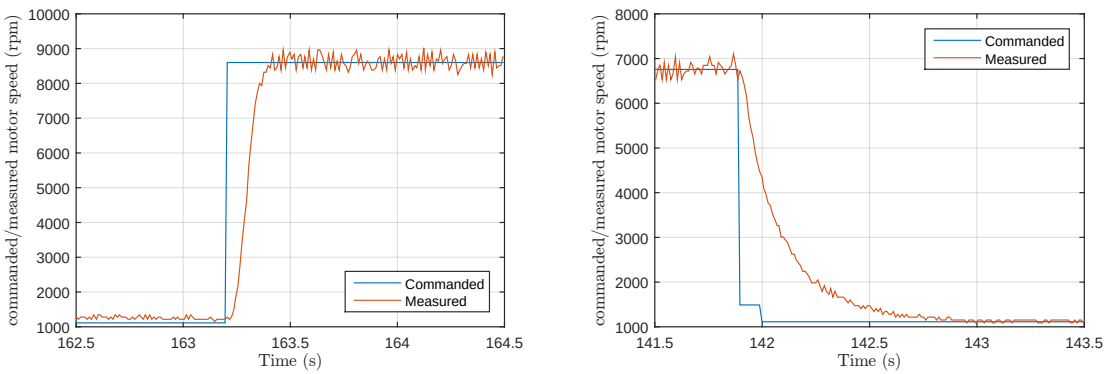


Figure 11.2: Plot of average measured motor speed vs. expected motor speed for DMC values.

Figure 11.2 groups the means for all four motors into one plot and plots Equation 11.1 on top. Once the DMC values is higher than the nonlinearity point at $\sim$25, the means for all of the measured motor speeds perfectly follow the expected equation line. The measured motor speeds for $DMC < 25$ were slightly higher than expected, but they were still largely clustered around the expected equation line. Based on the close match-up seen in Figure 11.2, Equation 11.1 is deemed to be sufficiently accurate for flying the quadrotor, and Equation 11.2 will be used to convert a desired motor speed to a direct motor command value.

11.2 Motor Step Response Plots

This section quickly examines the motor's response time to a commanded motor speed value. Figure 11.3a shows the motor speed response time for a sudden DMC jump from 1 to 200, while Figure 11.3b shows the motor speed response time for a sudden DMC jump from 150 to 1. When increasing the commanded motor speed, the settling time is quite short ($\sim$ 200 ms), even when transitioning from the lowest possible value to the highest possible value. When decreasing the commanded motor speed, the settling time is much longer ($\sim$ 1 s). The results in Figure 11.3 seem to indicate that commands involving motor speed increases don't cause any latency issues, but commands involving slowing the motors may take a while, so aggressive maneuvers downwards may be difficult to execute. Otherwise, given the relatively short response time from such large speed command changes, the rotational inertia effects of having to accelerate the propellers will largely be considered insignificant for state estimation and controls purposes.

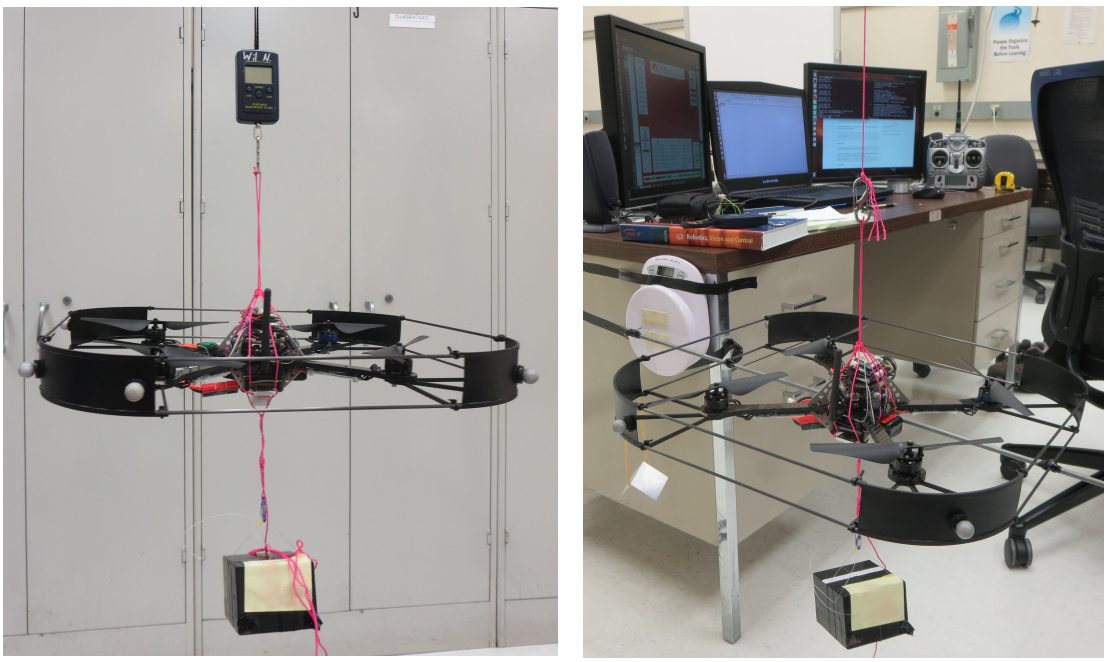


(a) Rise (b) Fall

Figure 11.3: Plots of the motor speed step response for rising and falling step commands. The blue lines represent the commanded values, while the orange lines represent the measured values.

11.3 Propeller Coefficient Evaluation

To test the performance of the Hummingbird quadrotor's flexible propellers, two tests were performed: a static thrust test, and a static drag test. Figure 11.4 shows the setups of the thrust and drag tests. Both tests involved weighing the quadrotor down and suspending it using a harness composed of rope. The static thrust test involved suspending the quadrotor harness from a hanging electronic scale and measuring the weight as motor speeds were varied. The drag test added a rigid arm with a point at a set radius, setting it to push against an electric scale, setting the motors such that it executed a strong yaw torque, and measuring the "weight" generated at the set radius.



(a) Thrust Coefficient Test

(b) Drag Coefficient Test

Figure 11.4: Pictures of the quadrotor while performing the thrust and drag coefficient tests. An aluminum mounting block can be seen under the bottom of the quadrotor.

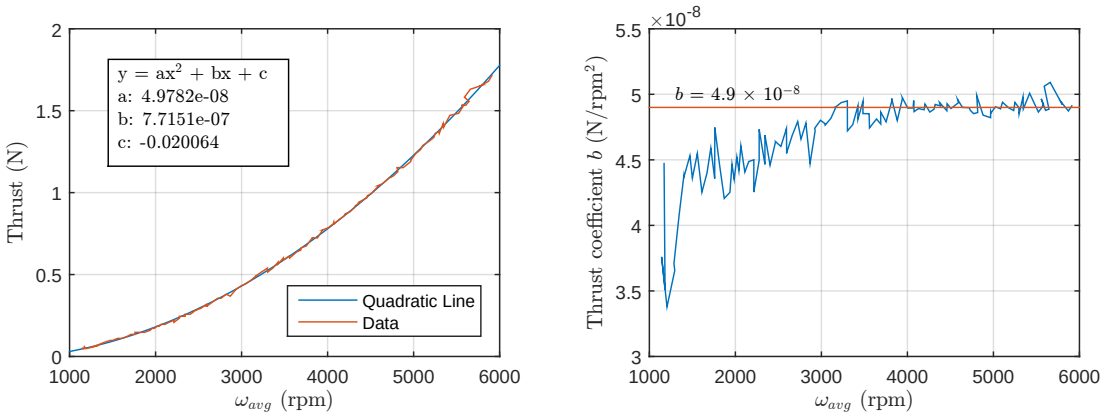
11.3.1 Propeller Thrust Coefficients

To evaluate the thrust coefficient of the flexible propellers, the quadrotor was weighed down and suspended from a hanging electric scale. The scale's weight reading was recorded with the motors off. The motors were all set with a DMC value of 1 for a period of time, and the scale's weight reading was recorded. This process was repeated for incrementing DMC values from 1 to 128, at which point the quadrotor started to become unstable within the harness shown in Figure 11.4a. The quadrotor's measured motor speeds as reported by the Autopilot were recorded during the entire process.

The methods described in Section 11.1 was used to generate average motor speeds for each DMC. These averages were grouped together to form one big set, visible outliers were removed, and an average motor speed for each DMC was generated. The scale readings were subtracted from the initial scale reading to give the net thrust contribution in kg, which was converted to Newtons. The net thrust value was divided by 4 to give the average thrust generated by each propeller.

Figure 11.5a shows a plot of the average motor speeds compared to the calculated average thrust for one propeller. As the average motor speeds were generated from noisy measurements, some jagged lines appear in the plot. A quadratic regression was applied to the set to match the relationship between propeller speed and generated thrust. A quadratic coefficient of 4.9782×10^{-8} N/rpm^2 resulted from the data set. This would imply a thrust coefficient of $\approx 5.0 \times 10^{-8}$ N/rpm^2.

Calculating the thrust coefficient on a per-DMC basis, however, presented a different potential coefficient. Figure 11.5b shows a plot of the calculated thrust coefficient versus average propeller speed. The thrust coefficient appears to be lower at low propeller speeds and slowly levels out as the motor speed passes 3,000 rpm. A trend line in Figure 11.5b shows that the calculated thrust coefficient values center



(a) Thrust vs. Motor Speed Squared (b) Thrust Coefficient vs. Motor Speed

Figure 11.5: Plots from thrust coefficient test. 11.5a: Trend of thrust vs. the average motor speed squared. 11.5b: Plot of the thrust coefficient vs the average motor speed.

around 4.9×10^{-8} N/rpm^2 past the 3,000 rpm mark. As the propellers will primarily be operating within the middle of their speed range, a thrust coefficient value of $b = 4.9 \times 10^{-8}$ N/rpm^2 will be used. This value is approximately 80% of the value specified in [20].

11.3.2 Propeller Drag Coefficients

To evaluate the drag coefficient of the flexible propellers, the quadrotor was weighed down and suspended by a rope. A metal bar was fixed to the quadrotor frame, and a bolt was attached to the bar's end such that it stuck out perpendicular to the bar. The quadrotor was suspended in such a way that the bolt was aligned to press on an electric scale secured to a vertical surface. Motors 2 and 4 were both set to DMC values of 50 while Motors 1 and 3 were left off, which generated a net negative torque. The bolt pressed against the scale and generated a force/"weight" reading at a given radius, which was a measurable torque. This process was repeated for DMC values of 100, 150, and 200 as well.

Table 11.1: Test Results for the Drag Coefficient Test

DMC	Ω_{avg} (rpm)	F_{scale} (kg)	$\tau_{\mathbf{z}}$ (N m)	k (N m/rpm^2)
50	2,922	0.003	0.013	7.6×10^{-10}
100	4,817	0.008	0.035	7.5×10^{-10}
150	6,518	0.017	0.074	8.7×10^{-10}
200	7,988	0.024	0.104	8.1×10^{-10}

The radius at which the bolt was fixed was measured to be $r = 17\frac{3}{8}$ inches, which corresponds to $r = 44.1325$ cm. Table 11.1 lists the measured average motor speeds for the test, the measured scale readings, the resulting torque, and the effective drag coefficient calculated using Equation 11.3.

$$k = \frac{rF_{scale}}{2\Omega_{avg}^2} = \frac{\tau_{\mathbf{z}}}{2\Omega_{avg}^2}. \tag{11.3}$$

The low precision of the electric scale (0.001 kg) made it difficult to gather any conclusive data on the flexible propellers' drag coefficient. Still, the general mean of the drag coefficients was loosely grouped around 8×10^{-10} N m/rpm^2, which is $\sim$53% of the specified value from [20].

11.4 Quadrotor Rotational Inertia

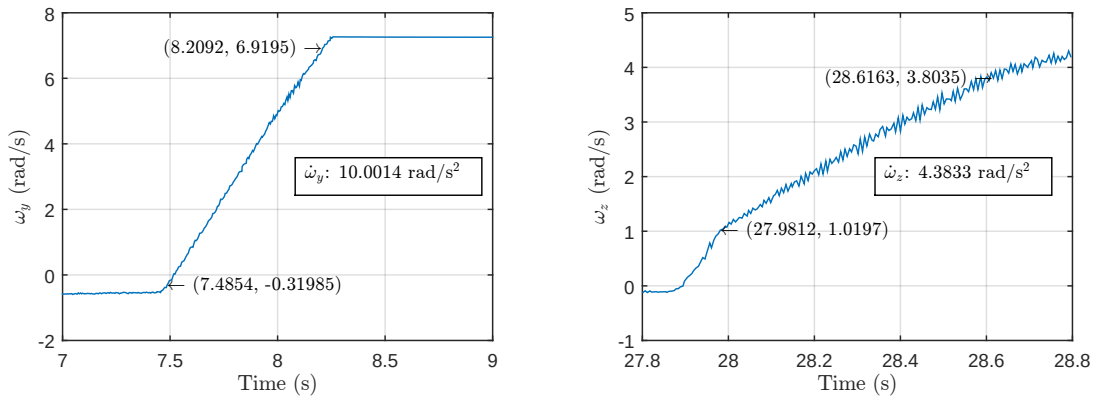
With properly characterized propellers, it becomes possible to evaluate the rotational inertia of the quadrotor. The on-board gyroscopes can quickly measure any changes in angular velocity, while specific motor commands can be sent to generate steady torques. For a single axis of rotation, an object's rotational inertia J, the net torque it experiences, and its angular acceleration are related through Equation 11.4. This equation stems from Euler's equation of motion (Equation 6.15) collapsing when only one angular velocity is nonzero. Using this relationship, the rotational inertia of the quadrotor around its Z^B and Y^B axes were evaluated. As the quadrotor is largely

Figure 11.6: Picture of the suspended quadrotor while testing its pitch rotational inertia.

symmetrical around its X^B and Y^B axes, the rotational inertias around both axes were assumed to be the same.

$$\tau = J\dot{\omega}. \tag{11.4}$$

Figure 11.6 shows a configuration used to evaluate the pitch (Y^B) rotational inertia. A similar configuration was used for the yaw (Z^B) rotational inertia. The quadrotor was suspended using fishing line such that it hung vertically in a balanced manner. The fishing line was loosely attached to a fixed metal bar using a zip tie. The zip tie prevented the quadrotor from swinging, but it allowed the fishing line to freely rotate.



(a) Pitch Inertia Test

(b) Yaw Inertia Test

Figure 11.7: Plots of the angular accelerations over time for the pitch and yaw rotational inertia tests.

For the pitch inertia test, Motor 3 was commanded to execute a constant DMC value of 50 while the rest of the motors were off. Figure 11.7a shows the resulting progression of $\omega_{\mathbf{y}}$ over time. The average angular acceleration over the course of 0.7 seconds was 10.0014 rad/s^2. Past this point, the gyroscope saturated and continued to report a constant angular velocity. Equation 11.5 calculates the torque applied using the average motor speed for a DMC of 50, while Equation 11.6 calculates the quadrotor's pitch rotational inertia. The resulting value of 7.1×10^{-4}kg m^2 shows that the added Edison block and protective frame effectively doubled the quadrotor's $\mathbf{J}_{yy}$ value.

$$\tau_{\mathbf{y}} = bl\Omega^2 = (4.9 \times 10^{-8}\ \text{N/rpm}^2)(0.17\ \text{m})(2,922\ \text{rpm})^2 = 0.0071\ \text{N m}, \tag{11.5}$$

$$\mathbf{J}_{yy} = \frac{\tau_{\mathbf{y}}}{\dot{\omega}_{\mathbf{y}}} = \frac{0.0071\ \text{N m}}{10.0014\ \text{rad/s}^2} = 7.1 \times 10^{-4}\ \text{kg m}^2. \tag{11.6}$$

For the yaw inertia test, Motors 1 and 3 were set to DMC values of 150 while Motors 2 and 4 remained off. Figure 11.7b shows the resulting progression of $\omega_{\mathbf{z}}$ over time. The average angular acceleration ofver the course of 0.6 seconds was 4.3833

rad/s^2. The measurements became fairly noisy, so points near the middle of the vibrations were selected. The torque for DMC values of 150 was already measured in Table 11.1. Equation 11.7 calculates the quadrotor's yaw rotational inertia. The resulting value of 0.017 kg m^2 shows that the Edison block and protective frame more than doubled the quadrotor's $\mathbf{J}_{zz}$ value.

$$\mathbf{J}_{zz} = \frac{\tau_{\mathbf{z}}}{\dot{\omega}_{\mathbf{z}}} = \frac{0.074 \text{ N m}}{4.3833 \text{ rad/s}^2} = 0.017 \text{ kg m}^2. \tag{11.7}$$

Chapter 12

Results

This chapter examines the flight performance of the quadrotor. Section 12.1 shows a simulation of the nonlinear controller's performance on an ideal quadrotor model. Section 12.2 examines the flight performance where the quadrotor was commanded to take off from the ground and hover. Section 12.3 examines the flight performance for a test case where the quadrotor was commanded to fly in a tilted figure eight pattern. The tracking error in all three cases is examined.

12.1 Controller Simulation Results

The feedback linearization-based controller was created in MATLAB and simulated with an ideal, frictionless, damping-free quadrotor model. To evaluate its performance in tracking a trajectory, a trajectory pattern in the form of a figure eight with varying altitude was constructed. The figure eight had a width (X^W) of 1.5 meters, a length (Y^W) of 2 m, and an altitude variation of 0.50 m. The figure eight had a period of 12 seconds. By using sinusoids for the trajectory signals, determining their derivatives for the feedback linearization's artificial inputs was simplified.

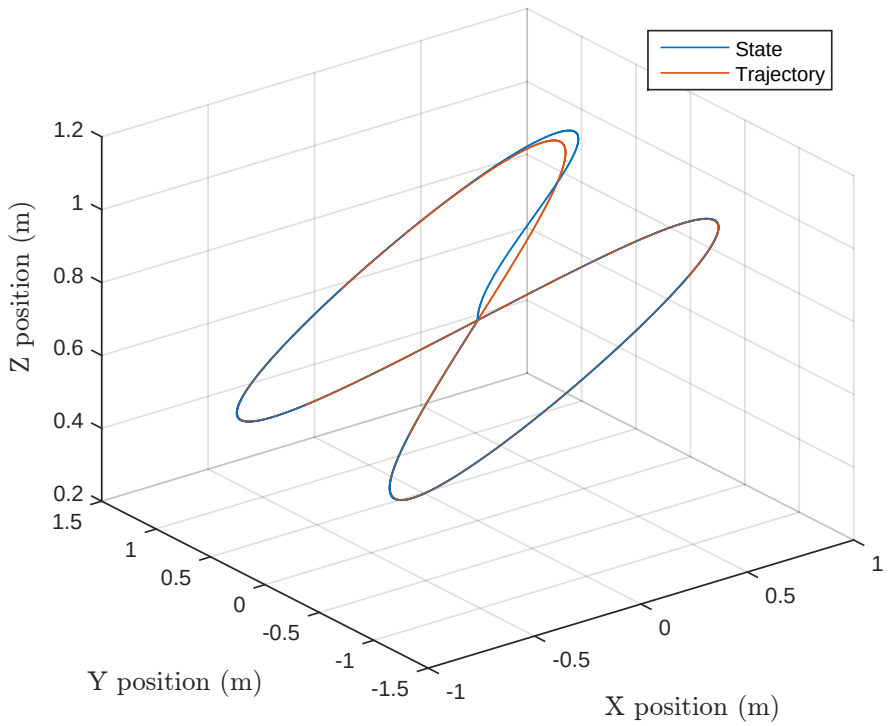


Figure 12.1: 3-D plot of the test trajectory and the simulated quadrotor's attempt to follow it.

Figure 12.1 shows a 3-D plot of the generated trajectory and the quadrotor's states as it tracks the trajectory. Figure 12.2 shows the evolution of each trajectory value and output state. The simulated controller manages to quickly converge to the target trajectory and lock on for the duration of the flight. As the feedback linearization process used the same model that was generated for performing state evolution calculations, the feedback linearization manages to perfectly invert the quadrotor's dynamics and control the quadrotor's position in a linear manner. Some initial oscillations in the various position states can be seen, but the oscillations quickly decay away until the system converges to the reference trajectory.

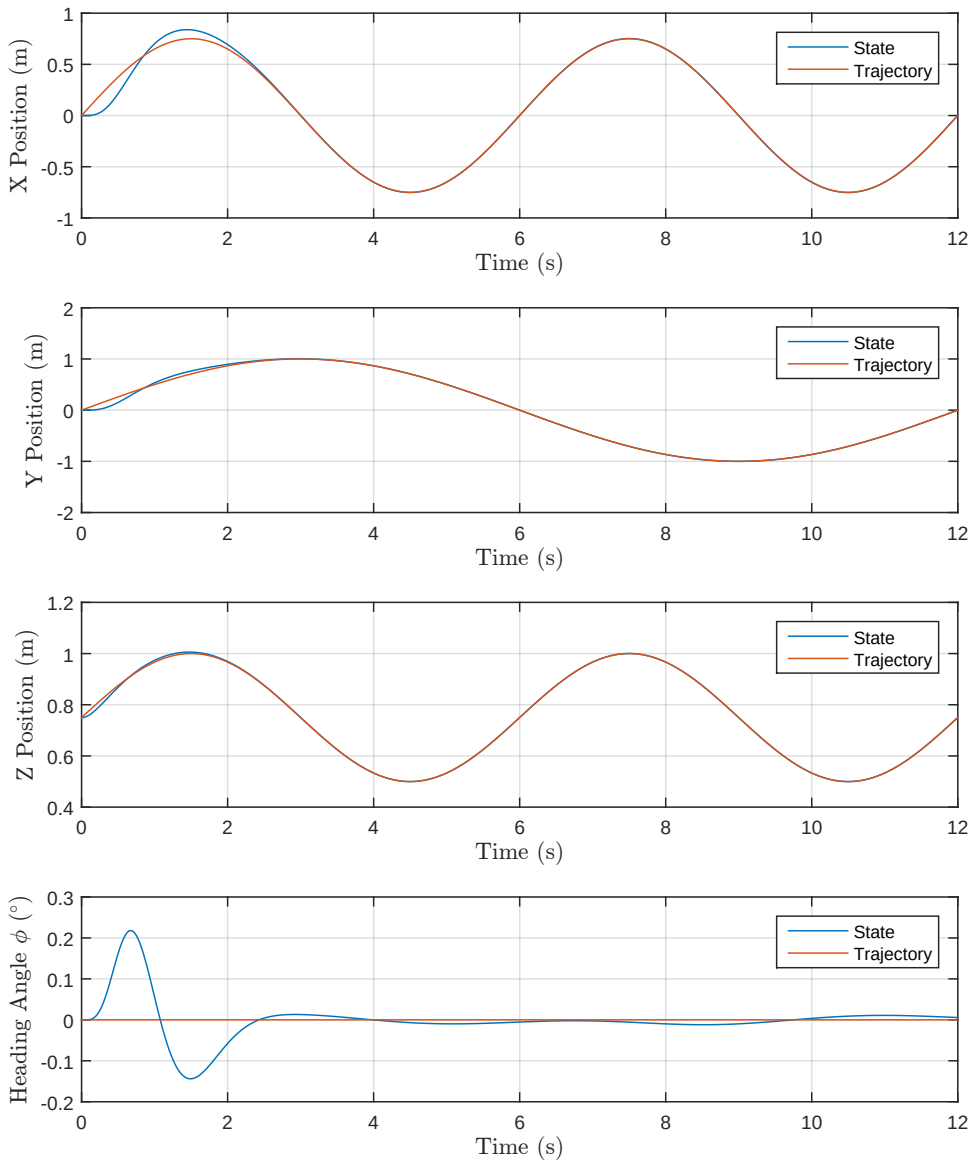


Figure 12.2: Plot of the test trajectory values and the simulated quadrotor's states as it attempt to follow.

12.2 Takeoff and Hover Results

For the takeoff procedure, the reference altitude was swept upwards in a sinusoidal manner from ground level to an altitude of 0.75 m. The x and y positions were commanded to hold constant from their starting positions at ground level, while the heading angle was commanded to sweep to $0°/0$ radians. Generating the sinusoidal sweeps in heading and altitude involved also generating the appropriate trajectory derivatives to serve as references for the feedback linearization's artificial input. After coming within 5 cm of the target altitude, the quadrotor was told to hover where it was until it received another command.

Figure 12.5 shows a 3-D plot of the quadrotor's position trajectory and states, while Figure 12.3 shows the individual states and trajectories. The takeoff procedure started around the 87 second mark and finished around the 89 second mark. The quadrotor's x and y states initially varied by a large margin, though this was caused by the quadrotor sitting at a slight angle on takeoff. Once the hover mode was reached, the x and y values fluctuated around the reference signal. The altitude and heading followed their reference signals with minor fluctuations, but there were no apparent unexpected behaviors.

Figure 12.4 shows the error associated with each state and trajectory signal. After the takeoff procedure and some brief fluctuations in the transition to hovering, the x and y errors never went above $\sim$ 2.5 cm/1 inch. While hovering, the altitude remained within 1 cm of the reference signal. After leaving the ground, the heading angle stayed within $\sim 1°/0.018$ radians of the given trajectory.

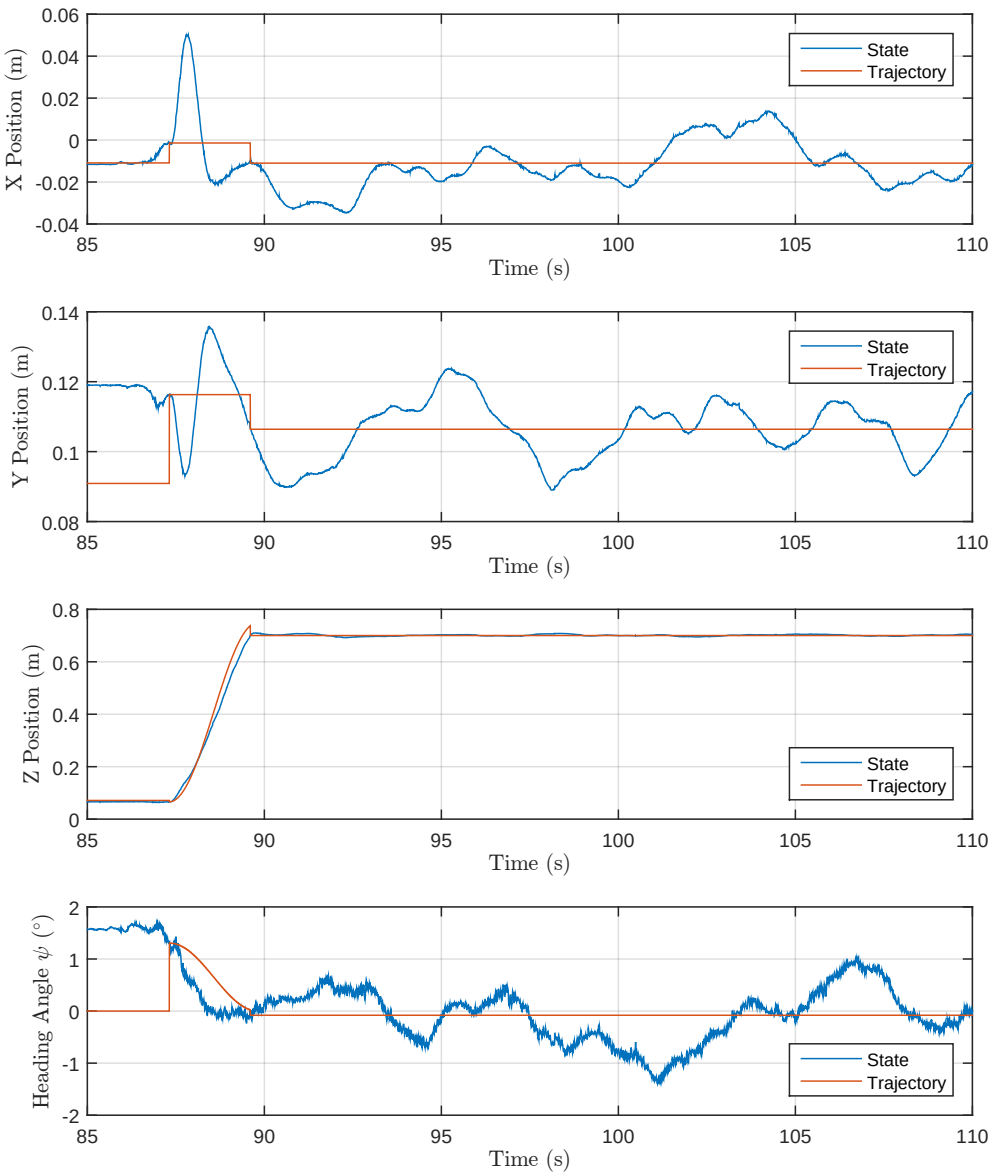


Figure 12.3: Plots of the position and heading trajectories vs. the quadrotor's states as the quadrotor takes off and hovers.

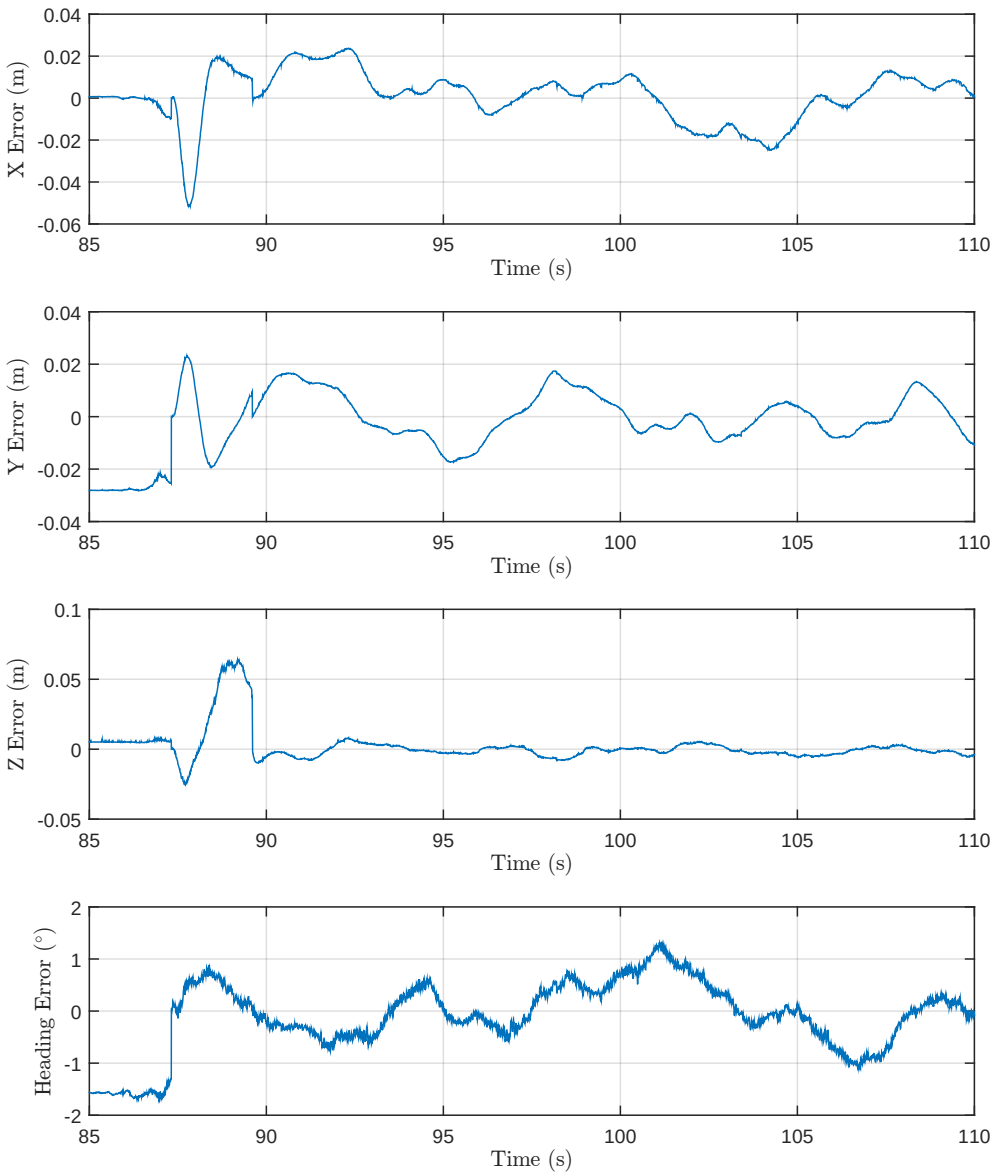


Figure 12.4: Plots of the error between the trajectory and the quadrotor's state while taking off and hovering.

12.3 Trajectory Tracking Results

To evaluate the quadrotor's hardware performance in tracking a trajectory, the same trajectory defined in Section 12.1 was used. Figure 12.6 shows a 3-D plot of the quadrotor's position trajectory and states during the figure eight pattern, while Figure 12.7 shows the individual states and trajectories. After a discontinuity in the trajectory and a resulting jump in the states, the quadrotor eventually converged and tracked the desired trajectory. A discontinuity in the quadrotor's state occurs around the 23.5 second mark. Examining the data set, the discontinuity resulted from the control program not receiving Vicon measurements for a period of ∼300 ms. In that time frame, the IMU and state prediction filtering took over and the quadrotor slowly drifted over time.

While Figure 12.6 shows that the quadrotor largely followed the desired pattern, Figure 12.7 shows that it did so with a delay. The delay worked out to be ∼200 ms. Curiously, this 200 ms delay looks similar to the one seen in the motor speed step response plots found in Chapter 11. This seems to imply that the assumption made in Section 11.2 about the motor acceleration dynamics being insignificant may not be valid. Still, other sources of error may be causing the problem, such as unmodeled dynamics (drag, gyroscopic effects, and motor accelerations) that are left out of the feedback linearization's inversion process.

Figure 12.8 shows the error between the reference trajectories and the quadrotor's states. The error for all three position states varies sinusoidally at a frequency approximately equal to that of its reference trajectory. While the altitude stayed within 5 cm of the trajectory, the y state varied by almost 10 cm, and the x state varied by almost 20 cm. These values largely stemmed from the lag seen between the trajectory and the states.

To account for the delay seen in Figure 12.7, the trajectory was artificially lagged and the quadrotor's error was recalculated. Figure 12.9 shows the resulting error with ∼175 ms of simulated delay. The error drops down substantially, with both the x and y errors dropping to within 5 cm. The effect on altitude error was not as drastic; it dropped to be within ∼3 cm. Still, the results of Figure 12.9 indicate that a latency of ∼175 ms exists somewhere within the system. The consistency of the latency is cause for concern.

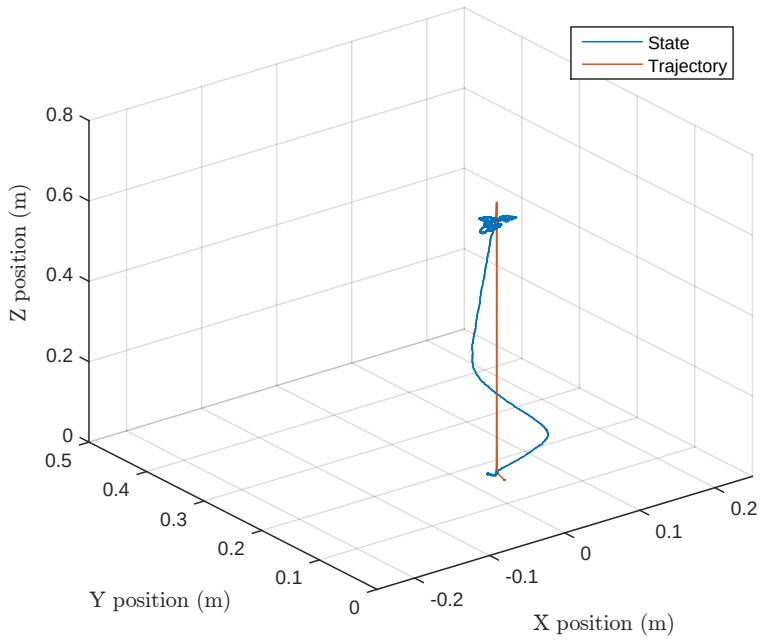


Figure 12.5: 3-D Plots of the position trajectories as the quadrotor takes off and hovers.

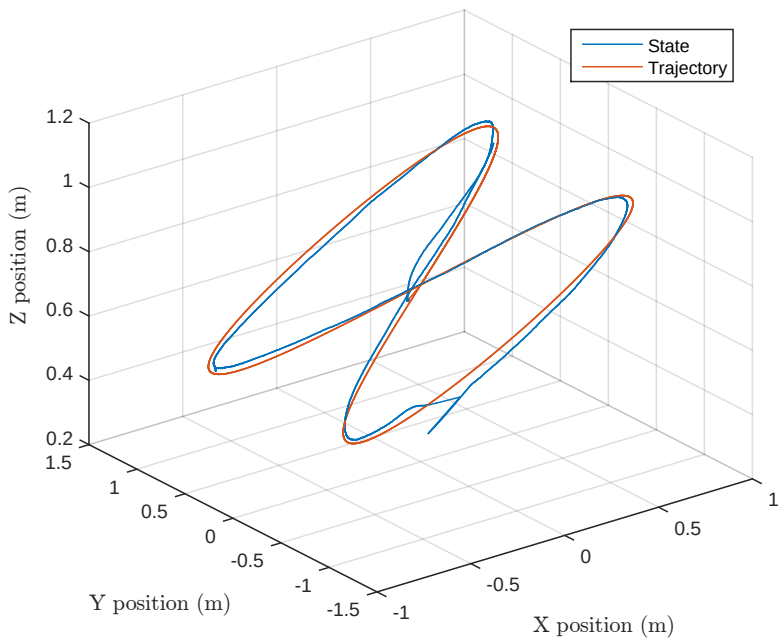


Figure 12.6: 3-D plot of the test trajectory and the quadrotor's attempt to follow it.

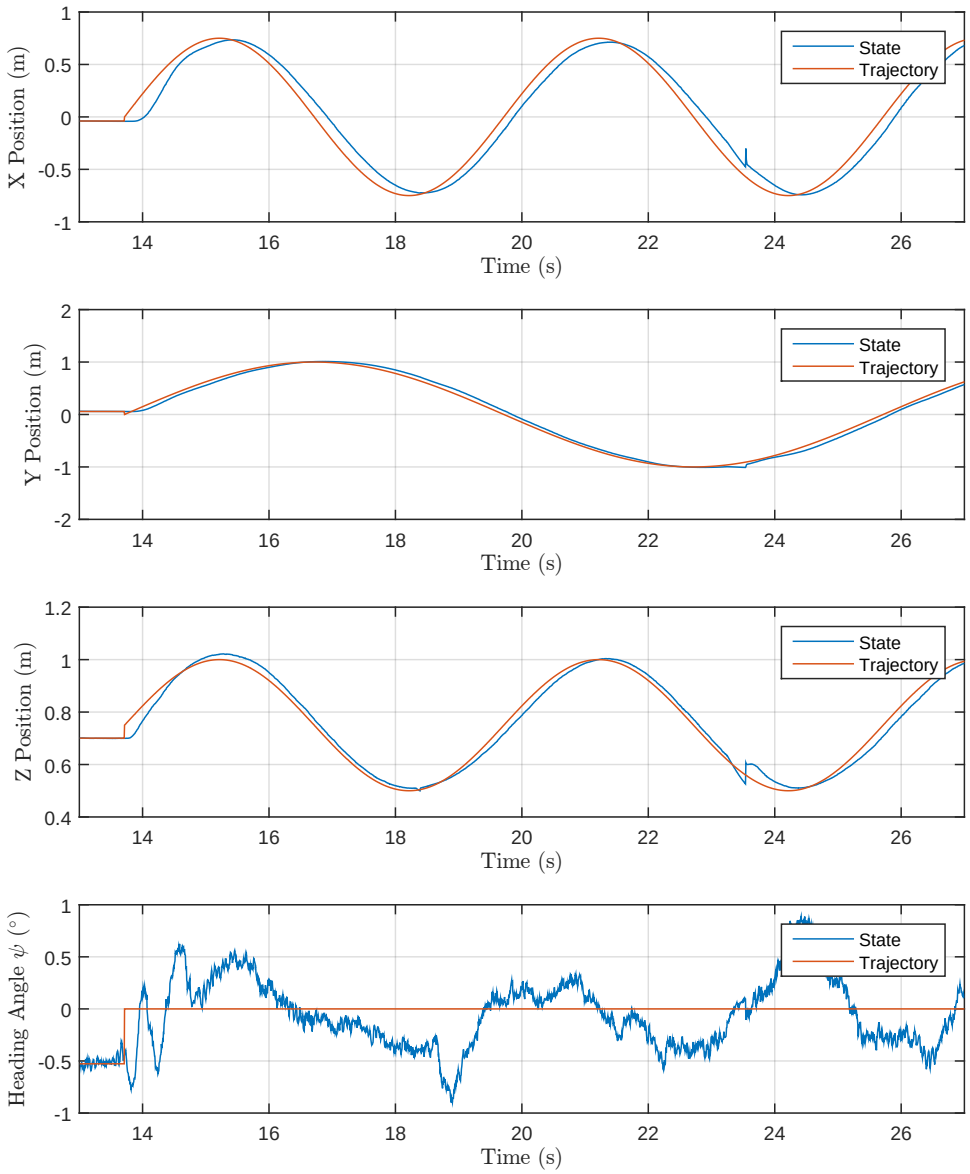


Figure 12.7: Plots of the position and heading trajectories and the quadrotor's attempt to follow them.

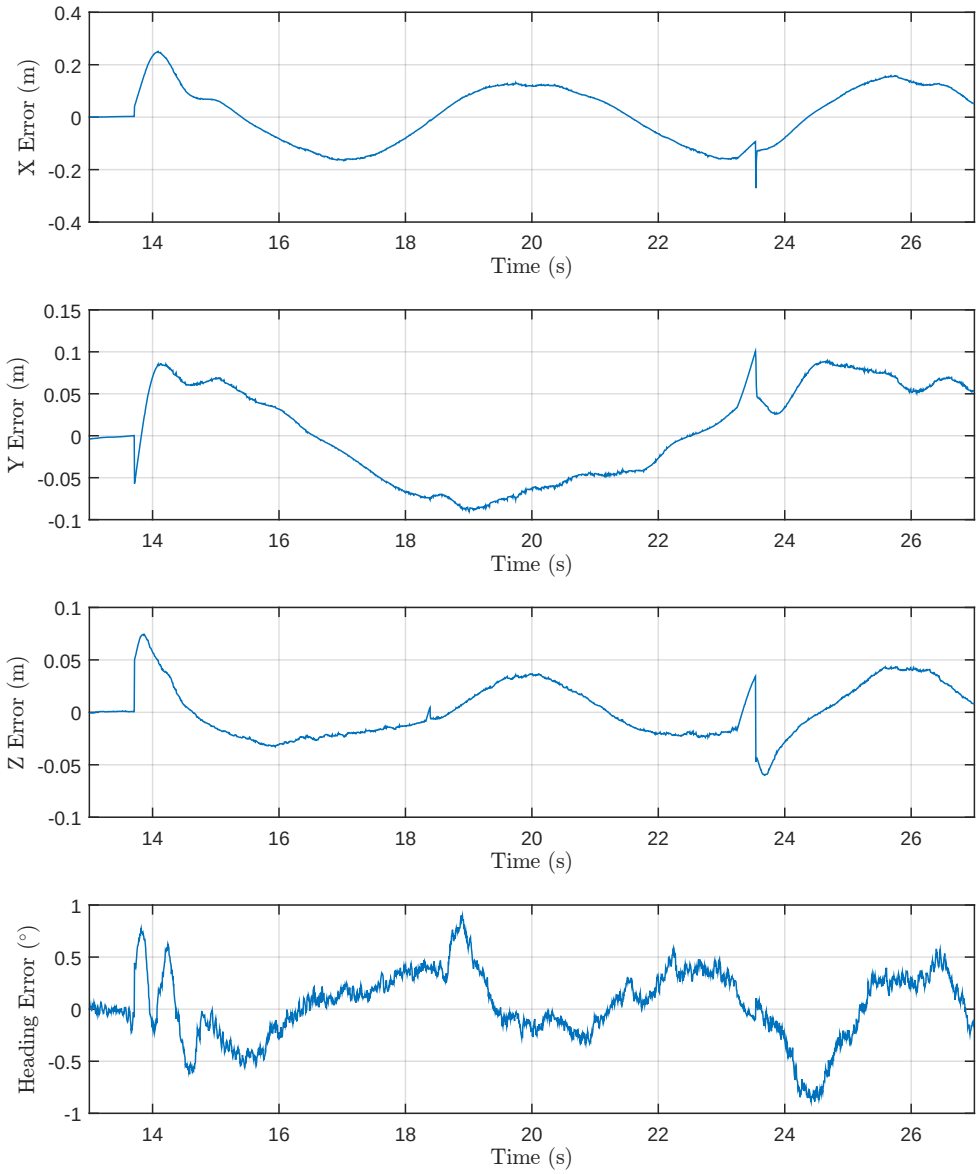


Figure 12.8: Plots of the error between the trajectory and the quadrotor's state.

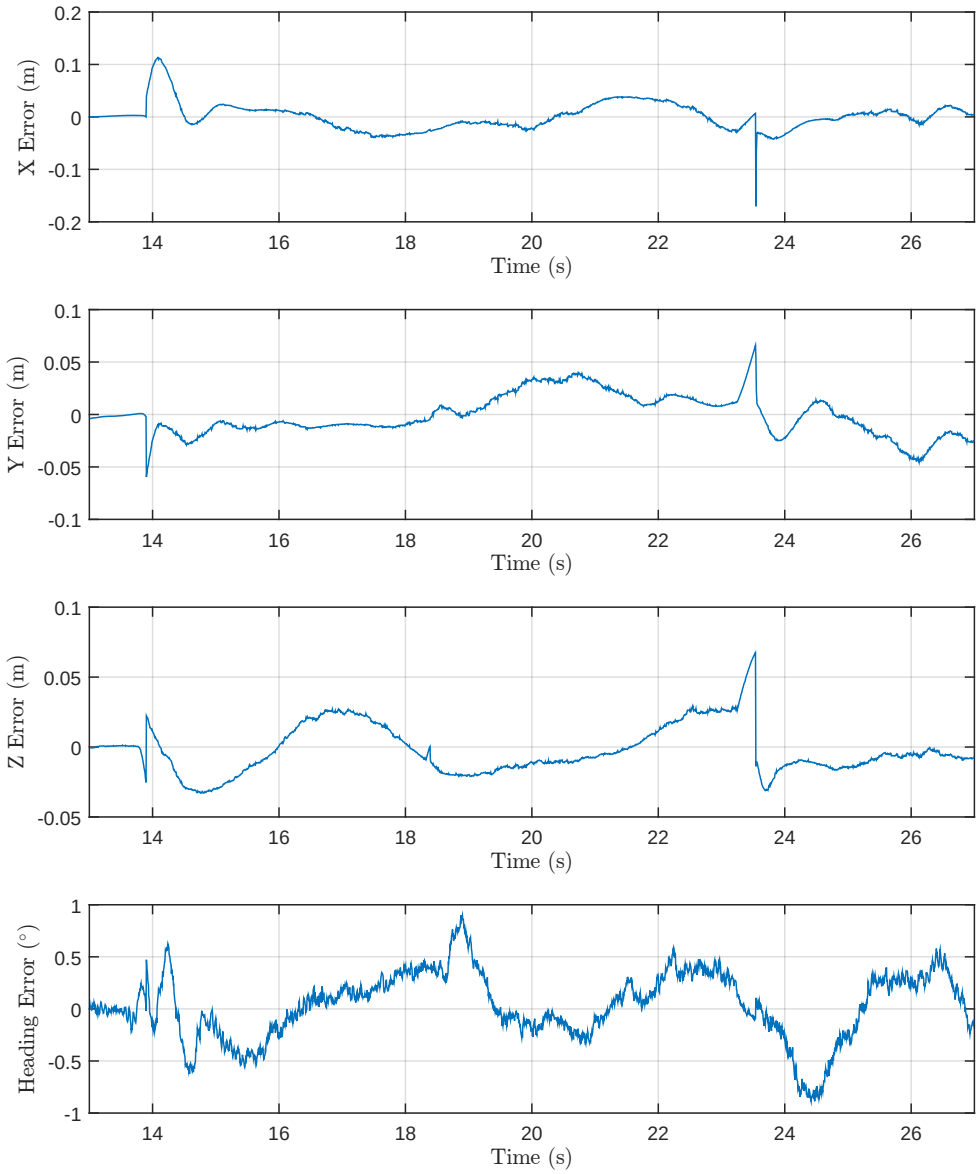


Figure 12.9: A reproduction of Figure 12.8 where the trajectory has been artificially lagged by 175 ms.

Chapter 13

Conclusions and Future Work

13.1 Conclusions

In this thesis, a high-performance quadrotor control system was developed for an AscTec Hummingbird quadrotor using direct motor speed control within a Vicon motion capture system environment. A custom circuit board was designed for interfacing the Hummingbird's Autopilot with an Intel Edison computer. A communication framework was developed between the Intel Edison and the Autopilot's HLP such that the Edison could receive IMU measurements and send motor speed commands to the motor controllers.

An explanation of Euler angles and quaternions is provided, and a split quaternion frame is described. A mathematical model for the quadrotor's dynamics was developed. Feedback linearization was applied to the quadrotor's dynamics with the split quaternion system to render the system input-output linear relative to artificial inputs. A controller based on developing an exponentially stable Lypaunov function was applied to the quadrotor's linearized dynamics. Design heuristics and the simplex method ensured the quadrotor could safely restrict infeasible input combinations.

the Vicon system was configured to transmit the quadrotor's position and orientation to the Edison over Wi-Fi. The Vicon measurements and the Autopilot's IMU measurements are meshed together on the Edison using a quick, Kalman-like filter that bypasses the lengthy Kalman filter and Extended Kalman filter calculations using a heuristic-based approach. A method for generating the quadrotor's velocities from noisy position measurements and inaccurate acceleration measurements was proposed and implemented.

A multithreaded C++ program was written to handle obtaining measurements, performing state estimation, calculating control inputs, limiting invalid inputs, sending motor commands to the quadrotor, and logging the program's variables. The Hummingbird quadrotor's propellers and physical model were evaluated.

The control system was applied to the quadrotor, flight patterns were performed, and the results were analyzed. Hovering control was found to work well within 2.5 cm of a desired hover point. Trajectory tracking came within 5 cm of the desired reference, albeit with $\sim$175 ms of latency.

13.2 Contributions

Below is a list of this thesis' main contributions:

- Circuit Board Design: This thesis developed a custom circuit board designed specifically to interface the Intel Edison with the AscTec Hummingbird's Autopilot while providing power, battery charging, and USB serial console functionalities.
- Communication Framework: A custom communication framework was implemented between the Edison and the Autopilot's HLP for passing data back and forth over UART in a fast, reliable, and organized manner.

- Feedback Linearization Rederivation: Extensive work was put into rederiving the feedback linearization process developed in [9] for a Z-up coordinate system.
- Input Limitation: A custom input restriction method was developed using optimization methods and some design choices to ensure only feasible input combinations were attempted.
- Velocity Estimation: A custom method for quickly estimating an object's velocity from position and acceleration data was developed and tested.
- Control Program: A modular multithreaded C++ program was designed for implementing the control process on the Edison. the modularity of the design facilitates replacing individual components as needed for future design iterations.
- Hummingbird Model Evaluation: unique sets of propeller coefficients and rotational inertias were found for the AscTec Hummingbird with an attached protective frame, attached Edison and circuit board, and the stock flexible propellers. The measured propeller coefficients provide conflicting numbers for the values provided by AscTec and other sources.

13.3 Areas for Improvement

Multiple areas exist for improving the current system. The most notable is a controller that incorporates the dynamics of the individual motors and propellers to increase the system responsiveness. Other parameters to be included in the modeling would be translational and rotational damping, the gyroscopic effects of the propellers, and the effects of accelerating the propellers. The increased model and controller accuracy would hopefully mitigate the $\sim$175 ms delay seen between a desired trajectory and the quadrotor's state.

Another improvement would involve shifting the control algorithm to the Autopilot's HLP to take advantage of the 1 kHz default loop rate and direct access to the IMU measurements. However, such an implementation would likely have to operate only on attitude control, while some type of attitude reference values are passed from a position controller. Using fixed point representations of numbers would keep the control algorithm efficient and tractable on the Autopilot's HLP, though developing a robust framework for fixed point notation would take a lot of work.

Sticking with the current control scheme, research into smooth trajectory generation that avoids discontinuities in the position states and their derivatives would help the nonlinear controller perform better. The discontinuities create rapid and sometimes violent responses as the system attempts to quickly correct the problem.

The custom Quadrotor Block circuit board for attaching the Edison to the Autopilot could use several improvements. Battery protection circuitry could prevent overly discharging the Edison's battery. Voltage measurement methods would provide insight into the power left within be battery. Additional sensors, such as gyroscopes that don't saturate when the Autopilot's gyroscopes saturate, could be added to the board to provide even more sets of measurements. Minor component changes in future iterations could help keep the cost down and simplify the board assembly.

To increase the amount of feasible maneuvers, the Vicon motion capture volume could be increased in size. The current size restricts the ability to perform large and/or fast maneuvers. More cameras would also provide a higher degree of accuracy and reduce the number of blind spots within the volume.

References

[1] P. J. C. Davalos, "Real-time control architecture for a multi uav test bed," Master's thesis, University of New Mexico Dept. of Electrical and Computer Engineering, December 2012. [Online]. Available: http://repository.unm.edu/handle/1928/21995

[2] (2014, May) Quadrotor's real-time controller using labview. The University of New Mexico - MARHES Laboratory. [Online]. Available: http://marhes.ece.unm.edu/index.php/LabView_Quadrotor

[3] I. Palunko, R. Fierro, and P. Cruz, "Trajectory generation for swing-free maneuvers of a quadrotor with suspended payload: A dynamic programming approach," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 2691–2697.

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

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

[6] P. Cruz and R. Fierro, "Autonomous lift of a cable-suspended load by an unmanned aerial robot," in *Control Applications (CCA), 2014 IEEE Conference on*. IEEE, 2014, pp. 802–807.

[7] K. Åström and K. Furuta, "Swinging up a pendulum by energy control," *Automatica*, vol. 36, no. 2, pp. 287 – 295, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109899001405

[8] S. Bouabdallah, A. Noth, and R. Siegwart, "Pid vs lq control techniques applied to an indoor micro quadrotor," in *Intelligent Robots and Systems, 2004. (IROS*

2004). Proceedings. 2004 IEEE/RSJ International Conference on, vol. 3, Sept 2004, pp. 2451–2456 vol.3.

[9] O. Fritsch, P. De Monte, M. Buhl, and B. Lohmann, "Quasi-static feedback linearization for the translational dynamics of a quadrotor helicopter," in *American Control Conference (ACC), 2012*, June 2012, pp. 125–130.

[10] Z. Shulong, A. Honglei, Z. Daibing, and S. Lincheng, "A new feedback linearization lqr control for attitude of quadrotor," in *Control Automation Robotics Vision (ICARCV), 2014 13th International Conference on*, Dec 2014, pp. 1593–1597.

[11] Y. Al-Younes, M. Al-Jarrah, and A. Jhemi, "Linear vs. nonlinear control techniques for a quadrotor vehicle," in *Mechatronics and its Applications (ISMA), 2010 7th International Symposium on*, April 2010, pp. 1–10.

[12] S. Bouabdallah and R. Siegwart, "Full control of a quadrotor," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, Oct 2007, pp. 153–158.

[13] P. De Monte and B. Lohmann, "Trajectory tracking control for a quadrotor helicopter based on backstepping using a decoupling quaternion parametrization," in *Control Automation (MED), 2013 21st Mediterranean Conference on*, June 2013, pp. 507–512.

[14] H. Voos, "Nonlinear control of a quadrotor micro-uav using feedback-linearization," in *Mechatronics, 2009. ICM 2009. IEEE International Conference on*, April 2009, pp. 1–6.

[15] M. Buhl and B. Lohmann, "Control with exponentially decaying lyapunov functions and its use for systems with input saturation," in *Control Conference (ECC), 2009 European*, Aug 2009, pp. 3148–3153.

[16] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea, "A platform for aerial robotics research and demonstration: The flying machine arena," *Mechatronics*, vol. 24, no. 1, pp. 41 – 54, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957415813002262

[17] D. Mellinger, "Trajectory generation and control for quadrotors," Ph.D. dissertation, University of Pennsylvania, January 2012, dissertations available from ProQuest. Paper AAI3509215. [Online]. Available: http://repository.upenn.edu/dissertations/AAI3509215

[18] A. Maksymiw, *AscTec Autopilot*, Ascending Technologies GmbH, January 2015. [Online]. Available: http://wiki.asctec.de/display/AR/AscTec+AutoPilot

[19] *Go Further with Vicon MX T-Series*, Revisioin 1.3, Vicon Motion Systems Ltd., January 2011.

[20] A. Ryll, *AscTec Hummingbird*, Ascending Technologies GmbH, April 2015. [Online]. Available: http://wiki.asctec.de/display/AR/AscTec+Hummingbird

[21] K. P. Valavanis and G. J. Vachtsevanos, *Handbook of Unmanned Aerial Vehicles*. Springer Science+Business Media, 2015.

[22] A. Ryll, *Pinout and Connections*, Ascending Technologies GmbH, January 2015. [Online]. Available: http://wiki.asctec.de/display/AR/Pinout+and+Connections

[23] A. Maksymiw, *AscTec SDK*, Ascending Technologies GmbH, April 2014. [Online]. Available: http://wiki.asctec.de/display/AR/SDK+Manual

[24] ——, *AscTec Communication Interface: List of all predefined variables, commands and parameters*, Ascending Technologies GmbH, April 2015. [Online]. Available: http://wiki.asctec.de/display/AR/List+of+all+predefined+variables%2C+commands+and+parameters

[25] *VICON MX Hardware System Reference*, Revision 1.7, Vicon Motion Systems Ltd., November 2007.

[26] *Vicon Datastream SDK Manual*, Revision 1.2.0, Vicon Motion Systems Ltd., March 2011.

[27] (2015, July) Community supported platforms. The Qt Company. [Online]. Available: https://doc.qt.io/qt-5/supported-platforms.html

[28] *Intel Edison Module Hardware Guide*, Revision 4, Intel Corporation, January 2015, document Number 331189-004. [Online]. Available: http://download.intel.com/support/edison/sb/edisonmodule_hg_331189004.pdf

[29] (2015, July) Yocto project homepage. The Yocto Project. [Online]. Available: http://www.yoctoproject.org

[30] (2015, July) Ubilinux. Emutex Ltd. [Online]. Available: http://emutexlabs.com/ubilinux

[31] S. Hymel. (2014, December) Loading debian (ubilinux) on the edison. Sparkfun Electronics. [Online]. Available: https://learn.sparkfun.com/tutorials/loading-debian-ubilinux-on-the-edison

[32] (2015, July) Compute module. Raspberry Pi Foundation. [Online]. Available: https://www.raspberrypi.org/products/compute-module/

[33] (2015, July) Gumstix homepage. Gumstix, Inc. [Online]. Available: https://www.gumstix.com/

[34] (2015, July) Odroid homepage. Hardkernel Co., Ltd. [Online]. Available: http://www.hardkernel.com/main/main.php

[35] *XBee® Multipoint RF Modules*, Digi International, 2011. [Online]. Available: http://www.digi.com/pdf/ds_xbeemultipointmodules.pdf

[36] (2015, March) Intel edison how-to guide v1.0. HELIOS Software GmbH. [Online]. Available: http://www.helios.de/heliosapp/edison/index.html#Power_consumption

[37] *MCP73831/2: Miniature Single-Cell, Fully Integrated Li-ion, Li-Polymer Charge Management Controllers*, Revision G, Microchip Technology Inc., July 2014, document DS20001984G. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/20001984g.pdf

[38] *FT232R USB UART IC*, Version 2.10, Future Technology Devices International Ltd., March 2012, document Number FT_000053. [Online]. Available: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf

[39] (2014, October) Sparkfun blocks template. Sparkfun Electronics. [Online]. Available: https://github.com/sparkfun/Sparkfun_Blocks_Template

[40] *UM10139: LPC214x User Manual*, Revision 4, NXP Semiconductors, April 2012. [Online]. Available: http://www.nxp.com/documents/user_manual/UM10139.pdf

[41] H. G. Wells, *War of the Worlds*. Project Gutenberg, October 2004. [Online]. Available: http://www.gutenberg.org/cache/epub/36/pg36.txt

[42] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, ser. Springer Tracts in Advanced Robotics. Springer, 2011.

[43] D. M. Henderson, "Euler angles, quaternions, and transformation matrices - working relationships," NASA, Lyndon B. Johnson Space Center, Tech. Rep., July 1977, mission Planning and Analysis Division. [Online]. Available: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770024290.pdf

[44] R. F. Stengel, *Flight Dynamics*. Princeton University Press, 2004.

[45] J. Huerta. (2015, April) Introducing the quaternions. UC Riverside Department of Mathematics. [Online]. Available: http://math.ucr.edu/~huerta/introquaternions.pdf

[46] B. L. Stevens and F. L. Lewis, *Aircraft Control and Simulation*. John Wiley & Sons, Inc., 2003.

[47] H. K. Khalil, *Nonlinear Systems*, 3rd ed. Prentice Hall, 2001.

[48] S. Sastry, *Nonlinear Systems: Analysis, Stability, and Control*. Springer, 1999.

[49] J.-J. E. Slotine and W. Li, *Applied Nonlinear Control*. Prentice-Hall, 1991.

[50] N. S. Nise, *Control Systems Engineering*, 6th ed. John Wiley & Sons, Inc., 2011.

[51] P. Dorato, C. Abdallah, and V. Cerone, *Linear Quadratic Control: An Introduction*. Krieger Pub Co, 2000.

[52] J. Hespanha, *Linear Systems Theory*. Princeton University Press, 2009.

[53] J. Nocedal and S. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. [Online]. Available: http://books.google.com/books?id=eNlPAAAAMAAJ

[54] G. Welch and G. Bishop. (2006, July) An introduction to the kalman filter. University of North Carolina at Chapel Hill Department of Computer Science. [Online]. Available: https://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[55] M. Al-Alaoui, "Al-alaoui operator and the new transformation polynomials for discretization of analogue systems," *Electrical Engineering*, vol. 90, no. 6, pp. 455–467, 2008. [Online]. Available: http://dx.doi.org/10.1007/s00202-007-0092-0

[56] (2015, July) About ros. The Open Source Robotics Foundation. [Online]. Available: http://www.ros.org/about-ros/

[57] B. Barney. (2015, January) Posix threads programming. Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[58] B. Hall. (2012, July) Beej's guide to network programming. [Online]. Available: http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf

[59] (2015, July) About opencv. Itseez. [Online]. Available: http://opencv.org/about.html

[60] A. Makhorin, *GNU Linear Programming Kit Reference Manual for Version 4.55*, GNU Project, August 2014.

[61] A. Ryll. (2015, January) Asctec sdk download. Ascending Technologies GmbH. [Online]. Available: http://wiki.asctec.de/display/AR/SDK+Downloads