Quadrotor Modeling and Control

Senior Design Final Report

Nikhil Ranganathan

Spring 2019

Contents

Figure 1: DJI Phantom 1

1 Introduction

1.1 Motivation

Applications for autonomous UAVs, specifically quadrotors, are numerous, with significant implications for socioeconomic growth. Autonomous quadrotors would allow farmers to continually monitor crop quality, grant energy companies the ability to survey pipelines, cables, and roads, without exhaustive labor costs, and allow infrastructure companies to continually assess bridge and building security. Additionally, quadrotors are increasingly being used as a research platform to test novel concepts in estimation, control, and autonomy, particularly because the parts are relatively cheap and well documented, and quadrotors do not require a large testing apparatus. As I intent to study control systems in graduate school, studying the quadrotor system would give me useful insight into not only quadrotors themselves, but practical control system design in general. Particularly, I wanted experience implementing a control system in hardware, as the control systems I have implemented in class have been generally in software. Thus, the primary educational objectives were to gain experience implementing a state-space controller in hardware, and to gain experience with quadrotor hardware as a research platform.

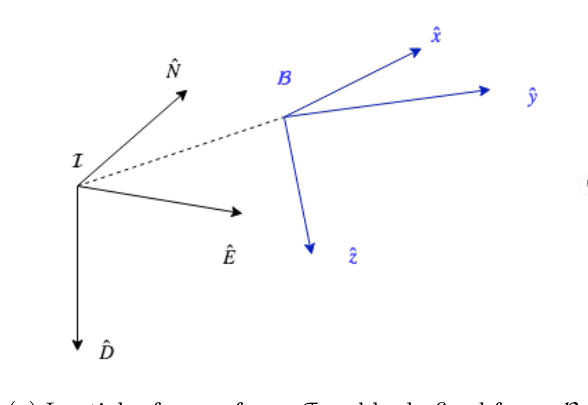
1.2 Project Concept

The primary project objective was to develop a quadrotor capable of stable hover and reaching waypoints autonomously. This was to be accomplished using a purpose-built attitude estimator from MEMS sensors and purpose-built control software. The only components allowed to be utilized from the DJI frame were the casing, motors, ESCs, and power distribution system. The components allowed to be COTS purchased were the sensors, transmitter, and reciever. All other elements were intended to be produced by me. In particular, I did not utilize a commercial flight controller or commercial attitude estimator.
The project was broken into several parts. The rough project timeline is shown in table 1.

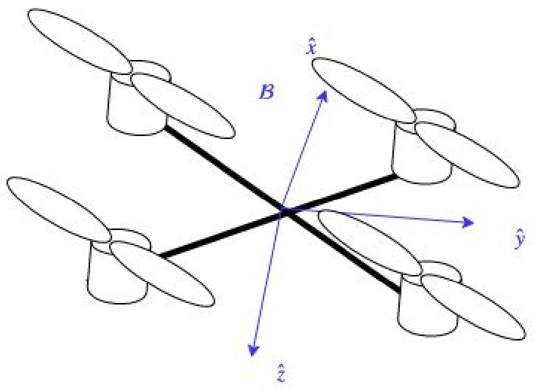
Project Component	Time given
Complete assembly and perform system ID	1 week
Develop simulator	3 weeks
Develop attitude estimator	4 weeks
Develop attitude controller	2 weeks
Integrate attitude estimator	1 week
Integrate attitude controller	1 week
Test and tune attitude controller	2 weeks
Develop and integrate position estimator	1 week
Develop and integrate position controller	1 week
Test and tune position controller	1 week

Table 1: Project timeline

I allocated only one week for hardware assembly and system identification because much of the assembly had been accomplished by me the previous semester. The bulk of the time was allocated to development of the simulator and attitude estimator, as those required the most novel work. I chose to schedule position controller and estimator towards the end of the project in case the attitude controller took the entire semester. Complete development and integration of an autonomous quadrotor in one semester was an ambitious concept, so I wanted to ensure a result.



(a) Inertial reference frame $\mathcal{I}$ and body-fixed frame $\mathcal{B}$



(b) Body Fixed Frame $\mathcal{B}$

Figure 2: Reference Frames

2 Quadrotor Modeling

2.1 Definitions and Formalisms

Figure 1 displays the two reference frames used throughout this report. First, there is an inertial earth-fixed North-East-Down (NED) frame $\mathcal{I}$, composed of unit vectors $\{\hat{N}, \hat{E}, \hat{D}\}$, where $\hat{N}$ points in the direction of magnetic north, $\hat{D}$ points in the direction of gravity, and $\hat{E} \equiv \hat{D} \times \hat{N}$. Second, there is a body fixed frame $\mathcal{B}$, composed of unit vectors $\{\hat{x}, \hat{y}, \hat{z}\}$, where $\hat{x}$ points in the quadrotor's 'forward' direction, $\hat{z}$ points downwards, and $\hat{y} \equiv \hat{z} \times \hat{x}$.
The quadrotor's attitude is represented in this report in two ways. First, all algorithms presented utilize the unit quaternion $\mathbf{q}$ which defines the rotation from the inertial frame to body frame, where

$$\mathbf{q} \in \mathbb{R}; \mathbf{q} \equiv \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}; q_0 \in \mathbb{R}; \bar{q} \in \mathbb{R}^3 \tag{1}$$

The quaternion $\mathbf{q}$, by definition, must satisfy

$$v_b = \mathrm{R}(\mathbf{q}) v_i \tag{2}$$

where v_i is a vector in the inertial frame, v_b is a vector in the body frame, and $\mathrm{R}(\mathbf{q})$ is the rotation matrix defined from the quaternion, which is given by

$$\mathrm{R}(\mathbf{q}) = \begin{bmatrix} 1 - 2q_2^2 - 2q_3^2 & 2(q_1 q_2 - q_3 q_0) & 2(q_1 q_3 + q_2 q_0) \\ 2(q_1 q_2 + q_3 q_0) & 1 - 2q_1^2 - 2q_3^2 & 2(q_2 q_3 - q_1 q_0) \\ 2(q_1 q_3 - q_2 q_0) & 2(q_2 q_3 + q_1 q_0) & 1 - 2q_1^2 - 2q_2^2 \end{bmatrix} \tag{3}$$

In addition, I utilize for convenience the 3-2-1 Euler angle sequence, defined as a yaw rotation ψ about $\hat{D}$, a pitch rotation θ about $\hat{E}'$, where $\hat{E}'$ is the rotated $\hat{E}$, and a roll rotation ϕ about

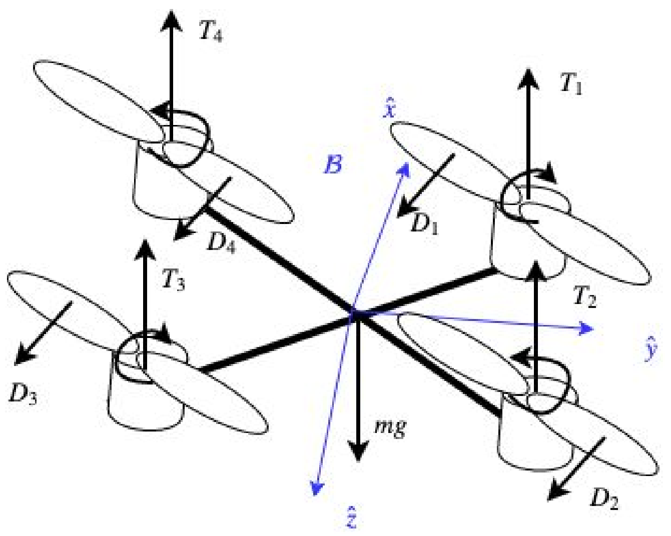


Figure 3: Free Body Diagram

$\hat{x} = \hat{N}''$. Euler angle representation is used only for intuition for data output, and is given by

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \mathrm{atan2}(2(q_0q_1 + q_2q_3), 1 - 2(q_1^2 + q_2^2)) \\ \mathrm{asin}(2(q_0q_2 - q_3q_q) \\ \mathrm{atan2}(2(q_0q_3 + q_1q_2), 1 - 2(q_2^2 + q_3^2)) \end{bmatrix} \tag{4}$$

where atan2 is the two argument arctangent function.

2.2 Force Modeling

The quadrotor system is modeled as a 3-dimensional rigid body. When modeling the forces acting on the body, I made the following assumptions:

1. Due to the quadrotor's size, inertial forces dominate viscous drag, and so the plant model includes no damping.
2. The only aerodynamic forces considered are the motor thrust and propeller drag.
3. The mass of the the motor casing and propellers is negligible in comparison to the overall quadrotor mass, and so the motor moment of inertia I_m is treated as zero.
4. The quadrotor center of mass is at the geometric center of the symmetric quadrotor. While this is likely inaccurate, an offset center of mass can be modeled as a disturbance torque.

Figure 2 shows the resulting model. The three forces considered are gravity, the motor thrust forces, and the propeller drag torque. The gravity force is given by

$$\mathbf{F_g} = m\mathbf{g} \tag{5}$$

and is assumed to act at the geometric center of the body in the direction $\hat{D}$, and is thus decoupled from the attitude model. To model the motor thrust forces, a quadratic relationship between propeller angular velocity, which is derived in [1], is assumed:

$$T_i = k\omega_i^2 \tag{6}$$

Practical modeling of the motor thrust is difficult to perform due to unsteady flows and propeller deformations that are complicated to model, so implementation is performed empirically as described in section 6.2. Propeller drag force can also be described as a quadratic function of angular velocity, as in

$$D_i = b\omega_i^2 \tag{7}$$

as derived in [1].

2.3 Equations of Motion

The rotational and translation equations of motion can be decoupled. I start the derivation with the Newton-Euler equations of motion

$$\begin{aligned} \mathbf{F} &= m\mathbf{a} \\ \boldsymbol{\tau} &= \mathbf{I}\dot{\omega} + \omega \times \mathbf{I}\omega \end{aligned} \tag{8}$$

where $\mathbf{I}$ is the moment of inertia matrix. The motor torque acting in the body frame is given by

$$\boldsymbol{\tau_m} = \begin{bmatrix} Lk(w_1^2 + w_2^2 - w_3^2 - w_4^2) \\ Lk(w_1^2 - w_2^2 - w_3^2 + w_4^2) \\ Lb(-w_1^2 + w_2^2 - w_3^2 + w_4^2) \end{bmatrix} \tag{9}$$

The motor force acting in the body frame is given by

$$\boldsymbol{F_m} = \begin{bmatrix} 0 \\ 0 \\ k(w_1^2 + w_2^2 + w_3^2 + w_4^2) \end{bmatrix} \tag{10}$$

where L is the distance from rotor to the x and y axes. The complete equations of motion are given by

$$\begin{bmatrix} \dot{q} \\ \dot{\omega} \\ \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\mathbf{q} \otimes \omega \\ \mathrm{I}^{-1}(\tau - \omega \times \mathrm{I}\omega) \\ v \\ \mathbf{q} \otimes F_m \otimes \mathbf{q}^* + \mathbf{g} \end{bmatrix} \tag{11}$$

3 Simulation

The aim of the simulator was to test the control laws in advance before implementing in hardware. The simulator is also used to perform model verification. The simulator is written as a script in MATLAB.

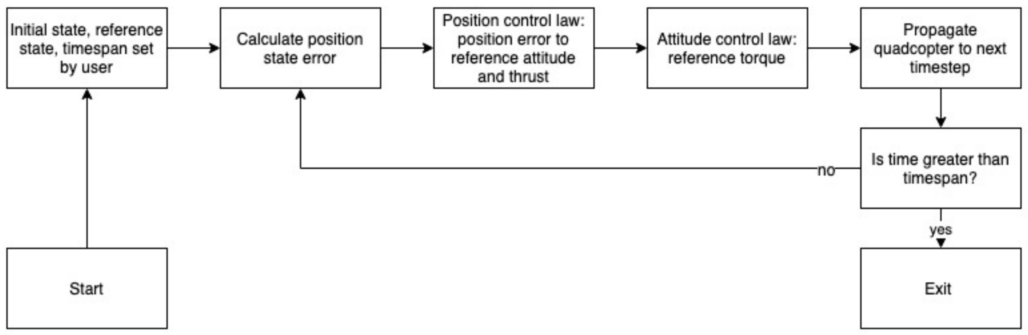

Initial state, reference state, timespan set by user
Calculate position state error
Position control law: position error to reference attitude and thrust
Attitude control law: reference torque
Propagate quadcopter to next timestep
Is time greater than timespan?
no
yes
Start
Exit

Figure 4: Simulator High Level Architecture

3.1 Simulator Architecture

Figure 3 shows the high level architecture of the simulator. The user sets the initial conditions, quadcopter properties, and integration timespan. The controller zero order hold is simulated by the timestep size, which is set to realistic values of the controller loop time. The controller logic is identical to the real controller, with several key exceptions.

1. The simulator assumes perfect attitude knowledge and does not currently model the attitude observer or attitude noise.
2. The simulator utilizes torque and thrust as the input parameters to the propagator, and does not incorporate the decomposition of thrust and torque to motor angular velocity.

Performance of the simulator will be discussed in section 7.

3.2 Propagator

The main difficulty in implementing the attitude simulator was choice of a numerical integrator. My first choice was to integrate the equations of motion for one timestep using MATLAB's `ode45`. This proved numerically unstable due to the unit norm constraint on the attitude quaternion. Some in the literature have resolved this by implementing an RK45 and normalizing the quaternion every timestep, but I chose to utilize a different approach which explicitly encodes the unit norm constraint. I followed the approach in [2] in implementing a quaternion variational integrator. the algorithm as derived in the paper is implemented as in Algorithm 1. A variation is taken on the intermediate quaternion

$$f_k = q_k q_{k+1} \tag{12}$$

such that the variation explicitly obeys the unit norm constraint. This can be converted into the unconstrained parameterization

$$f_k = \frac{\phi_k}{\sqrt{1 - \phi_k \cdot \phi_k}} \tag{13}$$

This parameterization is only valid assuming that the rotation for any timestep k is not greater than π. This can be enforced by using an arbitrarily small integration timestep.

Algorithm 1: Quaternion Integration

```
Result: $q_{t+1}$ given $q_t$,$\omega$,$\tau$,h
while $t_0 < t < t_f$ do
    $p_k = I\omega$;
    $\phi_0 = \frac{h}{2}\omega$;
    $f_0 = [1, 1, 1]^T$;
    while $f > tol$ do
        $f(\phi) = \sqrt{1 - \phi \cdot \phi} \cdot I \cdot \phi + \phi \times I\phi - p_k \frac{h}{2} + \tau \frac{h^2}{2}$;
        $f'(\phi) = \frac{2}{h}(\sqrt{1 - \phi^T \phi})I - \frac{I\phi\phi^T}{\sqrt{1-\phi^T\phi}} + \mathrm{crs}(\phi)I - \mathrm{crs}(I\phi)$;
        $\phi_{n+1} = \phi_n - f'(\phi)^{-1} f(\phi)$
    end
    $f_k = \begin{bmatrix} \sqrt{1 - \phi^T \phi} & \phi \end{bmatrix}$;
    $q_{t+1} = q_t f_k$;
    $w_{t+1} = I^{-1}(\frac{2}{h}\sqrt{1 - \phi^T \phi} I\phi + \phi \times I\phi + h\tau)$
end
```

Attitude Estimation	Quaternion EKF
Attitude Rate Estimation	Gyroscope
Altitude Estimation	LIDAR

Table 2: Attitude Estimation Summary

Algorithm 1 summarizes this approach. the parameter ϕ is solved using a Newton method, which displays good convergence within a few iterations. The distinction between the integration timestep and the simulation timestep is important; the integration timestep should be chosen to reduce integration error, while the simulation timestep should be representative of the system loop time, to accurately model the zero order hold. The position integration is trivial and is therefore not explicitly derived here, but is accomplished by extending the variational integrator.

4 Estimator Design

As complete state estimation was out of the project's scope, I constrained the estimator to measure attitude, attitude rates, and altitude. Discussion of further state estimation is in section 8. The design is summarized in Table 1.

4.1 Attitude Estimator

I decided to utilize the quaternion for attitude estimation for several reasons. First, quaternions do not exhibit singularities at large angles, and are a comparatively minimal representation to SO(3). Second, calculation of the body rates does not require differentiation and can be taken directly from the gyroscope.
Attitude estimation is performed through measurement of Earth's gravitational field and magnetic

field in the body frame. It is well known that two inertial vectors is sufficient to calculate the body attitude. The gravitational vector $\hat{g}$ is assumed to point in the $\hat{D}$ direction, while the magnetic field vector $\hat{m}$ is assumed to point in the $\hat{N}$ direction. Given an ideal acceleration signal $\mathbf{a}$ comprising of only gravity in the body frame, and a magnetometer signal $\mathbf{m}$, the problem is then reduced to solving for $\mathbf{q}$ in

$$\begin{aligned} R(\mathbf{q})\mathbf{g} &= \mathbf{a} \\ R(\mathbf{q})\mathbf{m} &= \mathbf{h} \end{aligned} \tag{14}$$

where $\mathbf{h}$ is the global magnetic field vector. Possible solutions to this overdetermined system are discussed below.

4.1.1 Quaternion Estimator

There are many popular solutions to (14), also known as Wahba's problem. To design my estimator, I looked at three solutions: the Quaternion Estimator (QUEST), the Algebraic Quaternion Algorithm, and the Factored Quaternion Algorithm (FQA).

The QUEST algorithm [3] determines the optimal rotation quaternion given a series of weighted vector observations. However, this solution is often overcomplicated for this problem, because for a quadcopter, the roll and pitch estimation ought to be invariant on the yaw estimation. Thus, I chose not to implement QUEST.

The Algebraic Quaternion Algorithm [4] produces an analytic formula for the attitude quaternion given acceleration and magnetometer observations. It solves Wahba's problem by assuming pitch and roll information are obtained only using the accelerometer data, and heading information is only obtained by the magnetometer data. This results in the following analytical formula for $\mathbf{q}_{acc}$, the pitch and roll quaternion,

$$\begin{aligned} \mathbf{q}_{acc} &= \begin{cases} \begin{bmatrix} \lambda_1 & -\frac{a_y}{2\lambda_1} & \frac{a_x}{2\lambda_1} & 0 \end{bmatrix}, a_z \geq 0 \\ \begin{bmatrix} -\frac{a_y}{2\lambda_2} & \lambda_2 & 0 & \frac{a_x}{2\lambda_2} \end{bmatrix}, a_z < 0 \end{cases} \\ \lambda_1 &= \sqrt{\frac{a_z + 1}{2}} \\ \lambda_2 &= \sqrt{\frac{1 - a_z}{2}} \end{aligned} \tag{15}$$

The two representations are used to avoid the singularity. Similarly, the heading quaternion $\mathbf{q}_{mag}$ is written as

$$\mathbf{q}_{mag} = \begin{cases} \begin{bmatrix} \frac{\sqrt{\Gamma + l_x\sqrt{\Gamma}}}{\sqrt{2\Gamma}} & 0 & 0 & \frac{l_y}{\sqrt{2}\sqrt{\Gamma + l_x\sqrt{\Gamma}}} \end{bmatrix}, l_x \geq 0 \\ \begin{bmatrix} \frac{l_y}{\sqrt{2}\sqrt{\Gamma - l_x\sqrt{\Gamma}}} & 0 & 0 & \frac{\sqrt{\Gamma - l_x\sqrt{\Gamma}}}{\sqrt{2\Gamma}} \end{bmatrix}, l_x < 0 \end{cases} \tag{16}$$

where

$$\begin{aligned} \mathbf{l} &= \mathbf{R^T}(\mathbf{q}_{mag})\mathbf{m} \\ \Gamma &= l_x^2 + l_y^2 \end{aligned} \tag{17}$$

Finally,

$$\mathbf{q} = \mathbf{q}_{acc} \otimes \mathbf{q}_{mag} \tag{18}$$

This parameterization has several benefits. First, the measurement vector is considered to be the true measurement, acceleration and magnetometer measurements, and the Jacobian of the quaternion estimate with respect to the measurement can be explicitly computed for use in the Kalman filter. Second, the algebraic formulation reduces computation time. However, while this parameterization was successful for me for $\mathbf{q}_{acc}$, but not for $\mathbf{q}_{mag}$, which in retrospect may have been caused by improper magnetometer calibration. Thus, I attempted method 3.
the Factored Quaternion Algorithm (FQA) [5] is a method that constructs the estimate quaternion from computed Euler angles. While this does suffer from singularities, the paper gives parameterizations that avoid these problems. The elevation quaternion is computed as

$$\begin{aligned} q_e &= \begin{bmatrix}\cos\frac{\theta}{2} & 0 & \sin\frac{\theta}{2} & 0\end{bmatrix} \\ \sin\theta &= a_x \end{aligned} \tag{19}$$

The roll quaternion is computed as

$$\begin{aligned} q_r &= \begin{bmatrix}\cos\frac{\phi}{2} & \sin\frac{\phi}{2} & 0 & 0\end{bmatrix} \\ \sin\phi &= \frac{-a_y}{\cos\theta} \\ \cos\phi &= \frac{-a_z}{\cos\theta} \end{aligned} \tag{20}$$

Finally, the azimuth quaternion is computed by rotating the magnetic field vector by the pitch and roll angles. Given a normalized magnetic field reference vector $n = \begin{bmatrix}n_x & n_y & n_z\end{bmatrix}^T$ and a rotated magnetic field measurement vector $m_e = q_e q_r m_b q_r^{-1} q_e^{-1}$, the problem is simplified to solving the system of equations

$$\begin{bmatrix}n_x \\ n_y\end{bmatrix} = \begin{bmatrix}\cos\psi & -\sin\psi \\ \sin\psi & \cos\psi\end{bmatrix}\begin{bmatrix}m_{ex} \\ m_{ey}\end{bmatrix} \tag{21}$$

The azimuth quaternion is then constructed as

$$q_a = \begin{bmatrix}\cos\frac{\psi}{2} & 0 & 0 & \sin\frac{\psi}{2}\end{bmatrix} \tag{22}$$

Finally, the estimate quaternion is computed by

$$q_{est} = q_a q_e q_r \tag{23}$$

I utilized FQA over the algebraic quaternion method not due to a specific benefit, but only because FQA worked first. Performance of the estimator is discussed further in section 7.

4.1.2 Extended Kalman Filter

Data from the magnetometer and accelerometer form an estimated quaternion, which can be optimally combined with the gyroscope information using an EKF to create a better estimate than gyroscope integration or two-vector estimation alone. There are generally two different architectures for a quaternion EKF. In both cases, the process model [6] gives the projected quaternion given the current quaternion and gyroscope measurement as

$$\dot{q} = \Omega(\omega)q \tag{24}$$

where

$$\Omega(\omega) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{25}$$

The discrete time form of equation (24) is given by

$$q_t = \exp(\Omega_{t-1}\Delta t))q_{t-1} + w_k \tag{26}$$

where w_k is the process noise. Equation 26 can be linearized to get

$$\bar{q_t} = (\mathrm{I}_{4x4} + \Omega(\omega)\Delta t)q_{t-1}^T + w_k \tag{27}$$

Assuming the process noise is solely a function of gyroscope noise, and assuming gyroscope noise can be modeled as $\omega = \bar{\omega} + \delta\omega$, where $\bar{\omega}$ is the true angular velocity and $\delta\omega$ is the gyroscope noise, the process noise matrix Q_t can be written as

$$Q_t = \frac{\Delta t^2}{4} G_t \Sigma_g G_t^T \tag{28}$$

where

$$G_t = \begin{bmatrix} q_1 & q_2 & q_3 \\ -q_0 & q_3 & -q_2 \\ -q_3 & -q_0 & q_1 \\ q_2 & -q_1 & -q_0 \end{bmatrix} \tag{29}$$

and Σ_g is the zero-mean white Gaussian noise covariance. The measurement model is treated in different ways in the literature. Some define the measurement vector $z = \begin{bmatrix} a_x & a_y & a_z & m_x & m_y & m_z \end{bmatrix}$ and define analytically $h(q) = z$ by using a parameterization that is amenable to such an analytic approach. The benefit of this approach is that the measurement uncertainty can be directly estimated from the noise characteristics of the accelerometer and magnetometer. For example, [4] explicitly derives the measurement covariance $\Sigma_q = J\Sigma_z J^T$, where $J = \frac{\partial q}{\partial z}$. The downside of this approach is difficulty in implementation and computational cost of computing this Jacobian. The alternate approach is to consider the measurement $z = q_{est}$, and thus the measurement equation is just

$$z = I_{4x4} q \tag{30}$$

which does not require linearization. The measurement noise Σ_z is just estimated empirically, which has disadvantages. This approach assumes a constant measurement uncertainty throughout the state space, which is in many cases is a bad assumption due to external accelerations and large rotations of the magnetic field vector. However, experiment has shown this simplifying assumption to be reasonable in the case of a quadcopter, which does not experience large pitch and roll angles often, and does not exhibit large unmodeled external accelerations. Figure 4 displays the high level architecture of the EKF. An intermediate measurement quaternion q_{est} is produced which is then used for the update, in comparison to a traditional EKF which would utilize $z = \begin{bmatrix} a & m \end{bmatrix}$ as the measurement. Implementation of the EKF is discussed in section 6.3 and performance of this EKF is discussed in section 7.

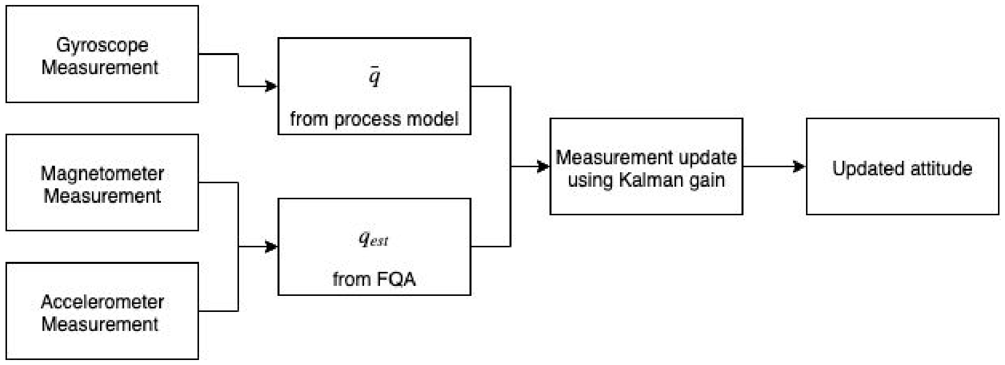

Gyroscope Measurement
$\bar{q}$ from process model
Magnetometer Measurement
Accelerometer Measurement
q_{est} from FQA
Measurement update using Kalman gain
Updated attitude

Figure 5: Extended Kalman Filter Architecture

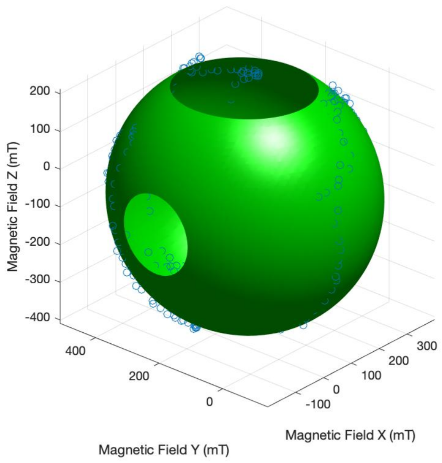
4.2 Altitude Estimator

While I considered utilizing a Kalman filter to profess my altitude state, I decided that this would introduce more noise than using the raw altitude signal provided from the LIDAR. Because the EKF would utilize the attitude estimate, it would compound the error, and accelerations induced by change in velocity were often close to the noise floor of the sensors and would further introduce noise into the measurement. Additionally, the high sample rate of the LIDAR gives generally good altitude information at every timestep. Performance is discussed in section 7.

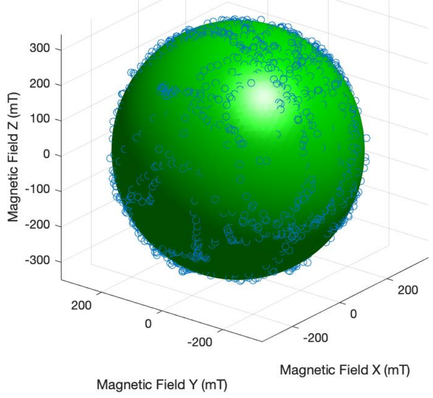
4.3 Implementation and Performance

4.3.1 Attitude Estimator

The attitude estimator was implemented utilizing an InvenSense MPU-9250 9-axis inertial measurement unit. Communication was performed using I^2C protocol. More details on IO pin control is found in Section 6.
The accelerometer is configured to the ± 2g setting, because a high resolution is desired. Bias removal was difficult, because bias determination required a level surface and I was without an electronic level. Discussion of possible effects of accelerometer bias is in section 7.
To calibrate the gyroscope, the stationary bias signal was subtracted away. For this chip, the bias was determined to be $\omega_{bias} = \begin{bmatrix} 0.0043 & 0.0185 & -0.0064 \end{bmatrix}$ rad/s. While this bias is small, it incurs an appreciable error on the EKF.
Magnetometer calibration was more involved. Magnetometers are subject to a number of disturbances, which can be classified into 'hard iron' and 'soft iron' offsets. Hard iron offsets involve magnetic fields other than the Earth's that the magnetometer senses. This could be magnetic fields induced on the IMU chip itself, or fields in the quadrotor motors and power supply. Soft iron offsets are offsets created by the interaction of Earth's magnetic field with objects in the environment, distorting the field. Ideally, with no distortions, points measured by the magnetometer should lie on a sphere of radius h, the strength of Earth's magnetic field at that location. Hard iron offsets have the effect of moving the center of the sphere from the origin, while soft iron offsets transform the sphere into an ellipsoid. The problem of magnetometer calibration is simply a transformation



(a) Uncalibrated Magnetometer Data



(b) Calibrated Magnetometer Data

Figure 6: Ellipsoidal fit and calibration of magnetometer data

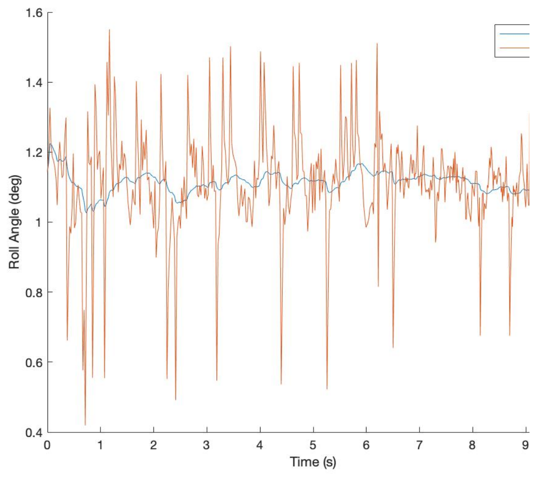
of this ellipsoid into an origin-centered sphere. Assuming the ellipsoid is described by equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{31}$$

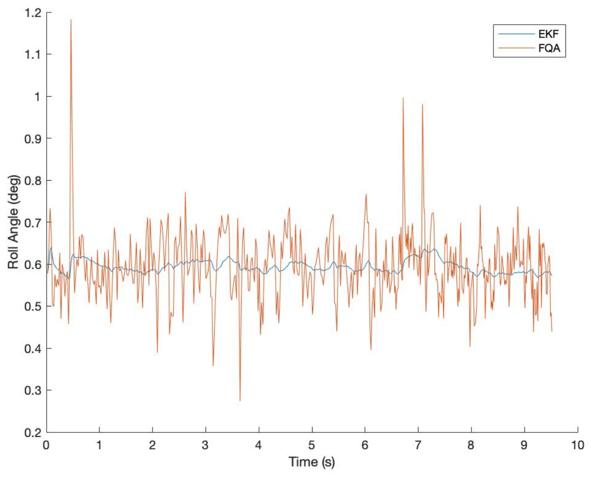
The volume-preserving transformation into a sphere is

$$(u, v, w) = \frac{1}{(abc)^{\frac{2}{3}}}(bcx, acy, abz) \tag{32}$$

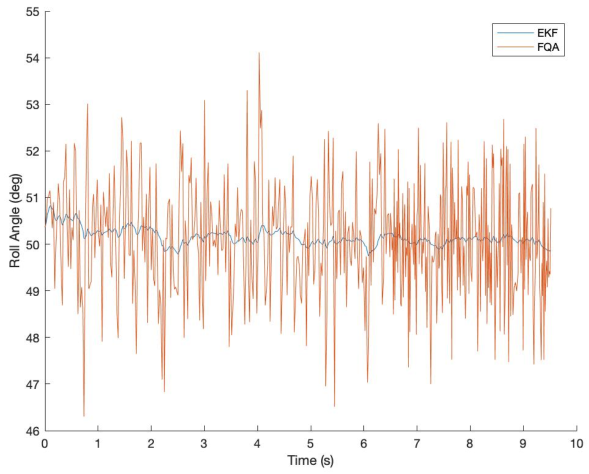
Figure 5 displays magnetometer data before and after calibration. The ellipsoidal fit was accomplished using a least-squares fit in MATLAB. It is clear immediately that in the testing environment, soft iron offsets were negligible, as the raw ellipsoid is actually approximately spherical already. However, hard iron offsets are significant, and are resolved in the calibrated picture. Without magnetometer calibration, yaw estimation is severely inaccurate.
The attitude estimator is successful in certain circumstances. Figure 6 shows data taken with the IMU held static. While these tests do not demonstrate the EKF's effectiveness compared to ground truth, as I did not have access to a gimbal setup with pose measurement, it allows comparison of the EKF to the direct measurement. It is clear that the considerable static noise introduced by the accelerometer and magnetometer is effectively rejected by the EKF. Figure 7 displays how the EKF performs in a dynamic environment. In general, the EKF tracks the measurement while rejecting noise. Additionally, figure 7 (b) demonstrates the EKF rejecting measurement disturbances created by external acceleration. The peaks of the FQA curve are induced by the acceleration at the peak of the motion, which are successfully rejected by the EKF. However, large external accelerations still induce severe disturbances. Figure 8 is the result of shaking the IMU without changing its orientation. The acceleration norm is seen in figure (b), which oscillates around 1g. It is clear that the EKF is not sufficient to reject disturbances of this magnitude and frequency. Luckily, these



(a) Roll Angle

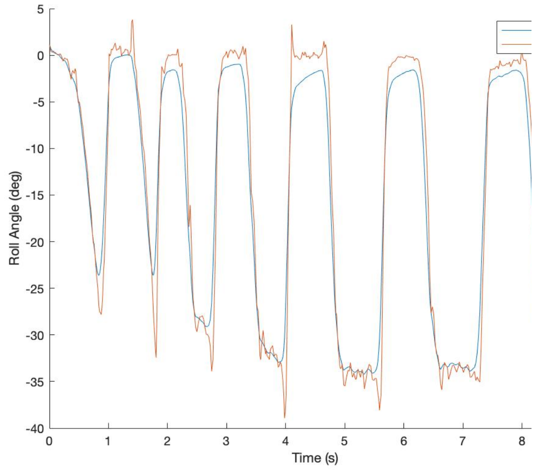


(b) Pitch Angle

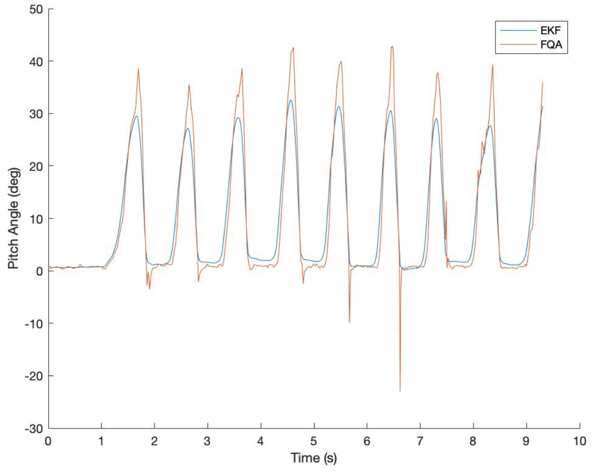


(c) Yaw Angle

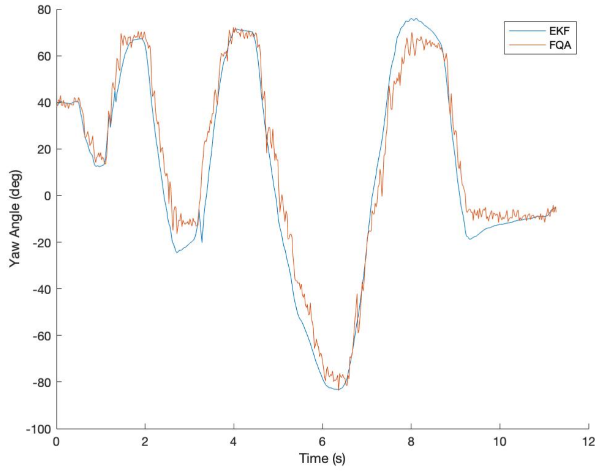
Figure 7: EKF vs FQA Estimates, static case



(a) Roll Angle

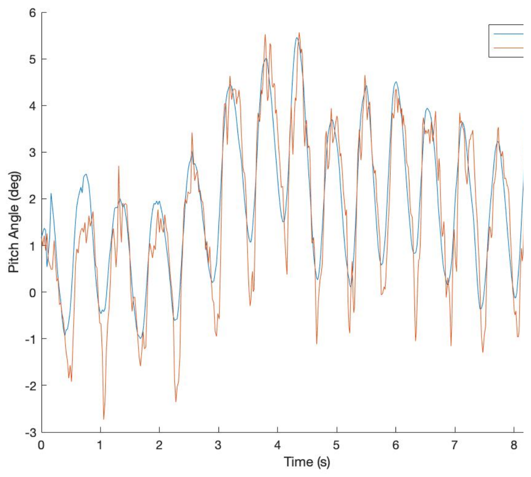


(b) Pitch Angle

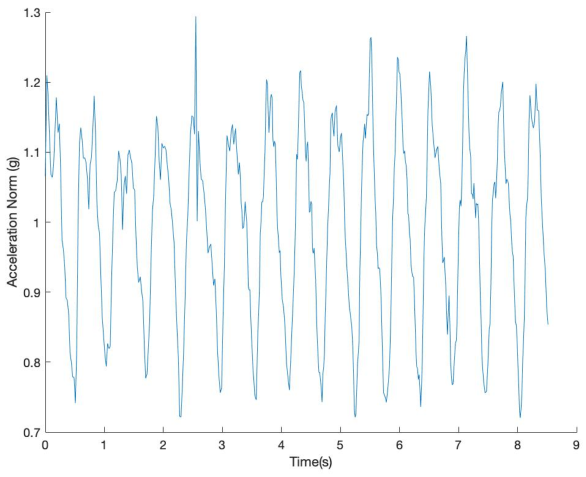


(c) Yaw Angle

Figure 8: EKF vs FQA Estimates, dynamic case



(a) Pitch Angle



(b) Accelerometer Norm

Figure 9: Pure external accelerations induce substantial disturbances

types of disturbances are generally not seen in quadrotors. However, I discuss this problem and potential solutions in section 8.

4.3.2 Altitude Estimator

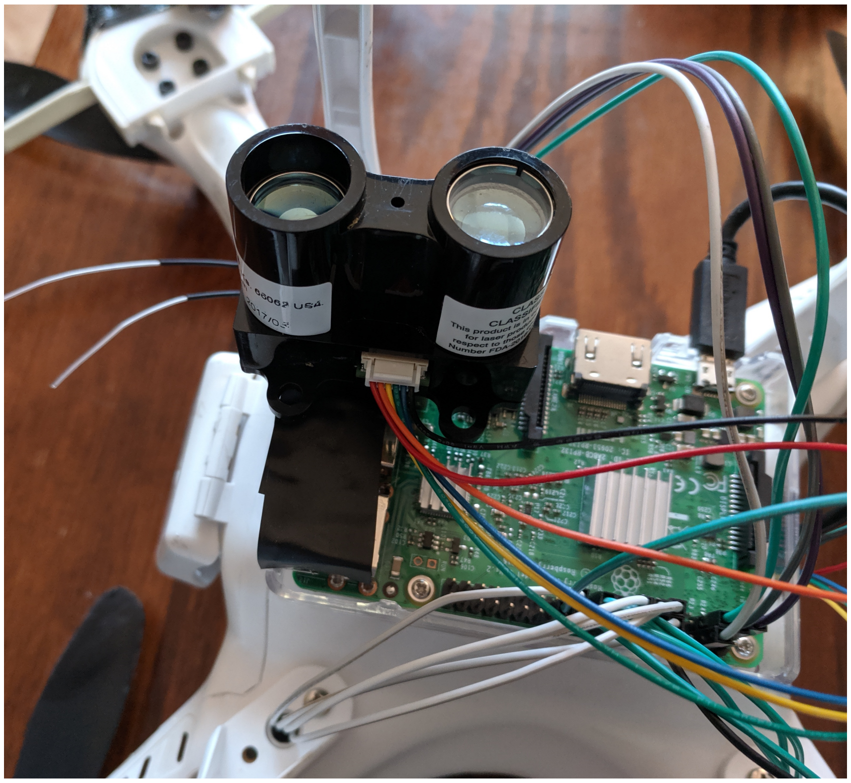


Figure 10: Implementation of altitude estimator

The altitude estimator, shown in Figuree 10, consists of a single LIDAR Lite rangefinder mounted to the bottom of the quadrotor. The sensor returns a pulse width which is proportional to the time of flight of the laser beam. I did have difficulties in implementation, as the I2C output of this sensor is not compatible with the Raspberry Pi 3. This motivated the choice of using PWM. The sensor's update rate is significantly faster than the loop time and thus imposes minimal time delay, and is relatively noise free when pointed at a flat surface.

5 Controller Design

5.1 Problem Setup

A quadcopter controller must regulate the attitude of the quadcopter, expressed in any parameterization, using four input thrust vectors. The system input, while often described as $u = \begin{bmatrix}\omega_{m1} & \omega_{m2} & \omega_{m3} & \omega_{m4}\end{bmatrix}$, is more conveniently expressed as $u = \tau$, the torque on the body frame. The motivation for this choice is discussed later.

The most common approach to designing a quadcopter controller in the literature is a Proportional-

Integral-Derivative (PID) controller. The first control law I attempted to implement was a PID on the Euler angles, with a control law

$$u = K_p\boldsymbol{\theta} + K_d\dot{\boldsymbol{\theta}} + K_i\int_0^t \boldsymbol{\theta}dt \tag{33}$$

While this has been successful in the literature, there are several issues with this method. First, parameterization using Euler angles has the disadvantages described earlier. Second, the PID requires tuning 9 parameters which are co-dependent, which can be an arduous task. Finally, the PID gives no guarantees on optimality. As a result, I have chosen to reframe the problem as an optimal controls problem.

5.2 Linear Quadratic Regulator (LQR)

A Linear Quadratic Regulator (LQR) is a control law designed to produce the optimal gain matrix K in the law

$$u = -Kx \tag{34}$$

given the linear system with state $\mathbf{x}$, output $\mathbf{y}$, and control input $\mathbf{u}$

$$\begin{aligned}\mathbf{x} &= \mathbf{Ax} + \mathbf{Bu}\\ \mathbf{y} &= \mathbf{Cx} + \mathbf{Du}\end{aligned} \tag{35}$$

This is performed by minimizing the cost function

$$J(\mathbf{x},\mathbf{u}) = \int_0^\infty \mathbf{x^TQx} + \mathbf{u^TRu}dt \tag{36}$$

by decomposing the gain matrix $\mathbf{K} = -\mathbf{R^{-1}B^TS}$, this can be reframed as a continuous-time algebraic Ricatti equation

$$0 = \mathbf{SA} + \mathbf{A^TS} - \mathbf{SBR^{-1}B^TS} + \mathbf{Q} \tag{37}$$

5.3 Controller Architecture

Figure 5 displays the final architecture of the controller. Because the attitude and position dynamics are decoupled, it makes sense to control them in separate loops. Additionally, if the position controller fails, the result is not always catastrophic, but if the attitude controller fails, it is much more problematic. This approach enables a faster loop time for the attitude controller, and prevents potentially costly position measurements from impacting the attitude loop time. For each position and attitude, there is an LQR gain matrix and integrator, which are summed to produce the control input. The position LQR produces an input u_{pos} consisting of a vector encoding both the reference quaternion q_{ref} and input thrust T_{ref}. This feeds into the attitude loop, containing both an LQR gain and integrator, which results in a reference torque, which is decomposed into the motor angular velocities. Specific design is discussed below.

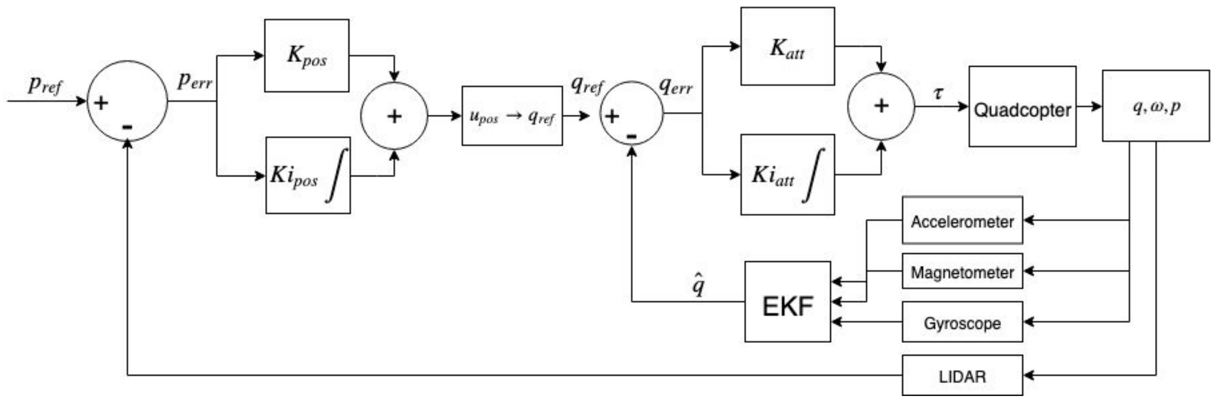


Figure 11: Controller Block Diagram

5.4 Attitude Controller

5.4.1 Error Linearization

In order to utilize the analysis in section 5.2, the nonlinear attitude equations in (11) must be linearized. The linearized state dynamics equation, with $\mathbf{x} = \begin{bmatrix} q & \omega \end{bmatrix}^T$, yields

$$A = \begin{bmatrix} \frac{\partial \dot{q}}{\partial q} & \frac{\partial \dot{q}}{\partial \omega} \\ 0 & \frac{\partial \dot{\omega}}{\partial \omega} \end{bmatrix} \tag{38}$$

The partial derivatives are given by

$$\frac{\partial \dot{q}}{\partial q} = \frac{1}{2} \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{39}$$

$$\frac{\partial \dot{q}}{\partial \omega} = \frac{1}{2} \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_2 \\ -q_2 & q_1 & q_1 \end{bmatrix} \tag{40}$$

$$\frac{\partial \dot{\omega}}{\partial \omega} = \omega^{\times} - I^{-1} \omega^{\times} I \tag{41}$$

With the control input $u = \tau$,

$$B = \begin{bmatrix} 0_{4x3} \\ I^{-1} \end{bmatrix} \tag{42}$$

Next, in order to control to an arbitrary reference point, it is necessary to control the attitude error. The attitude error, expressed as a quaternion, is given by

$$q_e = q^{-1} \otimes q_r \tag{43}$$

and angular velocity error given by

$$\omega_e = \omega_r - \omega \tag{44}$$

where q_r is the reference attitude quaternion and ω_r is the reference angular velocity. Due to the fact that the reference attitude is considered static, $\dot{q} = \dot{q_e}$, and thus the linearized error dynamics are equivalent to the linearized dynamics.

5.4.2 Ricatti Solution

As described in section 5.2, in order to solve for the gain matrix **K**, equation (37) must be solved. While there are numerous methods to solve the Continuous-time Algebraic Ricatti Equation (CARE), I chose the method described in [7], which utilizes the Schur decomposition. The method considers the Hamiltonian matrix

$$Z = \begin{bmatrix} A & -BRB^T \\ -Q & -A^T \end{bmatrix} \tag{45}$$

The Schur decomposition then produces a linear transformation U such that Z maps to real Schur form (RSF)

$$U^T Z U = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} \tag{46}$$

Additionally, the matrix U is arranged such that the real parts of the eigenvalues of S_{11} are negative and the real parts of S_{22} are positive. Then, the U can be partitioned into

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} \tag{47}$$

The matrix S is then given by solving the linear system

$$SU_{11} = U_{21} \tag{48}$$

Finally,

$$K = -R^{-1}B^T S \tag{49}$$

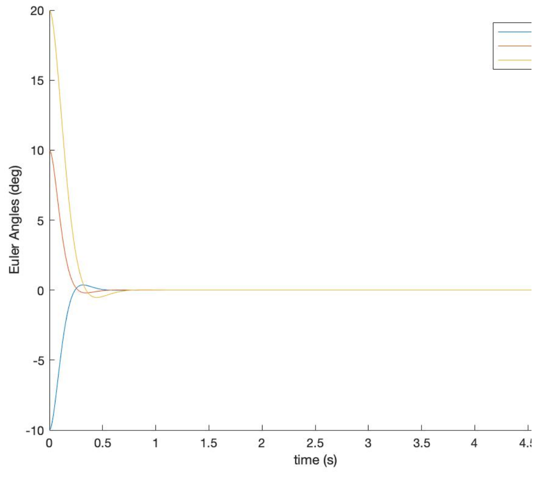
I chose this method for its speed and numerical stability, properties demonstrated by the [7].

5.4.3 Controllability

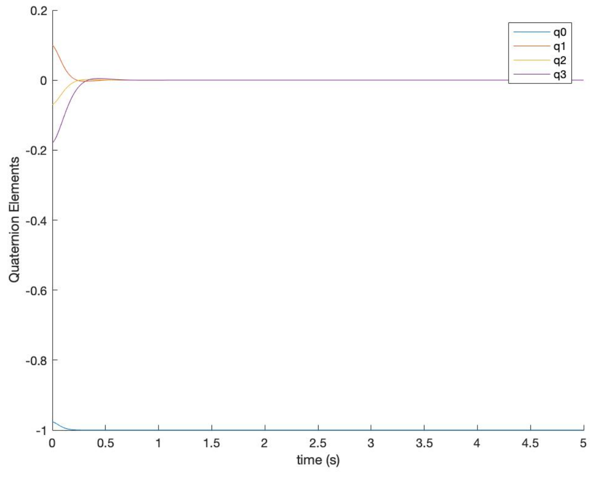
Unfortunately, the system described in 5.4.1 is not controllable, a fact that can be easily seen by examining the controllability Gramian

$$C = \begin{bmatrix} B & AB & A^2B \dots A^{n-1}B \end{bmatrix} \tag{50}$$

which is not full rank in the given system. This is caused by the fact that the quaternion space double covers SO(3). This is resolved in much of the literature by considering the reduced state space $x = \begin{bmatrix} q_{1:3} & \omega \end{bmatrix}$ utilizing the unit norm constraint to remove the quaternion real part. However, this approach exhibits variable stability throughout the state space, and is actually uncontrollable when the quaternion rotation angle $\theta = \pi$, a fact proven by [8]. The authors of that work propose



(a) Euler Angles



(b) Quaternion Elements

Figure 12: Step response in roll, pitch and yaw. Stability to hover is demonstrated.

an alternate solution; the addition of a virtual control input u_a renders the non-controllable mode controllable. This involves the creation of the augmented system

$$\begin{aligned} A_a &= A \\ B_a &= \begin{bmatrix} B & \begin{bmatrix} q \\ 0 \end{bmatrix} \end{bmatrix} \end{aligned} \tag{51}$$

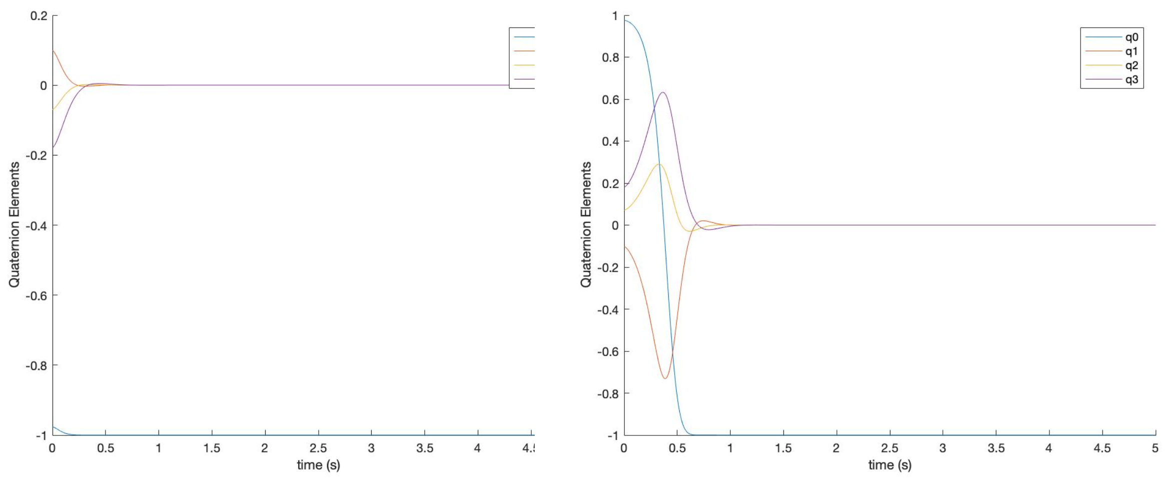
This guarantees the controllability as full rank. It makes intuitive sense to place the virtual control input in the direction of the quaternion, as this is the direct that the original model exhibits no control authority. This control input is simply ignored in the final dynamics.

5.4.4 Simulation Performance

The above control law was simulated using representative values for quadcopter properties, as discussed in 6.2. In this case, I used weighting matrices of $Q = 100C^TC$ and $R = 0.1I_{4x4}$. Figure 10 displays a representative step response in both Euler angles and quaternion elements. Clearly, the LQR successfully stabilizes the quadcopter about the reference point, while minimizing overshoot and ringing, which can be tuned by changing Q and R.

5.4.5 Global Stability Questions

The use of quaternions does introduce some unfavorable dynamics into the system. As explained previously, the quaternion double covers the space of orientations and SO(3). Thus, there are two quaternions that correspond to hover with no yaw, $q = \begin{bmatrix} \pm 1 & 0 & 0 & 0 \end{bmatrix}$. This results in two problems. First, while the LQR renders both of these equilibrium points in the closed loop system, stability for both is not guaranteed. In fact, only one of these equilibrium points will be stable. In



(a) Initial condition close to equilibrium point (b) Initial condition far from equilibrium point

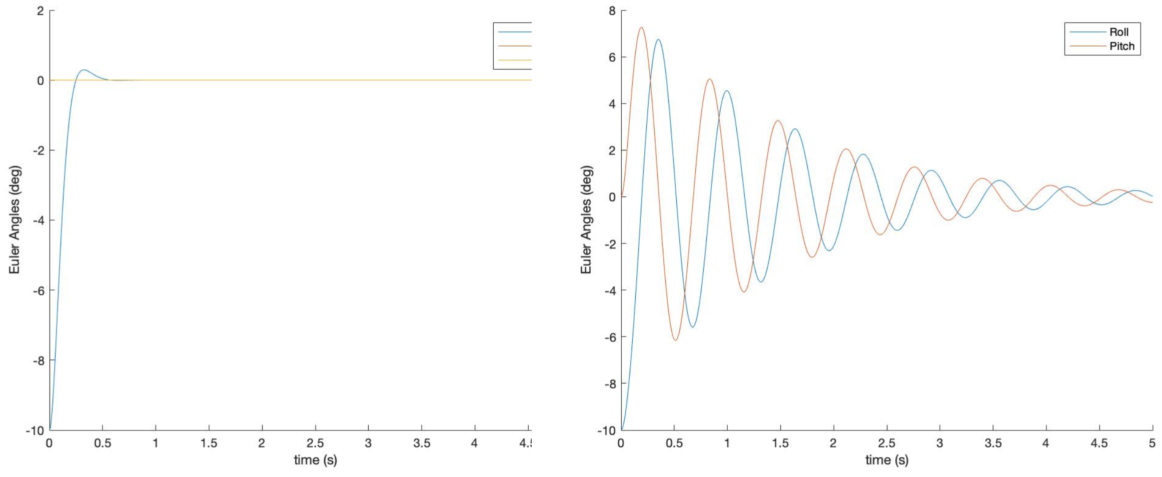
Figure 13: Unstable dynamics demonstrated by step response in quaternion space

this case, the stable equilibrium point is $q = \begin{bmatrix}-1 & 0 & 0 & 0\end{bmatrix}$. While the proof of this is beyond the scope of this work, it can be shown through numerical integration, as in Figure 11. While both of the initial orientations were identical in Euler space, they represented opposite quaternions, and it is clear that the second case jumped from a position around the unstable equilibrium to the stable equilibrium. This is demonstrated most clearly by q_3, which moves from 1 to -1. I resolved this issue in the simulation by ensuring the initial condition is always in the correct 'hemisphere' of quaternion space, which means choosing the sign of q_0 correctly. In the quaternion estimator, this requires forcing the estimated quaternion to always have negative real part, thus preventing this instability.

Another issue that quaternions introduce is more prohibitive. Under large rotations, the linearization becomes gradually less accurate, to the point of introducing significant instability. The impetus for this discussion is that my initial design does not control for yaw, as I wanted to ensure pitch and roll stability first. However, the performance proved extremely variable dependent on yaw angle, and under large yaw angles often failed completely. Figure 11 displays this behavior. With zero yaw, the control input is entirely in the roll direction, as expected. With nonzero yaw, the control input disturbs the pitch direction as well, resulting in significant ringing. It is important to note that the equilibrium state is still globally stable, but the path is substantially sub-optimal. This is an issue I was unable to resolve, but fixed by simply disregarding yaw information in the implemented controller.

5.4.6 Torque to Input Transformation

The input to the system was chosen as torque over rotor angular velocity because it was difficult to force the output of the LQR to be strictly positive. Thus, a transformation from torque to rotor thrust must be determined. This intuitive transformation was implemented only for τ_x,τ_y, and



(a) $\psi = 0$ (b) $\psi = 120^o$

Figure 14: Instability introduced by large yaw, as demonstrated by roll step response

throttle, but can be easily extended to include τ_z. This approach considers four independent cases depending on the signs of τ_x and τ_y. For example, in the case where $\tau_x > 0$ and $\tau_y > 0$, then only motors 1,3,and 4, as notated in Figure 3, will need to contribute to the total torque. If, in this case, $\tau_y > \tau_x$, then motor 4 is expected to generate $(\tau_x + \tau_y)/2$, and motor 1 is expected to generate $\tau_y/2 - \tau_x/2$. This produces the correct torque, and can be extended to all cases. Thus, only two rotors are required to produce any given τ_x and τ_y. Then, the required thrust is simply calculated by subtracting the desired thrust from the thrust required to generate the reference torque, and is distributed equally across each rotor. The full algorithm is presented in Algorithm 2.

5.5 Position Controller

The original intention of the project was to control all degrees of freedom of the quadrotor, including position. However, due to complications implementing GPS in the testing space, I chose to only implement altitude control. Full position control is still implemented in the simulator, and is thus presented here.

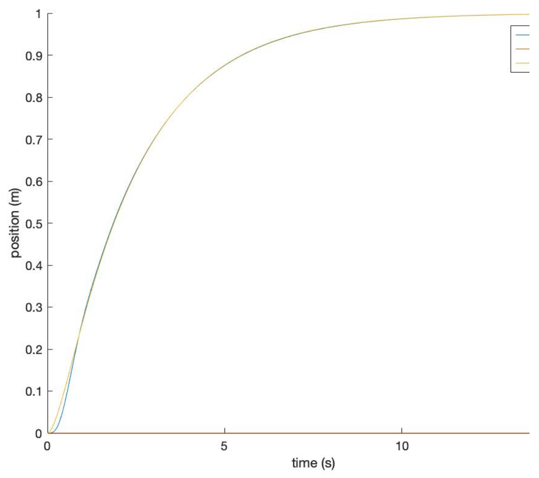
5.5.1 Linearization

Unlike the attitude controller, which is relinearized every step, the position controller is not. Thus, the linearization, expressed through the error dynamics $p_e = p_r - p$ and $v_e = v_r - v$, is simply

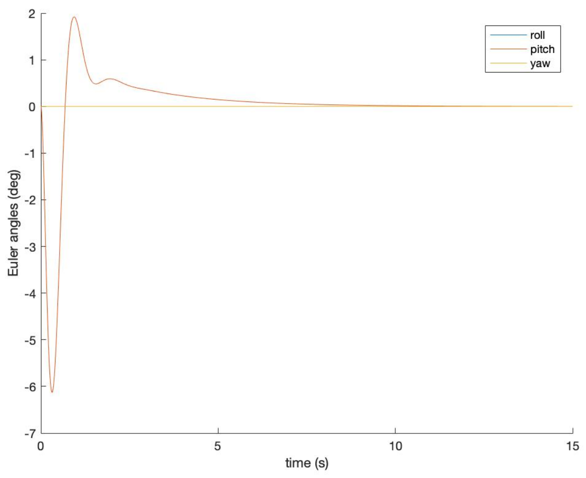
$$A = \begin{bmatrix} 0_{3x3} & I_{3x3} \\ 0_{3x3} & 0_{3x3} \end{bmatrix} B = \begin{bmatrix} 0_{3x3} \\ \frac{1}{m} I_{3x3} \end{bmatrix} \tag{52}$$

Algorithm 2: Input torque and thrust to input pulse width transformation

```
Result: PW1,PW2,PW3,PW4 given $\tau_x$,$\tau_y$,$T_{ref}$,$k$,$l$,PW0
Initialize PW1=0,PW2=0,PW3=0,PW4=0;
if $\tau_x >= 0 \;\; \tau_y >= 0$ then
    PW4 = PW4+$(\tau_x + \tau_y)$/(2kl);
    if $\tau_x - klPW4 > 0$ then
        PW3 = PW3+($\tau_x$−klPW4)/(kl)
    else
        PW1 = PW3+($\tau_y$−klPW3)/(kl)
    end
else if $\tau_x >= 0 \;\; \tau_y < 0$ then
    PW3 = PW3+$(\tau_x - \tau_y)$/(2kl);
    if $-\tau_x - klPW3 > 0$ then
        PW4 = PW4+($\tau_x$−klPW3)/(kl)
    else
        PW2 = PW2+($-\tau_y$−klPW3)/(kl)
    end
else if $\tau_x < 0 \;\; \tau_y >= 0$ then
    PW1 = P1+($-\tau_x$+$\tau_y$)/(2kl);
    if $-\tau_x - klPW3 > 0$ then
        PW2 = PW2+($-\tau_x$−klPW1)/(kl)
    else
        PW4 = PW4+($\tau_y$−klPW1)/(kl)
    end
else if $\tau_x < 0 \;\; \tau_y < 0$ then
    PW2 = P2+$(-\tau_x - \tau_y)$/(2kl);
    if $-\tau_x - klPW3 > 0$ then
        PW1 = PW1+($-\tau_x$−klPW2)/(kl)
    else
        PW3 = PW3+($-\tau_y$−klPW2)/(kl)
    end
$T_{res} = T_{ref} - k(PW1 + PW2 + PW3 + PW4)$;
if $T_{res}$¿0 then
    PW1 = PW1+$T_{res}/4$;
    PW2 = PW2+$T_{res}/4$;
    PW3 = PW3+$T_{res}/4$;
    PW4 = PW4+$T_{res}/4$;
end
```



(a) Position



(b) Euler Angles

Figure 15: Example position control step response from [0,0,0] to [1,0,1]

The input vector thrust vector u_p then must be decomposed into an input thrust and reference quaternion. This transformation was computed by [9]. The reference quaternion is computed as

$$\begin{aligned} q'_r &= (\hat{z} \cdot u_p + ||u_p||) + \hat{z} \times u_p \\ q_r &= \frac{q'_r}{||q'_r||} \end{aligned} \tag{53}$$

The reference thrust is computed simply by $\hat{z}||u_p||m$.

5.5.2 Position Control Performance

Position control, unlike attitude control, is not sensitive to rise time, but is highly sensitive to overshoot. For example, when landing, it is desirable for the quadrotor to have zero vertical velocity upon touchdown. To accomplish this, the closed loop eigenvalues of the position system must all have zero imaginary parts. While there is no analytical method to accomplish this, it is possible by simply weighting the velocity components sufficiently in the Q matrix. As is clear from Figure 13(a), the closed loop poles are heavily damped and exhibit zero overshoot. For this application, the large rise time is acceptable. Figure 13(b) shows the resulting control orientation, which approaches level hover in steady state.

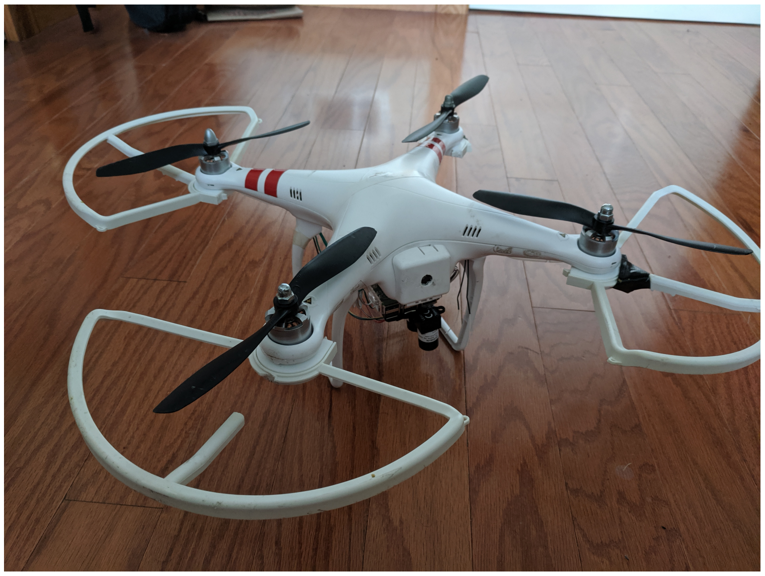


Figure 16: Quadrotor Setup

6 Implementation

In this section, I will discuss implementation of these algorithms both in hardware and software.

6.1 Hardware Setup

The quadrotor setup, as pictured in Figure 14, consists of a stripped down DJI Phantom 1 frame and custom electronics. As described in the introduction, the only parts original to the Phantom 1 are the frame, motors, speed controllers, and power distribution board. The high level wiring diagram is shown in figure 15. The main hardware components are the motors, electronic speed controllers (ESCs), battery, power distribution board (PDB), reciever, IMU, LIDAR, and computation unit. Implementation of these subsystems are discussed below.

6.1.1 Computation

Computation is performed by a Raspberry Pi Model 3, as pictured in figure 18. The Pi runs the Unix-based operating system Raspbian. I chose the Raspberry Pi because they are cheap, easy to implement, and can communicate over WiFi, precluding a complicated communications protocol. Utilizing a non real time operating system is considered dangerous for two main reasons. First, there are no hardware timed GPIO pins on the Raspberry Pi, and so precision timing is difficult. In particular, the Pi must generate precise PWM waveforms, and measure PWM waveforms from the LIDAR and reciever. Second, it is possible for the operating system to interrupt the flight controller, which is undesirable. I resolved the first issue through the library PIGPIO, which runs a daemon to simulate hardware timed PWM on software timed pins. In practice, this enables accuracy to around $\pm 6\mu s$, which is sufficient for measuring PWM signals on the order of milliseconds. While

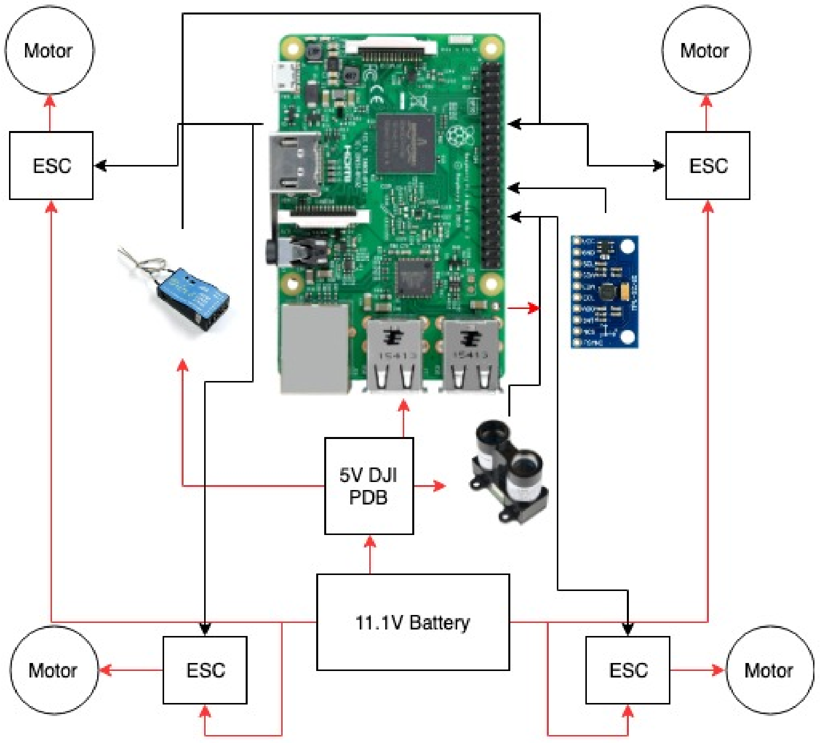


Figure 17: Quadrotor Setup

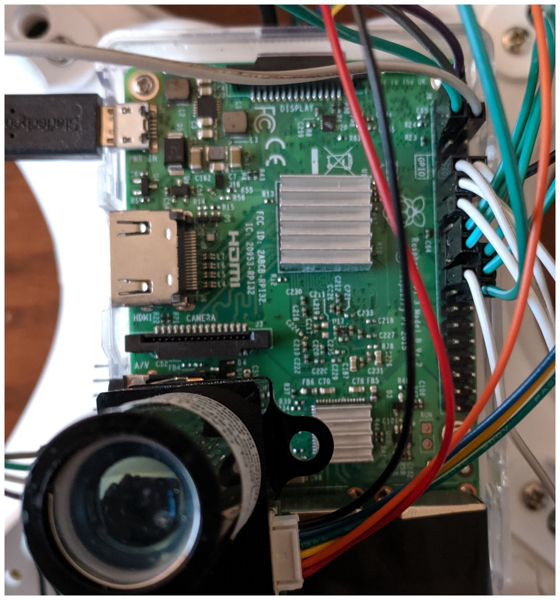


Figure 18: Raspberry Pi onboard computation

I did not explore the effects of the operating system on the computation, I did not see significant interruptions during flight tests.
The computer communicates to the quadrotor components through the GPIO pins. Table 3 shows the usage of each pin on the Pi, by CPU pin number (Broadcom).

Function	GPIO Pin # (BCM)
RECIEVER CHANNEL 1	17
RECIEVER CHANNEL 2	27
RECIEVER CHANNEL 3	22
RECIEVER CHANNEL 4	18
RECIEVER CHANNEL 5	11
MOTOR 1	10
MOTOR 2	9
MOTOR 3	25
MOTOR 4	11
LIDAR	7
IMU SDA	2
IMU SCL	3

Table 3: Raspberry Pi GPIO Pin Usage

6.1.2 Propulsion

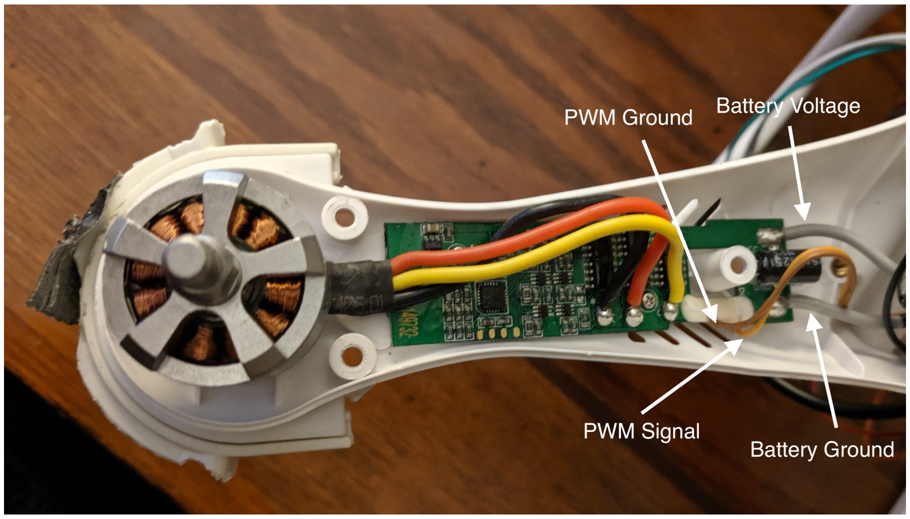


Figure 19: Propulsion system components

The propulsion subsystem consists of the motors and ESCs. Both of these components are leftover from the original DJI Phantom frame. The motors are given three-phase control signals from the speed controllers. The ESCs are powered directly from the battery voltage, and the control signal is supplied by the Raspberry Pi. The input to the DJI ESCs, found through trial and error due to lack of documentation, is a PWM signal operating at 400 Hz. Zero throttle is a 1ms pulse and full throttle is a 2ms pulse. The PWM duty cycle is given by

$$DC = \frac{255}{2.5}T \tag{54}$$

Where DC is the supplied duty cycle and T is the desired pulse width in milliseconds. The square wave is generated using the PIGPIO package for the Raspberry Pi.

6.1.3 Communications

There are two relevant communication links. The transmitter, a FrSky Taranis Pro, must communicate with the Raspberry Pi information regarding desired input attitude and throttle, as well as arm/kill information. Additionally, a ground station computer must provide commands and receive data from the quadrotor.

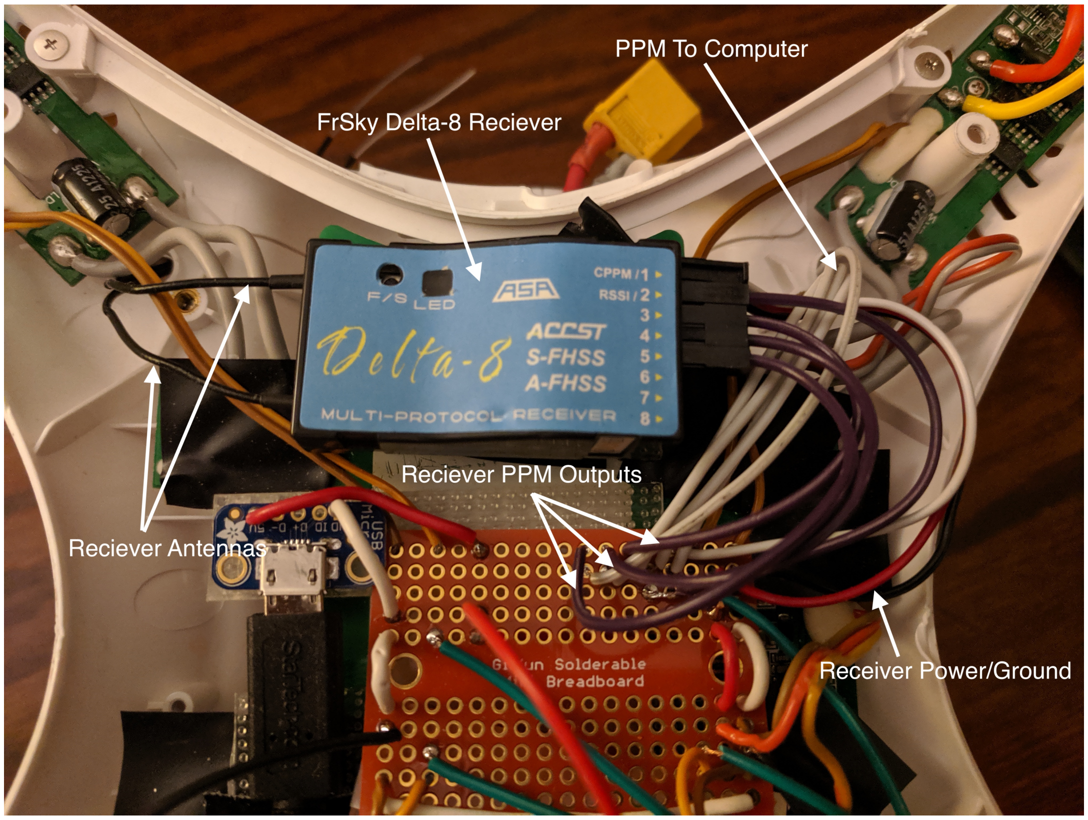


Figure 20: Communications system components

The receiver (FrSky Delta-8) has several possible communication protocols: PPM, CPPM, and RSSI. CPPM, or combined pulse position modulation, is the most compact, as it requires only one signal wire, and each pulse is modulated on one period. However, the code required to read this protocol was more complex, so I decided to utilize only PPM (pulse position modulation), and utilize one signal wire per channel. In this case, the five channels are roll, pitch, yaw, throttle, and arm/kill. Each of the roll, pitch, and yaw sticks gives a pulse position in the range of 1-2ms. The roll and pitch sticks both map to $\pm 10^o$, such that the sticks at the center is 0^o. The arm/kill switch is binary, such that if the channel width is greater than 1.4ms, the command is ARM, and if it is less than 1.4ms, the command is KILL.
In order to measure the transmitter command signals, the pulse widths of each channel had to be measured. I ultimately decided to utilize 'callback functions' in PIGPIO, which function similarly to interrupts. These callback functions are configured to fire on a rising edge or falling edge of the receiver GPIO pins. This algorithm is summarized in Algorithm 2.
The communication between ground station and computer is done through WiFi. The nmap package is utilized to locate the Raspberry Pi network, and ssh is used to communicate. I used sftp to transfer files between the ground station and the Pi.

Algorithm 3: Pulse Width Measurement Algorithm

```
Result: Pulse Width PW
if $level == RISING$ then
    risingtime = tick;
else
    PW = tick-risingtime
end
```

6.1.4 Power

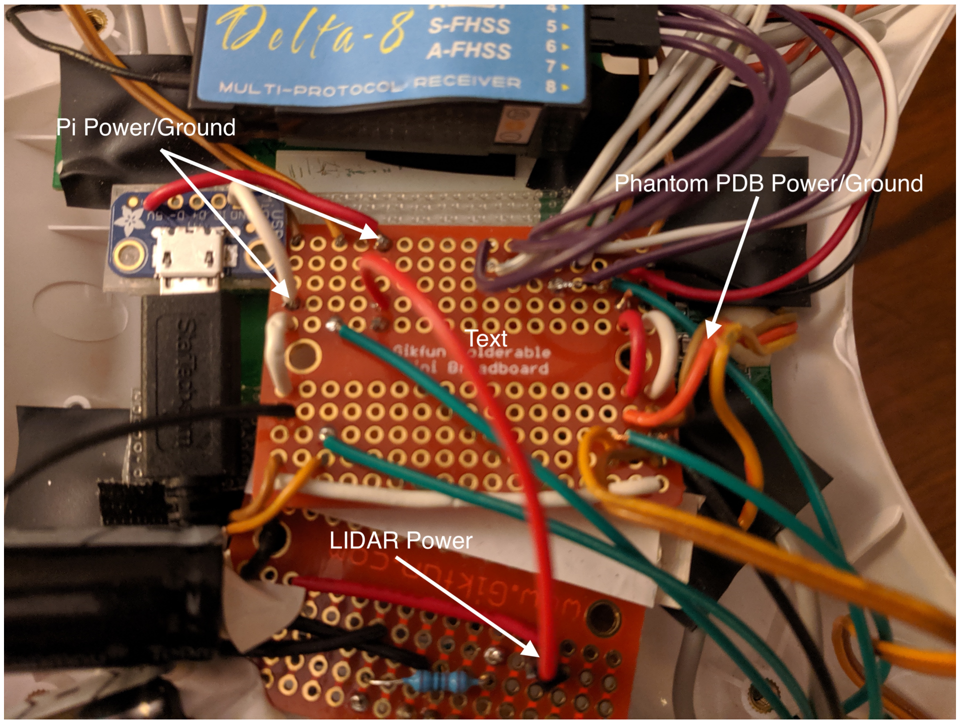


Figure 21: Power system components

The objective of the power subsystem is to ensure each system component receives the correct voltage and sufficient current. The components requiring power are the ESCs, computer, Receiver, LIDAR, and IMU. Implementation is shown in Figure 21. The DJI Phantom contains onboard power distribution for the ESCs, supplying the battery 11.1V. It also contains a 5V power supply circuit, which converts the variable battery voltage into a constant, reliable 5V supply. This is routed into the central breadboard, and distributed to the LIDAR breadboard, receiver, and Pi. One important consideration was how to power the Raspberry Pi; while it is possible to power through the GPIO pins, this provides no protection against current spikes and could damage the Pi. Instead, I utilized a microUSB breakout board and soldered power and ground directly from the

Phantom PDB. Testing showed that the DJI PDB was reliable and relatively noise free. Another concern was the relatively small gauge wire connecting to the DJI PDB, as the ~1A drawn by the Pi could overheat the wires. However, while a larger gauge wire is recommended for this amperage, I did not experience issues.

6.1.5 Sensing

The sensing subsystem consists of the IMU and LIDAR. The IMU, pictured in Figure 22, communicates with the Raspberry Pi through I2C. Reading the IMU data through I2C requires the bulk of the loop time, and so raising the I2C bitrate substantially improved the controller performance. The IMU was generally configured to default state, and the IMU startup routine was as follows:

- Open I2C connection at bus 1 to MPU9250 address
- Restart IMU to default by writing 0x02 to 0x6B
- Calculate accelerometer and gyroscope offsets
- Set MPU9250 magnetometer bypass byte to 1 by reading MPU9250 address 0x37 and writing (0x37—0b00000010) to 0x37, and writing 0x01 to 0x38
- Open I2C connection at bus 1 to AK8960
- sleep 0.1s

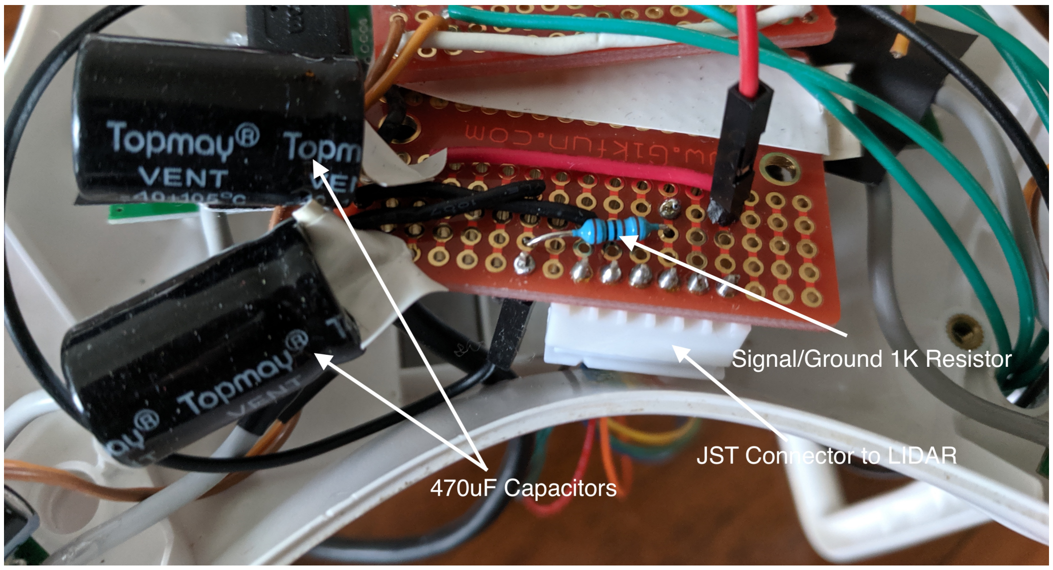


Figure 22: LIDAR circuitry

The LIDAR required extra onboard circuitry to implement, as shown in Figure 22. The LIDAR can draw current unpredictably, so to prevent the power subsystem from being overloaded, two

$470\mu F$ capacitors are placed in parallel across power and ground, effectively low-pass filtering the power supply. As suggested by the datasheet, a 1KΩ resistor is added between signal and ground. Additionally, I constructed a JST connector for the LIDAR, so the sensor could be removed with ease.

6.2 Software Setup

The onboard software is all written in Python, due to its accessibility and my familiarity with it. One potential issue with Python as compared to lower-level languages such as C is speed. However, after porting some system components into C as a test, there was a minimal effect on the loop time, as the majority of the loop is spent on I2C communication with the IMU.
As explained above, the package PIGPIO is used for GPIO control. The flight controller utilizes the Python bindings to the C software. Integration is done in Sublime and pushed to the Pi through sftp.

6.3 System Identification

In order to simulate the quadrotor model, the physical properties of the drone must be found. The relevant constants to be calculated are the motor thrust constant k, the motor drag constant b, the quadrotor mass m, the moment of inertia I.
The mass of the drone was determined by a scale to be 1kg. The moment of inertia matrix was difficult to estimate because of a lack of an air bearing table or accurate CAD models. Thus, the inertia matrix used in [10] was used: $I_{xx} = 0.081kgm^2$,$I_{yy} = 0.081kgm^2$,$I_{zz} = 0.142kgm^2$. The motor thrust constant was found by empirics. Rather than find the thrust as a function of rotor angular velocity, it is found as a function of input pulse width. The experiment was performed by placing the quadrotor on a scale and measuring the change in weight for various input pulse widths. The total thrust was then divided by four to find the thrust for a single rotor. The results are shown in figure 23. The ESCs are clearly designed to have a linear thrust response to input pulse width. The thrust to pulse width relationship is given by the linear fit as

$$T = 5.73PW + 1.16 \tag{55}$$

where the pulse width PW is in ms and T is in kg. The motor drag constant b is more difficult to compute experimentally and theoretical models are highly unreliable, so this property was not found for this project. A potential experiment would involve a motor mounted perpendicular to the scale, with a lever arm pressing against the scale. The motor drag torque would then exert a force on the scale which could be isolated.

6.4 Estimator Integration

Integration of the attitude estimator proved troubling for a few reasons. First, there was not a convenient, level surface to mount the IMU to. It was important to place the IMU close to the center of mass to minimize external accelerations generated by rotation of the quadrotor. Additionally, the surface must minimize vibrations generated by the rotors and minimize possible interference with the magnetometer. I finally chose to place the IMU as shown in figure 18. This placement has several effects. First, there is significantly more vibration in the x and y directions (in the plane of the surface) than the z direction because the ability of the surface to warp in the z direction

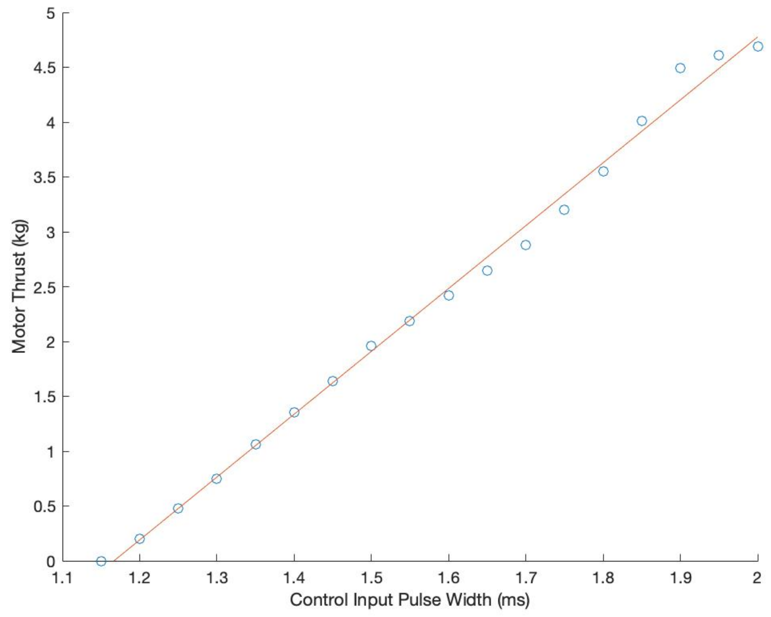


Figure 23: Rotor thrust as a function of input pulse width

functionally acts as a vibration damper. Second, the rotation of the sensor frame with respect to the body frame necessitates that the quaternion measured by the estimator be rotated into the body frame, as

$$q_{adj} = q_{est} \otimes q_{static} \tag{56}$$

where q_{adj} is the adjusted body quaternion, and q_{static} is the static rotation between sensor and body frames, measured to be $q_{static} = [0.57444037, -0.8178944, 0.01441685, 0.02727644]$.
Additionally, the placement of the IMU was not sufficient to damp out rotor vibrations. When ramped up, the motors produced significant noise in the estimated quaternion. This is caused by small imbalances in the rotors producing an external acceleration on the IMU. The solution to this was to low-pass filter the IMU, utilizing the formula for the discrete time low pass filter

$$\hat{a_t} = (1-\alpha)a_{t-1} + \alpha a_t \tag{57}$$

where

$$\alpha = \frac{\Delta t}{2\pi f + \Delta t} \tag{58}$$

and f is the low-pass cutoff frequency.

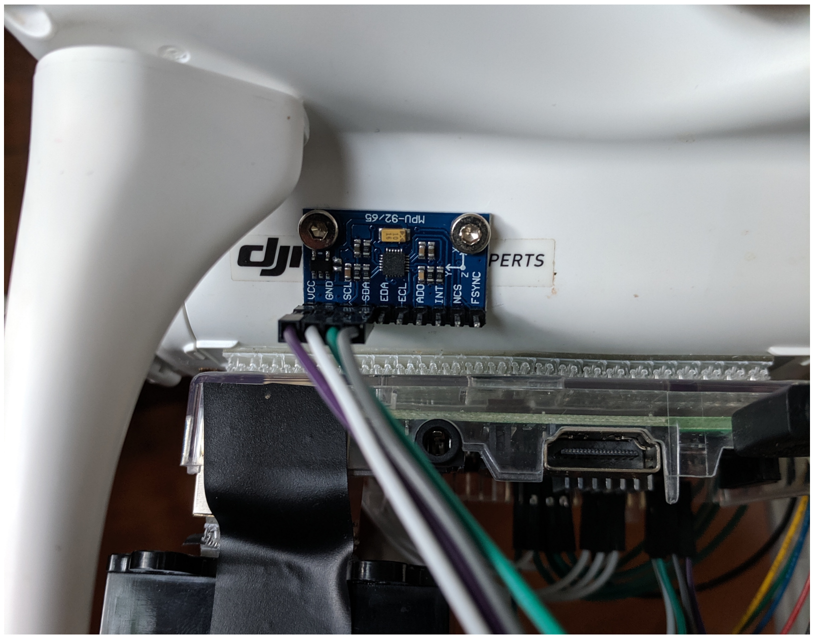


Figure 24: IMU placed on non-level surface on the side of the quadrotor

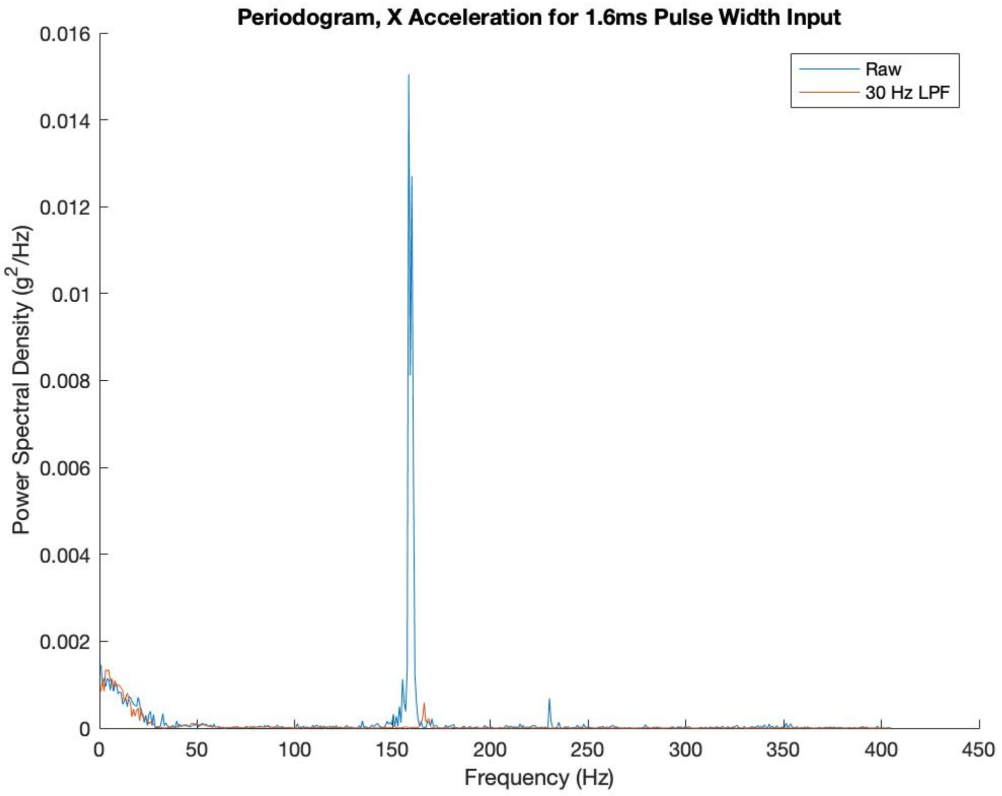


Figure 25: Periodogram of accelerometer x direction, with and without low pass filter

Figure 19 shows a periodogram, plotting the power spectral density as a function of frequency, of the accelerometer, with the rotors running at a 1.6ms PWM input. In order to plot up to 500Hz, I raised the I2c bitrate to 600kbps from the default 100 kbps, in order to achieve a loop speed of around 1KHz. The result shows a very clear peak in the signal, placed exactly at the expected rotor frequency. Thus, by placing a low pass filter with a corner frequency of 30Hz, the noise is all but eliminated, with a small time delay cost. This also proves the useful result that an accelerometer can be used to measure rotor speed relatively accurately, rather than using an expensive tachometer. The low pass filter successfully eliminated most of the EKF noise.
The altitude estimator also required filtering. While the sensor provided relatively noise-free data, the altitude rate was computed through numerical differentiation, and was extremely noisy when applied to the raw data. To resolve this, two low pass filters were utilized; a filter with cutoff frequency of 1Hz for the altitude data, and another filter of cutoff frequency 1.5Hz applied to the resulting velocity data. This produced sufficiently noise-free velocity data.

6.5 Controller Implementation

Safety was a key priority when implementing the flight controller. The key requirements can be summarized as:

- The pilot must have complete ARM/KILL authority authority at all times and must be able to kill the rotors at any time during operation.
- The controller must autokill if the receiver loses connection with the transmitter.
- The controller must autokill if the receiver loses connection with the IMU or LIDAR.
- The controller must autokill if the controller crashes during operation.

This was resolved by using two global variables ARM and AUTOARM. The controller will kill if either of these variables are set to zero. The transmitter arm/kill switch controls the ARM variable, and flight checks control the AUTOARM variable. If the receiver, IMU, or LIDAR loses connection, then AUTOARM is zeroed and the controller self kills. Additionally, if the controller crashes, the script executes code to kill the motors before the program exits. The full flight controller flow diagram is shown in figure

7 Results

In this section, I present data taken from flight tests. For attitude control tests, I controlled only the throttle and arm/kill swtich, and for altitude control tests, I controlled only the arm/kill switch. These tests were performed outdoors, and effects of wind disturbances are present and will be discussed.

7.1 Attitude Control

Tuning the attitude controller proved much easier than my attempts to tune the PID controller. Initially, the only adjustments I had to make were tuning the Q and R matrices. Generally, I held R fixed and adjusted Q. If the gains were too high, the quadrotor would either go visibly unstable or would incur significant ringing. In this case, I would lower Q. If the gains were too low, the quadrotor would act sluggish and drift significantly, in which case I would raise Q. The following analysis was produced with $Q = 100CC^T$ and $R = 0.2I_{4x4}$.
Another issue that was that due to bias in the estimator, the integrator would result in significant drift. With the integrator turned off, the controller did not regulate about the origin, but still hovered relatively well, a phenomenon I was unable to explain.

Figure 16 demonstrates the quadrotor operating at stable hover. The quadrotor took off at approximately 1.5s with a slight initial pitch. The quadrotor regulates the roll and pitch angles around approximately $\pm 4^o$ in both direction. At around 6s, a gust of wind occurred, and the quadrotor successfully rejected this disturbance. The noise present at hover is likely due to several factors. First, slight environmental disturbances, such as wind, result in noise. Second, the attitude estimator introduces noise and time delay. This noise could be attenuated by improving the attitude estimator.

Figure 17 displays an example step response in pitch, as compared to a similar step response in the simulator. While the steady state value is not pictured, it is around 5^o, likely due to estimator bias.

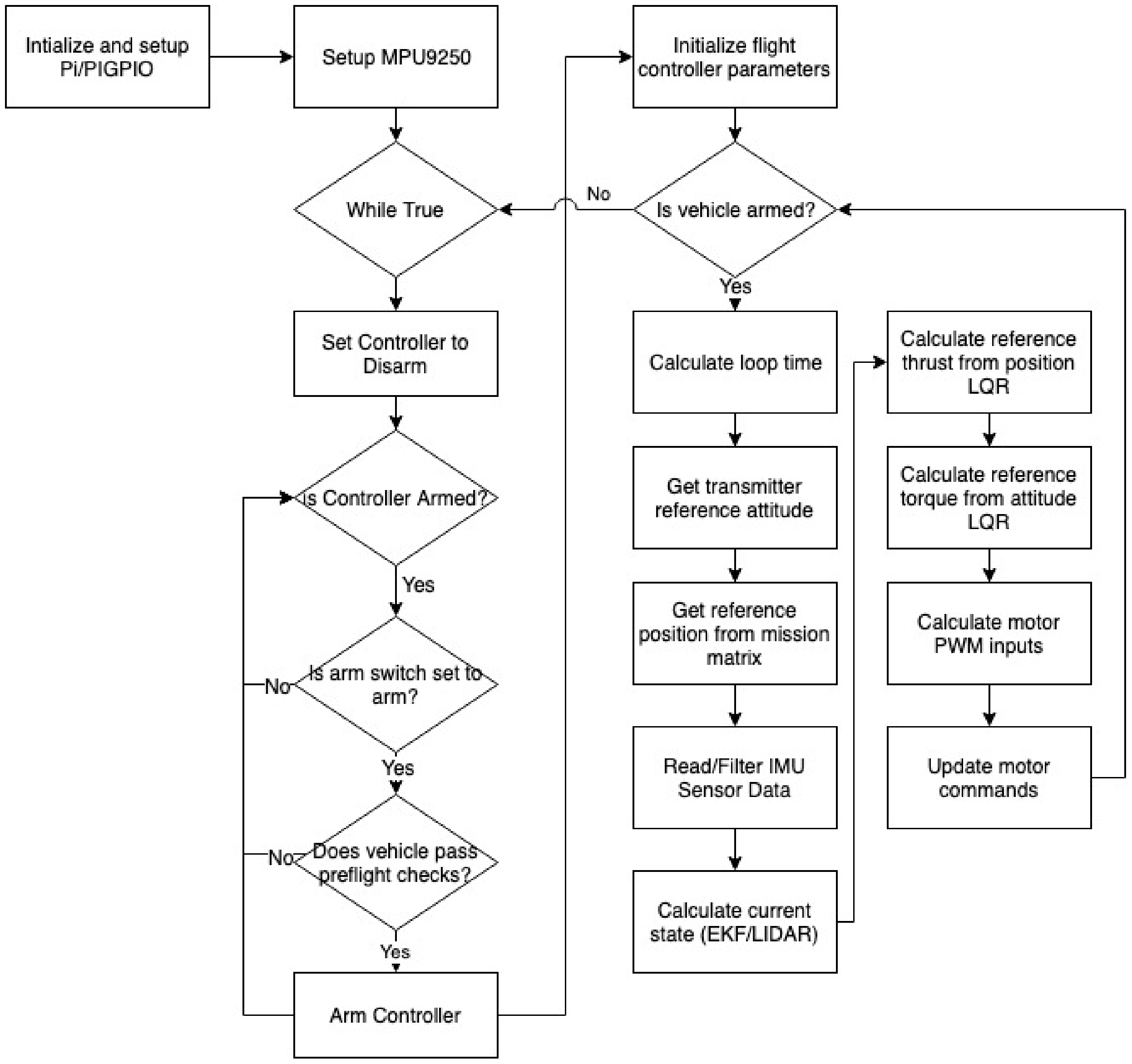

Intialize and setup Pi/PIGPIO
Setup MPU9250
Initialize flight controller parameters
While True
No
Is vehicle armed?
Yes
Set Controller to Disarm
Calculate loop time
Calculate reference thrust from position LQR
Get transmitter reference attitude
Calculate reference torque from attitude LQR
Is Controller Armed?
Yes
Get reference position from mission matrix
Calculate motor PWM inputs
No
Is arm switch set to arm?
Yes
Read/Filter IMU Sensor Data
Update motor commands
No
Does vehicle pass preflight checks?
Yes
Calculate current state (EKF/LIDAR)
Arm Controller

Figure 26: Full flight controller flow diagram

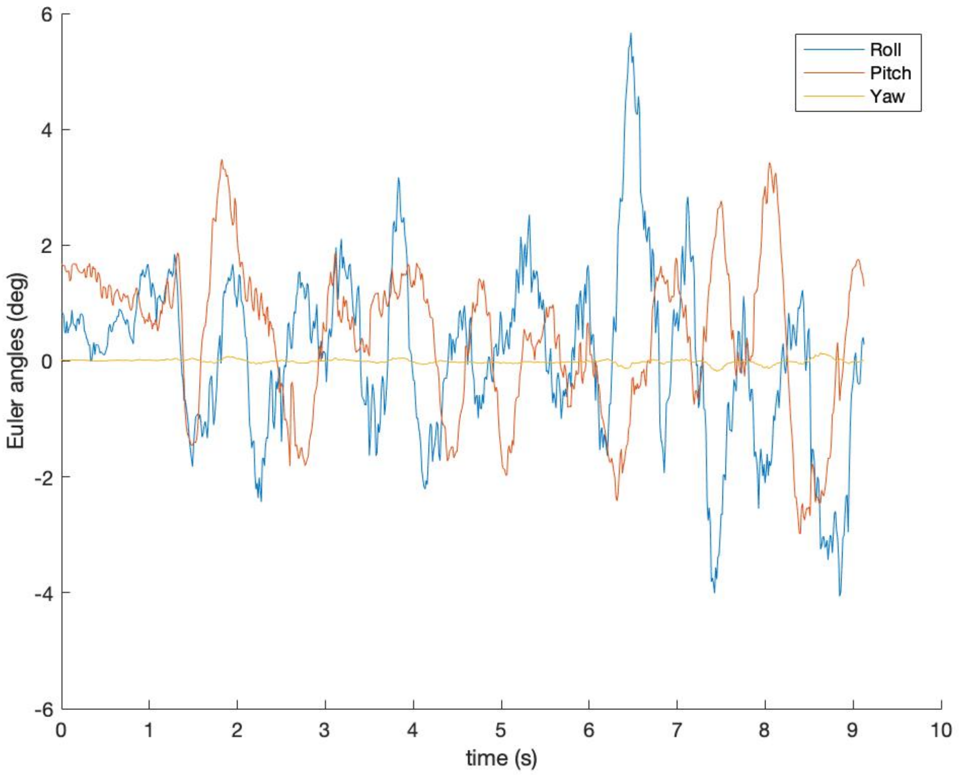
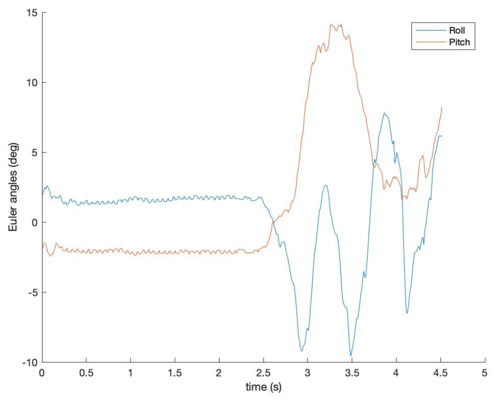
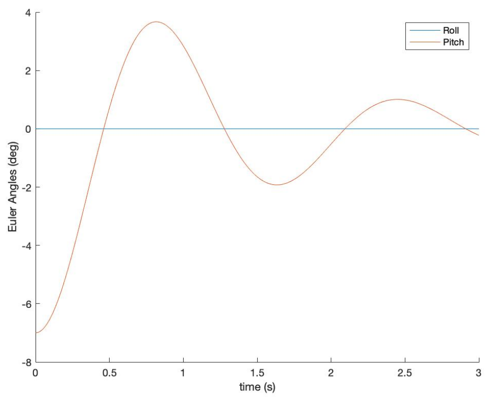


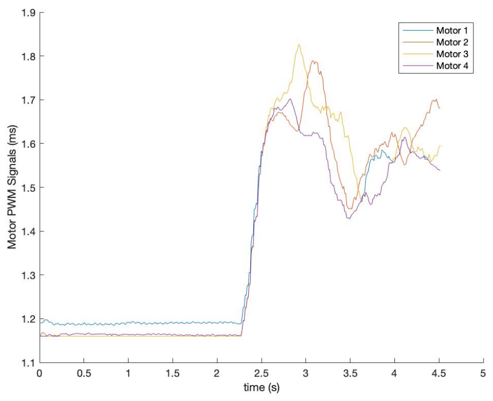
Figure 27: Euler angles, Stable Hover



(a) Step response, Euler angles



(b) Simulated step response



(c) Motor Control Signal

Figure 28: Step response in pitch, as compared to simulated step response

First, a main difference between experiment and simulation is the large disturbance in roll angle, which is likely due to disturbances during takeoff. These disturbances, though not shown in the plot, are damped out eventually, but with a very large settling time. Second, it is clear that there is an undesirably large overshoot of approximately 100%, while the simulator exhibits an overshoot of 48%. The experimental rise time is approximately 0.5 seconds, and the theoretical rise time is approximately 0.5 seconds as well. There is a significant difference in theoretical and experimental overshoots, which is likely due to unmodeled time delay. Time delay results in an increase in phase lag, which increases both overshoot and settling time. Sources of unmodeled time delay include the estimator and zero order hold computation delay. One possible method to model this delay is to utilize the frequency response of the zero order hold:

$$L(s) = \frac{1 - \exp(-sT)}{sT} \tag{59}$$

where T is the zero order hold period, taken as the loop time. Additionally, the estimator could be modeled directly in the simulation. To resolve the issues with overshoot, a more precisely timed compute unit could be used, such as a microprocessor. Additionally, utilizing a faster sensor protocol, such as SPI, would cut down on the loop time. Finally,a less aggressive controller would produce a smaller overshoot.

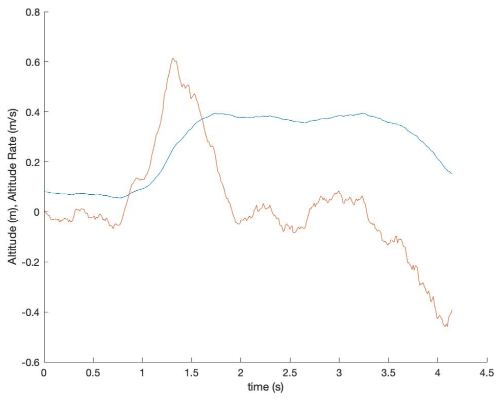
The near identical rise times suggest that the quadcopter model is very reasonable, and that the values for I are near correct. Additionally, it implies that the relationship between control input and motor torque is approximately accurate.

The control inputs for each motor, in PWM (ms), are show in in 17(c). It is clear that the control inputs are never saturated during the step response, but that there is also significant jitter. This would increase the effect of motor slew rate, which is also unmodeled, and would be accounted for in a more accurate model.

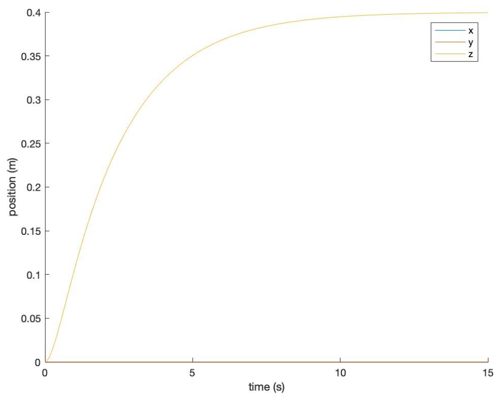
7.2 Altitude Control

The altitude control is implemented as explained in section 6.2. For these tests, the quadrotor is instructed to execute a simple autonomous mission: take off to a certain altitude, hold altitude for a certain time, and land. Performance is compared to the simulator. The tolerance is set to 2cm for the first waypoint and altitude hold, and 15cm for landing. Figure 18 shows an example mission versus a simulated altitude step response. In this mission, the quadrotor was instructed to take off to 0.4m altitude, hold altitude for 2 seconds, then land. It successful completed this trajectory and held altitude to the prescribed tolerance and timespan. The jitter in altitude was surprisingly low, likely due to high quality measurements from the LIDAR. Often, the quadrotor would tip on landing due to uncontrolled horizontal velocities, but when horizontal velocity was low, the quadrotor would land consistently upright.

The altitude step response appears to match the experiment well. Both exhibit zero overshoot, suggesting that if the closed loop poles are sufficiently damped, the added phase lag due to time delay would not cause overshoot. This also allowed the quadrotor to land softly from the 0.15m tolerance. The experimental rise time is 0.64s, compared to a theoretical rise time of 4.44s. These values are extremely different, suggesting an error in the position integrator or incorrect values for motor thrust. I was unfortunately restricted in performing tests for a longer duration due to significant horiziontal position drift and a relatively small testing space. However, this altitude



(a) 2 second altitude hold at 0.4m, altitude and altitude rate



(b) Simulated altitude step response to 0.4m

Figure 29: Experimental altitude hold as compared to simulated altitude step response

control appears capable of performing arbitrarily complex mission absent failure of the attitude control system.

8 Conclusion

The quadrotor is increasingly being used in industry as well as research, because it is a convenient platform for testing control technology on an under-actuated system that exhibits strong nonlinearities. For this project, I successfully designed, simulated, and constructed a quadcopter with an estimator and controller. The key design choices I made were the utilization of a kalman filter for orientation estimation, the use of onboard LQR control, and the use of quaternions for attitude representation. The attitude estimator is successful at producing a quaternion estimate while rejecting disturbances from external accelerations and gyroscope drift. While I initially chose a complementary filter, the Kalman filter more generally handle nonlinear quaternion schemes. The LQR controller is more successful than the PID in terms of tuning, but ultimately suffers from significant nonlinearities in quaternion space, which do not allow direct quaternion feedback with yaw information. In the end, the quadrotor faced several key obstacles that allow for further research.

1. The quadrotor was unable to fly with yaw information.
2. The quadrotor exhibited significant drift in horizontal position due potentially to estimator bias.
3. Estimator noise resulted in significant oscillations during level flight.
4. Performance was highly variable on battery charge.

Further research into nonlinear performance in quaternion space could allow for direct quaternion feedback even in the presence of large rotations. Better modeling of accelerometer and magnetometer noise and bias could resolve position drift and improve estimator noise characteristics. Finally, better system identification, including the power subsystem, could help resolve battery nonlinearities. If I were to repeat this project, I would likely approach the problem differently. First, I would implement the controller on a real time operating system to improve loop time. Second, I would develop a transformation to better linearize the quaternion model. In summary, I have learned an appreciable amount about implementing a real control system, and the challenges associated with hardware.

References

[1] P. Bristeau et al., "The Role of Propeller Aerodynamics in the Model of a Quadrotor UAV", Proceedings of the European Control Conference, 2009.

[2] Z. Manchester and M. Peck, "Quaternion Variational Integrators for Spacecraft Dynamics", Journal of Guidance, Control, and Dynamics, Vol. 39, No. 1, 2016.

[3] F. Markley and D. Mortari, "QUATERNION ATTITUDE ESTIMATION USING VECTOR OBSERVATIONS", Journal of Guidance and Control, 2010.

[4] R. Valenti et al.,"A Linear Kalman Filter for MARG Orientation Estimation Using the Algebraic Quaternion Algorithm", IEEE Transactions on Instrumentation and Measurement, vol. 65, no. 2, 2016.

[5] X. Yun et. al, "A Simplified Quaternion-Based Algorithm for Orientation Estimation From Earth Gravity and Magnetic Field Measurements", IEEE Transactions on Instrumentation and Measurement, vol. 57, no. 3, 2008.

[6] K. Feng et al., "A New Quaternion-Based Kalman Filter for Real-Time Attitude Estimation Using the Two-Step Geometrically-Intuitive Correction Algorithm", Sensors, vol. 17, 2017.

[7] A. Laub, "A Schur Method for Solving Algebraic Ricatti Equations", IEEE Transactions on Automatic Control, Vol. AC-24,no.6, 1979.

[8] L. Lustosa et al., "A new look at the uncontrollable linearized quaternion dynamics with implications to LQR design in underactuated systems", European Control Conference, June 2018.

[9] J. Carino et al., "Quadrotor Quaternion Control", International Conference on Unmanned Aircraft Systems, 2015.

[10] F. Morbidi et al., "Minimum-Energy Path Generation for a Quadrotor UAV", IEEE International Conference on Robotics and Automation, May 2016, Stockholm, Sweden. ICRA16 - International Conference on Robotics and Automation, 2016.