Intuitive and Ergonomic Gesture-Based Drone Controller

By

Elaine Houha

Adam Poindexter

Final Report for ECE 445, Senior Design, Spring 2019

TA: Channing Philbrick

1 May 2019

Team No. 22

Abstract

This project details the creation of a gesture-based controller for commercially available drones. The device is a glove worn by the user that serves as a complete replacement for a standard controller. The user controls the drone by holding out their hand, palm down, and miming the desired drone movements. For example, tilting the hand to the right would cause the drone to move to the right. The device passed all verification procedures and successfully flew the available drone on multiple occasions throughout testing.

Contents

1 Introduction

There are currently 1.4 million Unmanned Ariel Vehicles in the United States as of 2018, and the Federal Aviation Administration expects that to double by 2021 [1]. Currently, many market available drones are flown from a standard radio transmitter much like how one would drive an RC car. Our product intends to change this method of controller-based flight to one that is gesture-based with a controller mounted on a glove that uses the users own movements control the flight of the drone. The glove will be compatible with a wide variety of hobbyist drones, eliminating the need for a separate controller for each drone a person may own. See Appendix C for compatible products. Additionally, this provides users a new novel way to fly. Through much trial and error, a successful product was built where a person could fly a drone using their hand movements, and the next chapters describe the process of this design from concept to fabrication, then verification and testing, and finally flying.

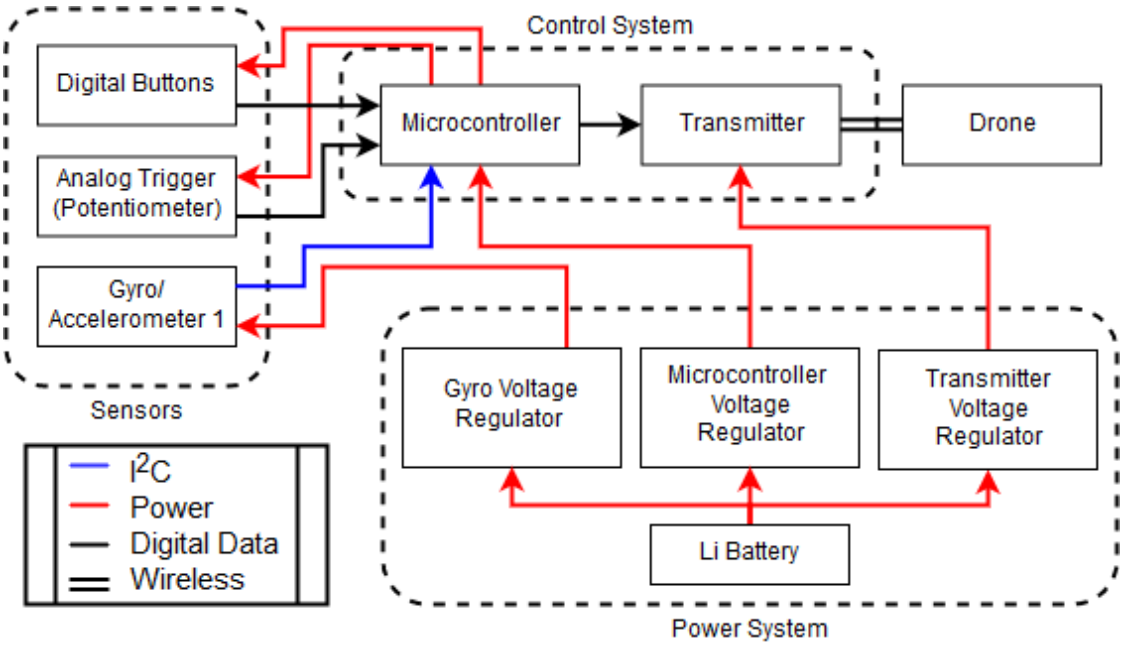

Figure 1: Block Diagram

Figure 1 shows the block diagram for the overall system and the three subsystems that make up the controller. The subsystems are power, control, and sensors. The power system steps the battery voltage down to the required voltages needed for the micro-controller, transmitter, and inertial measurement unit (IMU). The control system filters the data from the sensors and formats in a way the transmitter will be able to send it to the drone. The sensor system is a combination of debounced buttons, a potentiometer, and an IMU. This is a change from our original design where two IMUs were used; the change was made based off more research into the type of filtering algorithms used on IMU data this is discussed in detail in Section 2.4.

2 Design

Before delving into the electrical components of the glove there were considerations made to the physical placement of the glove as well as how the analog trigger would function on the glove. See Figure 2 below for pictures of the final glove. The IMU unit is in the top left corner of the PCB. While the control subsystem is centrally located in the middle of the PCB with the transmitter resting above the micro-controller. The digital buttons are in the bottom center of the PCB and to the right of the silver reset button. The analog trigger mechanism is on the underside of the glove between the index and middle finger spots with a lever that faces towards the thumb for ease of control when flying. In addition to the hardware components, we also designed a software filtering and control system.

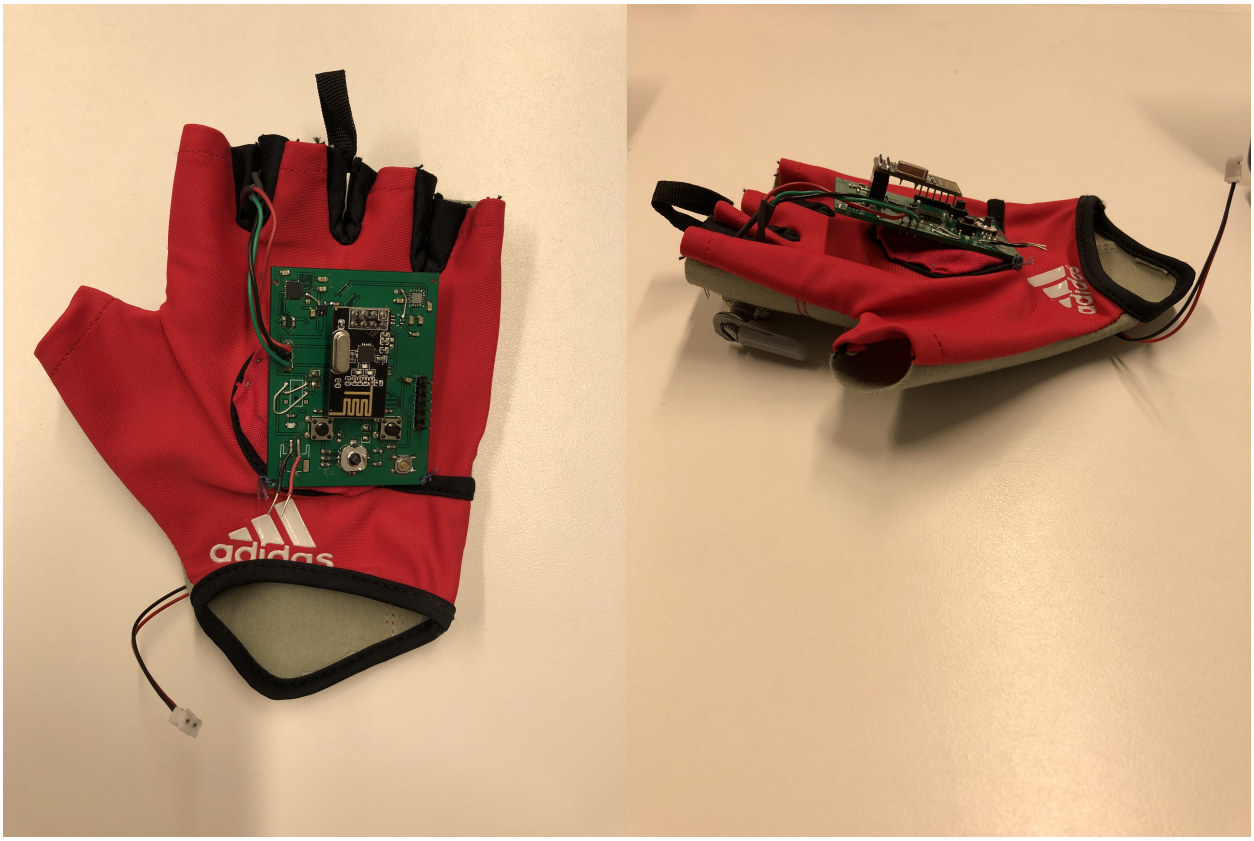

Figure 2: Top and side views of the glove

2.1 Power Subsystem

The power systems consists of two sub-components, a battery and voltage regulators. See Appendix B for schematic and Appendix A for full requirements and verification procedures.

2.1.1 Battery

The main requirements for the battery where it had to last for a period of about one hour, so the user could have decent amount of flight time with the drone. It also had to be compact so it would add unnecessary bulk to the glove which could cause user discomfort. A 3.7V Lithium-Polymer battery was chosen for these

reasons.

2.1.2 Voltage Regulators

The battery's 3.7V output provided a higher voltage than what many of the components' specifications allowed. Regulators were used to consistently reduce the voltage to 3.3V operating voltage of the IMU, micro-controller and transmitter.

2.2 Control Subsystem

The control subsystem consists of two sub-components, a micro-controller and a transmitter. See Appendix B for the schematic and Appendix A for full requirements and verification procedures.

2.2.1 Micro-controller

The requirement for the micro-controller is that it needed to be able to process the filtering algorithm within 6ms. The controller also had to have enough inputs and outputs for the specified design to ensure proper function. Hence the choice for the ATMega328PB, the same chip that is on the Arduino Pro Mini, as it has 13 digital pins, 6 analog pins, MOSI/MISO pins for SDI communication, and SDA/SCL pins for I2C communication.

2.2.2 Transmitter

The requirement for the transmitter is it needed to communicate with the drone, so we could control it from the glove. In this case we had two design options, design our own transmitter and antenna or use a stock ready-made breakout board transmitter. For the sake of time and ease we chose a ready-made transmitter, the NRF2401+, as it has an Arduino library to communicate with the micro-controller and there's an opened-sourced library developed by hobbyists that uses this chip to communicate with the type of drone we are using in the project [2].

2.3 Sensor Subsystem

This subsystem provides all the raw data for the micro-controller to process and then package for the transmitter to transmit to the drone. It consists of three sub-components: digital buttons; analog trigger; and an inertial measurement unit. See Appendix B for a schematic and Appendix A for full requirements and verification procedures.

2.3.1 Digital Buttons

The digital buttons in the design will control trim. This is important as it provides error corrections for variations in motor power on each of the propellers on the drone to provide a more stable flight. The requirement for the buttons is they only register one input per press, also known as debouncing.

2.3.2 Analog Trigger

The analog trigger in the block diagram is described as a potentiometer and is used to control the thrust on the drone. Originally, the design was to have discreet power levels where the user would push a lever which would rotate the potentiometer and based on the angle of the lever would provide fraction of the maximum thrust available. Figure 3 shows the division of power as the potentiometer turned as described for

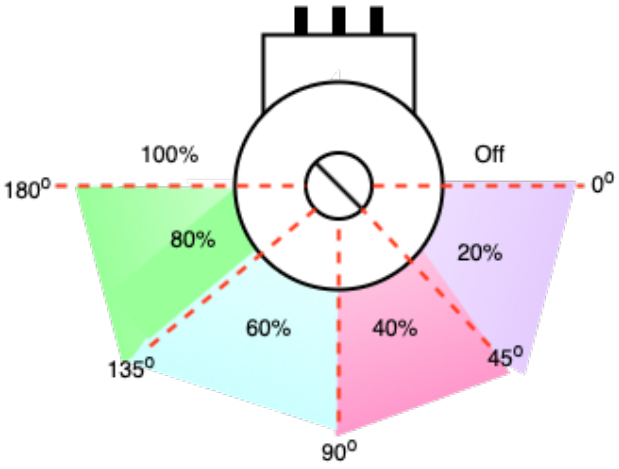

Figure 3: Original design of discrete thrust values

the original design. However, during testing it was discovered we could linearly scale the value read by the micro-controller from the potentiometer to create continuous thrust values. This is discussed in detail in Section 3.3.2. The trigger also needed a lever so the user could easily turn the potentiometer by pushing and pulling the lever with the thumb. Figure 4 is a rendering of the lever which was 3D printed.

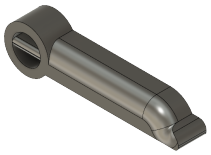

Figure 4: Design of Trigger Lever

2.3.3 Inertial Measurement Unit

The inertial measurement unit (IMU) is the crux of the design. The requirements for the unit are the accelerometer had to measure hand movements within 0.2g of force and the gyroscope had to be to measure hand movements within 3°. During the creation of the design there were several options to consider: a separate gyroscope and accelerometer; a combined 6-axis gyroscope-accelerometer; and 9-axis gyroscope-accelerometer-magnetometer combination. Since the data coming from this unit is filtered in the micro-controller, we could use a relatively inexpensive IMU and to save space on the board decided a combined 6-axis unit would best fit our needs.

2.4 Filtering

Our original design called for the combination of data from two different inertial measurement units (IMUs) through a Kalman filter in order to acquire stable data for flight control. However, after researching Kalman

filters in greater depth, we realized that this was a poor approach. We came to understand that Kalman filters excelled in combining data points into a stable output by examining the mathematical relationships between the differing data sources. In the case of a single 6-axis gyro-accelerometer, relationships between acceleration and rate of rotation are developed geometrically. For example, a rotation about the x-axis will cause changes in the acceleration of the y-axis and z-axis.

In order to design a Kalman filter for the purpose of combining the sensor data from two identical sensor units, a relationship between the two sensors (and each of their 6 axes) would need to be developed. The best relationship we could create was that of a simple average. Other than the additional computations for the averaging, the filter mathematics would be merely extensions of the single sensor algorithm. Upon discovering this, we scraped the two sensor idea altogether. The cost of the extra sensor and the greatly increased processing time did not add enough value to be worth it.

Deciding to look in another direction, we began researching how to make one IMU as stable as possible through an alternative filtering method. The filtering system we found is called the Madgwick filter [3]. This filter is specifically designed to increase the stability of low-cost IMU data for measuring the orientation of an object. See Figure 5 for the block diagram of the algorithm.

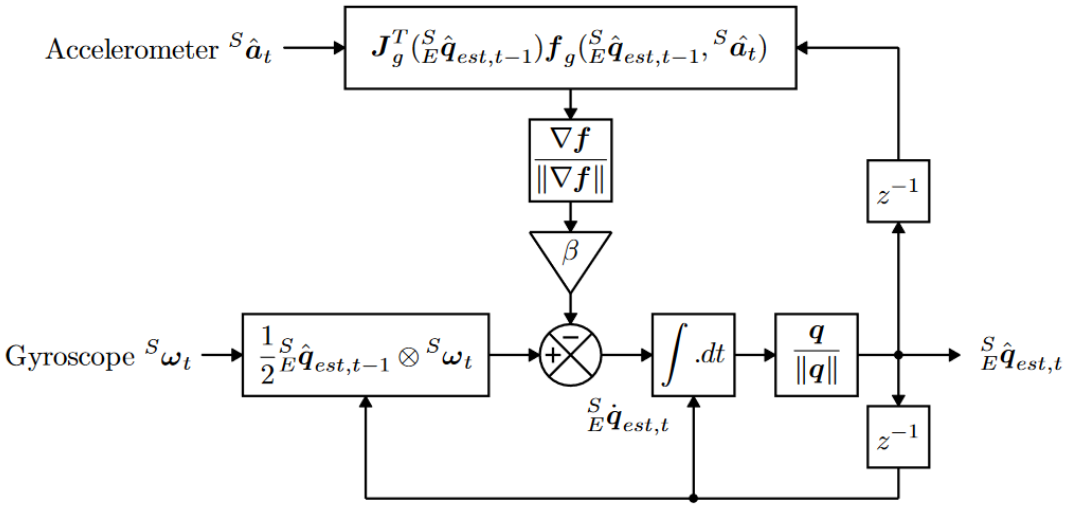

Figure 5: Madgwick Filter Block Diagram [3]

The algorithm first calculates the direction of the gravity vector using a gradient decent algorithm. From this, orientation with respect to the earth can be easily determined with simple trigonometry. The gradient decent operation outputs a quaternion matrix that is then normalized and scaled by a tuning factor β. This information is then combined with the change of orientation information provided by the gyroscope. If the two information sources do not correspond, the result is canceled out. The output is integrated over the time between algorithm runs and then normalized once more. This final quaternion is used to calculation roll, pitch, and yaw using conversion formula. The system operates in continuous feedback so that momentary noise can be accounted for.

The original report by Madgwick contained graphs comparing the designed filter against a Kalman filter in the same setting. These are included in Figures 6 and 7 below. As you can see, the Madgwick filter is

superior.

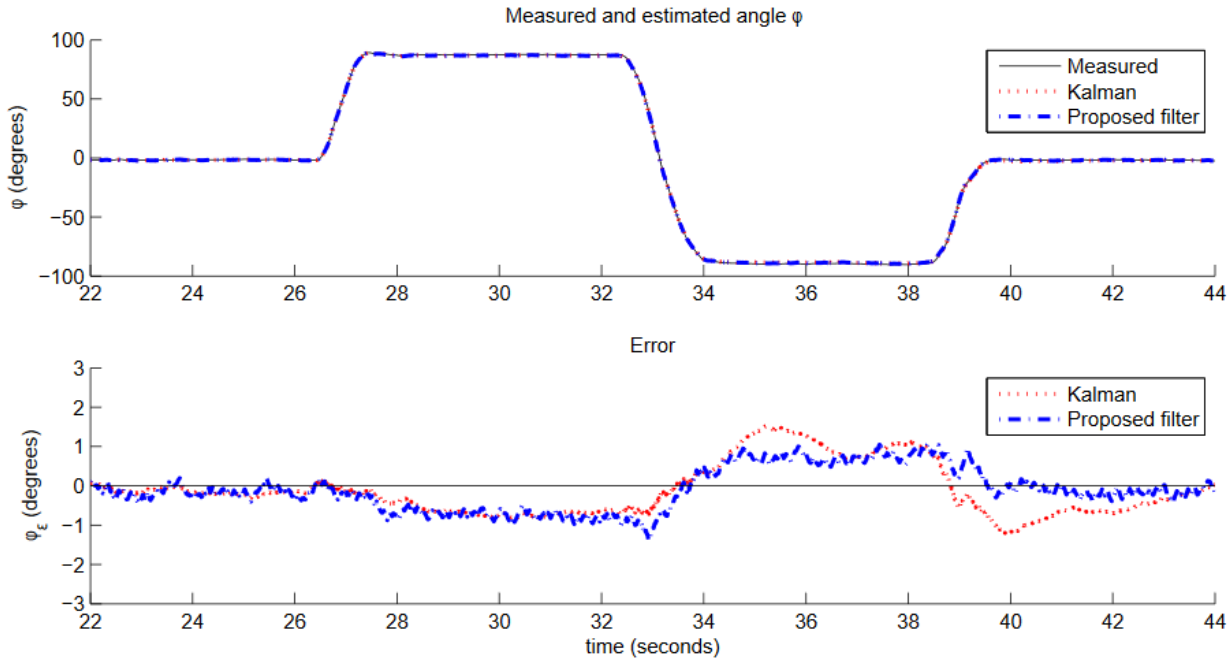

Figure 6: Comparison 1 of the Kalman and Madgwick filters [3]

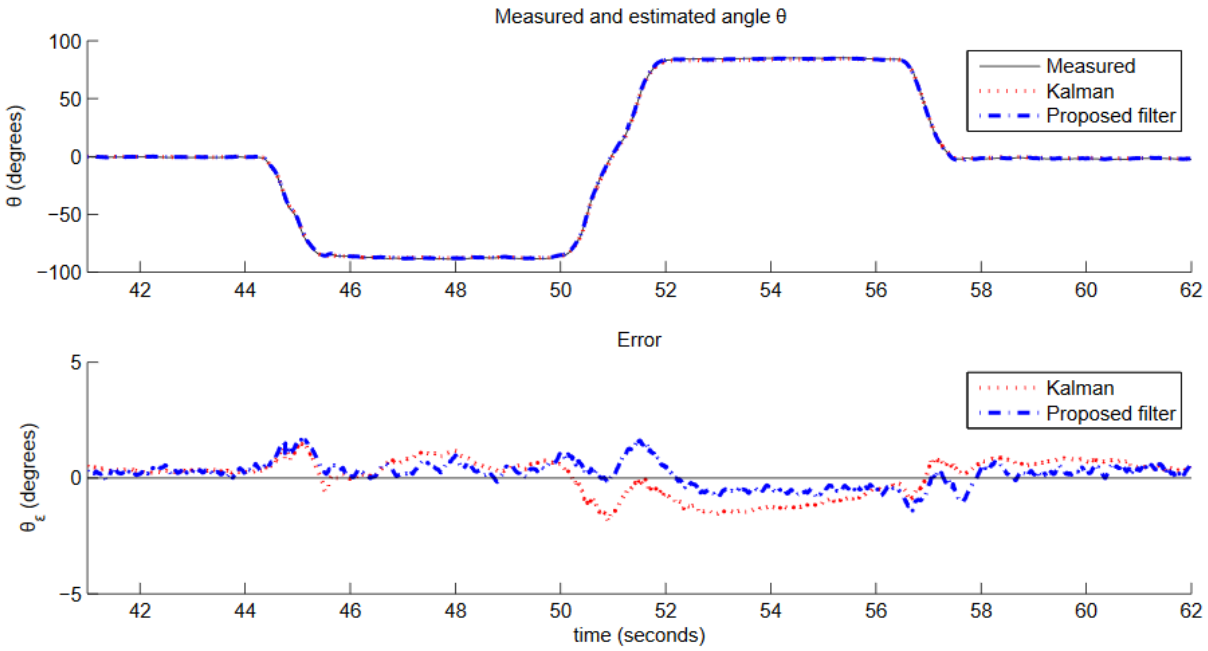

Figure 7: Comparison 2 of the Kalman and Madgwick filters [3]

3 Design Verification

For full details on the procedures to verify the requirements set in the previous chapter please see Appendix A for the requirements and verification tables.

3.1 Power Subsystem

3.1.1 Battery

To verify the battery provided the required 3.7V over at least an hour, a rechargeable 3.7V battery was placed in the circuit and the drone was flown. It passed this requirement by being able to control the drone for an hour and did not need charging.

3.1.2 Voltage Regulators

The voltage regulators were tested by running a voltage larger than 3.3V into them and making sure the output was consistently around 3.3V. See the graphs in Figures 8 below.

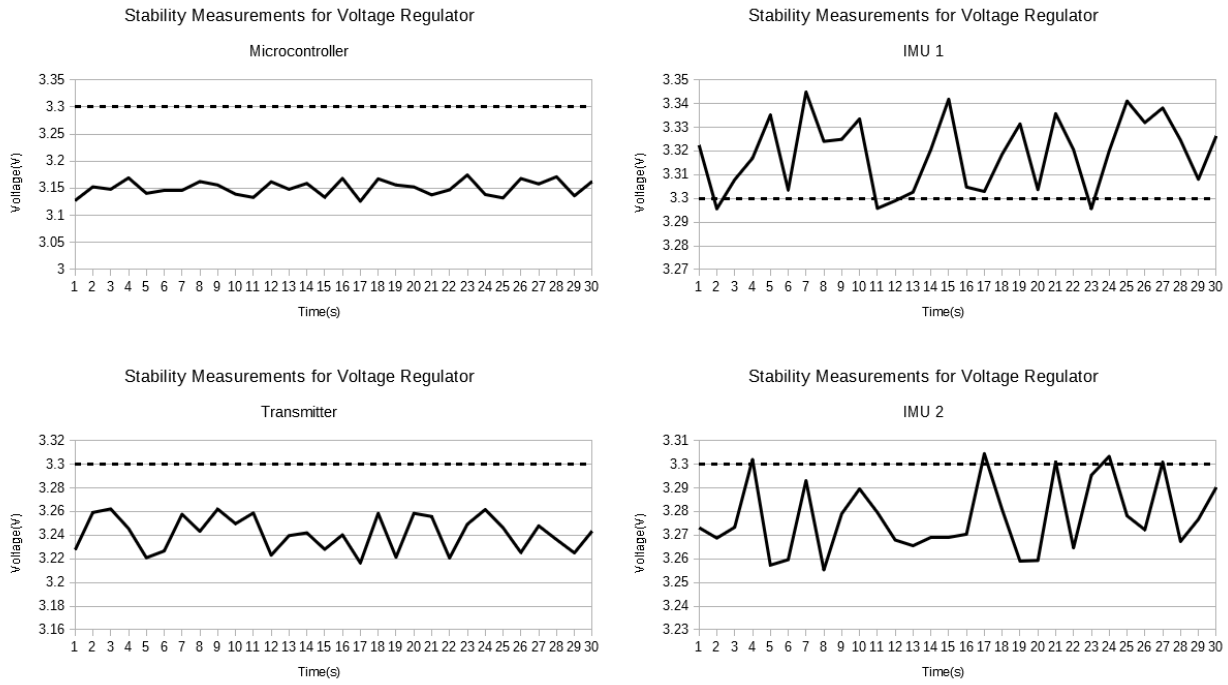

Figure 8: Voltage Test

3.2 Control Subsystem

3.2.1 Micro-controller

To the test the micro-controller we created a circuit with a breakout board that had the chosen IMU on it and loaded the chosen filtering algorithm onto the controller. Then we added a timer to the serial print how long the full operation took for each point of data received by the controller from the IMU. The output of the monitor consistently showed between 6ms and 8ms to run the algorithm. See Listing 4 in Appendix D for

the verification code.

3.2.2 Transmitter

The transmitter was tested by powering on the drone and controller while in proximity to one another. After the drone calibrates its internal gyroscope it begins to the search for an RF signal from a controller. The transmitter then sends out a pairing command and if the lights on the drone went from blinking to solid then the controller was successfully paired with the drone. This occurred correctly in roughly 80% of trials and in the remaining 20% a simple reboot of the controller solved the issue.

3.3 Sensor Subsystem

3.3.1 Digital Buttons

To test the buttons with created a circuit according to the Figure 11 in Appendix B and ran the script in Listing 3 in Appendix D then looked at the serial output on the monitor to ensure each button only registered once per press. This worked flawlessly.

3.3.2 Analog Trigger

After verifying the chosen 10 kΩ potentiometer had a full actuation between (1.5 ± 0.1) V with a multi-meter; testing the analog trigger required creating a circuit and measuring the read values from the analog-in pin on the micro-controller and mapping it the angle the potentiometer was turned. From these values the data was then converted to linearly map to values for thrust on the controller. Below in Table 1 shows the angle the potentiometer was at along with raw data value the micro-controller read. After finding these values a test script was made to linearly translate the raw data into a number from 0-1000 which is the range of values the library we used to communicate with the drone uses for thrust. Equation 1 is the equation used to translate the raw data into a usable number by the micro-controller where T is average of ten raw thrust values.

$$Thrust = (T - 50)/625 * 1000 \tag{1}$$

Table 1: Angle of potentiometer and Raw Data Value

Angle (°)	Data Value
0	50
45	312
90	430
135	545
180	675

3.3.3 Inertial Measurement Unit

Much like the other sensors in the this section to test the requirements of the IMU we built a circuit according to the schematic in Appendix B. Then we ran the specified scripts in Appendix D. The gravitational measurements displayed correctly and the failsafe for motions over 2g triggered as planned. The angular measurements were well within the tolerance typically 1° or less off the real value.

4 Cost

4.1 Parts

Table 2: Costs for Parts

Part	Manufacturer	Retail ($)	Bulk ($)	Count	Actual ($)
MPU-6050	Invensense	12.95	8.31	4	42.52
ATMega328	Atmel	4.25	1.46	4	14.21
NRF24L01+	Nordic Semiconductor	2.37		4	9.50
Resonator	Murata	0.28		3	0.84
LEDs	ROHM Semiconductor	0.40		3	1.20
Resistors-10K	Yageo	0.58	0.41	10	4.12
Resistors-1K	Yageo	0.68	0.38	20	7.62
MIC5205	Microchip	0.41		10	4.06
Capacitors-Varied	Yaego	0.27		50	13.54
Custom PCB	PCB way	1.00		5	5.00
Battery	Sparkfun	9.95		1	9.95
Physical Glove	Adidas	5.00		1	5.00
Poteniometer	Panasonic	0.95		1	0.95
3D Printed Trigger	Illinois Maker Lab	2.00		1	2.00
Total					120.51

4.2 Labor

Table 3: Costs for Labor

Worker	Rate	Hours	Multiplier	Cost
Elaine	$38/hr	60	2.5	$5,700
Adam	$38/hr	60	2.5	$5,700
Total				$11,400

5 Conclusion

5.1 Accomplishments

In the end we were able to build a low-profile glove to control a drone. While flying a drone still takes practice, using hand movements to map controls gives a more intuitive sense of how to fly. For example, tilt your hand to the left the drone will also follow to the left. Achieving a latency of less than 50 ms is an achievement because it means the delay between the movement of the controller and the drone making the movement is not seen by the human eye improving user experience. Also by only using one IMU and the Madgwick filter instead of two IMUs and the Kalman filter the overall cost of the design is reduced. This bodes well if the product was to go to market as the market price could be lowered.

5.2 Ethical Considerations

Our project poses possible safety concerns in two ways. The first, electrocution, rises from the fact that we are attaching electrical components to a wearable device; however, the battery size we are using (3.7 V) is not powerful enough to cause a harmful current across human skin. The minimum resistance for dry skin is $\sim$1000 Ω which results in a current much too small to cause damage [4]. This resistance is greatly reduced when wet so a warning would be attached to the product advising not to operate when wet.

The second possible concern is the drone injuring someone while being controlled by our product. Unlike the former issue, this one cannot be dismissed. All drone operation is accompanied by some hazard for injury to oneself and others. In addition to warnings advising users to not fly in crowded areas or near people, we included an emergency shut-off feature as part of the glove. This disables the drone in the event of user becoming incapacitated or device malfunction.

Addressing both of these concerns falls under the IEEE Code of Ethics Policy 1, "hold paramount the safety[...]of the public[...]and to disclose promptly factors that might endanger the public[...]". We feel that the warnings and preventative measures we have enacted adequately satisfy this mandate.

Drones for public and private use are regulated by the Federal Aviation Administration (FAA). According to current guidelines, all "Unmanned Aircraft Systems" being piloted in public airspace for non-recreational, non-commercial purposes fall under "Part 107" [5]. This regulation mandates that all drones used for educational purposes be registered with the FAA and all operators be "FAA-Certified Drone Pilots". As both of these processes require time and money, we decided to confine our testing to within campus buildings. The regulations do not cover indoor spaces allowing for a streamlined project timetable.

Additionally, the University of Illinois at Urbana-Champaign has a policy on drones used for non-recreational purposes that mandates operators obtain approval from the Division of Public Safety if operating outdoors on campus property and from Code Compliance & Fire Safety if operating indoors on campus property [6]. We obtained these authorizations will little difficulty.

5.3 Future work

The future for this project would include a redesign of the circuit board to make it smaller and fix the minor mistakes in wiring that were made in the first version of the design. By making the board smaller we improve placement on the top of the hand and the overall glove lighter in design. Another consideration would be

making it easier for the end user to switch what type of drone they are flying with the glove. This could be with more buttons or an app that would sync with the glove and pick the right programming based on the drone being flown.

Since the control signals are being sent through a relatively strong processor before being sent to the drone, there are a large amount of software improvements that could be made to ease the control of the drone. One of the trickiest parts of flying a drone is thrust compensation—the act of slightly throttling up when other movements are made in order to counteract the slight dip in thrust that occurs. With relatively little work this task could be, at least partially, automated by the controller.

References

[1] A. Pasztor, "Faa projects fourfold increase in commercial drones by 2022," Mar 2018. [Online]. Available: https://www.wsj.com/articles/faa-projects-fourfold-increase-in-commercial-drones-by-2022-1521407110

[2] Goebish, "nrf24_multipro," Feb 2019. [Online]. Available: https://github.com/goebish/nrf24_multipro

[3] S. O. H. Madgwick, "An efficient orientation filter for inertial and inertial / magnetic sensor arrays," 05 2010.

[4] "Q & a: The human body's resistance," Oct 2007. [Online]. Available: https://van.physics.illinois.edu/qa/listing.php?id=6793

[5] "Recreational fliers & modeler community-based organizations," Feb 2019. [Online]. Available: https://www.faa.gov/uas/recreational_fliers/

[6] E. D. of Public Safety, "Aerial activities over, on, or in campus property," Sep 2015. [Online]. Available: https://cam.illinois.edu/policies/fo-05/

Appendix A Requirement and Verification Tables

All verification tests were successful and therefore individual success/failure indicators were excluded from this appendix.

Table 4: Requirements and Verification

Requirement	Verification
Each button should register exactly one input per press (debouncing).	1. Attach USB adapter and power on device. 2. Load and run *buttonTest.ino*. 3. Press the button and observe the on-screen data. The counter should only increment by one. 4. Repeat step 3 five times. 5. Repeat steps 2-4 for all buttons.
Full actuation of the trigger mechanism should have a range of $(1.5 \pm 0.1)\,\mathrm{V}$.	1. Power on the device. 2. Attach a multi-meter to the potentiometer output and ground. 3. Actuate the trigger from full extension to full retraction noting the voltages at the extremes. 4. The difference between the maximum voltage and the minimum voltage should match the requirement. $\left(\left\|1.5 - (V_{max} - V_{min})\right\| \leq 0.1\right)$
The accelerometer must be able to measure user hand movements within $0.2\,\mathrm{g}$.	1. Attach USB adapter and power on device. 2. Load and run *generalTest.ino*. 3. Orient the glove so that the z-axis is up. 4. Verify the z acceleration display on the computer reads $(1.0 \pm 0.2)\,\mathrm{g}$. 5. Repeat steps 3-4 for the x and y axes.
The gyroscope must be able to measure user hand orientation within $3°$.	1. Attach USB adapter and power on device. 2. Load and run *generalTest.ino*. 3. Using a protractor or digital leveling tool, position the device at $20°$. Verify the angle displayed on the computer matches the requirement and positioning. 4. Repeat step 3 for $40°$ and $60°$.

Table 5: Requirements and Verification (cont.)

Requirement	Verification
Must be able to preform the Madgwick filtering algorithm on the IMU data in less than 6 ms.	1. Attach USB adapter and power on device. 2. Load and run *generalTest.ino*. 3. Slowly rotate device from $-90°$ to $90°$ along the pitch axis. Verify that the times displayed on the computer match the requirement. 4. Repeat step 3 for roll and yaw axes. 5. Repeat steps 3-4 for quick rotations.
The transmitter must be able to communicate with the drone.	1. Power on drone and wait for lights to stabilize to a slow blink. 2. Power on device. 3. Wait appx. 5 seconds for device to pair. 4. Verify that the lights transitioned from a slow blink to a solid state signifying a connection.

Appendix B Schematics

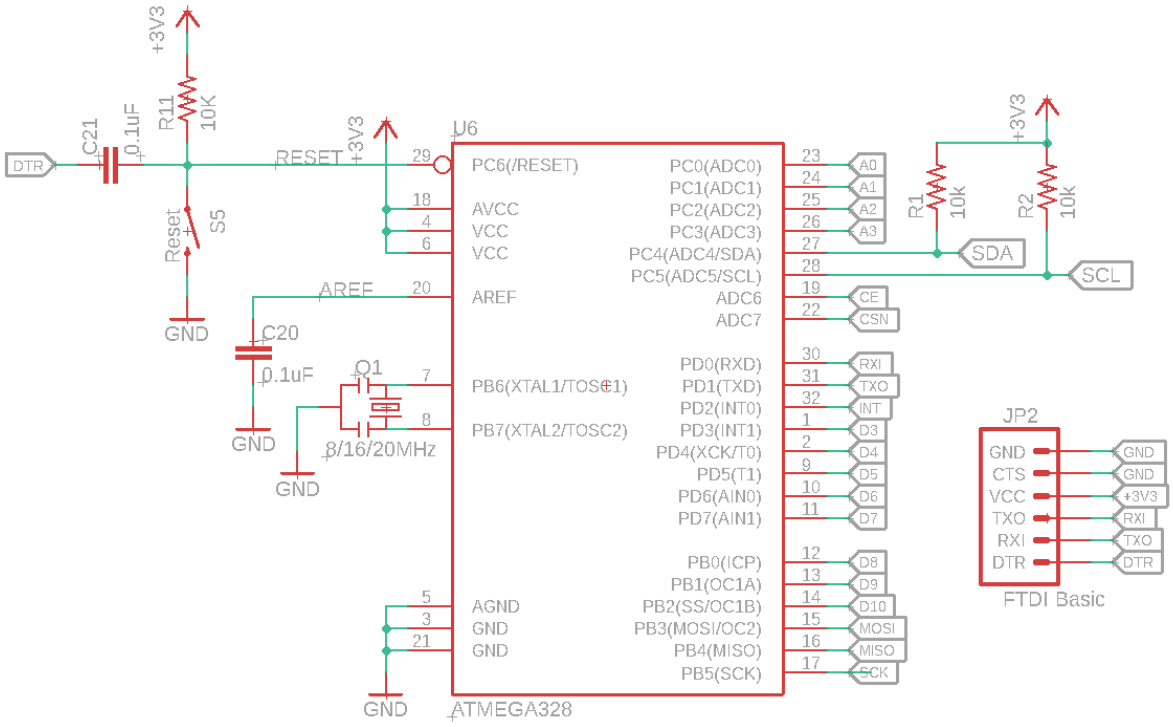

Figure 9: Microcontroller

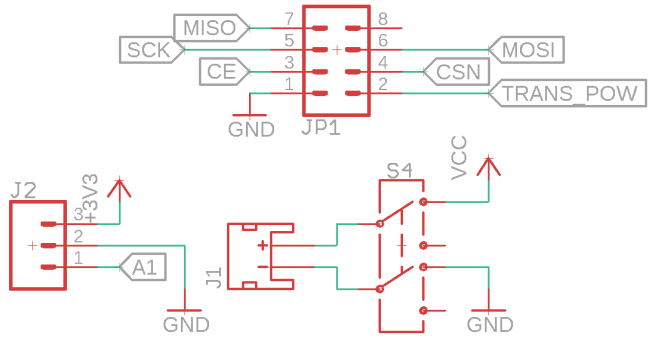

Figure 10: Transmitter, Battery, and Trigger Connections

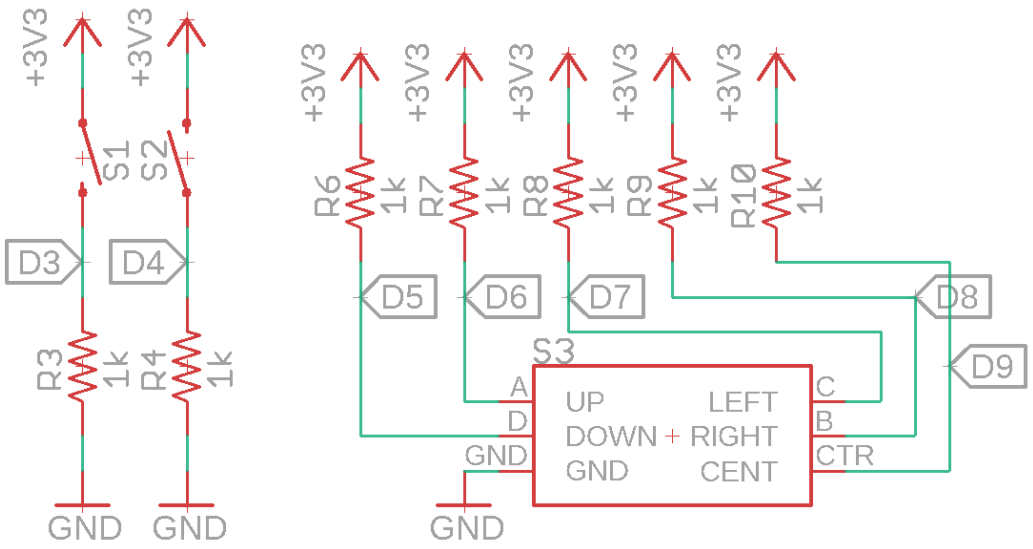

Figure 11: Buttons

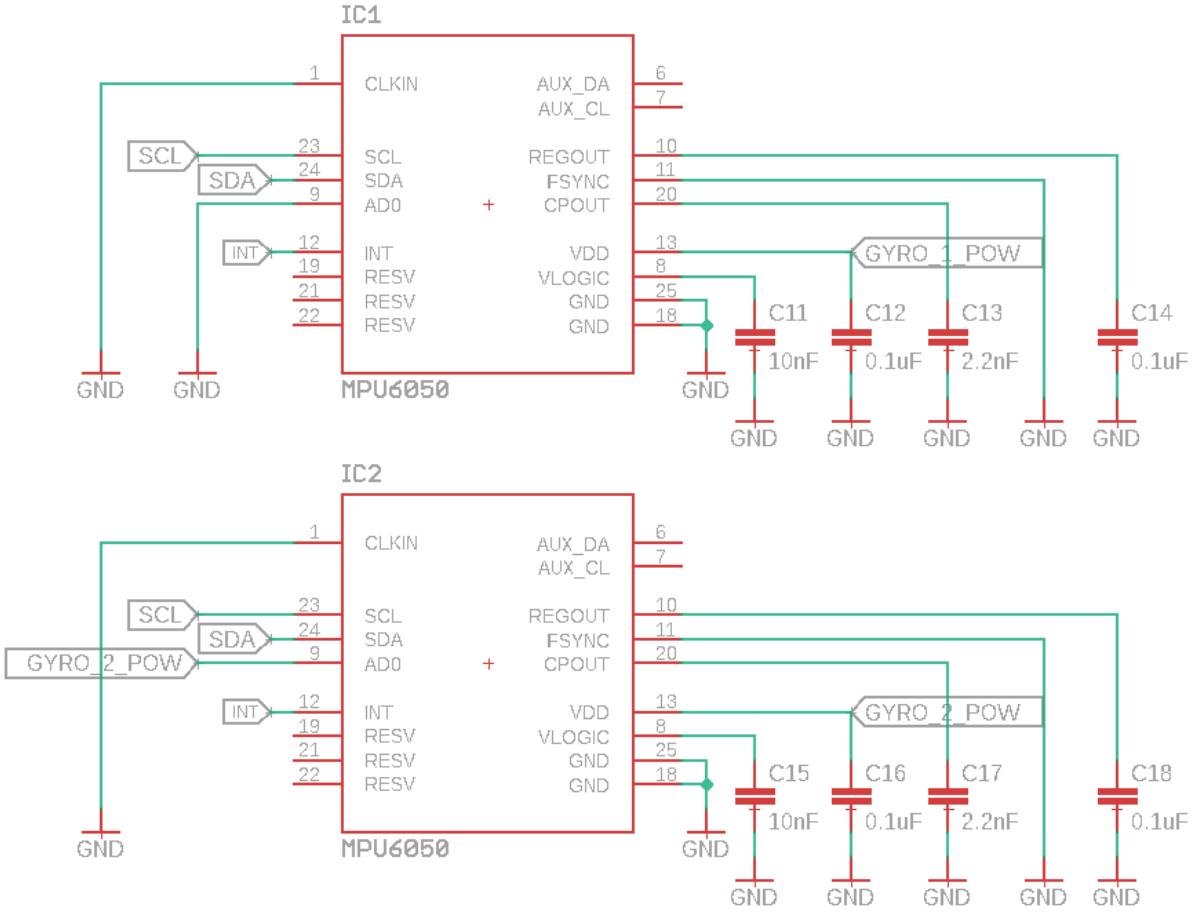

Figure 12: IMUs

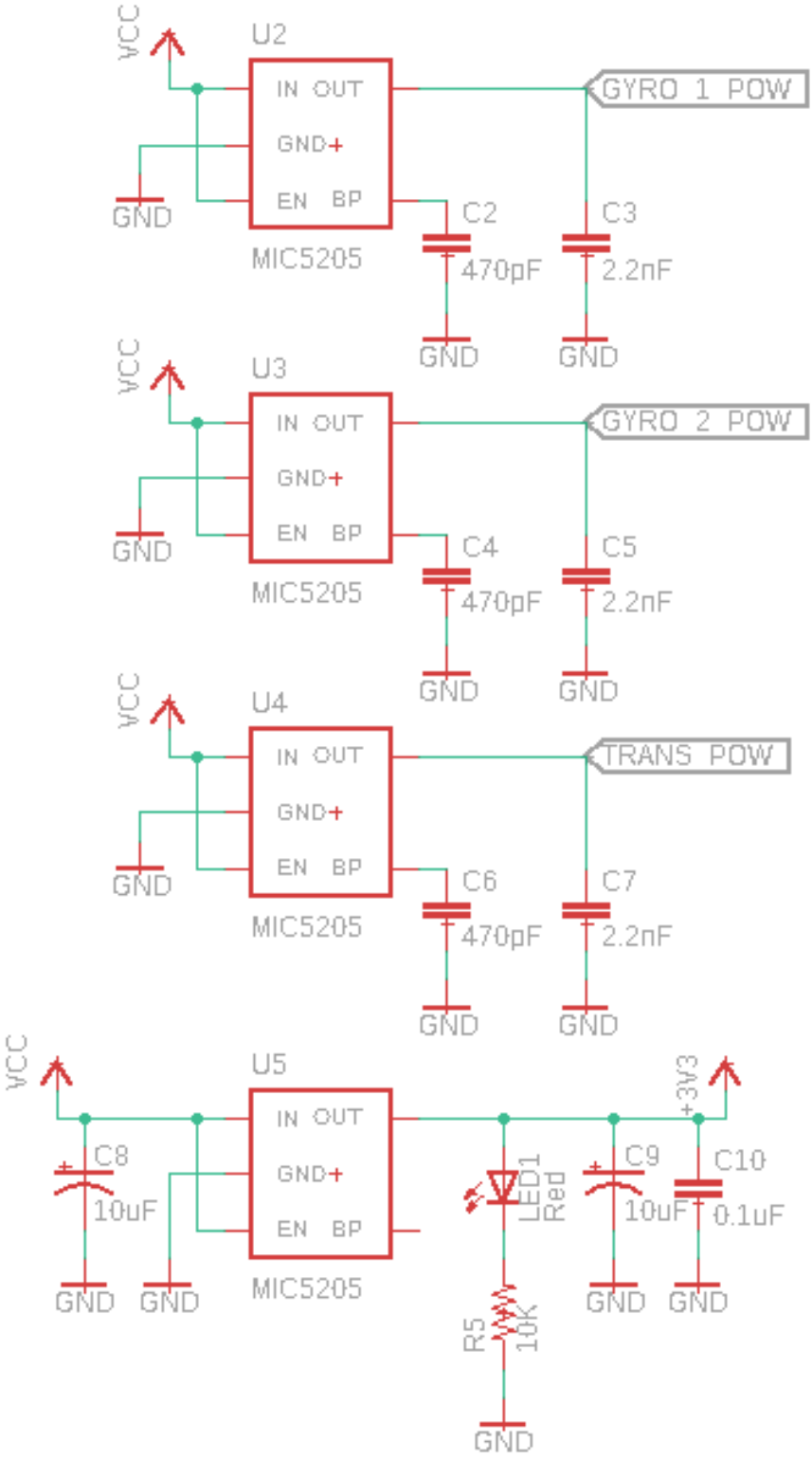

Figure 13: Power Controllers

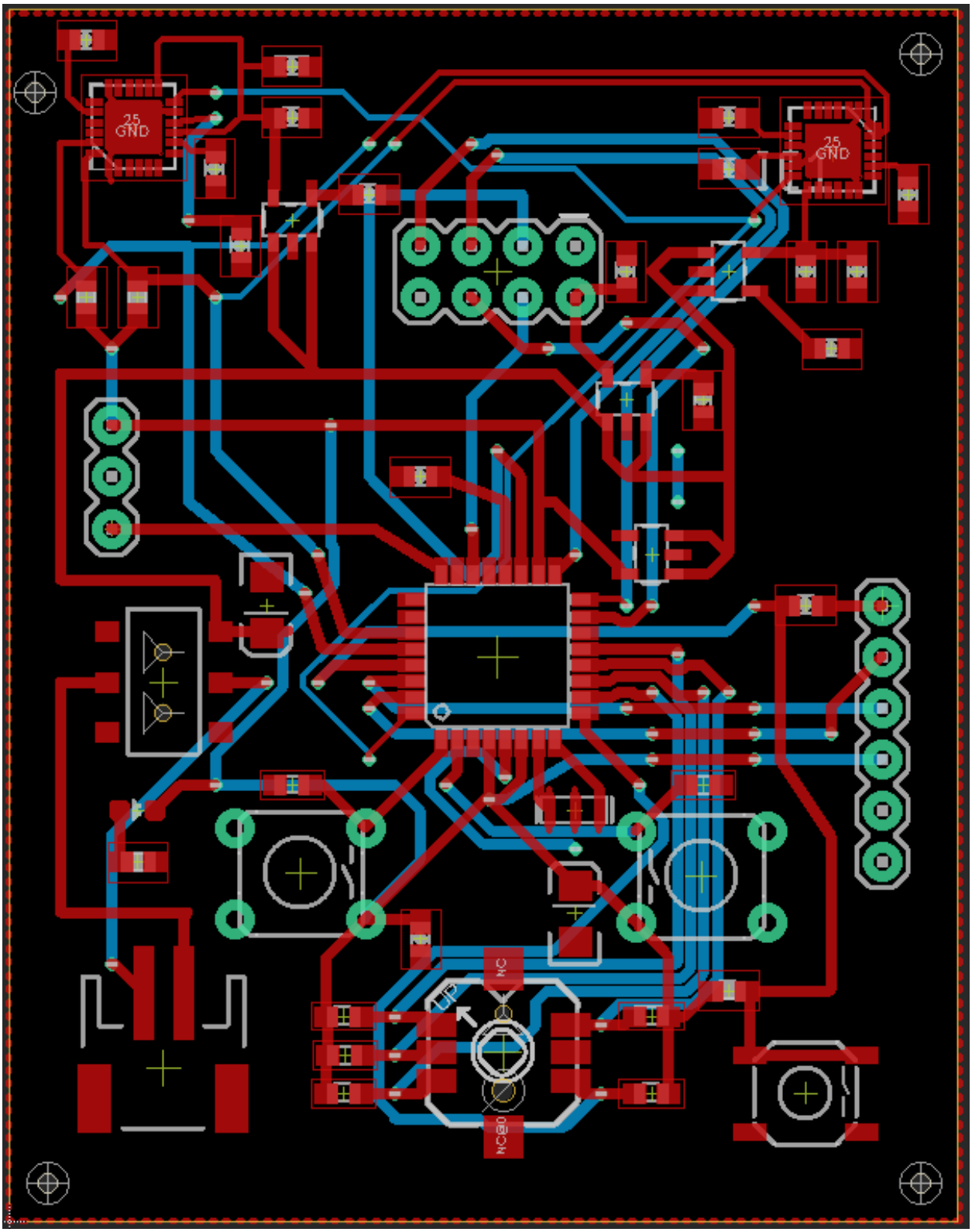

Figure 14: PCB (Ground Plane Not Shown in This Figure)

Appendix C Compatible Products

Attop YD-822/YD-829/YD-829C

BayangToys X6/X7/X9

BWhoop B03

Cheerson CX-10/CX11/CX12/CX205/CX30/SH6057/SH6043/SH6044/SH6046/SH6047

EAchine CG023/CG031/3D X4/E010/H7/H8 mini/H8 mini 3D/JJRC H20/JJRC H22

Floureon FX10/H101

FQ-777-124 Pocket Drone

FY326Q7

HiSky RXs/HFP80/HCP80/HCP100/FBL70/FBL80/FBL90/FBL100/FF120/HMX120

JJRC DHD D1/H36 mini/H6C/JJ850

JXD 385/388/389/391/393

MJX X600

NiHui NH-010

Syma X5C/X5C-1/X11/X11C/X12/X2

WLToys V202/252/252 Pro/272/343/930/931/939/966/977/988/933/944/955

XinXun X28/X30/X33/X39/X40

Yizhan Tarantula X6

Appendix D Software

In addition to the following files and the libraries included in them, files were directly taken from this github repository and added to our sketch: https://github.com/goebish/nrf24_multipro. These files contain the communication protocols for controller-drone interfacing.

Listing 1: fullController.ino

```
#include <EEPROM.h>
#include "iface_nrf24l01.h"
#include <ButtonDebounce.h>
#include "MPU6050.h"
#include "Wire.h"
#include "customMadgwick.cpp"

// ############ Wiring ################
#define MOSI_pin  11
#define SCK_pin   13
#define CE_pin    A2
#define MISO_pin  12
#define CS_pin    A0

// SPI outputs
#define MOSI_on PORTB |= _BV(3)  // PB3
#define MOSI_off PORTB &= ~_BV(3)// PB3
#define SCK_on PORTB |= _BV(5)   // PB5
#define SCK_off PORTB &= ~_BV(5) // PB5
#define CE_on PORTC |= _BV(2)    // PC2
#define CE_off PORTC &= ~_BV(2)  // PC2
#define CS_on PORTC |= _BV(0)    // PC0
#define CS_off PORTC &= ~_BV(0)  // PC0
// SPI input
#define  MISO_on (PINB & _BV(4)) // PB4

#define yaw_ccw_pin 3
#define yaw_cw_pin  4
#define pitch_b_pin 5
#define pitch_f_pin 6
#define roll_l_pin  7
#define roll_r_pin  8
#define trim_rst_pin 9

ButtonDebounce yaw_ccw(yaw_ccw_pin, 50);
ButtonDebounce yaw_cw(yaw_cw_pin, 50);
ButtonDebounce pitch_b(pitch_b_pin, 50);
```

```
38 ButtonDebounce pitch_f(pitch_f_pin, 50);
39 ButtonDebounce roll_l(roll_l_pin, 50);
40 ButtonDebounce roll_r(roll_r_pin, 50);
41 ButtonDebounce trim_rst(trim_rst_pin, 50);
42
43 #define SAMPLES_TAKEN 10
44 #define TWENTY_THRESH 545
45 #define FOURTY_THRESH 430
46 #define SIXTY_THRESH 312
47 #define EIGHTY_THRESH 181
48
49 #define RF_POWER TX_POWER_80mW
50
51 // PPM stream settings
52 #define CHANNELS 12 // number of channels in ppm stream, 12 ideally
53 enum chan_order{
54     THROTTLE,
55     AILERON,
56     ELEVATOR,
57     RUDDER,
58     AUX1,  // (CH5)  led light, or 3 pos. rate on CX-10, H7, or inverted
           flight on H101
59     AUX2,  // (CH6)  flip control
60     AUX3,  // (CH7)  still camera (snapshot)
61     AUX4,  // (CH8)  video camera
62     AUX5,  // (CH9)  headless
63     AUX6,  // (CH10) calibrate Y (V2x2), pitch trim (H7), RTH (Bayang, H20),
           360deg flip mode (H8-3D, H22)
64     AUX7,  // (CH11) calibrate X (V2x2), roll trim (H7)
65     AUX8,  // (CH12) Reset / Rebind
66 };
67
68 #define PPM_MIN 1000
69 #define PPM_SAFE_THROTTLE 1050
70 #define PPM_MID 1500
71 #define PPM_MAX 2000
72 #define PPM_MIN_COMMAND 1300
73 #define PPM_MAX_COMMAND 1700
74 #define GET_FLAG(ch, mask) (ppm[ch] > PPM_MAX_COMMAND ? mask : 0)
75
76 // supported protocols (stripped all but SYMAX5C1)
77 enum {
```

```
78      PROTO_V2X2 = 0,      // WLToys V2x2, JXD JD38x, JD39x, JJRC H6C, Yizhan
            Tarantula X6 ...
79      PROTO_CG023,         // EAchine CG023, CG032, 3D X4
80      PROTO_CX10_BLUE,     // Cheerson CX-10 blue board, newer red board, CX-10A,
             CX-10C, Floureon FX-10, CX-Stars (todo: add DM007 variant)
81      PROTO_CX10_GREEN,    // Cheerson CX-10 green board
82      PROTO_H7,            // EAchine H7, MoonTop M99xx
83      PROTO_BAYANG,        // EAchine H8(C) mini, H10, BayangToys X6, X7, X9,
            JJRC JJ850, Floureon H101
84      PROTO_SYMAX5C1,      // Syma X5C-1 (not older X5C), X11, X11C, X12
85      PROTO_YD829,         // YD-829, YD-829C, YD-822 ...
86      PROTO_H8_3D,         // EAchine H8 mini 3D, JJRC H20, H22
87      PROTO_END
88  };
89
90  // EEPROM locations
91  enum{
92      ee_PROTOCOL_ID = 0,
93      ee_TXID0,
94      ee_TXID1,
95      ee_TXID2,
96      ee_TXID3
97  };
98  uint32_t timeout;
99
100 int16_t trim_rpy[3] = {0, 0, 0};
101
102 int16_t trigger_sample_sum = 0;
103 float trigger_sample_average = 0;
104 unsigned trigger_val;
105
106 uint8_t transmitterID[4];
107 uint8_t packet[32];
108 volatile uint16_t Servo_data[12];
109 static uint16_t ppm[12] = {PPM_MIN,PPM_MID,PPM_MID,PPM_MID,PPM_MID,PPM_MID,
110                            PPM_MID,PPM_MID,PPM_MID,PPM_MID,PPM_MID,PPM_MID,};
111
112 MPU6050 gyro(0x69);
113 int16_t gx, gy, gz, ax, ay, az;
114 SF gyro_filter;
115 float gyro_rpy[3];
116 float delta_t;
117 float gRes = 250.0/32768.0*DEG_TO_RAD;
```

```
118 float aRes = 2.0/32768.0*9.8;
119
120 float scale_rpy[3] = {2, 2, 1};
121
122 void setup() {
123     randomSeed((analogRead(A4) & 0x1F) | (analogRead(A5) << 5));
124     pinMode(MOSI_pin, OUTPUT);
125     pinMode(SCK_pin, OUTPUT);
126     pinMode(CS_pin, OUTPUT);
127     pinMode(CE_pin, OUTPUT);
128     pinMode(MISO_pin, INPUT);
129
130     TCCR1A = 0;  //reset timer1
131     TCCR1B = 0;
132     TCCR1B |= (1 << CS11);  //set timer1 to increment every 1 us @ 8MHz, 0.5
          us @16MHz
133
134     delay(1000);
135
136     Wire.begin();
137     do{
138         gyro.initialize();
139         delay(250);
140     }while(!gyro.testConnection());
141     // Modify gyro offsets
142     gyro.setXAccelOffset(191);
143     gyro.setYAccelOffset(1205);
144     gyro.setZAccelOffset(573);
145     gyro.setXGyroOffset (-30);
146     gyro.setYGyroOffset (-16);
147     gyro.setZGyroOffset (10);
148     gyro.setRate(0);
149
150
151     for(int i=0; i<4; i++) {
152         transmitterID[i] = random() & 0xFF;
153         EEPROM.update(ee_TXID0+i, transmitterID[i]);
154     }
155     EEPROM.update(ee_PROTOCOL_ID, PROTO_SYMAX5C1);
156     NRF24L01_Reset();
157     NRF24L01_Initialize();
158     Symax_init();
159     SymaX_bind();
```

```
160
161     Serial.begin(38400);
162 }
163
164 void loop() {
165     // Update buttons and trim
166     yaw_ccw.update();
167     yaw_cw.update();
168     pitch_b.update();
169     pitch_f.update();
170     roll_l.update();
171     roll_r.update();
172     trim_rst.update();
173
174     if(!roll_l.state()){
175       trim_rpy[0]--;
176     } else if(!roll_r.state()){
177       trim_rpy[0]++;
178     } else if(!pitch_b.state()){
179       trim_rpy[1]--;
180     } else if(!pitch_f.state()){
181       trim_rpy[1]++;
182     } else if(yaw_ccw.state()){
183       trim_rpy[2]--;
184     } else if(yaw_cw.state()){
185       trim_rpy[2]++;
186     } else if(!trim_rst.state()){
187       trim_rpy[0] = 0;
188       trim_rpy[1] = 0;
189       trim_rpy[2] = 0;
190     }
191
192     // Update trigger
193     for(int i=0; i < SAMPLES_TAKEN; i++){
194         trigger_sample_sum += (1023-analogRead(A1));
195         delay(2);
196     }
197     trigger_sample_average = float(trigger_sample_sum) / float(SAMPLES_TAKEN);
198     trigger_sample_sum = 0;
199
200     if(trigger_sample_average > 50 and trigger_sample_average < 675){
201         trigger_val = (trigger_sample_average - 50) / 625 * 1000;
202     } else if(trigger_sample_average > 675 and trigger_sample_average < 800) {
```

```
203         trigger_val = 1000;
204     } else {
205         trigger_val = 0;
206     }
207 
208     // Update gyro
209     gyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
210 
211     if(abs(ax*aRes) >= 2*9.8 or abs(ay*aRes) >= 2*9.8 or abs(az*aRes) >=
            2*9.8) {
212       exit(0);
213     }
214     Serial.println(gyro_rpy[2]);
215     delta_t = gyro_filter.deltatUpdate();
216     gyro_filter.customMadgwickUpdate(gx*gRes, gy*gRes, gz*gRes, ax*aRes, ay*
            aRes, az*aRes, delta_t);
217     gyro_rpy[0] = gyro_filter.getRoll();
218     gyro_rpy[1] = gyro_filter.getPitch();
219     gyro_rpy[2] = gyro_filter.getYaw()-180;
220 
221     // Thottle
222     ppm[THROTTLE] = trigger_val;
223 
224     // Aileron/Roll
225     ppm[AILERON] = PPM_MID + gyro_rpy[0]*scale_rpy[0] + trim_rpy[0];
226 
227     // Elevator/Pitch
228     ppm[ELEVATOR] = PPM_MID + gyro_rpy[1]*scale_rpy[1] + trim_rpy[1];
229 
230     // Rudder/Yaw
231     ppm[RUDDER] = PPM_MID - gyro_rpy[2]*scale_rpy[2] + trim_rpy[2];
232 
233     timeout = process_SymaX();
234     while(micros() < timeout) { } // wait for drone to process
235 }
```

Listing 2: customMadgwick.cpp

```
#include "Arduino.h"

#define betaDef     0.1f

class SF {
public:
    SF(){
        beta = betaDef;
        q0 = 1.0f;
        q1 = 0.0f;
        q2 = 0.0f;
        q3 = 0.0f;
        anglesComputed = 0;
    }
    void customMadgwickUpdate(float gx, float gy, float gz,
                              float ax, float ay, float az, float deltat){
        float recipNorm;
        float s0, s1, s2, s3;
        float qDot1, qDot2, qDot3, qDot4;
        float _2q0, _2q1, _2q2, _2q3, _4q0, _4q1, _4q2;
        float _8q1, _8q2, q0q0, q1q1, q2q2, q3q3;

        // Rate of change of quaternion from gyroscope
        qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
        qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
        qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
        qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

        // Normalise accelerometer measurement
        recipNorm = invSqrt(ax * ax + ay * ay + az * az);
        ax *= recipNorm;
        ay *= recipNorm;
        az *= recipNorm;

        // Auxiliary variables to avoid repeated arithmetic
        _2q0 = 2.0f * q0;
        _2q1 = 2.0f * q1;
        _2q2 = 2.0f * q2;
        _2q3 = 2.0f * q3;
        _4q0 = 4.0f * q0;
        _4q1 = 4.0f * q1;
        _4q2 = 4.0f * q2;
```

```
43          _8q1 = 8.0f * q1;
44          _8q2 = 8.0f * q2;
45          q0q0 = q0 * q0;
46          q1q1 = q1 * q1;
47          q2q2 = q2 * q2;
48          q3q3 = q3 * q3;
49
50          // Gradient decent algorithm corrective step
51          s0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
52          s1 = _4q1 * q3q3 - _2q3 * ax + 4.0f * q0q0 * q1 - _2q0 *
53                  ay - _4q1 + _8q1 * q1q1 + _8q1 * q2q2 + _4q1 * az;
54          s2 = 4.0f * q0q0 * q2 + _2q0 * ax + _4q2 * q3q3 - _2q3 *
55                  ay - _4q2 + _8q2 * q1q1 + _8q2 * q2q2 + _4q2 * az;
56          s3 = 4.0f * q1q1 * q3 - _2q1 * ax + 4.0f * q2q2 * q3 - _2q2 * ay;
57          recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3);
58
59          // normalise step magnitude
60          s0 *= recipNorm;
61          s1 *= recipNorm;
62          s2 *= recipNorm;
63          s3 *= recipNorm;
64
65          // Apply feedback step
66          qDot1 -= beta * s0;
67          qDot2 -= beta * s1;
68          qDot3 -= beta * s2;
69          qDot4 -= beta * s3;
70
71          // Integrate rate of change of quaternion to yield quaternion
72          q0 += qDot1 * deltat;
73          q1 += qDot2 * deltat;
74          q2 += qDot3 * deltat;
75          q3 += qDot4 * deltat;
76
77          // Normalise quaternion
78          recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
79          q0 *= recipNorm;
80          q1 *= recipNorm;
81          q2 *= recipNorm;
82          q3 *= recipNorm;
83          anglesComputed = 0;
84      }
85
```

```
86      float deltatUpdate (){
87          Now = micros();
88          deltat = ((Now - lastUpdate) / 1000000.0f);
89          // set integration time by time elapsed since last filter update
90          lastUpdate = Now;
91          return deltat;
92      }
93
94      float getRoll() {
95          if (!anglesComputed) computeAngles();
96          return roll * RAD_TO_DEG;
97      }
98      float getPitch() {
99          if (!anglesComputed) computeAngles();
100         return pitch * RAD_TO_DEG;
101     }
102     float getYaw() {
103         if (!anglesComputed) computeAngles();
104         return yaw * RAD_TO_DEG + 180.0f;
105     }
106
107 private:
108     float beta;
109     float q0, q1, q2, q3; // quaternion of sensor frame relative to auxiliary
            frame
110     bool anglesComputed;
111
112     static float invSqrt(float x){
113         float halfx = 0.5f * x;
114         float y = x;
115         long i = *(long*)&y;
116         i = 0x5f3759df - (i>>1);
117         y = *(float*)&i;
118         y = y * (1.5f - (halfx * y * y));
119         y = y * (1.5f - (halfx * y * y));
120         return y;
121     }
122
123     void computeAngles(){
124         roll = atan2f(q0*q1 + q2*q3, 0.5f - q1*q1 - q2*q2);
125         pitch = asinf(-2.0f * (q1*q3 - q0*q2));
126         yaw = atan2f(q1*q2 + q0*q3, 0.5f - q2*q2 - q3*q3);
127         anglesComputed = 1;
```

```
128     }
129
130     float roll, pitch, yaw;
131     float Now,lastUpdate,deltat;
132 };
```

Listing 3: buttonTest.ino

```
1  #include <ButtonDebounce.h>
2
3  #define yaw_ccw_pin 3
4  #define yaw_cw_pin 4
5  #define pitch_b_pin 5
6  #define pitch_f_pin 6
7  #define roll_l_pin 7
8  #define roll_r_pin 8
9  #define trim_rst_pin 9
10
11 ButtonDebounce yaw_ccw(yaw_ccw_pin, 50);
12 ButtonDebounce yaw_cw(yaw_cw_pin, 50);
13 ButtonDebounce pitch_b(pitch_b_pin, 50);
14 ButtonDebounce pitch_f(pitch_f_pin, 50);
15 ButtonDebounce roll_l(roll_l_pin, 50);
16 ButtonDebounce roll_r(roll_r_pin, 50);
17 ButtonDebounce trim_rst(trim_rst_pin, 50);
18
19 int yaw_ccw_count = 0;
20 int yaw_cw_count = 0;
21 int pitch_b_count = 0;
22 int pitch_f_count = 0;
23 int roll_l_count = 0;
24 int roll_r_count = 0;
25 int trim_rst_count = 0;
26
27 bool yaw_ccw_state = HIGH;
28 bool yaw_cw_state = HIGH;
29 bool pitch_b_state = HIGH;
30 bool pitch_f_state = HIGH;
31 bool roll_l_state = HIGH;
32 bool roll_r_state = HIGH;
33 bool trim_rst_state = HIGH;
34
35 void setup()
36 {
```

```
37   Serial.begin(38400);
38 }
39
40 void loop()
41 {
42   yaw_ccw.update();
43   yaw_cw.update();
44   pitch_b.update();
45   pitch_f.update();
46   roll_l.update();
47   roll_r.update();
48   trim_rst.update();
49
50   if (yaw_ccw_state != yaw_ccw.state())
51   {
52     if (yaw_ccw.state() == HIGH)
53     {
54       yaw_ccw_count++;
55       yaw_ccw_state = HIGH;
56     }
57     else
58     {
59       yaw_ccw_state = LOW;
60     }
61   }
62
63   if (yaw_cw_state != yaw_cw.state())
64   {
65     if (yaw_cw.state() == HIGH)
66     {
67       yaw_cw_count++;
68       yaw_cw_state = HIGH;
69     }
70     else
71     {
72       yaw_cw_state = LOW;
73     }
74   }
75   if (pitch_b_state != pitch_b.state())
76   {
77     if (pitch_b.state() == LOW)
78     {
79       pitch_b_count++;
```

```
80        pitch_b_state = LOW;
81      }
82      else
83      {
84        pitch_b_state = HIGH;
85      }
86    }
87    if (pitch_f_state != pitch_f.state())
88    {
89      if (pitch_f.state() == LOW)
90      {
91        pitch_f_count++;
92        pitch_f_state = LOW;
93      }
94      else
95      {
96        pitch_f_state = HIGH;
97      }
98    }
99    if (roll_r_state != roll_r.state())
100   {
101     if (roll_r.state() == LOW)
102     {
103       roll_r_count++;
104       roll_r_state = LOW;
105     }
106     else
107     {
108       roll_r_state = HIGH;
109     }
110   }
111   if (roll_l_state != roll_l.state())
112   {
113     if (roll_l.state() == LOW)
114     {
115       roll_l_count++;
116       roll_l_state = LOW;
117     }
118     else
119     {
120       roll_l_state = HIGH;
121     }
122   }
```

```
123   if (trim_rst_state != trim_rst.state())
124   {
125     if (trim_rst.state() == LOW)
126     {
127       trim_rst_count++;
128       trim_rst_state = LOW;
129     }
130     else
131     {
132       trim_rst_state = HIGH;
133     }
134   }
135
136   Serial.print("yaw_ccw:␣");
137   Serial.print(yaw_ccw_count);
138   Serial.print("\t");
139
140   Serial.print("yaw_cw:␣");
141   Serial.print(yaw_cw_count);
142   Serial.print("\t");
143
144   Serial.print("pitch_b:␣");
145   Serial.print(pitch_b_count);
146   Serial.print("\t");
147
148   Serial.print("pitch_f:␣");
149   Serial.print(pitch_f_count);
150   Serial.print("\t");
151
152   Serial.print("roll_l:␣");
153   Serial.print(roll_l_count);
154   Serial.print("\t");
155
156   Serial.print("roll_r:␣");
157   Serial.print(roll_r_count);
158   Serial.print("\t");
159
160   Serial.print("trim_rst:␣");
161   Serial.print(trim_rst_count);
162   Serial.println();
163
164   delay(50);
165 }
```

Listing 4: generalTest.ino

```
#include "MPU6050.h"
#include "Wire.h"
#include "customMadgwick.cpp"

MPU6050 gyro(0x69);
int gx, gy, gz, ax, ay, az;
SF gyro_filter;
float gyro_rpy[3];
float delta_t;
float gRes = 250.0 / 32768.0 * DEG_TO_RAD;
float aRes = 2.0 / 32768.0 * 9.8;

int startTime = 0;
int endTime = 0;

void setup()
{
    Wire.begin();
    do
    {
        gyro.initialize();
        delay(250);
    } while (!gyro.testConnection());
    // Modify gyro offsets
    gyro.setXAccelOffset(191);
    gyro.setYAccelOffset(1205);
    gyro.setZAccelOffset(573);
    gyro.setXGyroOffset(-30);
    gyro.setYGyroOffset(-16);
    gyro.setZGyroOffset(10);
    gyro.setRate(0);

    Serial.begin(38400);
}

void loop() {
    startTime = millis();
    gyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
    delta_t = gyro_filter.deltatUpdate();
    gyro_filter.customMadgwickUpdate(gx*gRes, gy*gRes, gz*gRes, ax*aRes, ay*
        aRes, az*aRes, delta_t);
    gyro_rpy[0] = gyro_filter.getRoll();
```

```
    gyro_rpy[1] = gyro_filter.getPitch();
    gyro_rpy[2] = gyro_filter.getYaw()-180;
    endTime = millis();

    Serial.print("Time(ms):");
    Serial.print(endTime - startTime);
    Serial.print("\t");

    Serial.print("X Accel(m/s):");
    Serial.print(ax*aRes);
    Serial.print("\t");
    Serial.print("Y Accel:");
    Serial.print(ay*aRes);
    Serial.print("\t");
    Serial.print("Z Accel:");
    Serial.print(az*aRes);
    Serial.print("\t");

    Serial.print("Roll(deg):");
    Serial.print(round(gyro_rpy[0]));
    Serial.print("\t");
    Serial.print("Pitch:");
    Serial.print(round(gyro_rpy[1]));
    Serial.print("\t");
    Serial.print("Yaw:");
    Serial.println(round(gyro_rpy[2]));
}
```