Department of Mechanical Engineering
University of Washington

Stevens Way, Box 352600
Seattle, WA 98195 USA
206-543-5090

ME 495 Report

Michael Janicki
Michael.david.janicki@gmail.com

Karl Kintner-Meyer
kkintnermeyer6@gmail.com

Nicholas Kunst
nicholas.m.kunst@gmail.com

Hallvard Ng
hallvardng@yahoo.com.hk

Phillip Rudolph
phillipprud11@gmail.com

Mechatronics Capstone Design Report — June 8, 2018

Contents

List of Figures

1 Nomenclature

b	Thrust Coefficient $[N\ s^2]$
d	Drag Coefficient $[Nm\ s^2]$
g	Gravity $[9.81\ m/s^2]$
I_x	x-axis Inertia $[kg\ m^2]$
I_y	y-axis Inertia $[kg\ m^2]$
I_z	z-axis Inertia$[kg\ m^2]$
J_r	Motor Rotor Polar Inertia $[kg\ m^2]$
l	Radius Arm Length $[m]$, measured from Center of Motor to Quadcopter Center of Gravity
m	Quadcopter Mass $[kg]$
x	x-axis $[m]$
y	y-axis $[m]$
z	z-axis $[m]$
ϕ	Roll Angle $[rad]$
θ	Pitch Angle $[rad]$
ψ	Yaw Angle $[rad]$
Ω	Motor Angular Velocity $[rad/s]$

2 Executive Summary

This report describes our design process, mathematical model, simulation, physical implementation, testing, and results of building a quadcopter from the ground up. This report compares the theoretical simulation of the quadcopter to the experimental results under the same conditions, justifies any differences found, and ultimately provides considerations on how the design could be improved.

We developed a mathematical model for the dynamic behavior of the quadcopter and designed linear controllers for the attitude and height according to our functional specifications. Functional specifications, models, and controllers can be found according to the table of contents. The controllers and mathematical model were implemented in Simulink and Matlab to simulate the response of the quadcopter based on initial conditions, feedback noise, reference changes, and disturbances.

State estimation and feedback was accomplished by incorporating a Sparkfun 9Dof Stick and a Garmin LidarLite distance sensor. The 9DoF sensor is an IMU sensor that we used to estimate the angular position of the quadcopter, and the distance sensor was used for height feedback. A complementary filter was implemented to combine the accelerometer and gyroscope data from the 9Dof sensor to provide a more accurate estimation of the angles, and to help reduce noise. Mechanical damping was also added between different components of the frame and sensor mounts to additionally reduce noise in the feedback.

Computation, data analysis, and control were accomplished using a C program run on a NI MyRIO attached to the quadcopter. The brushless DC motors were controlled using 20 Amp Electronic Speed Controllers that received PWM control signals from the MyRIO, and they were powered, along with the sensors and MyRIO, with an onboard LiPo battery.

A series of tests were conducted with the prototype, such as coefficients of lift and drag for the propellers, correlations between % thrust for the motors and speed, and also moments of inertia, to verify the mathematical model and simulation. The testing was performed using an arrangement of testing setups that constrained the quadcopter such that one axis could be analyzed at a time. These setups were used to record response data for step reference changes in angle, and then compared to the theoretical responses with good agreement. Further analysis was done to explain differences between theoretical and experimental data, and the limitations of our testing setups. The team did not perform unrestrained hover testing, as no real-time user feedback devices were implemented to control the x and y-axes of the quadcopter. However, the results of the constrained tests had strong agreements between the simulation responses and actual responses.

3 Problem Statement

Quadcopters are becoming increasingly more popular. They can be used to carry light payloads over short distances and can carry important measurement and recording equipment. They are also very popular among recreational users. However, a common expectation for quadcopters is that they remain stable in flight such that they do not move unless commanded to by the user. This expectation drives an important area of quadrotor design: stabilization control. Ensuring that a quadcopter hovering in the air remains in this state until commanded to move by a user controller demands that an electric controller appropriately responds to small disturbances such as winds blowing or light weights (such as rain, snow or other airborne materials) falling unevenly on the quadcopter.

4 Project Deliverables

The goals of this project are to design and implement embedded C controllers that can control the pitch and roll axes of a quadcopter as well as control the quadcopter's altitude. The controllers should also have good agreement between the results of the control simulation and the physical control testing.

5 Functional Specifications and Design Variables

Functional requirements for the output response were determined from Leong et. al [1]. These requirements were

generally similar across multiple papers, so they were used as benchmark metrics for controller design. Design variable benchmarks were determined from Bouabdallah [2] and from a personally owned quadcopter. The requirements and design variables are tabulated below:

Functional Requirements for ϕ, θ, ψ, and z	**Benchmark Metric**
Settling Time	7.3 s
Overshoot	17.9 %
Steady State Error	0 cm, 0 rad

Design Parameters	**Benchmark Value**
Quadcopter	
I_x,I_y	0.0124 $kg\,m^2$
I_z	0.0245 $kg\,m^2$
l	0.2250 m
m	1 kg
Propellers	
b	1.4865E-07 $N\,s^2$
d	2.925e-09 $Nm\,s^2$

6 Mathematical Model

The mathematical model for a quadcopter is derived by summing the torques and forces caused by each individual motor on the drone frame shown in Figure 1.

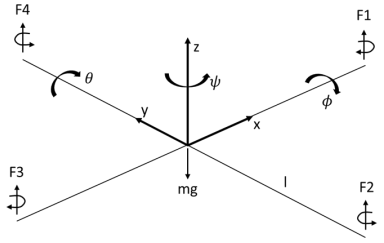


Figure 1: Schematic of Quadcopter Showing Positive Directions for Six Degrees of Freedom and Positive Motor Rotation Directions.

Equations of motion of a Quadcopter are taken from Zulu and John [3], in which the derivation of the equations of motion have already been done. Analysis of other technical paper derivations yields the same dynamics equations. Gyroscopic effects due to the propellers are small for small angles and low motor rotational inertias and are thus ignored for this project. Additionally, stabilization of the x and y coordinates are ignored as they are beyond the scope of this project. The remaining dynamics are shown in Equations 1-8.

$$\ddot{\phi} = \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right) + \frac{l}{I_x}U_2 \tag{1}$$

$$\ddot{\theta} = \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right) + \frac{l}{I_y}U_3 \tag{2}$$

$$\ddot{\psi} = \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right) + \frac{1}{I_z}U_4 \tag{3}$$

$$\ddot{z} = -g + (\cos\phi\cos\theta)\frac{U_1}{m} \tag{4}$$

$$U_1 = b\left(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2\right) \tag{5}$$

$$U_2 = b\left(-\Omega_2^2 + \Omega_4^2\right) \tag{6}$$

$$U_3 = b\left(-\Omega_1^2 + \Omega_3^2\right) \tag{7}$$

$$U_4 = d\left(\Omega_1^2 - \Omega_2^2 + \Omega_3^2 - \Omega_4^2\right) \tag{8}$$

7 Controller Design

The Quadcopter dynamics were linearized around the steady-state operating point of the quadcopter that corresponds to the quadcopter hovering in midair. The linearization is done using a first-derivative jacobian linearization method.

The linearized state space model is defined to be:

$$\frac{d}{dt}\begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\\ \dot{z}\end{bmatrix} = \begin{bmatrix} 0 & \frac{-2b\Omega_0 l}{I_x} & 0 & \frac{2b\Omega_0 l}{I_x} \\ \frac{-2b\Omega_0 l}{I_y} & 0 & \frac{2b\Omega_0 l}{I_y} & 0 \\ \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} & \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} \\ \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} \end{bmatrix}\begin{bmatrix}\Omega_1 - \Omega_0\\ \Omega_2 - \Omega_0\\ \Omega_3 - \Omega_0\\ \Omega_4 - \Omega_0\end{bmatrix} \tag{9}$$

Where: $\Omega_0 = \sqrt{\frac{mg}{4b}}$

Equation 9 can be reduced to:

$$\begin{bmatrix}\phi\\ \theta\\ \psi\\ z\end{bmatrix} = \frac{1}{s^2}\begin{bmatrix} 0 & \frac{-2b\Omega_0 l}{I_x} & 0 & \frac{2b\Omega_0 l}{I_x} \\ \frac{-2b\Omega_0 l}{I_y} & 0 & \frac{2b\Omega_0 l}{I_y} & 0 \\ \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} & \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} \\ \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} \end{bmatrix}\begin{bmatrix}\Omega_1 - \Omega_0\\ \Omega_2 - \Omega_0\\ \Omega_3 - \Omega_0\\ \Omega_4 - \Omega_0\end{bmatrix} \tag{10}$$

We using a Motor Correction Mixing Matrix shown in Equation 11 to isolate transfer functions and turn the 4-input 4-output system into 4 decoupled Single-Input Single-Output (SISO) systems:

$$\begin{bmatrix} \Omega_1 - \Omega_0 \\ \Omega_2 - \Omega_0 \\ \Omega_3 - \Omega_0 \\ \Omega_4 - \Omega_0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 1 & 1 \\ -1 & 0 & -1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \phi_{corr} \\ \theta_{corr} \\ \psi_{corr} \\ z_{corr} \end{bmatrix} \tag{11}$$

Substituting Equation 11 into Equation 10 and multiplying matrices we get the decoupled system transfer functions shown in Equation 12:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \\ z \end{bmatrix} = \frac{1}{s^2} \begin{bmatrix} \frac{4b\Omega_0 l}{I_x} & 0 & 0 & 0 \\ 0 & \frac{4b\Omega_0 l}{I_y} & 0 & 0 \\ 0 & 0 & \frac{8b\Omega_0}{I_z} & 0 \\ 0 & 0 & 0 & \frac{8b\Omega_0}{m} \end{bmatrix} \begin{bmatrix} \phi_{corr} \\ \theta_{corr} \\ \psi_{corr} \\ z_{corr} \end{bmatrix} \tag{12}$$

The controllers were designed using the built-in "pidtune" function in MATLAB. The controllers were specified to be PIDF controllers with a bandwidth of 1 Hz. These parameters were chosen because they ensure that the quadcopter has a good response to input reference changes, but is resistant to signal noise. The Controllers will be implemented in the quadcopter system as shown in Figure 2.

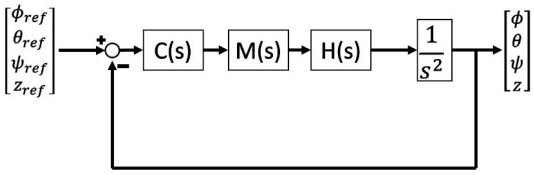


Figure 2: Block Diagram Schematic for Controller Design

Where:

$$C(s) = \begin{bmatrix} PhiC(s) & 0 & 0 & 0 \\ 0 & ThetaC(s) & 0 & 0 \\ 0 & 0 & PsiC(s) & 0 \\ 0 & 0 & 0 & ZC(s) \end{bmatrix}$$

$$M(s) = \begin{bmatrix} 0 & -1 & 1 & 1 \\ -1 & 0 & -1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 1 \end{bmatrix}$$

$$H(s) = \begin{bmatrix} 0 & \frac{-2b\Omega_0 l}{I_x} & 0 & \frac{2b\Omega_0 l}{I_x} \\ \frac{-2b\Omega_0 l}{I_y} & 0 & \frac{2b\Omega_0 l}{I_y} & 0 \\ \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} & \frac{2b\Omega_0}{I_z} & \frac{-2b\Omega_0}{I_z} \\ \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} & \frac{2b\Omega_0}{m} \end{bmatrix}$$

$$PhiC(s) = ThetaC(s) = \frac{57.67s^2 + 67.33s + 19.63}{0.02918s^2 + s}$$

$$PsiC(s) = \frac{507.4s^2 + 592.3s + 172.8}{0.02918s^2 + s}$$

$$ZC(s) = \frac{380.6s^2 + 444.3s + 129.6}{0.02918s^2 + s}$$

8 Simulation Results

The above-mentioned non-linear differential equations and linear controllers were implemented using MATLAB and Simulink to make a simulation of the behavior of the quadcopter. Shown in Figures 3 and 4 are the simulated controller responses to 10-degree initial conditions for the roll, pitch, and yaw axes, with reference positions all at zero, including the z-axis. As can be seen from the plots, the angles respond quickly with approximately 10% overshoot and a settling time of 7 seconds, which meet our functional requirements.

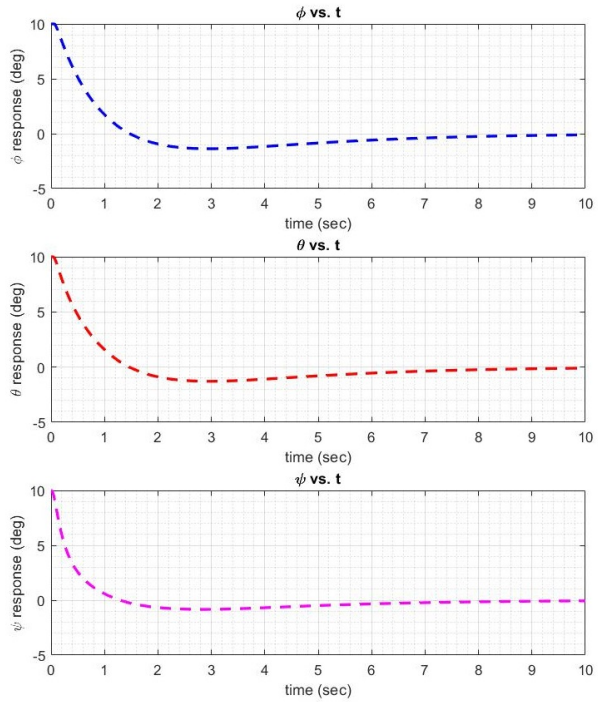


Figure 3: Responses for all Angles

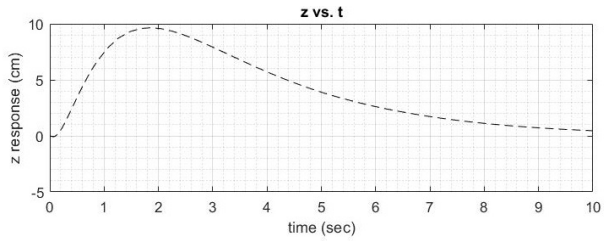


Figure 4: Responses for Z-axis

The simulation also predicts the motor speeds, motor currents, and applied torques to the model. Figure 5 is a plot of the simulated motor speeds to the same initial conditions. This shows that the primary effort of the controllers is to counteract the force of gravity. The motor speed saturations due to physical limitations of the ESCs were implemented as well to illustrate the small effect the saturations have on the performance of the quadcopter. As shown later in the report, this simulation will be used to compare the theoretical response of the quadcopter to the experimental one performed in a constrained testing setup.

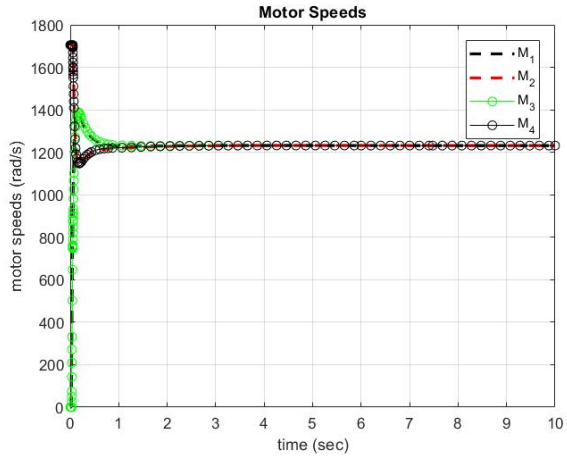


Figure 5: Motor Speeds

9 Feedback Methods

An integral part of the control scheme is the ability to feedback the current state of the quadcopter to the controller. To that end, we use an IMU sensor to sense the angular position of the quadcopter in the air. We chose to use a SparkFun 9DoF Stick, shown in Figure 6, which utilizes a LSM9DS1 chip.

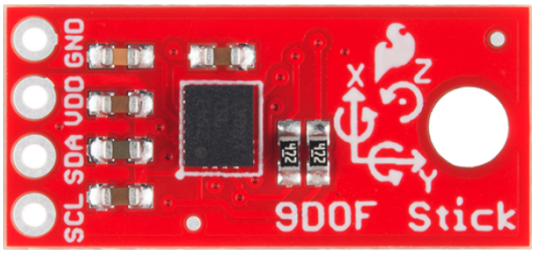


Figure 6: 9 DoF Sensor Stick

The LSM9DS1 chip has 3 accelerometers, 3 gyroscopes, and 3 magnetometers. This 9DoF sensor will be connected to the MyRio through an I2C bus, where we can receive all the sensor readings. However, before use, the 9DoF sensor had some potential problems that needed to be addressed prior to placement on the quadcopter. The sensor was found to operate on a left-hand ruled coordinate axis as shown in Figure 6. The quadcopter dynamics, however, were derived in a right-hand ruled coordinate system with the positive y-axis being the opposite direction of the positive y-axis on the 9DoF sensor. From this knowledge, we found that the angular speed measured around the y-axis and the acceleration measurements in the positive x-axis needed to be negated to match the signs that were expected of the measurements in the right-hand coordinate frame.

The z-axis feedback was sensed by the Garmin LidarLite v3 sensor shown in Figure 7. The sensor obtains a distance reading by sending out a pulse signature and then waiting until the reciever picks up the reflected pulse signature. Since the speed of light is programmed in, the sensor can determine the distance that the light pulse traveled.

Figure 7: Lidar z-axis sensor

One caveat to using this sensor is that it can only measure distance along the axis that the light travels. As a result, the readings from the sensor tend to be incorrect when the quadcopter has anything other than a non-zero pitch or roll angle. This can easily corrected if the angle of the quadcopter is known. A schematic of the correction is shown in Figure 8, where $h_{measured}$ is the height returned from the sensor.

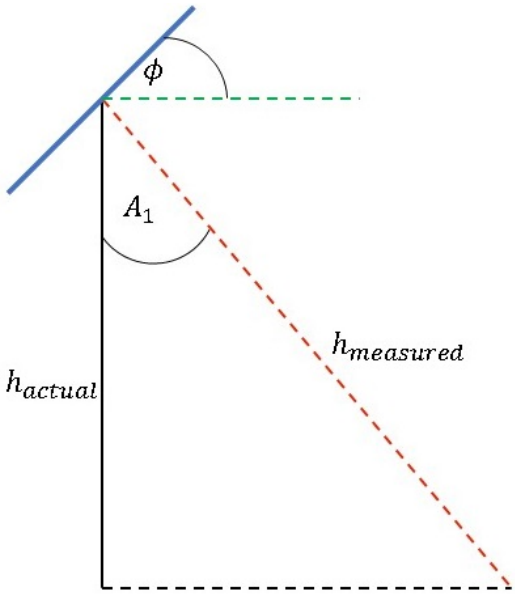


Figure 8: Height Sensor Calculation Schematic

Assuming that the quadcopter (represented in Figure 8 by the blue line) has a roll angle ϕ, we can show that the angle A_1 between the sensor axis and the real z-axis is equal to ϕ. From this we get the following relation.

$$h_{actual} = cos(\phi) \cdot h_{measured}$$

Since this relationship also holds true for the quadcopter's pitch angle, θ, the final relationship bewteen h_{actual} and $h_{measured}$ can be written:

$$h_{actual} = cos(\theta) \cdot cos(\phi) \cdot h_{measured} \tag{13}$$

10 Noise Reduction

To best obtain the angular position in space and achieve the most accurate results from our 9DoF sensor, we had to limit the noise in our system as much as possible. Accelerometers are extremely prone to high frequency noise, especially the noise caused by the vibration of the motors. Before we added any mechanical damping to our prototype, we saw fluctuations of +/- 6 degrees when the quadcopter remained level and the motors were running at their steady-state value. It was determined that the motors were spinning at a higher frequency than the control loop frequency sampling rate, and as a result, the high-frequency vibrations due to motor spin were being Nyquist shifted down to low frequencies. Since the sampled vibrational frequencies were at similar frequencies to the quadcopter angle signal, we could not simply filter out the vibrational noise through software. In order to limit the noise from these high frequency motor vibrations we took a number of steps to isolate each system from the noise. Between the 4 arms and the main body of the quadcopter we placed a piece of foam-core, double-sided tape through which the screws were placed, such that the arms and the main body would not be in direct contact with one another. In our research we found that many quadcopters used earplugs to limit noise due to vibration. We placed earplugs under each of the 4 motors so that the motors themselves were not directly in contact with the arms. We also created a vibration-isolating bed (shown in Figure 9) for the 9DoF sensor, so that any remaining high-frequency noise would be attenuated by the earplugs and not be seen in the 9DoF accelerometer readings. After implementing these 3 measures of vibration isolation, we saw significant improvements in the peak-to-peak amplitude of the angular position readings. With these improvements, there was only a fluctuation of +/- 1.5 degrees, a 75% reduction of noise amplitude.

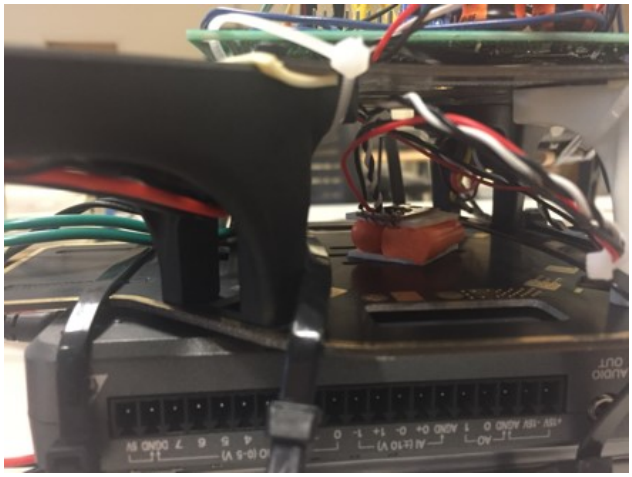


Figure 9: Vibration-Isolation Bed for 9Dof Sensor

Another concern was interference due to all of the electrical components of our quadcopter. These electrical interferences could lead to electromagnetic waves that could affect the readings of our 9DoF sensor, especially the magnetometer readings. In order to limit any of these interferences we completely rearranged the layout of our prototype. We previously had our circuit board, all of the wire connections, and the 9DoF sensor on top of the quadcopter. We moved the 9DoF sensor into the center of the

quadcopter main body, so that it was the only electrical component within that area of the quadcopter. We also used a twisted wire pair for our I2C bus to further limit any interference that could have been affecting our measurements.

11 Filtering

Crucial to the success of our controllers is being able to always know its angular position in space. The angular position of the quadcopter accounts for 3 of the 4 state variables in our model: θ, ϕ, and ψ; and it also is used in calculation of the fourth state variable, z. To calculate the θ and ϕ angles of the quadcopter, we used a combination of the angles calculated by the 3-axis accelerometer and the 3-axis angular rate-gyroscope. Each of these sensors could measure the angle on their own, but they are only reliable under certain conditions. Accelerometers are reliable in the long run because they are very prone to high frequency noise. Gyroscopes are accurate in the short term, but due to the constant integration from angular rate to angles, the gyroscope angular reading will start to drift over time shown in Figure 10.

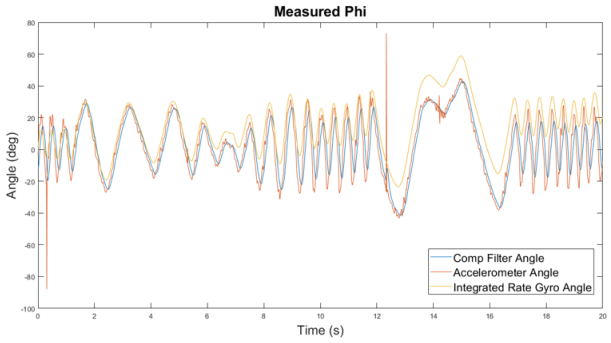


Figure 10: Graph Showing Accelerometer, Integrated Rate-Gyro, and Complimentary Filter Angles

To create a continuous and accurate angular reading, we utilized a fusion of these angular readings using a Complementary Filter. A complementary filter low-passes the accelerometer reading to filter out the high frequency noise, while high-pass filtering the gyroscope readings to limit some of the drift. Theoretically, the angular reading covers the entire frequency spectrum because the low-pass filter and high-pass filter have the same break frequency, so that when the accelerometer angle starts to roll off, the gyroscope reading will roll on and create the continuous angular reading. This is illustrated in Figure 11.

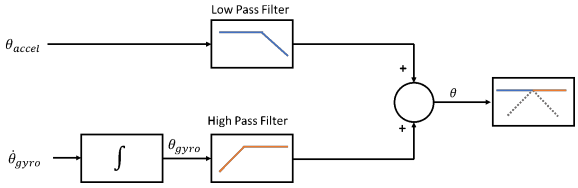


Figure 11: Complementary Filter Schematic

Our first iteration of the complementary filter was based on the filters of the same name that we found in literature during our research phase. The basic filtering equation is given in Equation 14:

$$Angle = 0.98(Angle_{prev} + gyro \cdot dt) + 0.02 \cdot Accel \quad (14)$$

The 0.98 value represents the filtering coefficient (shown in Equation 15), where τ is the time constant of the break frequency, and dt is the update rate of the control loop.

$$\alpha = \frac{\tau}{\tau + dt} \quad (15)$$

We tested the functionality of the complementary filter using a path planning C code script to control a motor to follow an angular position sine wave of specified frequency and amplitude. We attached the 9Dof sensor to the flywheel of the motor so we could compare the predicted angle from the complementary filter to the actual angle of the encoder. Using this experimental apparatus, we conducted 14 separate tests: 2 different amplitudes (30 degrees peak-to-peak, and 60 degrees peak-to-peak), and 7 different frequencies (0.2-2.0 Hz). From these tests, we were able to conclude that the complementary filter is very reliable at low frequency oscillations, but started to lose accuracy at a frequency of 2.0 Hz. Shown in Figures 12 and 13 are the plots showing the agreement between the Encoder reading of the flywheel and the angle measured via the complementary filter at low frequencies.

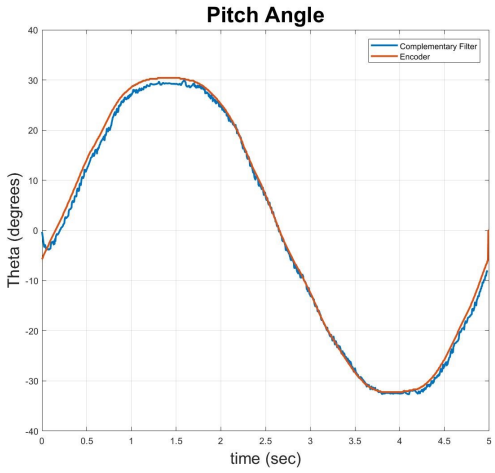


Figure 12: Complementary Filter Test

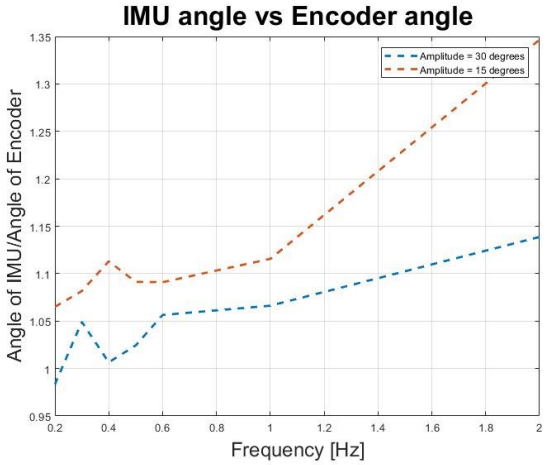


Figure 13: IMU vs. Encoder Magniutdes for Various Frequencies

After testing the complimentary filtering equation in Equation 14 in a physical quadcopter test, we found that it was inadequate for some of the higher frequency angular corrections that our quadcopter needed to make. A new complimentary filter was designed in MATLAB and exported as a discrete time difference equation structure for use in the C-programmed control loop.

Through testing, we discovered that a first-order filter adequately attentuated the noise without adding a lot of phase shift in the signal. Equations 16 and 17 represent the first-order, continuous-time equations we used to design the new complimentary filter.

$$High\ Pass\ Filter = \frac{\tau s}{\tau s + 1} \tag{16}$$

$$Low\ Pass\ Filter = \frac{1}{\tau s + 1} \tag{17}$$

We then proceeded to tune the complimentary filter parameter τ such that noise was eliminated and the output angle of the complimentary filter was an accurate representation of the true angle without phase shift. We eventually settled on a τ value of $1.326s$.

12 Electronic Speed Controllers

To control the speed of the motors, we use Electronic Speed Controllers (ESCs) to communicate between the MyRIO and the motors. Since the ESC recieves Pulse Width Modulation (PWM) signals, we can use the built in MyRIO PWM module to control the motors. An ESC usually receives a PWM signal ranging from 50 to 100 Hz, but it can receive a signal up to 500 Hz. Since our control loop frequency is 200 Hz, we needed the PWM signal frequency to also be 200 Hz to prevent any signal lag. An ESC receives a pulse ranging from 1 ms to 2 ms for turning the motor from 0 to 100 % throttle (shown in Figure 14). To adjust the speed of the motors, we change the length of the PWM pulse signal sent to the ESCs. Using experimental data, we were able to obtain an equation for converting angular velocity of the motor to % throttle, which we can use to convert the controller outputs to the appropriate ESC signal.

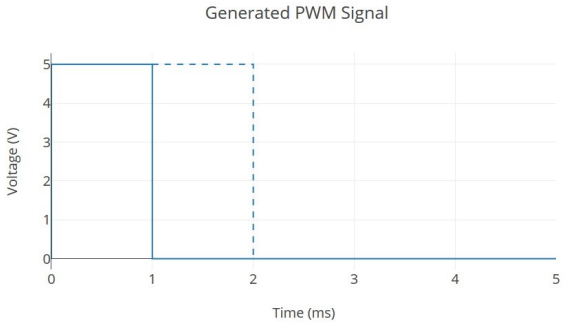


Figure 14: PWM Signal Duration, 0-100% Thrust

13 C Program Implementation

There are three main components in our C control-loop code: the main function, the timer interrupt request (IRQ) function, and the discrete time controller and filter structures.

The main function is used to initialize the input and output channels of the MyRIO. Since we are using the I2C signals for the sensors and PWM signals for the ESCs, we

changed the registers on output pin 7 of the MyRIO to enable the I2C functionality and output pins 2, 3 and 4 on the MyRIO for enabling the PWM functionality. The main function then sets up the I2C block and sets the frequency of the PWM signal. Afterwards, it initializes the Timer IRQ thread, and sets up various control parameters so that the user can switch between manual and automatic control modes for controlling the quadcopter. Manual mode allows the user to change the % throttle of one or all of the motors at the same time while automatic mode enables the controllers to alter the motor speed.

The Timer Interrupt Request thread implements the control loop for the quadcopter. In this loop, raw data is queried from the 9DoF sensor and the altitude sensor. The raw data is passed through filters to obtain filtered angle data. The filtered angle data will then be passed to the controller to calculate the error signal and determine the necessary motor speed corrections for each of the four motors. These corrections will be converted into % throttle signals and be output as PWM signals to the ESCs, which then commutate the motors. Circular buffers store data after every loop iteration. This loop is repeated until the main function signals the thread to terminate. Before terminating, all data buffers are stored in a MATLAB data file for further analysis.

To implement the controllers and filters, a biquad cascade method is used. The continuous-time transfer functions are converted to discrete-time transfer functions in MATLAB using Tustin's method and exported as c-structures. Coefficient values for the discrete-time difference equation for the controllers and filters are stored in these c-structures. These structures also store all the necessary input-output data for the controller and filter difference equations to be solved on every iteration through the control loop and update the stored input-output data every iteration through the control loop.

14 Physical Design

Our quadcopter uses a drone frame with 450mm wheelbase, 4 20A Electronic Speed Controllers, 4 2300KV motors and 5x4x4 propellers. The MyRIO is mounted on the bottom of the frame, with a 3000mAh LiPo battery secured below the MyRIO. The 9DoF sensor is placed in the center of the frame. Placing the sensor close to the center of the frame, away from electrical and mechanical interference, helps the sensor to receive more accurate data for calculations, and places the sensor as close as possible to the center of rotation. The LiDar sensor is placed on the bottom of the frame next to the myRIO for altitude measurments. A protoboard is placed on the top of the frame to make the necessary connections between the MyRIO and the ESCs, sensors, and battery. The ESCs are mounted along the bottom of the four arms with a motor mounted on the end of each arm. Memory-foam earplugs are added between the motor and the arms, as well as between the 9Dof sensor and the frame for additional physical damping of the motor vibrations. Double-sided foam-core tape is also added to some of the frame joints for more damping. To control the quadcopter without any physical connections from the computer, a Wi-Fi connection is enabled between the MyRIO and the computer for communication and the MyRIO is powered by the LiPo battery.

15 Coefficient Testing

In order to complete our plant model and design our controllers, we first had to conduct a series of tests to determine the performance of our motors and propellers. These tests verified our values of the thrust and drag coefficients, which were integral to our simulations and overall controller design. From the tests we ran, our values were on the same order of magnitude as the literature values, and all of the plots had the same characteristics we expected based on the corresponding mathematical models. We ran three main tests, which are further explained in the sections below.

Angular Speed vs. Percent Throttle

The first test that we had to run was to create a correlation between the % throttle that was sent from our PWM code to our ESCs and the angular speed of each motor. To find this correlation, we measured the angular speed of the propellers using a stroboscope for each specified percent throttle. To conduct this test, we placed a piece of black electrical tape on one of the propeller blades for easier identification, then we would enter a specified % throttle to begin spinning the propellers. We would change the flash frequency of the stroboscope such that when its frequency matched the angular speed of the propeller, the black tape on the illuminated propeller would appear to stay in the same position (shown in Figure 15). To verify we had the correct speed and not a harmonic, we would double and halve the frequency of the stroboscope. This would ensure that we had the correct angular speed because at double the frequency the black tape appears to flip positions, while at half the frequency it would seem that the black tape was also always in a constant position. We performed this test every 5% throttle from 0-95% to obtain an accurate equation that we then implemented into our c-code to send the correct inputs to the ESCs. This correlation curve can be seen in Figure 16.

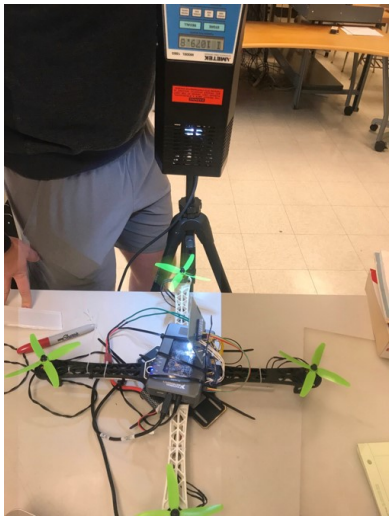


Figure 15: % Thrust vs. Motor Speeds Testing Setup

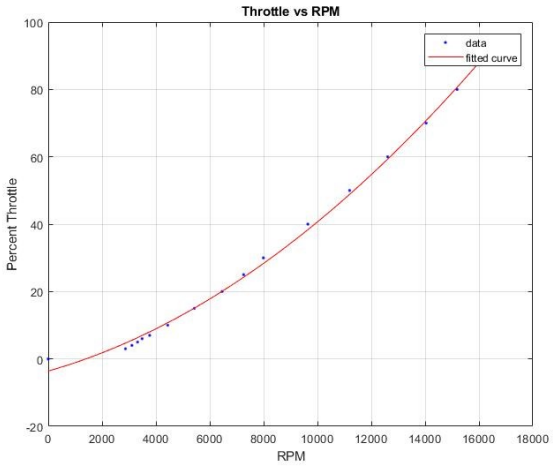


Figure 16: ESC % Thrust vs. Motor Speeds

Thrust Coefficient Testing

Once the correlation between % throttle and angular speed was known, we could then continue with the verification of our plant model. The next unknown that we had to calculate was the thrust coefficient of our motor and propeller combination. Our test consisted of placing our quadcopter on the initial testing setup, shown in Figure 17, resting one of the frame legs on a gram scale, and then zeroing the scale. We would then turn on the motor on the opposite side of the quadcopter as the gram scale so that the motor created a thrust force that pushed the frame leg down onto the scale. Repeating this process several times for different % thrusts, we were able to plot a thrust curve from the force and angular speed data, which can be seen in Figure 18. This thrust curve was then used to calculate the thrust coefficient needed for our plant model using Equation 18.

$$F_{thrust} = b\Omega^2 \tag{18}$$

The coefficient that we calculated was on the same order of magnitude as the literature values, which reassured us that the tests we were conducting and the results we were obtaining were an accurate representation of our system.

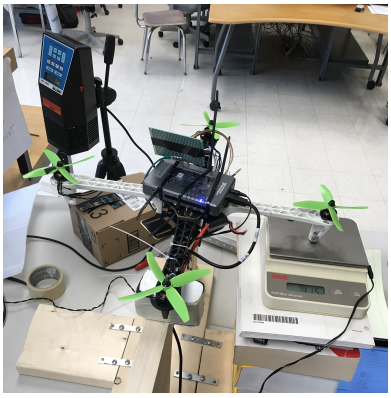


Figure 17: Motor Thrust vs. Angular Speed Testing Setup

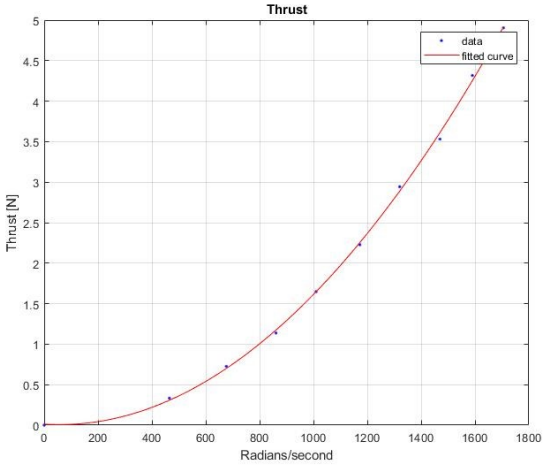


Figure 18: Motor Thrust vs. Angular Speed

Drag Coefficient Testing

The final test that we had to perform was to calculate the drag coefficient of our propellers. To calculate the drag coefficient, we strapped our quadcopter to a turntable that allowed it to freely rotate about the z-axis. We could not

use a gram scale for this test since the gram scale did not function properly when place on its side. Instead we used a calibrated cantilever beam so that we could find a correlation between the strain in the beam and the force that the motors exerted on the end of the beam (see Figure 19).

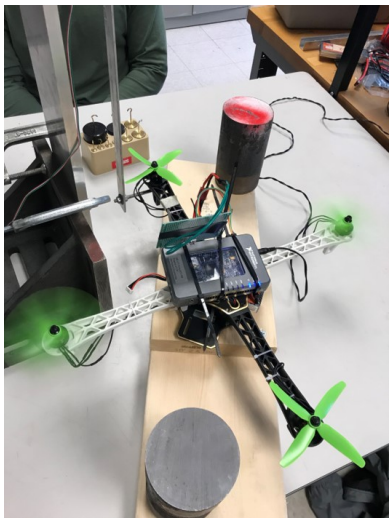


Figure 19: Drag Testing Setup

To perform this test we turned on two opposing motors to create a moment about the z-axis. Having two motors on was key to the accuracy of this test so that the turntable would stay level and limit as much friction in the system as possible. Once again, as with the thrust test, we could measure the force applied to the cantilever beam, and thus the yaw torque due to the motors for specified angular speeds. From these measured data points we were able to find a fitted correlation curve shown in Figure 20. The drag could be calculated from this curve using Equation 19

$$T_{drag} = d\Omega^2 \tag{19}$$

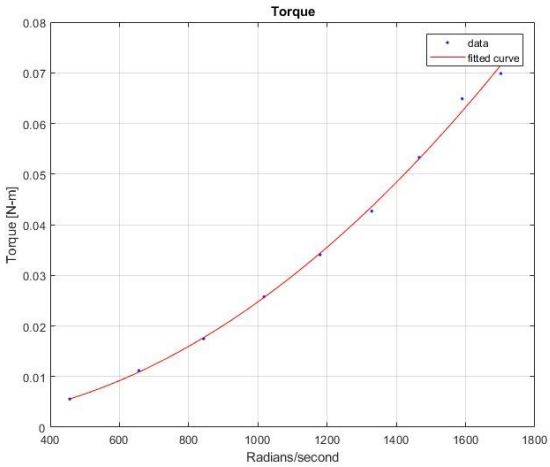


Figure 20: Motor Torque vs. Angular Speed

16 Testing Setup

After the plant model was characterized by drag coefficient, lift coefficient, and % thrust vs. angular speed testing, the flight control algorithms were tested. To test the ability of the designed control algorithms of the quadcopter to maintain midair stability, a multitude of testing setups were constructed to allow testing of control in all three angular directions (roll, pitch, and yaw) as well as height (z-direction). The testing of the controllers was completed in stages, where the quadcopter was isolated to move only about a single degree of freedom in a restrained testing setup. This improved the safety during testing as the quadcopter can only move a limited distance and therefore is not able to danger anyone or pose a risk of itself being damaged. When each controller had been developed as desired from the restrained testing setups, the controls of the quadcopter needed far less tuning during unrestrained hover testing and thus there is a much lower chance of damage to the quadcopter during this unrestrained testing.

First, the pitch and roll control of the quadcopter was tested using the same testing setup, as shown in Figure 21. Using an aluminum rod that was inserted through the center of the quadcopter frame allowed the quadcopter to move about its axis in only the pitch or roll direction, the pitch controller and the roll controller could individually be tested using the same setup. While we could achieve desirable controller responses in this setup, this setup was susceptible to a few drawbacks. A significant amount of friction was inherent in this setup that would not occur in unrestrained testing, which caused the quadcopter to experience a significant damping in its motion about the pitch and roll axis which further caused responses that did

not follow our simulated model. Due to the position of the rod placement through the frame, it was not possible to get the quadcopter to rotate about center of mass as is expected in the plant model and during unrestrained flight.

Figure 21: Initial Roll/Pitch Testing Setup

The drawbacks of the testing setup shown in Figure 21 led our group to create another testing setup as shown in Figure 22, where a solid-core insulated wire was strung through the center of the quadcopter frame. Through this setup we were able to significantly reduce the friction and allow the quadcopter to rotate about its center of mass. Some desirable responses were realized through this setup, although the wire was not of ideal stiffness and thus the wire would bend and oscillate as the quadcopter was making angle corrections and ultimately caused the quadcopter to be unstable. This drawback was significant enough to force us to create another pitch and roll testing setup.

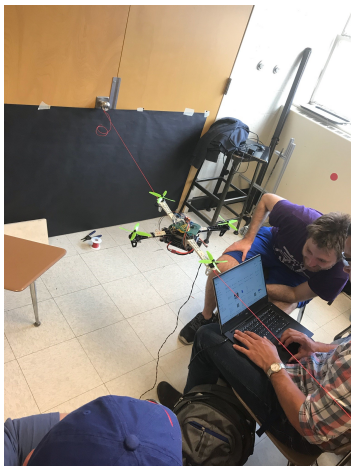


Figure 22: Second Roll/Pitch Testing Setup

The final pitch and roll testing setup, as shown in Figure 23, used two large aluminum rods that two opposing quadcopter arms would rest on, allowing the quadcopter to rotate up to +/- 90 degrees in the pitch or roll direction. This setup had negligible friction and the large aluminum rods removed oscillations transmitted to the quadcopter, although the quadcopter was forced to rotate slightly above its center of mass thus causing a slight pendulum effect. Despite not rotating exactly about the center of mass of the quadcopter, we were able to acquire experimental data that was agreeable with our simulated responses allowing for us to move forward with other testing.

Figure 23: Final Roll/Pitch Testing Setup

After the pitch and roll controllers performed as desired in the testing setups, z-direction controller testing begun. This setup restricted the quadcopter from pitch, roll, yaw or horizontal-translational movements, only allowing for the quadcopter to control its movements in the z-direction. This setup, as shown in Figure 24, included two brass rods attached to the feet of the quadcopter frame which would side through two vertical aluminum beams. The LiDar (z-directional) sensor would read the distance from the quadcopters frame to the ground and the reference position was set about 20 centimeters above the top of the aluminum beams. This setup allowed the quadcopter to achieve accurate height measurements and adequately restrained the quadcopter to only move in the z-direction, although there was a considerable amount of friction between brass rods and the vertical beams. While the friction in the system caused some damping in the z-directional movement, desirable responses were achieved and it showed the quadcopter could adequately rise to and hover at a reference height position.

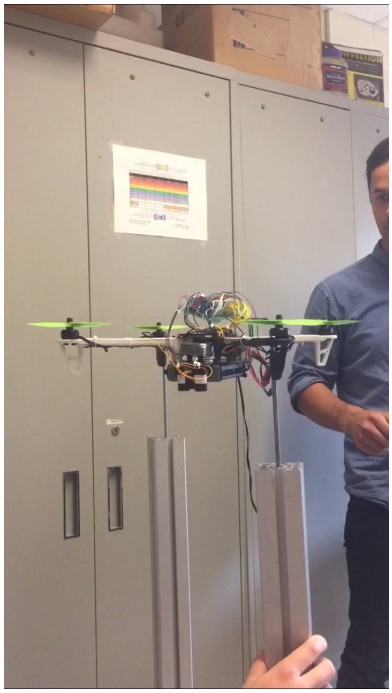


Figure 24: Altitude Testing Setup

The yaw control was tested by mounting the quadcopter to a ball bearing turn table, the setup is shown in Figure 25. The turn table connected to the bottom of the quadcopter frame using zip ties and restricted pitch, roll, x, y, and z-direction motion. While the setup performed as desired, the magnetometer within the IMU sensor experienced too much magnetic interference caused by the wiring throughout the quadcopter prototype to allow for appropriate readings. From this drawback we decided to change the scope of our project to focus on the control of pitch, roll, and z-directional motion as yaw control is not necessary for the quadcopter to hover.

Figure 25: Yaw Testing Setup

Once it was ensured that the pitch, roll, and z-directional controllers were acting as expected, we performed restricted testing of hover control using the setup in Figure 26. The quadcopter was restrained by a hanging wire to restrict excessive movement in the x and y directions while allowing for pitch, roll and z-directional movements and prevented the quadcopter from falling and being damaged. Although the quadcopter for some instances would hover freely the quadcopter would tend to drift in the x and y directions forcing the wire to tighten and pull the on the top of the quadcopter. This pulling force caused the quadcopter to snap back and rotate franticly, creating a hazardous testing circumstance. Due to the hazardous nature of the testing setup we were forced to move on to a new limited restraint testing setup.

Figure 26: Initial Restrained Hover Testing Setup

For the new limited restraint setup we decided to restrict the quadcopter from moving laterally in the x or y-direction while allowing for movement in the pitch, roll and z-direction. This setup uses a brass rod that slides through a vertical aluminum beam and is connected using a universal joint to the bottom of the quadcopter and is shown in Figure 27. We could achieve some reasonable responses using this setup, but due to the universal joint being connected to the bottom of the quadcopter, the quadcopter was not able to rotate about its center of mass and thus the quadcopter would seldom maintain a stable hover position.

Figure 27: Second Restrained Hover Testing Setup

Since we were incapable of finding a limited restraint setup that could appropriately reflect in flight conditions, we could not prove our quadcopter would have a safe and desirable unrestrained performance and thus we did not move forward with unrestrained testing. Although we achieved desirable pitch, roll, a z-directional control individually, we did not want to hastily place the quadcopter in a circumstance that could harm someone or itself, therefore unrestrained testing is something that would be conducted in the future given more time.

17 Testing Results

In order to determine the validity of our simulations and controllers, we perfomed a series of constrained tests where we analyzed the experimental response of the quadcopter to a step change in angle reference in the roll axis, and compared it to the simulated theoretical response to the same reference change. Figures 28 and 29 compare simulated and experimental step changes of 10 degrees and -20 degrees, respectively. As can be seen from the plots, the controller responds very quickly, but then undershoots and slowly climbs to the correct angle. Based on our analysis, we have determined that this error is due to the quadcopter not rotating about its center of mass. Since the real center of mass is lower than the point of rotation, the center of mass acts as a disturbance torque about the controlled axis and acts against the change for non-zero angles. The integral term in the PIDF controller ensures that the quadcopter will have zero steady state error after a period of time, but since the modeled plant is assumed to be ideal it takes much longer with our test apparatus than simulated. The high frequency oscillations can be attributed to the motor vibration aliasing that was mentioned previously.

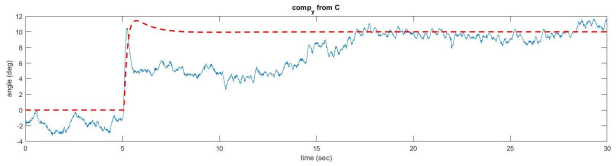


Figure 28: 10 Degree Step Theoretical and Measured Responses

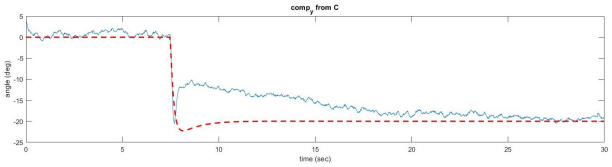


Figure 29: -20 Degree Step Theoretical and Measured Responses

In order to verify our assumptions about this undershoot, the simulation was changed to reflect the disturbance torque enacted by the center of mass as the quadcopter rotates away from zero degrees. This disturbance torque was added to the model input torque U3 in Simulink.

$$T_{dist} = -mglsin(\theta) \tag{20}$$

We realize that rotating about a point other than the center of mass affects the moment of inertia as well, but we wanted to get a good idea about whether or not our assumption was correct, not necessarily the exact response.

Figures 30 and 31 show the responses of the simulation compared to the experimental responses when the disturbance torque is accounted for. As can be seen from the plots, there is extremely good agreement, which verifies our assumptions.

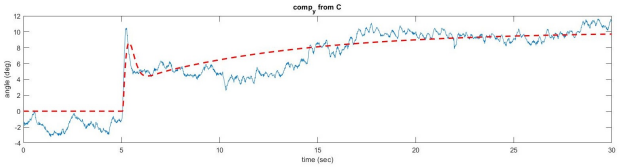


Figure 30: 10 Degree Step Theoretical and Measured Responses with Center of Mass Offset

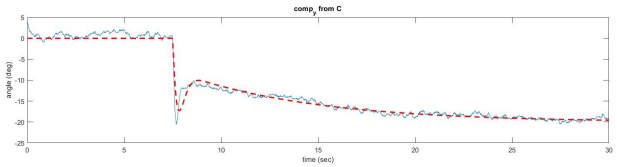


Figure 31: -20 Degree Step Theoretical and Measured Responses with Center of Mass Offset

Based on the very good agreement between simulated and experimental results, we feel very strongly that with a bit more effort the quadcopter could be fully functional in all axes for an unconstrained testing setup. As we saw with the plots of the step reference changes, the testing setups can have a profound impact on the performance of the system, and can add unintentional dynamic elements that were not modeled when we designed the controllers. Since we did not have a way of making x-axis and y-axis corrections during a flight, we did not feel confident that we could perform an unconstrained test flight safely and successfully without risking damage to the quadcopter and ourselves. With added inputs from a controller such as a RC radio controller, we could easily do a test flight and ensure that the risks could be mitigated effectively, but this was out of scope of the project.

18 Risk and Liability

Due to the high rate of speed that the quadcopter motors operate at as well as the high rate of speed of the overall quadcopter, there are serious risks involved with flying the quadcopter if it is operated with malicious intent or if malfunctions occur during flight. Foremost, the quadcopter poses a risk of harming the user, and other people in the vicinity, when the quadcopter is in flight. While serious injury is very unlikely, the propellers moving at high speed are capable of wounding anyone if they were to touch someone during operation. Further, the quadcopter poses a risk of harming or destroying other airborne vehicles and physical structures that the quadcopter may encounter. These risks are already combatted by the multitude of regulations set by the FAA and include the limitation of drone size, flight speed, where the quadcopter can be operated, and operating conditions [4]. To individually combat these risks, our capstone group has set out to prove the quadcopters airworthiness though an exorbitant amount of flight control tests, as described previously, before unrestrained operation of the quadcopter.

19 Ethical Issues

The major ethical implication surrounding quadcopters and drones are stemmed from the possible applications of drone technology. While the scope of our capstone project does not involve any of these applications, some current applications for drones include aerial videography and military uses. Aerial videography is a purposeful application for many drone hobbyists and has been tied to the ethical implications that someone may use a drone to take video footage of personal property and of other people (amongst other implications), thus necessitating regulation and code of conduct for how drones can be used for aerial videography. As for governmental and military applications, many ethical concerns surround the ability and ease for a drone to be used for surveillance and military strikes. While surveillance and military strikes occur with or without drones, many people are concerned with the ease and limited reliability that is associated with the use of drones for these applications [5]. Although these implications are not directly related to the scope of this capstone project, it is important to understand the extent of ethical impact this technology has.

20 Impact on Society

Beyond the negative implications of quadcopters and drones discussed in the Ethical Issues section of this report, quadcopters do serve the ability to benefit society in various ways. For instance, Amazon is currently developing its Amazon Prime Air delivery system to incorporate quadcopters in the fast delivery of its merchandise to users [6]. Similarly, Boeing is currently developing a high payload (500 pound) quadcopter to be used for various life-altering applications [7]. These advancements in quadcopter technology could increase the speed that it takes for consumers and companies to receive their shipments without ever requiring them to leave the house. While the method of online shopping already has the ability for quick shipment, this technology will increase the speed further to where shipments can be received within 30 minutes of ordering. Further, this technology would increase safety and decrease cost of shipment by lowering the number of manned-cargo vehicles used for shipments. In all, advancing quadcopter technology has the ability to increase the ease of consumerism and air travel, lower labor costs, increase safety, amongst many other benefits.

21 Impact on the Environment

Quadcopters have a very interesting impact on the environment. With the increase in quadcopters in recent years, they have begun to be developed for a numerous number of roles. As explained in the Impact on Society section of this report, one big avenue for quadcopters is package delivery systems, such as Amazon and UPS. Both of these companies are looking into quadcopters as a new means of delivering packages. This could cut down significantly on costs

of fuel for current ground-based transportation systems. Quadcopters would be able to take more direct and efficient routes, but at current advancements, only be able to handle 1-2 packages at a time. Like most electric-battery-based vehicles, the main concern is what happens to the batteries once they are no longer of use. Overall, when comparing emissions from quadcopters and ground-based transportation, smaller quadcopters will produce much less emissions that steel trucks [8].

22 Codes and Standards

Codes:
FAA 14 CFR Part Part 107 The Small UAS Rule
FAA Public Law 112-95, Section 336 - Special Rule for Model Aircraft

Standards:
ISO/TC 20/SC 16 Unmanned aircraft systems
ARP4754A Guidelines For Development Of Civil Aircraft and Systems
DO-178C, Software Considerations in Airborne Systems and Equipment Certification

23 Cost and Engineering Economics

For our quadcopter, we have purchased all our parts, except for the MyRIO. Below is a table showing the cost to build our 450mm quadcopter with the parts that we used.

Item	Name	Cost	Quantity	Total
Frame	RipaFire F450	$18.99	1	$18.99
ESC	RipaFire LittleBee	$17.99 (includes 2)	2	$35.98
Motor	Cobra 2204	$79.99 (includes 4)	1	$79.99
Propeller	AvatarRC Geniune HQ 5040-4	$8.99 (includes 8)	4	$8.99
Battery	Floreon 3S Lipo 3000mAh	$49.99 (includes 2)	1	$49.99
Wires	-	In House	-	-
Protoboard	-	$11.99	1	$11.99
IMU	Sparkfun 9DoF	$14.95	1	$14.95
Altitude Sensor	Garmin LidarLite v3	$158.75	1	$158.75

The list provided shows that our quadcopter cost is very competitive, as it cost less than other same size quadcopters on the market. Since we have created a controller based on the weight, thrust and drag coefficient, we can also implement the same controller with little adjustment, meaning we can increase the number of sales by selling a variety of quadcopters with different sizes.

24 Conclusions

With further time on this project, certain improvements could be made on our current quadcopter model that could further increase our certainty that our quadcopter could successfully fly unrestrained. First, the quadcopter could be implemented with an RC radio controller as the signal input for reference values. Ideally, the RC controller could be used to override controller commands should the quadcopter controllers fail to function correctly on any given unrestrained flight test. Naturally, the biggest obstacle for our project was the lack of robust testing setups that could adequately simulate the airborne quadcopter while simultaneously carrying little risk of unsafe operation. For the

controllers to truly be tested, an unrestrained testing setup where the tester could instantaneously take manual control of the quadcopter would be ideal. Unfortunately, we never had the opportunity to utilize such a setup. With an RC controller connected however, we could be more confident that the quadcopter could be kept under control should a controller malfunction.

The other main change that would be implemented is the knowledge of the center-of-mass disturbance torque. With more time, we could add this torque into the simulation and tune the controllers to account for the center-of-mass disturbance torque. This in turn would give us more assurance that our controllers were ready to be tested unrestrained and would allow us to place fewer constraints on the quadcopter testing setups.

References

[1] L. B. T. Meng, S. M. Low, and M. P.-L. Ooi, "Low-cost microcontroller-based hover control design of a quadcopter," *Procedia Engineering*, vol. 41, pp. 458–464, 2012.

[2] S. Bouabdallah, "Design and control of quadrotors with application to autonomous flying," *SomeJournal*, 2007.

[3] A. Zulu and S. John, "A review of control algorithms for autonomous quadrotors," *Open Journal of Applied Sciences*, no. 4, pp. 547–556, 2014.

[4] "Summary of small unmanned aircraft rule (part 107)," tech. rep., Federal Aviation Administration, 2016.

[5] R. L. Wilson, "Ethical issues with use of drone aircraft," in *2014 IEEE International Symposium on Ethics in Science, Technology and Engineering*, 2014.

[6] Amazon.com, "Amazon prime air." Website.

[7] A. Hawkins, "Boeing built a giant drone that can carry 500 pounds of cargo," *The Verge*, 2018.

[8] C. Samaras, "Is drone delivery good for the environment?," *The Smithsonian*, 2018.

25 Appendices

Appendix A: Control Loop Flowchart

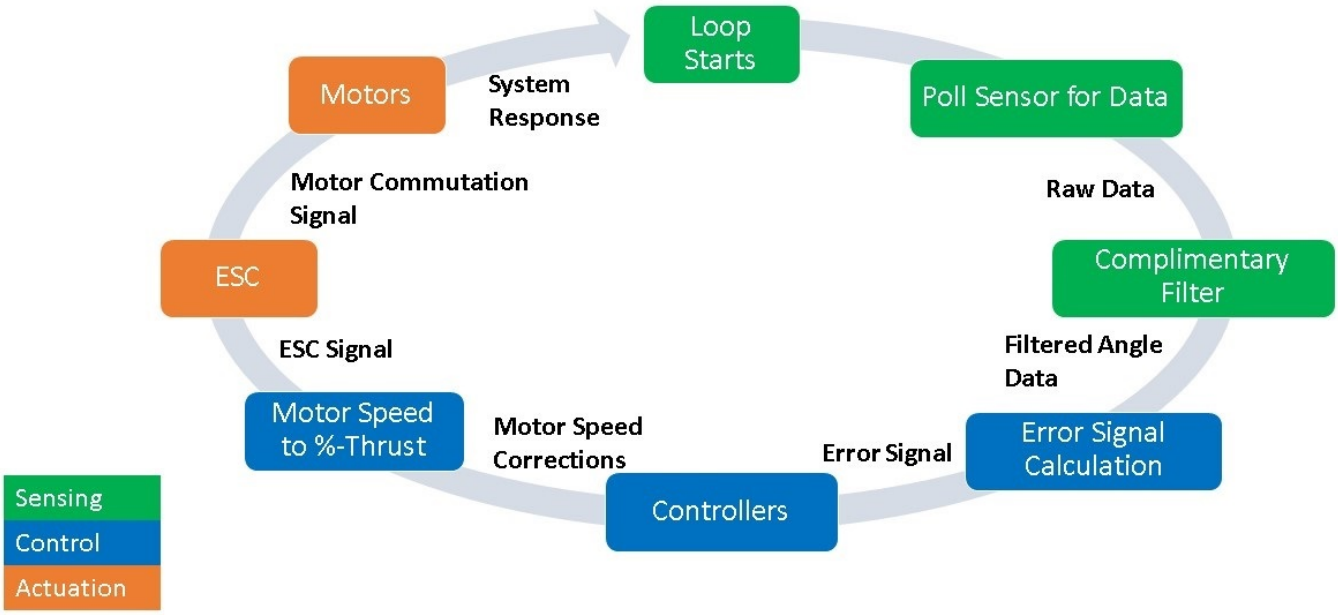
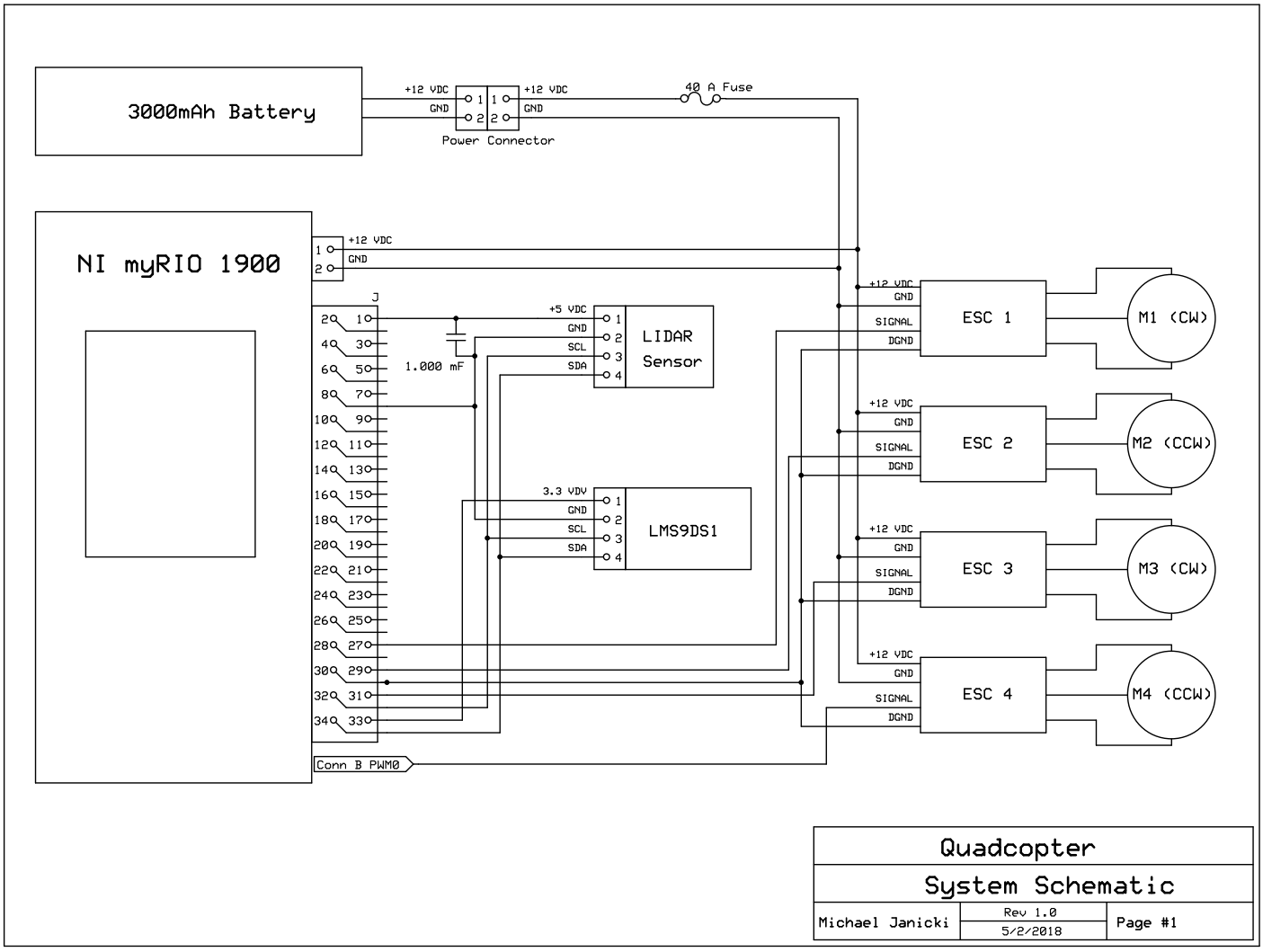


Figure 32: Control Loop Sequence

Appendix B: Electrical Design Schematic



Appendix C: Capstone Control Code (.c)

```c
/* Quadcopter Capstone Team
 * Authors:
 * Michael Janicki
 * Karl Kintner-Meyer
 * Nick Kunst
 * Hallvard Ng
 *
 *
 *
 * This code allows the user to control a quadcopter.
 * The code first initializes the ESCs for the motors.
 * Then, the user is prompted "Select Motor."
 * By specifying different cases when prompted, the
 * user can decide whether of not to manually select
 * the speeds of the motors individually or together,
 * or select automatic mode.
 * Modes 0-4 are manual.
 * Mode 5 turns on all controllers and goes to the
 * reference positions written in the code.
 * Mode 6 changes the Roll reference.
 * Mode 7 allows the user to specify the roll
 * reference manually.
 * Mode 9 ends the program.
 *
 *This program also saves buffers containing data on
 *the state estimation, controller outputs,
 *motor speeds, etc. to matlab.
 *
 * Sensor communication is done using I2C
 * communication protocols and the ESC signals
 * are output using PWM signals.
 * */

/* includes */
#include <stdio.h>
#include "MyRio.h"
#include "me477.h"
#include <string.h>
#include "TimerIRQ.h"
#include <pthread.h>
#include "matlabfiles.h"
#include "math.h"
#include "I2C.h"
#include <math.h>
#include <IRQConfigure.h>
#include <unistd.h>
#include "PWM.h"

/*
 * Declare the myRIO NiFpga_Session so that it can be used by any function in
 * this file. The variable is actually defined in myRIO.c.
 */
extern NiFpga_Session myrio_session;

// Biquad declaration for controllers/filters
struct biquad  {
    double  b0; double  b1; double  b2; // numerator
    double  a0; double  a1; double  a2; //denominator
    double  x0; double  x1; double  x2; //input
    double  y1; double  y2; };          //output

// Controllers and filters
#include "PhiController.h"
#include "ThetaController.h"
#include "PsiController.h"
#include "ZController.h"
#include "LowPassFilterPhi.h"
#include "HighPassFilterPhi.h"
#include "LowPassFilterTheta.h"
```

```
#include "HighPassFilterTheta.h"
#include "LowPassFilterPsi.h"
#include "HighPassFilterPsi.h"
#include "LowPassFilterZ.h"

//TimeoutValue declaration
uint32_t    timeoutValue = 5000;  // time interval - us; f_s = 100 Hz

// Prototypes
void *Timer_Irq_Thread(void* resource);
double  cascade( double xin,            //input
                 struct biquad *fa,     //biquad array
                 int ns);               //no. segments

// Definitions
#define PI 3.141592654
#define IMAX 1000
#define GAINPHI 0
#define GAINTHETA 1
#define GAINPSI 0
#define GAINZ 0

typedef struct {
    NiFpga_IrqContext irqContext;   //Context
    NiFpga_Bool irqThreadRdy;       //ready flag
} ThreadResource;

// I2C variable declarations
MyRio_I2c i2cA;

extern NiFpga_Session myrio_session;
NiFpga_Status status;

// PWM variable declarations
MyRio_Pwm pwmA0;
MyRio_Pwm pwmA1;
MyRio_Pwm pwmA2;
MyRio_Pwm pwmB0;
uint8_t selectRegA;
uint8_t selectRegB;
time_t currentTime;
time_t finalTime;

// PWM Control
Pwm_ClockDivider Divider = Pwm_64x;
uint16_t MaxCounter = 3125;

// Set each ESC with their corresponding duty cycles
uint16_t DutyCycleA0 = 625;
uint16_t DutyCycleA1 = 625;
uint16_t DutyCycleA2 = 625;
uint16_t DutyCycleB0 = 625;
float SelectDutyCycle =  0;
int ac = 0;

double stepInput = 0;
double ThetaRef = 0;

#if !defined(LoopDuration)
#define LoopDuration    60  // How long to output the signal, in seconds
#endif

#define SATURATE(x,lo,hi) ((x) < (lo) ? (lo) : (x) > (hi) ? (hi) : (x)) //Saturation
MACRO

/*---------------------------------------------------------------
Function AngVeltoCDP()
Purpose: Converts angular velocity of motors to servo signal duty cycle
Parameters: (in) - omega - angular velocity
```

```
Returns:   (out) - ConvDutyCycle - servo signal to ESCs
*-----------------------------------------------------------------*/
double AngVeltoCDP(double omega){
    double ConvDutyCycle;       // Converted Duty Cycle
    double ConvPercThSat;
    double ConvPercTh = 0.000014341525649*omega*omega+0.032702124122*omega-8.1151624437;
    if(omega<0){
        ConvPercTh = 0;
    }
    ConvPercThSat = SATURATE(ConvPercTh,0,95);
    ConvDutyCycle = (20+ConvPercThSat*0.2)*31.25;
    return ConvDutyCycle;
}

/*-----------------------------------------------------------------
Function initializePWM()
Purpose: Initializes the pwm output channels
Parameters: (in) - void
Returns:   (out) - void
*-----------------------------------------------------------------*/
void initializePWM(void){
    /*
     * Initialize the PWM struct with registers from the FPGA personality.
     */
    pwmA0.cnfg = PWMA_0CNFG;
    pwmA0.cs = PWMA_0CS;
    pwmA0.max = PWMA_0MAX;
    pwmA0.cmp = PWMA_0CMP;
    pwmA0.cntr = PWMA_0CNTR;

    pwmA1.cnfg = PWMA_1CNFG;
    pwmA1.cs = PWMA_1CS;
    pwmA1.max = PWMA_1MAX;
    pwmA1.cmp = PWMA_1CMP;
    pwmA1.cntr = PWMA_1CNTR;

    pwmA2.cnfg = PWMA_2CNFG;
    pwmA2.cs = PWMA_2CS;
    pwmA2.max = PWMA_2MAX;
    pwmA2.cmp = PWMA_2CMP;
    pwmA2.cntr = PWMA_2CNTR;

    pwmB0.cnfg = PWMB_0CNFG;
    pwmB0.cs = PWMB_0CS;
    pwmB0.max = PWMB_0MAX;
    pwmB0.cmp = PWMB_0CMP;
    pwmB0.cntr = PWMB_0CNTR;
}

/*-----------------------------------------------------------------
Function PWMset()
Purpose: Sets PWM parameters
Parameters: (in) - void
Returns:   (out) - status
*-----------------------------------------------------------------*/
int PWMset(void){
        /*
         * Set the waveform, enabling the PWM onboard device.
         */
        Pwm_Configure(&pwmA0, Pwm_Invert | Pwm_Mode,
                Pwm_NotInverted | Pwm_Enabled);
        Pwm_Configure(&pwmA1, Pwm_Invert | Pwm_Mode,
                Pwm_NotInverted | Pwm_Enabled);
        Pwm_Configure(&pwmA2, Pwm_Invert | Pwm_Mode,
                Pwm_NotInverted | Pwm_Enabled);
        Pwm_Configure(&pwmB0, Pwm_Invert | Pwm_Mode,
                Pwm_NotInverted | Pwm_Enabled);

        /*
```

```
207             * Set the clock divider. The internal PWM counter will increments at
208             * f_clk / 4
209             *
210             * where:
211             *  f_clk = the frequency of the myRIO FPGA clock (40 MHz default)
212             */
213            Pwm_ClockSelect(&pwmA0, Divider);
214            Pwm_ClockSelect(&pwmA1, Divider);
215            Pwm_ClockSelect(&pwmA2, Divider);
216            Pwm_ClockSelect(&pwmB0, Divider);
217
218            /*
219             * Set the maximum counter value. The counter counts from 0 to 1000.
220             *
221             * The counter increments at 40 MHz / 4 = 10 MHz and the counter counts
222             * from 0 to 1000. The frequency of the PWM waveform is 10 MHz / 1000
223             * = 10 kHz.
224             */
225            Pwm_CounterMaximum(&pwmA0, MaxCounter);
226            Pwm_CounterMaximum(&pwmA1, MaxCounter);
227            Pwm_CounterMaximum(&pwmA2, MaxCounter);
228            Pwm_CounterMaximum(&pwmB0, MaxCounter);
229
230            /*
231             * PWM outputs are on pins shared with other onboard devices. To output on
232             * a physical pin, select the PWM on the appropriate SELECT register. See
233             * the MUX example for simplified code to enable-disable onboard devices.
234             *
235             * Read the value of the SYSSELECTA register.
236             */
237            status = NiFpga_ReadU8(myrio_session, SYSSELECTA, &selectRegA);
238            MyRio_ReturnValueIfNotSuccess(status, status,
239                "Could not read from the SYSSELECTA register!")
240
241            status = NiFpga_ReadU8(myrio_session, SYSSELECTB, &selectRegB);
242            MyRio_ReturnValueIfNotSuccess(status, status,
243                "Could not read from the SYSSELECTB register!")
244
245            /*
246             * Set bit 2 and 3 of the SYSSELECTA register to enable PWMA_0 functionality.
247             * The functionality of the bit is specified in the documentation.
248             */
249
250            selectRegA = selectRegA | (1 << 2) | (1 << 3) | (1 << 4);
251
252            /*
253             * Write the updated value of the SYSSELECTA register.
254             */
255            status = NiFpga_WriteU8(myrio_session, SYSSELECTA, selectRegA);
256            MyRio_ReturnValueIfNotSuccess(status, status,
257                "Could not write to the SYSSELECTA register!")
258
259            selectRegB= selectRegB | (1 << 2);
260            status = NiFpga_WriteU8(myrio_session, SYSSELECTB, selectRegB);
261            MyRio_ReturnValueIfNotSuccess(status, status,
262                "Could not write to the SYSSELECTB register!")
263
264            return status;
265    }
266
267    /*---------------------------------------------------------------
268    Function main
269        Purpose: Set up and enable the IRQ interrupt.
270         Signal the Timer Thread to terminate using the irqThreadRdy flag,
271         then wait for ISR to terminate.
272         9 will end the code.
273    *---------------------------------------------------------------*/
274    int main(int argc, char **argv) {
275        NiFpga_Status status;
```

```
276       int32_t status2;
277       uint8_t selectReg;
278
279       /*
280        * Initialize the I2C struct with registers from the FPGA personality.
281        */
282       i2cA.addr = I2CAADDR;
283       i2cA.cnfg = I2CACNFG;
284       i2cA.cntl = I2CACNTL;
285       i2cA.cntr = I2CACNTR;
286       i2cA.dati = I2CADATI;
287       i2cA.dato = I2CADATO;
288       i2cA.go = I2CAGO;
289       i2cA.stat = I2CASTAT;
290
291       status = MyRio_Open();          /*Open the myRIO NiFpga Session.*/
292       if (MyRio_IsNotSuccess(status)) return status;
293
294       MyRio_IrqTimer irqTimer0;  //Initialize the IRQ Timer
295       ThreadResource irqThread0; //Initialize the thread resource structure
296       pthread_t thread; //Initialize the ISR thread
297
298       /*
299        * Enable the I2C functionality on Connector A
300        *
301        * Read the value of the SELECTA register.
302        */
303       status = NiFpga_ReadU8(myrio_session, SYSSELECTA, &selectReg);
304
305       MyRio_ReturnValueIfNotSuccess(status, status,
306                   "Could not read from the SYSSELECTA register!");
307
308       /*
309        * Set bit7 of the SELECT register to enable the I2C functionality. The
310        * functionality of this bit is specified in the documentation.
311        */
312       selectReg = selectReg | (1 << 7);
313
314       /*
315        * Write the updated value of the SELECT register.
316        */
317       status = NiFpga_WriteU8(myrio_session, SYSSELECTA, selectReg);
318
319       MyRio_ReturnValueIfNotSuccess(status, status,
320                       "Could not write to the SYSSELECTA register!");
321
322       /*
323        * Set the speed of the I2C block.
324        *
325        * Standard mode (100 kbps) = 187.
326        * Fast mode (400 kbps) = 51.
327        *
328        * These values are calculated using the formula:
329        *   f_SCL = f_clk / (2 * CNTR) - 4
330        *
331        * where:
332        *   f_SCL = the desired frequency of the I2C transmission
333        *   f_clk = the frequency of the myRIO FPGA clock (40 Mhz default)
334        *
335        * This formula and its rationale can be found in the documentation.
336        */
337       I2c_Counter(&i2cA, 187);
338
339        /*
340         * Enable the I2C block.
341         */
342       I2c_Configure(&i2cA, I2c_Enabled);
343
344       //Specify IRQ channel settings
```

```
345         irqTimer0.timerWrite = IRQTIMERWRITE;
346         irqTimer0.timerSet = IRQTIMERSETTIME;
347 
348         //Configure Timer IRQ. Terminate if not successful
349         status2 = Irq_RegisterTimerIrq(
350                 &irqTimer0,
351                 &irqThread0.irqContext,
352                 timeoutValue);
353         if (status2 != 0) return status2;       //check the status of the register
354 
355         //Set the indicator to allow the new thread
356         irqThread0.irqThreadRdy = NiFpga_True;
357 
358         initializePWM();
359         PWMset();
360 
361         //Create new thread to catch the IRQ
362         status2 = pthread_create(
363                 &thread,
364                 NULL,
365                 Timer_Irq_Thread,
366                 &irqThread0);
367         if (status2 != 0) return status2;       //check the status of the register
368 
369         float percentage = 0;
370         int MotorSelection;
371         float ThetaRef1 = 0;
372 
373         // Mode loop
374         while (percentage!=-1) {
375             printf("Select Motor: ");
376             scanf("%d",&MotorSelection);
377             switch(MotorSelection){
378             case 0:     printf("\nSelect Percentage: ");
379                         scanf("%e",&percentage);
380                         ac = 0;
381                         DutyCycleA0=(20+percentage*0.2)*31.25;
382                         DutyCycleA1=(20+percentage*0.2)*31.25;
383                         DutyCycleA2=(20+percentage*0.2)*31.25;
384                         DutyCycleB0=(20+percentage*0.2)*31.25;
385                         printf("\n");
386                         break;
387             case 1:     printf("\nSelect Percentage: ");
388                         scanf("%e",&percentage);
389                         DutyCycleA0=(20+percentage*0.2)*31.25;
390                         printf("\n");
391                         break;
392             case 2:     printf("\nSelect Percentage: ");
393                         scanf("%e",&percentage);
394                         DutyCycleA1=(20+percentage*0.2)*31.25;
395                         printf("\n");
396                         break;
397             case 3:     printf("\nSelect Percentage: ");
398                         scanf("%e",&percentage);
399                         DutyCycleA2=(20+percentage*0.2)*31.25;
400                         printf("\n");
401                         break;
402             case 4:     printf("\nSelect Percentage: ");
403                         scanf("%e",&percentage);
404                         DutyCycleB0=(20+percentage*0.2)*31.25;
405                         printf("\n");
406                         break;
407             case 5:     ac = 1;
408                         ThetaRef = 0;
409                         break;
410             case 6:     ac = 1;
411                         ThetaRef = 20.0*PI/180.0;
412                         break;
413             case 7:     printf("\nSelect Angle: ");
```

```
414                     scanf("%e",&ThetaRef1);
415                     ThetaRef = (double)ThetaRef1*PI/180.0;
416                     printf("\n");
417                     break;
418         case 9: percentage = -1;
419                     break;
420         default:    printf("\nWrong Selection\n");//
421                     break;
422         }
423     }
424     irqThread0.irqThreadRdy = NiFpga_False; //signal ISR to terminate by setting
        irqThreadRdy flag in the ThreadResource structure
425     pthread_join(thread, NULL);             //wait for interrupt thread to return NULL
426
427     //Unregister the ISR
428     status2 = Irq_UnregisterTimerIrq(
429             &irqTimer0,
430             irqThread0.irqContext);
431
432     status = MyRio_Close();  /*Close the myRIO NiFpga Session. */
433     return status;
434 }
435
436 /*---------------------------------------------------------------------
437 Function *Timer_Irq_Thread()
438 Purpose: To perform any necessary function in response to the interrupt. In this
    program, the interrupt thread we are reading and
439 computing a dynamic signal
440 Parameters: resource - ThreadResource that is declared above
441 Returns: N/A
442 *---------------------------------------------------------------------*/
443 void *Timer_Irq_Thread(void* resource) {
444     uint8_t data[12] = {0x60,0x00,0x70,0x60,0x50,0x40,0x30,0x20,0x10,0x90,0xA0,0xB0};
445     uint8_t AccelStartStop[2] = {0xC0,0x00};
446     uint8_t GyroStartStop[2] = {0xC0,0x00};
447     uint8_t MagnetStartStop[2] = {0b01011100,0b00010000};
448     uint8_t MagnetStartStop2[2] = {0b00001000,0b00000000};
449     uint8_t subAddresses[5]={0x28/*Accel Data Out*/,0x0F/*Who Am
        I*/,0x20/*CtrlReg6*/,0x18/*Gyro Data Out*/,0x10/*CtrlReg1*/};
450     uint8_t subAddresses2[3]={0x20/*CtrlReg1*/,0x23/*CtrlReg4*/,0x28/*Magnetometer Data
        Out*/};
451     uint8_t slaveReadAddress = 0x6B;
452     uint8_t slaveReadAddress2 = 0x1E;
453
454     //I2C variables for LidarLite Z-sensor
455     uint8_t slaveWriteAddress3 = 0x62;
456     uint8_t slaveReadAddress3 = 0x62;
457     uint8_t subaddress3[5] = {0x02,0x04,0x1C,0x00,0x8F};
458     uint8_t values3[4] = {0x80,0x08,0x00,0x04};
459     uint8_t data2[6] = {0x40, 0x50,0x60,0x70,0x80,0x90};
460
461     // Controller reference values
462     double PhiRef = 0;
463     //double ThetaRef = 0;
464     double PsiRef = 0;
465     double ZRef = 1.4;
466     double Zact;
467
468     // Buffer declarations
469     double aXBuffer[IMAX];
470     double *pointerX = aXBuffer;
471     double aYBuffer[IMAX];
472     double *pointerY = aYBuffer;
473     double aZBuffer[IMAX];
474     double *pointerZ = aZBuffer;
475
476     double gXBuffer[IMAX];
477     double *pointerGX = gXBuffer;
478     double gYBuffer[IMAX];
```

```
479         double *pointerGY = gYBuffer;
480         double gZBuffer[IMAX];
481         double *pointerGZ = gZBuffer;
482 
483         double mXBuffer[IMAX];
484         double *pointerMX = mXBuffer;
485         double mYBuffer[IMAX];
486         double *pointerMY = mYBuffer;
487 
488         double CompxBuffer[IMAX];
489         double *compX = CompxBuffer;
490         double CompyBuffer[IMAX];
491         double *compY = CompyBuffer;
492         double CompzBuffer[IMAX];
493         double *compZ = CompzBuffer;
494         double HeightBuffer[IMAX];
495         double *HeightPointer = HeightBuffer;
496 
497         double Omega1Buffer[IMAX];
498         double *Omega1Pointer = Omega1Buffer;
499         double Omega2Buffer[IMAX];
500         double *Omega2Pointer = Omega2Buffer;
501         double Omega3Buffer[IMAX];
502         double *Omega3Pointer = Omega3Buffer;
503         double Omega4Buffer[IMAX];
504         double *Omega4Pointer = Omega4Buffer;
505 
506         double PercentThrust1Buffer[IMAX];
507         double *PercentThrust1Pointer = PercentThrust1Buffer;
508         double PercentThrust2Buffer[IMAX];
509         double *PercentThrust2Pointer = PercentThrust2Buffer;
510         double PercentThrust3Buffer[IMAX];
511         double *PercentThrust3Pointer = PercentThrust3Buffer;
512         double PercentThrust4Buffer[IMAX];
513         double *PercentThrust4Pointer = PercentThrust4Buffer;
514 
515         double AngleAccXBuffer[IMAX];
516         double *AngleAccXPointer = AngleAccXBuffer;
517         double AngleAccYBuffer[IMAX];
518         double *AngleAccYPointer = AngleAccYBuffer;
519         double AngleMagZBuffer[IMAX];
520         double *AngleMagZPointer = AngleMagZBuffer;
521 
522         double xOut;
523         double yOut;
524         double zOut;
525 
526         int16_t xOutTemp;
527         int16_t yOutTemp;
528         int16_t zOutTemp;
529 
530         double sqrtval;
531         double sqrtval2;
532         double angle_accelx;
533         double angle_accely;
534         double angle_magz;
535         double CompAnglex;
536         double CompAngley;
537         double CompAnglez;
538 
539         ThreadResource* threadResource = (ThreadResource*) resource;
540 
541         // 9dof sensor initialization
542         I2c_WriteByteToSub(&i2cA, slaveReadAddress,subAddresses+2,AccelStartStop,1);     //
        Turn on Accel.
543         I2c_WriteByteToSub(&i2cA, slaveReadAddress,subAddresses+4,GyroStartStop,1);      //
        Turn on Gyro.
544         I2c_WriteByteToSub(&i2cA, slaveReadAddress2,subAddresses2,MagnetStartStop,1);    //
        Turn on Magnetometer
```

```
545         I2c_WriteByteToSub(&i2cA, slaveReadAddress2,subAddresses2+1,MagnetStartStop2,1); //
            Turn on Magnetometer
546
547         // Z sensor initialization
548         I2c_WriteByteToSub(&i2cA, slaveWriteAddress3,subaddress3,values3,1);
549         I2c_WriteByteToSub(&i2cA, slaveWriteAddress3,subaddress3+1,values3+1,1);
550         I2c_WriteByteToSub(&i2cA, slaveWriteAddress3,subaddress3+2,values3+2,1);
551
552         MATFILE *MatFile; //initializing MATFILE type
553         int err; //initialize the error in saving to MATLAB
554
555         double Omega0 = 1186;
556         uint32_t irqAssert = 0;
557         while(threadResource->irqThreadRdy == NiFpga_True){ //while the main thread does
            not signal this thread to stop
558             Irq_Wait( threadResource->irqContext,
559                       TIMERIRQNO,
560                       &irqAssert,
561                       (NiFpga_Bool*) &(threadResource->irqThreadRdy));
562             NiFpga_WriteU32( myrio_session,
563                              IRQTIMERWRITE,
564                              timeoutValue );
565             NiFpga_WriteBool( myrio_session,
566                               IRQTIMERSETTIME,
567                               NiFpga_True);
568             if(irqAssert){
569                 //Interrupt service code here
570
571                 //Read Accel-----------------------------------------------
572                 I2c_ReadByteFromSub(&i2cA, slaveReadAddress,subAddresses,data,6);
573                 zOutTemp = (data[5]<< 8) | data[4];
574                 yOutTemp = (data[3]<< 8) | data[2];
575                 xOutTemp = (data[1]<< 8) | data[0];
576                 xOut = (double)xOutTemp * -0.000061*9.81;
577                 yOut = (double)yOutTemp * 0.000061*9.81;
578                 zOut = (double)zOutTemp * -0.000061*9.81;
579                 *pointerX++ = xOut;
580                 *pointerY++ = yOut;
581                 *pointerZ++ = zOut;
582
583                 // Calculate acceleration angles
584                 sqrtval = sqrt((yOut*yOut + zOut*zOut));
585                 angle_accely = atan2(xOut,sqrtval);//changed accely to accelx and vice
                    versa here
586                 sqrtval2 = sqrt((xOut*xOut + zOut*zOut));
587                 angle_accelx = atan2(yOut,sqrtval2);
588                 angle_accelx = -angle_accelx;
589
590                 // Save acceleration angle data to buffer
591                 *AngleAccXPointer++ = angle_accelx;
592                 *AngleAccYPointer++ = angle_accely;
593
594                 // Circular buffer
595                 if(AngleAccXPointer == (AngleAccXBuffer + IMAX)){
596                     AngleAccXPointer = AngleAccXBuffer;
597                     AngleAccYPointer = AngleAccYBuffer;
598                 }
599
600                 //Read Magnetometer
601                 I2c_ReadByteFromSub(&i2cA, slaveReadAddress,subAddresses,data,6);
602                 yOutTemp = (data[3]<< 8) | data[2];
603                 xOutTemp = (data[1]<< 8) | data[0];
604                 xOut = (double)xOutTemp * 0.00014;
605                 yOut = (double)yOutTemp * 0.00014;
606
607                 // Save to buffer
608                 *pointerMX++ = xOut;
609                 *pointerMY++ = yOut;
610
```

```
611                 // Calculate Z angle
612                 angle_magz = atan2(yOut,xOut)*180/PI;
613
614
615                 // Save magnetometer angle data to buffer
616                 *AngleMagZPointer++ = angle_magz;
617
618                 // Circular buffer
619                 if(AngleMagZPointer == (AngleMagZBuffer + IMAX)){
620                     AngleMagZPointer = AngleMagZBuffer;
621                 }
622
623                 //Read Gyro-----------------------------------------------------
624                 I2c_ReadByteFromSub(&i2cA, slaveReadAddress,subAddresses+3,data,6);
625                 zOutTemp = (data[5]<< 8) | data[4];
626                 yOutTemp = (data[3]<< 8) | data[2];
627                 xOutTemp = (data[1]<< 8) | data[0];
628                 xOut = (double)xOutTemp * 0.00875*PI/180;
629                 yOut = (double)yOutTemp * -0.00875*PI/180;
630                 zOut = (double)zOutTemp * 0.00875*PI/180;
631
632                  // Save to buffer
633                 *pointerGX++ = xOut;
634                 *pointerGY++ = yOut;
635                 *pointerGZ++ = zOut;
636
637                 // Circular buffer
638                 if(pointerX == (aXBuffer + IMAX)){
639                     pointerX = aXBuffer;
640                     pointerY = aYBuffer;
641                     pointerZ = aZBuffer;
642                     pointerGX = gXBuffer;
643                     pointerGY = gYBuffer;
644                     pointerGZ = gZBuffer;
645                     pointerMX = mXBuffer;
646                     pointerMY = mYBuffer;
647                 }
648
649                 //Complementary Filter
650                 CompAnglex = cascade(angle_accelx,LowPassFilterPhi,LowPassFilterPhi_ns) +
                    cascade(xOut,HighPassFilterPhi,HighPassFilterPhi_ns)+(PI/180.0);
651                 CompAngley = cascade(angle_accely,LowPassFilterTheta,LowPassFilterTheta_ns)
                    + cascade(yOut,HighPassFilterTheta,HighPassFilterTheta_ns);
652                 CompAnglez = cascade(angle_magz,LowPassFilterPsi,LowPassFilterPsi_ns)   +
                    cascade(zOut,HighPassFilterPsi,HighPassFilterPsi_ns);
653
654                 // Save data to buffers
655                 *compX++ = CompAnglex;//
656                 *compY++ = CompAngley;
657                 *compZ++ = CompAnglez;
658                 // Circular buffer
659                 if (compY == (CompyBuffer + IMAX)) {
660                     compX = CompxBuffer;
661                     compY = CompyBuffer;
662                     compZ = CompzBuffer;
663                 }
664
665                 //Read Z-Sensor
666                 I2c_WriteByteToSub(&i2cA, slaveWriteAddress3,subaddress3+3,values3+3,1);
667                 I2c_ReadByteFromSubLidarLite(&i2cA,slaveReadAddress3,subaddress3+4,data2,2);
668
669                 uint16_t distance = data2[0]<< 8 | data2[1];
670                 double ZactUF = (double)distance;
671
672                 double ZactF = cascade(ZactUF,LowPassFilterZ,LowPassFilterZ_ns);
673
674                 Zact = ZactF*(1.0/100.0)*cos(CompAnglex)*cos(CompAngley);
675
676                 // Save height data to buffer
```

```
677                 *HeightPointer++ = Zact;
678
679                 // Circular buffer
680                 if(HeightPointer == (HeightBuffer + IMAX)){
681                     HeightPointer = HeightBuffer;
682                 }
683
684                 // Calculate controller errors
685                 double ePhi = (PhiRef - CompAnglex);
686                 double eTheta = (ThetaRef - CompAngley);
687                 double ePsi = (PsiRef - CompAnglez);
688                 double eZ = (ZRef - Zact);
689
690                 // Calculate controller corrections
691                 double PhiCorr = GAINPHI*cascade(ePhi,PhiController,PhiController_ns);
692                 double ThetaCorr =
                    GAINTHETA*cascade(eTheta,ThetaController,ThetaController_ns);
693                 double PsiCorr = GAINPSI*cascade(ePsi,PsiController,PsiController_ns);
694                 double ZCorr = GAINZ*cascade(eZ,ZController,ZController_ns);
695
696                 // Motor mixing logic
697                 double Omega1 = -ThetaCorr + PsiCorr + ZCorr + Omega0;
698                 double Omega2 = -PhiCorr - PsiCorr + ZCorr + Omega0;
699                 double Omega3 = ThetaCorr + PsiCorr + ZCorr + Omega0;
700                 double Omega4 = PhiCorr - PsiCorr + ZCorr + Omega0;
701
702                 // Save motor speeds to buffer
703                 *Omega1Pointer++ = Omega1;
704                 *Omega2Pointer++ = Omega2;
705                 *Omega3Pointer++ = Omega3;
706                 *Omega4Pointer++ = Omega4;
707
708                 // Circular buffer
709                 if (Omega1Pointer == (Omega1Buffer + IMAX)) {
710                     Omega1Pointer = Omega1Buffer;
711                     Omega2Pointer = Omega2Buffer;
712                     Omega3Pointer = Omega3Buffer;
713                     Omega4Pointer = Omega4Buffer;
714                 }
715
716                 if(ac == 1){
717                 // Change the calculated speed into Duty Cycle
718                     DutyCycleA0=AngVeltoCDP(Omega2);
719                     DutyCycleA1=AngVeltoCDP(Omega1);
720                     DutyCycleA2=AngVeltoCDP(Omega4);
721                     DutyCycleB0=AngVeltoCDP(Omega3);
722                 }
723
724                 // Save percent thrust to buffers
725                 *PercentThrust1Pointer++ = ((DutyCycleA1/31.25)-20)/0.2;
726                 *PercentThrust2Pointer++ = ((DutyCycleA0/31.25)-20)/0.2;
727                 *PercentThrust3Pointer++ = ((DutyCycleB0/31.25)-20)/0.2;
728                 *PercentThrust4Pointer++ = ((DutyCycleA2/31.25)-20)/0.2;
729
730                 // Circular buffers
731                 if (PercentThrust1Pointer == (PercentThrust1Buffer + IMAX)) {
732                     PercentThrust1Pointer = PercentThrust1Buffer;
733                     PercentThrust2Pointer = PercentThrust2Buffer;
734                     PercentThrust3Pointer = PercentThrust3Buffer;
735                     PercentThrust4Pointer = PercentThrust4Buffer;
736                 }
737
738                 // Write duty cylces to ESCs
739                 Pwm_CounterCompare(&pwmA0, DutyCycleA0);
740                 Pwm_CounterCompare(&pwmA1, DutyCycleA1);
741                 Pwm_CounterCompare(&pwmA2, DutyCycleA2);
742                 Pwm_CounterCompare(&pwmB0, DutyCycleB0);
743
744                 //Acknowledge the IRQ has been asserted
```

```
            Irq_Acknowledge(irqAssert);
        }
    }

    // 9dof close down sequence
    I2c_WriteByteToSub(&i2cA, slaveReadAddress,subAddresses+2,AccelStartStop+1,1); //
    Turn off Accel.
    I2c_WriteByteToSub(&i2cA, slaveReadAddress,subAddresses+4,GyroStartStop+1,1); //
    Turn off Gyro.
    I2c_WriteByteToSub(&i2cA, slaveReadAddress2,subAddresses2,MagnetStartStop+1,1); //
    Turn off Magnetometer
    I2c_WriteByteToSub(&i2cA, slaveReadAddress2,subAddresses2+1,MagnetStartStop2+1,1);
    // Turn off Magnetometer

    // Method for saving input and out values to a MATLAB file
    MatFile = openmatfile("DataCollection19.mat", &err);
    if(!MatFile)  {
        printf_lcd("Can't open mat file %d\n", err);///////
    }

    // Save buffers to matlab
    matfile_addstring(MatFile, "comment", "change of theta ref of +10 deg");///////
    matfile_addmatrix(MatFile, "a_x", aXBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "a_y", aYBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "a_z", aZBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "g_x", gXBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "g_y", gYBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "g_z", gZBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "comp_x", CompxBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "comp_y", CompyBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "comp_z", CompzBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "Height", HeightBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "Omega1", Omega1Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "Omega2", Omega2Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "Omega3", Omega3Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "Omega4", Omega4Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "PercentThrust1", PercentThrust1Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "PercentThrust2", PercentThrust2Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "PercentThrust3", PercentThrust3Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "PercentThrust4", PercentThrust4Buffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "AngleAccX", AngleAccXBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "AngleAccy", AngleAccYBuffer, IMAX, 1, 0);
    matfile_addmatrix(MatFile, "AngleMagZ", AngleMagZBuffer, IMAX, 1, 0);
    matfile_close(MatFile);

    pthread_exit(NULL); //terminate the new thread and return from the function
    return NULL;//
}

/*---------------------------------------------------------------
Function cascade()
Purpose: Calculate the output value of the signal using the biquad cascade method
Parameters: (in) - xin, input signal
            (in) - *fa, biquad array
            (in) - ns, number of segments
            (in) - ymin, minimum saturation voltage
            (in) - ymax, maximum saturation voltage
Returns: y0 - output signal value
*---------------------------------------------------------------*/
double  cascade( double xin, struct biquad *fa, int ns)  {
    double y0 = 0; //initialize the output to zero
    int i;
    struct biquad *f; //initialize pointer for biquad structure
    f = fa; //set the biquad pointer to myFilter
    f->x0 = xin; //set the initial x0 to the input signal
    for (i=0; i < ns; i++)  {
        if (i != 0) { //every biquad except the first
            f->x0 = y0; //set the x0 to the output value of the previous biquad
        }
```

```
810             y0 = (f->b0*f->x0 + f->b1*f->x1 + f->b2*f->x2 - f->a1*f->y1 - f->a2*f->y2)
                /(f->a0); //calculate the biquad cascade
811             f->x2 = f->x1; //set x2 equal to x1 of the previous iteration
812             f->x1 = f->x0; //set x1 equal to x0 of the previous iteration
813             f->y2 = f->y1; //set y2 equal to y1 of the previous iteration
814             f->y1 = y0; //set y1 equal to the output y0 of the biquad cascade
815             f++; //increment the biquad pointer
816         }
817         return y0; //return the output value
818     }
```

Appendix D: Controller Design Code (.m)

```
close all; clear all; clc
%% Code for calculating Ix, Iy, and Iz
% Lengths (m)
x1 = 0.225; % arm length
x2 = 0.025; % motor height
x3 = 0.025; % motor radius
x4 = 0.060; % hight of copter center
x5 = 0.060; % radius of copter center
x6 = 0.020; % width of arms
x7 = 0.010; % hight of arms

% Masses (kg)
m1 = 0.400; % mass of center
m2 = 0.050; % mass of arm
m3 = 0.100; % mass of motor

Iy1 = m1/12*(3*x5^2 + x4^2);
Iy2 = m2/12*(x1^2 + x7^2) + m2/4*x1^2;
Iy3 = m3/12*(3*x3^2 + x2^2) + m3*x1^2;

Ymoment = Iy1 + 2*Iy2 + 2*Iy3 + m2/6*(x6^2 + x7^2) + m3/6*(3*x3^2 + x2^2);
Xmoment = Ymoment;
Zmoment = m1/2*x5^2 + 4*(m3/2*x3^2 + m3*x1^2 + m2/12*(x1^2 + x6^2) + m2/4*x1^2);

MASS = (m1 + 4*m2 + 4*m3)*1.1;
%% Main Code Script for Simulink Simulation
%% Motor Parameters
kv = 2300;                  % -- RPM/v
kt = 1/(kv*2*pi/60);        % -- v-s/rad

%% Current and Voltage Limits
maxCurrent = 20;            % -- Amps
maxVoltage = 12;            % -- Volts

%% Quadcopter Parameters
copter.Ix = (0.015)*1.25;%Xmoment;          % -- moment of inertia about x-axis
copter.Iy = (0.015)*1.25;%Ymoment;          % -- moment of inertia about y-axis
copter.Iz = Zmoment;                        % -- moment of inertia about z-axis
copter.L = x1;                              % -- length of arms (m)
copter.J = 0e-6;                            % -- polar moment of inertia of propellers
copter.m = MASS;                            % -- mass of quadcopter (kg)
copter.g = 9.81;                            % -- acceleration of gravity (m/s^2)
copter.ct = 1.779e-06;                      % -- prop lift coefficient
copter.cq = 2.96948e-08;                    % -- prop drag coefficient
copter.B = 0.75;                            % -- copter drag coefficient

%% Design Controllers
% Redifine Variables for simplicity
b = copter.ct;
d = copter.cq;
Ix = copter.Ix;
Iy = copter.Iy;
Iz = copter.Iz;
m = copter.m;
g = copter.g;
L = copter.L;

s = tf('s');

% Operating point for linearization
w0 = 1286;%(m*g/(4*b))^(1/2);

% Define State-Space
A = zeros(4,4);
B = [0            -2*b*w0*L/Ix    0               2*b*w0*L/Ix;
    -2*b*w0*L/Iy   0              2*b*w0*L/Iy     0;
     2*d*w0/Iz    -2*d*w0/Iz      2*d*w0/Iz      -2*d*w0/Iz;
     2*b/m*w0      2*b/m*w0       2*b/m*w0        2*b/m*w0];
C = eye(4);
```

```
D = zeros(4,4);

% Control Mixing Matrix
convert = [0  -1   1  1;
          -1   0  -1  1;
           0   1   1  1;
           1   0  -1  1];
% New B matrix for converting to Controllers as state inputs
B2 = B*convert;

% (s*I - A)^(-1)
S = eye(4)*1/s;

% Matrix of transfer functions between controllers and the state variables
TFs = (C*S*B2 + D)/s;

% Disturbances
D1 = (C*S*B + D)/s;
D2 = D1*ones(4,1);

% Steady State disturbances. Gravity is the only disturbance.
% This is just D2 written out by hand
Disturbances = [0;
                0;
                0;
                0.00483/s^2];
% Plants
PhiP = TFs(1,1);
TheP = TFs(2,2);
PsiP = TFs(3,3);
ZP = TFs(4,4);

%% Notes
% Functional specifications like steady-state error and response
% time should be specified here.

% Functional requirements for angle controllers
Ts = 4;             % - settling time (sec)
OS = 10;            % - percent overshoot

% System Requirements
z = -log(OS/100)/sqrt(pi^2+log(OS/100)^2);
wn = 4/(z*Ts);

%% Phi Controller Design
% Generate controller
wc = 1*2*pi;
PhiC1 = pidtune(PhiP,'PIDF',wc);
Tfphi = PhiC1.Tf;
kiphi = PhiC1.Ki;
kdphi = PhiC1.Kd;
kpphi = PhiC1.Kp;

% Assign variables
copter.Tfphi = Tfphi;
copter.kiphi = kiphi;
copter.kdphi = kdphi;
copter.kpphi = kpphi;

% Controller
PhiC = kpphi + kdphi*s/(Tfphi*s+1) + kiphi/s
figure('Name', 'Phi Controller', 'Visible', 'on')
rlocus(PhiC*PhiP)
sgrid(z, wn)
axis([-35 1 -2 2])
% KPHI = rlocfind(PhiC*PhiP);
KPHI = 1;%2.61145e+03/2.5;
rphi = rlocus(PhiC*PhiP,KPHI);
rphi = rlocus(PhiC*PhiP,KPHI);
```

```
hold on
plot(rphi,'gs')

phiClosed = feedback(PhiC*PhiP*KPHI,1);
[wnPhi, zPhi] = damp(phiClosed);
figure('Name', 'Phi Step', 'Visible', 'on')
step(KPHI*PhiC)

figure('Name', 'Phi Controller', 'Visible', 'on')
bode(PhiC)

figure('Name', 'Phi Step Closed Loop', 'Visible', 'on')
step(phiClosed)

%% Theta Controller Design
% Theta controller is identical to Phi controller

%% Psi Controller Design
% Generate controller
wcpsi = 1*2*pi;
PsiC1 = pidtune(PsiP,'PIDF',wcpsi);
Tfpsi = PsiC1.Tf;
kipsi = PsiC1.Ki;
kdpsi = PsiC1.Kd;
kppsi = PsiC1.Kp;

% Assign variables
copter.Tfpsi = Tfpsi;
copter.kipsi = kipsi;
copter.kdpsi = kdpsi;
copter.kppsi = kppsi;

% Controller
PsiC = kppsi + kdpsi*s/(Tfpsi*s+1) + kipsi/s
figure('Name', 'Psi Controller', 'Visible', 'on')
rlocus(PsiC*PsiP)
sgrid(z, wn)
axis([-11 1 -2.5 2.5])
%KPSI = rlocfind(PsiC*PsiP);
KPSI = 1;%7*7.025128821795129e+03;
rpsi = rlocus(PsiC*PsiP, KPSI);
hold on
plot(rpsi, 'gs')

psiClosed = feedback(PsiC*PsiP*KPSI,1);
[wnPsi, zPsi] = damp(psiClosed);

%% Z Controller Design
% % Functional Requirements for Altitude
% OSz = 15;
% Tsz = 5;
%
% % System Requirements
% zz = -log(OSz/100)/sqrt(pi^2+log(OSz/100)^2);
% wnz = 4/(zz*Tsz);
%
% % Pole/Zero Locations
% z1z = -0.6;
% z2z = -1.2;
% p1z = -20;


% % Controller
% ZC = (s-z1z)*(s-z2z)/(s*(s-p1z));
% figure('Name', 'z Controller', 'Visible', 'on')
% rlocus(ZC*ZP)
% sgrid(zz, wnz)
% KZ = rlocfind(ZC*ZP);
% %KZ = 1.0929494e+04/3.5;
```

```
% rz = rlocus(ZC*ZP,KZ);
% axis([-21 1 -10 10])
% hold on
% plot(rz, 'go')

wcz = 1*2*pi;
ZC1 = pidtune(ZP,'PIDF',wcz);

Tfz = ZC1.Tf;
kiz = ZC1.Ki;
kdz = ZC1.Kd;
kpz = ZC1.Kp;

% Assign variables
copter.Tfz = ZC1.Tf;
copter.kiz = ZC1.Ki;
copter.kdz = ZC1.Kd;
copter.kpz = ZC1.Kp;

% Controller
ZC = kpz + kdz*s/(Tfz*s+1) + kiz/s

% Controller
%ZC = (s-z1z)*(s-z2z)/(s*(s-p1z));
figure('Name', 'z Controller', 'Visible', 'on')
rlocus(ZC*ZP)
sgrid(z, wn)
%KZ = rlocfind(ZC*ZP);
KZ = 1;%1.0929494e+04/3.5;
rz = rlocus(ZC*ZP,KZ);
axis([-21 1 -10 10])
hold on
plot(rz, 'go')

%% Set Controller Parameters for Simulink
copter.KPHI = KPHI;
copter.KTHETA = copter.KPHI;
copter.KPSI = KPSI;
copter.KZ = KZ;
copter.lag = 0.0;
copter.w0 = w0;
copter.dist = 0.015;
copter.cutoff = 1705;
copter.stopTime = 30;

%% Quadcopter Initial Conditions
IC.P = 0;
IC.Q = 0;
IC.R = 0;
IC.phi = 0;
IC.theta = 0;
IC.psi = 0;
IC.X = 0;
IC.Y = 0;
IC.Z = 0;
IC.w1 = 0;
IC.w2 = 0;
IC.w3 = 0;
IC.w4 = 0;

%% Set State Values
set_param('model_of_quadcopter/Phi_cmd','after','0');
%set_param('model_of_quadcopter/Theta_cmd','after','0');
copter.thetaref = 0;
copter.thetaref1 = -20*pi/180;
copter.thetasteptime = 7.475;%5.05;
set_param('model_of_quadcopter/Psi_cmd','after','0');
set_param('model_of_quadcopter/Alt_cmd','after','0');

```

```
%% Set Disturbance Values
copter.M1D = 0;
copter.M2D = 0;
copter.M3D = 0;
copter.M4D = 0;

stepForce = .1; % Newton

copter.U1D = 0;%stepForce*L;
copter.U2D = 0;
copter.U3D = 0;%stepForce;
copter.U4D = 0;%-(abs(copter.U1D)/L + abs(copter.U2D)/L);

copter.s1 = 6;
copter.s2 = 6;
copter.s3 = 6;
copter.s4 = 6;

stepInput = sqrt(stepForce/b);

%% Simulate
%open('model_of_quadcopter');
sim('model_of_quadcopter');

%% Set Variables
% State Variables
Phi = simout.States.Phi;
Theta = simout.States.Theta;
Psi = simout.States.Psi;
x = simout.States.x;
y = simout.States.y;
zz = simout.States.z;

% Motor Inputs
M1 = simout.States.M1;
M2 = simout.States.M2;
M3 = simout.States.M3;
M4 = simout.States.M4;

% Torque Inputs
U2 = simout.States.U1;
U3 = simout.States.U2;
U4 = simout.States.U3;
U1 = simout.States.U4;

% Motor Torques
Tj1 = simout.signal2.Tj1;
Tj2 = simout.signal2.Tj2;
Tj3 = simout.signal2.Tj3;
Tj4 = simout.signal2.Tj4;

%% Torque and Current Calcs
T1 = Tj1.Data + M1.Data.^2*copter.cq;
T2 = Tj2.Data + M2.Data.^2*copter.cq;
T3 = Tj3.Data + M3.Data.^2*copter.cq;
T4 = Tj4.Data + M4.Data.^2*copter.cq;

I1 = T1/kt;
I2 = T2/kt;
I3 = T3/kt;
I4 = T4/kt;
ENERGY = 11.1*(trapz(M1.Time,I1) + trapz(M2.Time,I2) + trapz(M3.Time,I3) +
trapz(M4.Time,I4));
BATTERY = 5000/1000*1*3600*11.1;
times = BATTERY/ENERGY;
%% Plots
% State Variables
figure('Name', 'States', 'Color', [0, 0.4470, 0.7410], 'Visible', 'on')

```

```
subplot(3,2,1)
plot(Phi.Time, Phi.Data*180/pi,'--b','linewidth',2)
grid on
grid minor
xlabel('time (sec)')
ylabel('\phi response (deg)')
title('\phi vs. t')

subplot(3,2,3)
plot(Theta.Time, Theta.Data*180/pi,'--r','linewidth',2)
grid on
grid minor
xlabel('time (sec)')
ylabel('\theta response (deg)')
title('\theta vs. t')

subplot(3,2,5)
plot(Psi.Time, Psi.Data*180/pi,'--m','linewidth',2)
grid on
grid minor
xlabel('time (sec)')
ylabel('\psi response (deg)')
title('\psi vs. t')

subplot(3,2,2)
plot(x)
grid on
grid minor
xlabel('time (sec)')
ylabel('x response (m)')
title('x vs. t')

subplot(3,2,4)
plot(y)
grid on
grid minor
xlabel('time (sec)')
ylabel('y response (m)')
title('y vs. t')

subplot(3,2,6)
plot(zz.Time, zz.Data*100,'--k')
grid on
grid minor
xlabel('time (sec)')
ylabel('z response (cm)')
title('z vs. t')

% % Motor Inputs
figure('Name', 'Motor Speeds', 'Visible', 'on')
plot(M1.Time,M1.Data, '--k','linewidth',2);
hold on
plot(M2.Time,M2.Data, '--r','linewidth',2);
hold on
plot(M3.Time,M3.Data, '-og','linewidth',0.5);
hold on
plot(M4.Time,M4.Data, '-ok','linewidth',0.5);
legend('M_1', 'M_2', 'M_3', 'M_4')
xlabel('time (sec)')
ylabel('motor speeds (rad/s)')
title('Motor Speeds')
grid on

% Torque Inputs
figure('Name', 'Applied Torques', 'Visible', 'on')
plot(U1, '--or')
hold on
plot(U2, '--b')
hold on
```

```
plot(U3, '--r')
hold on
plot(U4, '--k')
legend('U_1', 'U_2', 'U_3', 'U_4')
grid on
xlabel('time (sec)')
ylabel('copter torques (N-m)')
title('Applied Torques')

% Motor Torques
figure('Name', 'Torques', 'Visible', 'on')
plot(M1.Time, T1, '--k','linewidth',2);
hold on
plot(M2.Time, T2, '--r','linewidth',2);
hold on
plot(M3.Time, T3, '-og','linewidth',0.5);
hold on
plot(M4.Time, T4, '-ok','linewidth',0.5);
grid on
legend('T_1', 'T_2', 'T_3', 'T_4')
xlabel('time (sec)')
ylabel('motor torques (N-m)')
title('Motor Torques')

% Motor Currents
figure('Name', 'Currents', 'Visible', 'on')
plot(M1.Time, I1, '--k','linewidth',2);
hold on
plot(M2.Time, I2, '--r','linewidth',2);
hold on
plot(M3.Time, I3, '-og','linewidth',0.5);
hold on
plot(M4.Time, I4, '-ok','linewidth',0.5);
grid on
legend('I_1', 'I_2', 'I_3', 'I_4')
xlabel('time (sec)')
ylabel('motor currents (A)')
title('Motor Currents')

maxSpeed = max([max(M1.Data) max(M2.Data) max(M3.Data) max(M4.Data)]);
maxTorque = max([max(T1) max(T2) max(T3) max(T4)]);

maxV = maxSpeed/kv;
MaxI = maxTorque/kt



% %% Test for Controller vs. Simulation
% load('5_23_Test_4.mat');
% dt = 0.005;
% tTest = transpose(dt*(0:1:length(a_x)-1));
%
% feedback1 = 0-comp_x;
%
% xoutTest = lsim(KPHI*PhiC,feedback1,tTest);
% m2Test = -xoutTest+1280;
% m4Test = xoutTest+1280;
%
% %r111 = rms(m2Test)
%
% figure('Name', 'Test Motor Speeds', 'Visible', 'on')
% % subplot(2,1,1)
% plot(tTest, m2Test, 'k--', 'linewidth', 1.5)
% hold on
% plot(tTest, Omega2)
% xlabel('time (sec)')
% hold on
% plot(tTest, m4Test, 'b--', 'linewidth', 1.5)
% hold on
```

```
% plot(tTest, Omega4)
% ylabel('Speed (rad/s)')
% title('M2')
% legend('M2 Actual','M2 Sim','M4 Actual','M4 Sim')
% %legend('M2 Sim','M4 Sim')


% subplot(2,1,2)
% plot(tTest, m4Test, 'k', 'linewidth', 1.5)
% hold on
% plot(tTest, Omega4)
% xlabel('time (sec)')
% ylabel('Speed (rad/s')
% title('M4')
% legend('Simulated','Actual')

load('DataCollection17.mat');
dt = 0.005;
ttest = transpose(dt*(0:1:length(comp_x)-1));
tshift = 0;%5-2.55-3.55+3.15;

datshift = 1376;%+751;
comp_temp = comp_y(1:datshift);
comp_temp2 = comp_y(datshift+1:end);
comp_y(1:length(comp_temp2)) = comp_temp2;
comp_y(length(comp_temp2)+1:end) = comp_temp;

comp_temp3 = comp_x(1:datshift);
comp_temp4 = comp_x(datshift+1:end);
comp_x(1:length(comp_temp4)) = comp_temp4;
comp_x(length(comp_temp4)+1:end) = comp_temp3;

figure(3)
subplot(2,1,1)
plot(ttest + tshift,comp_x)
hold on
plot(Phi.Time, Phi.Data*180/pi,'--r','linewidth',2)
xlabel('time (sec)')
ylabel('angle (deg)')
title('comp_x from C')

subplot(2,1,2)
plot(ttest + tshift,(comp_y)*180/pi)
hold on
plot(Theta.Time, Theta.Data*180/pi,'--r','linewidth',2)
xlabel('time (sec)')
ylabel('angle (deg)')
title('comp_y from C')

%% Define Low Pass and High Pass Filters for Comp
% Break Frequency
wb = 0.12*2*pi;
tau = 1/wb;
wb2 = wb;
tau2 = 1/wb2;
% Filters
low = 1/(tau*s+1);
high = (tau2*s)/((tau2*s+1)*s);

% Z Filters
wb3 = 5*2*pi;
tau3 = 1/wb3;
lowz = 1/(tau3*s+1)^2;
%% Convert Controllers and Filters to C Header Files
T = 0.005;
K_motor = 1;
%--------------Phi Controller-------------------------------
c = KPHI*PhiC;
cd = c2d(c,T,'tustin');
```

```
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\PhiController.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'PhiController',T,comment);
fclose(fid);
%--------------Theta Controller-----------------------------
c = KPHI*PhiC;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\ThetaController.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'ThetaController',T,comment);
fclose(fid);
%--------------Psi Controller-----------------------------
c = KPSI*PsiC;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\PsiController.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'PsiController',T,comment);
fclose(fid);
%--------------Z Controller-----------------------------
c = KZ*ZC;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\ZController.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'ZController',T,comment);
fclose(fid);
%--------------Low Pass Filter Phi-----------------------------
c = low;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\LowPassFilterPhi.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'LowPassFilterPhi',T,comment);
fclose(fid);
%--------------High Pass Filter Phi-----------------------------
c = high;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\HighPassFilterPhi.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'HighPassFilterPhi',T,comment);
fclose(fid);
```

```
%--------------Low Pass Filter Theta-----------------------------
c = low;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\LowPassFilterTheta.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'LowPassFilterTheta',T,comment);
fclose(fid);
%--------------High Pass Filter Theta-----------------------------
c = high;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\HighPassFilterTheta.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'HighPassFilterTheta',T,comment);
fclose(fid);
%--------------Low Pass Filter Psi-----------------------------
c = low;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\LowPassFilterPsi.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'LowPassFilterPsi',T,comment);
fclose(fid);
%--------------High Pass Filter Psi-----------------------------
c = high;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\HighPassFilterPsi.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'HighPassFilterPsi',T,comment);
fclose(fid);
%--------------Low Pass Filter Z-----------------------------
c = lowz;
cd = c2d(c,T,'tustin');
cdt = tf(cd);
[b,a] = tfdata(cdt);
b = b{1}; a = a{1};
sos = tf2sos(b,a);
HeaderFileName = 'C:\Users\michael.Local\Desktop\LowPassFilterZ.h';
comment = 'Michael Janicki';
fid = fopen(HeaderFileName,'W');
sos2header(fid,sos,'LowPassFilterZ',T,comment);
fclose(fid);
```