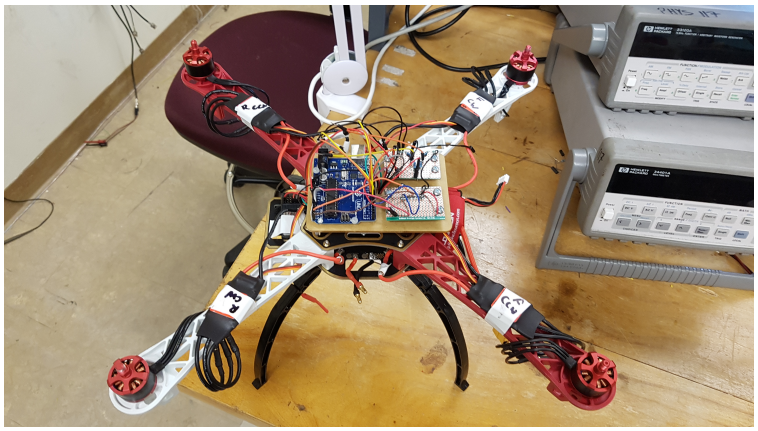
Drone Project

Physics 118 - Final Report

Krish Kabra,
Stefan Orosco



Physics & Astronomy Department
University of California, Los Angeles
June 2018

Contents

1 Introduction

1.1 Objective

The project objective was to build a quadcopter drone that uses an Arduino as its main flight controller. This build was largely inspired by electronic hobbyist Joop Brokking, whose YouTube videos we often referred to [1].

We set both primary and secondary objectives, with the secondary objectives existing simply to serve as an interesting bonus outgrowth of our original project plan. Our primary objectives were as follows:

1. Build a remote-controlled quadcopter
2. Stabilize the flight of the quadcopter using a feedback control loop

Although the primary objectives appear concise, there are many intricate aspects to the building such an "Auto-leveling" quadcopter. Firstly, the remote-control aspect of the quadcopter requires seamless, lag-free and stable communication between the transmitter and quadcopter motors for the quadcopter to fly smoothly. Secondly, due to the inherent unstable nature of quadcopter flight, there is a necessity for some stabilization mechanism in-built to the quadcopter flight controller. For this, we plan on using a proportional-integral-derivative (PID) feedback control that uses input for an onboard internal measurement unit (IMU) to communicate with the motors in such a way the quadcopter flight is less unstable, if not completely stable.

If these primary objectives were completed in time, we had modular secondary objectives as add-on features for the quadcopter, which included:

1. Sonar-based collision avoidance system
2. Voice control system

These objectives are currently based upon ongoing developments in industry standard quadcopters, and were to be bonus features we could add onto our main quadcopter build to enhance the barebone drone.

1.2 Applications

Autonomous drones technology has an extremely promising future in the evolving transportation, retail and entertainment markets. The rapid development of deep learning and the relative affordability of microcontrollers and other single board computing technology has made the development of the consumer facing drone industry economically feasible. Hardware companies like DJI have been steadily growing their own consumer/hobbyist product line. However, more importantly is the growth of the business facing drone industry for both hardware and software development. The promising advancements in neural net processing and open source AI projects has pushed the boundaries on the concept of possible. For example, companies like Skydio R1 are pushing forward drones with powerful facial recognition software for video tracking and photography. While facial recognition and autonomy is a game changer to the entertainment industry with smart cinema cameras and selfie drones, the true utility for this technology lies in the agriculture and shipping industries. Autonomous drones present an extremely convenient method for mapping and irrigation for large crop fields. Any sizeable farm will employ more than a dozen people for the sole purpose of watering and mapping the crop fields. This work could be given to a drone (or fleet of drones) and save time/energy. The commercial shipping industry is also investing in drone technology for quicker package delivery (Amazon being the prominent company). Large tech conglomerates have introduced large capital investments , for example Intel recently invested 60 million dollars into Yuneec an electrical aviation company specializing in drones.

2 Design

2.1 Overview

The overall project can be split up into 5 main components: developing a battery control circuit that powers our entire quadcopter, controlling the motors and electronic speed controllers (ESCs), retrieving pulse width signals from a radio frequency transmitter and receiver, obtaining data from an internal measurement unit (IMU), and, finally, building a flight controller on an MCU, specifically an ATmega328p (which we work using an Arduino Uno development board).

The design of the primary objective part of the project is very modular. The secondary objective features can easily be added to this 'barebone' quadcopter; for example, the voice control kit or sonar-based collision system can be set up with an external MCU, such as a Raspberry Pi or another Arduino, and be synced with the main Arduino flight controller via the I2C communication bus (corresponding to the SDA and SCL pins), which the MPU-6050 is currently connected to.

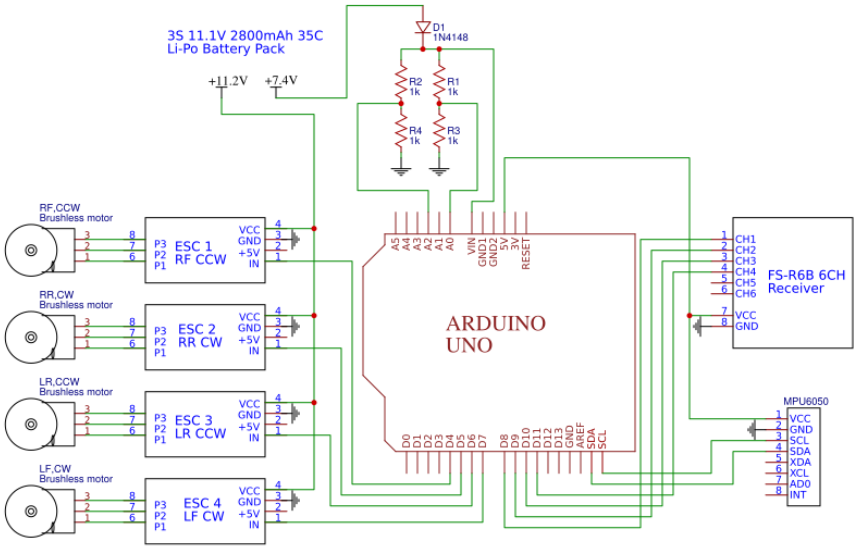


Figure 1: Full schematic of the electronics for the quadcopter.

2.2 Battery Control Circuit

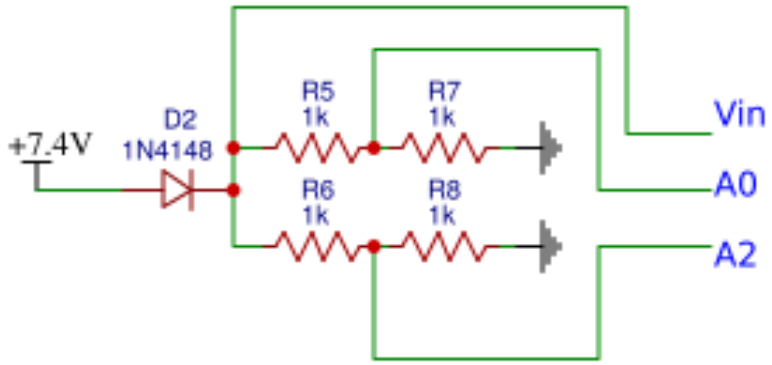


Figure 2: Schematic of battery control circuit

The battery control circuit is the heart of the quadcopter. You can reference figure 2 with the Vin, A0 and A2 pins representing Arduino connections pins. The four resistors used are identical precision resistors which are all within two ohms of each other. This was done to provide the relative voltage outputs to the Arduino's analog input pins, A0/A2. The battery circuit code is quite simple. The raw voltage output of the LiPo battery is roughly 13.1V (much higher than its rated 11.2V). This is too high for the on board voltage regulator on the Arduino UNO (the voltage regulator on the Arduino can handle 12.0V max). To accommodate for this we use the 3 pin MOLEX header that is used to charge each cell of the battery to draw only 7.4V from 2 cells of the 3 cell battery pack. The voltage divider is read the Arduino using analog pins

A0/A2 and below a given threshold the Arduino will flash an on board LED to indicate that the battery is low. This same effect could be achieved with just one pair of resistors, however, the second pair was added as a precaution in case there is a hardware malfunction mid flight. This second pair also allows for a more accurate measurement from the Arduino, because the values read at the analog inputs can be averaged to obtain a more consistent value. Finally, the diode is connected directly to the battery so as to prevent backflow current from entering the battery.

2.3 Motors & Electronic Speed Controllers

The electronic speed controllers (ESC) and motors are the muscles and limbs of the quadcopter respectively. We used LHI's a 2212 920KV Brushless DC Motor (BLDC) and SimonK's 30A ESC [2, 3].

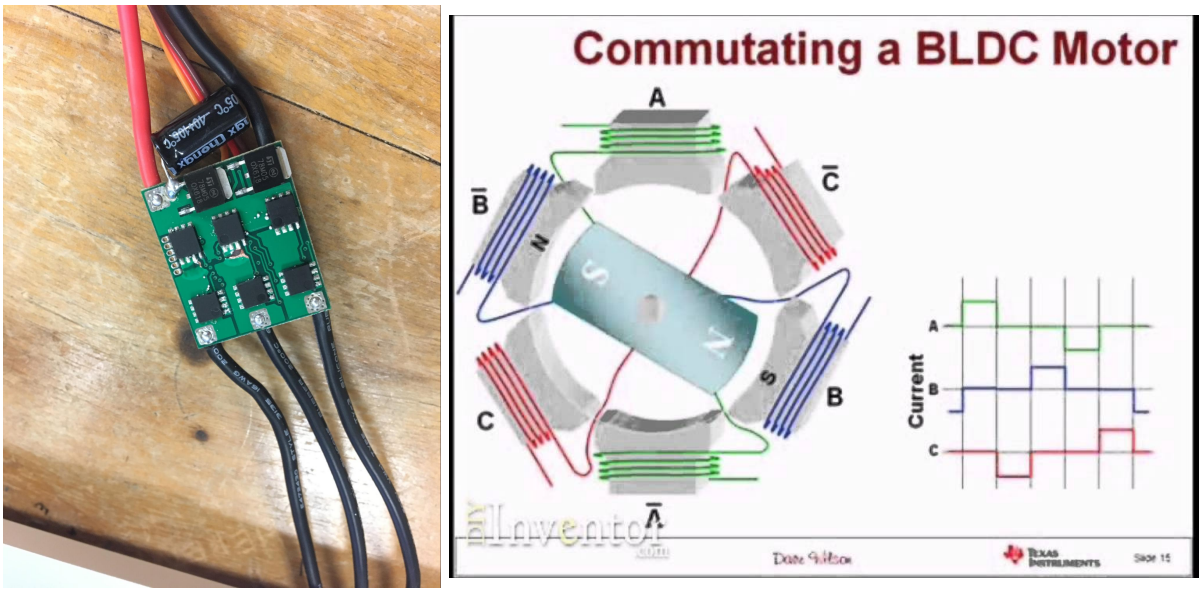


Figure 3: Right: ESC - middle top MOSFET chip has a visible short on the bottom three pins.
Left: Brushless Motor - Internal magnet rotated by the induced alternating magnetic field from wire loops. ESC used to output A-B-C pulses at frequency which matches desired speed[4].

BLDC motors work in a primarily different manner to commutator (brushed) DC motors. Instead of having a rotor shaft rotate by having an alternating current passing through a wire loop in between a permanent magnet, BLDC motors rely on switching the direction of current passing through electromagnets that surround a central permanent magnet which acts as the rotor. For the simple case of having 1 electromagnet, the current through the coil windings of the electromagnet is only alternated with every 180°. Typical BLDC motors nowadays, however, have 3 or more electromagnets, and so the current is alternated between the different coil windings (electromagnets) at different phases depending on the number of electromagnets and desired rotor RPM (refer to Figure 3, left). Due to the lack of friction in them, BLDC motors are more efficient, longer lasting, and faster in comparison to the older commutator DC motors.

The way we correctly output the necessary multiple phased pulses to the BLDC motor is with an ESC. The ESC interprets pulse-width-modulation (PWM) signals and converts them into the correct pulse currents to be sent to the electromagnets within the BLDC. This is done using multiple transistors, which are typically power MOSFET modules. Therefore, we are able to control the motors of our quadcopter using an Arduino by sending the calibrated PWM signal to the ESC, which corresponds to the RPM we would like the motor to spin at.

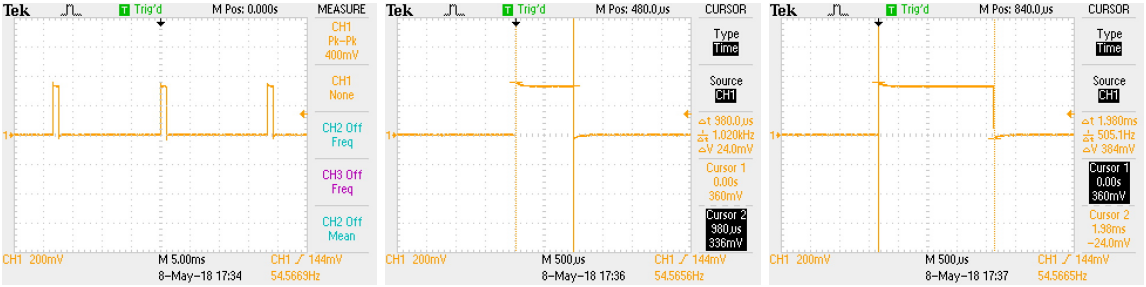
One issue that we have encountered during our project was a faulty ESC shipped from the manufacturer. There was a short through one of the attached MOSFETs on the ESC's circuit board. Unfortunately, when connecting this to a motor for testing, the motor shorted out and began smoking. We naively believed that the issue occured with the motor and not the ESC and so we tested the ESC on another motor and the same issue occurred. At this point we disassembled the ESC and the motors to troubleshoot the issues. The motors were tested using inductance meters because of the large quantity of copper wire windings associated with brushless motors. A working motor should read similar inductances around each of the leads, however if there has been a short then the inductance will vary extremely. After doing this process on each of the motors, we found that two of the motors had a short circuit. To test the remaining ESCs we checked the connection between the MOSFET source and drain pins on the circuit board and found that two of our ESC's contained shorts (refer to Figure 3, right). We promptly ordered a new set of ESCs and brushless motors and checked the ESCs and motors before testing.

Combined with knowledge of the specifications of our battery (11.1V, 2800mAh, 35C)[5], we can make an estimate on the minimum flight time of our quadcopter by assuming it is running at full-speed for the entire flight:

$$T = \frac{\text{Battery Capacity} \times \text{Battery Discharge}}{\text{Max. Total Amp Draw}}$$

$$T = \frac{2800mAh \times 35C}{30A \times 4 \text{ motors}} = 49 \text{ minutes}$$

2.4 Transmitter & Receiver



(a) Multiple pulses of constant time and frequency being received from the radio transmitter.

(b) Pulse length for 0% throttle corresponds to approximately 1000us.

(c) Pulse length for 100% throttle corresponds to approximately 2000us.

Figure 4: Example output from receiver channel 3, which corresponds to the transmitter's throttle.

The radio transmitter and receiver are the nervous system of the quadcopter. We use a 6-channel 2.4Ghz radio controller transmitter and receiver from Flysky [6], although we only need to utilize 4 sticks (i.e 4 channels) which correspond to the throttle, pitch, yaw and roll movements. The Flysky FS-T6 controller communicates with the FS-R6B receiver via radio frequency signals. Although the physics behind this communication is interesting, we will not choose to explain how the transmitter sends these signals in more detail.

Instead, what the receiver outputs as we move the sticks on the transmitter is of utmost importance. As can be seen in figure 4, the receiver channels output a constant frequency of pulses whose width is modulated according to the stick movements - this is exactly the pulse width modulation needed by the ESCs to control the motors.

This raises the question of what PWM signal we should send to the ESCs - the PWM generated by the Arduino, or the PWM generated by the receiver? The correct answer is the PWM from the receiver. This is for a few reasons: firstly, the Arduino PWM function is very slow, and not very stable. This makes it hard to control a quadcopter whose motor speeds need to be precisely controlled and be responsive to a change in user control, otherwise it will crash very easily. Secondly, using the receiver PWM signals allows for a quicker interaction between the pilot and motors.

```
ISR(PCINT0_vect){
  current_time = micros();
  //Channel 1=========================================
  if(PINB & B00000001){                                        //Is input 8 high?
    if(last_channel_1 == 0){                                   //Input 8 changed from 0 to 1.
      last_channel_1 = 1;                                      //Remember current input state.
      timer_1 = current_time;                                  //Set timer_1 to current_time.
    }
  }
  else if(last_channel_1 == 1){                                //Input 8 is not high and changed from 1 to 0.
    last_channel_1 = 0;                                        //Remember current input state.
    receiver_input[1] = current_time - timer_1;                //Channel 1 is current_time - timer_1.
  }
  //Channel 2=========================================
  if(PINB & B00000010 ){                                       //Is input 9 high?
    if(last_channel_2 == 0){                                   //Input 9 changed from 0 to 1.
      last_channel_2 = 1;                                      //Remember current input state.
      timer_2 = current_time;                                  //Set timer_2 to current_time.
    }
  }
  else if(last_channel_2 == 1){                                //Input 9 is not high and changed from 1 to 0.
    last_channel_2 = 0;                                        //Remember current input state.
    receiver_input[2] = current_time - timer_2;                //Channel 2 is current_time - timer_2.
  }
  //Channel 3=========================================
  if(PINB & B00000100 ){                                       //Is input 10 high?
    if(last_channel_3 == 0){                                   //Input 10 changed from 0 to 1.
      last_channel_3 = 1;                                      //Remember current input state.
      timer_3 = current_time;                                  //Set timer_3 to current_time.
    }
  }
  else if(last_channel_3 == 1){                                //Input 10 is not high and changed from 1 to 0.
    last_channel_3 = 0;                                        //Remember current input state.
    receiver_input[3] = current_time - timer_3;                //Channel 3 is current_time - timer_3.

  }
  //Channel 4=========================================
  if(PINB & B00001000 ){                                       //Is input 11 high?
    if(last_channel_4 == 0){                                   //Input 11 changed from 0 to 1.
      last_channel_4 = 1;                                      //Remember current input state.
      timer_4 = current_time;                                  //Set timer_4 to current_time.
    }
  }
  else if(last_channel_4 == 1){                                //Input 11 is not high and changed from 1 to 0.
    last_channel_4 = 0;                                        //Remember current input state.
    receiver_input[4] = current_time - timer_4;                //Channel 4 is current_time - timer_4.
  }
}
```

Figure 5: Complete interrupt sequence code contained in Arduino flight controller sketch.

In order to store the receiver pulse lengths, we use an interrupt sequence, as seen in figure 5. This interrupt sequence triggers on a state change. For example, let's look at how the pulse time will be obtained for channel 1, which is connected to the digital input pin 8 on the Arduino. When the channel goes from low to high, the interrupt sequence is triggered and we go through our control flow statements (note that for the first trigger, we initialize the last channel variable to be 0). Since the last state of the channel was a low, we collect the current time. After the interrupt is triggered again when the state falls from high to low, we go through the second control flow statement, which collects the current time and subtracts it off the previous time we had collected when our state had risen from low to high. This lets us calculate how long our pulse high state was on for in microseconds.

We can now use these pulse times to output the corresponding PWM to the ESCs, as seen in figure 6. In this code, we simply turn each ESC to a high state for a period of time that corresponds to the pulse time recorded from the receiver by the interrupt sequence using a loop function that exits when the corresponding pulse time has elapsed.

```
void esc_pulse_output(){
  zero_timer = micros();
  PORTD |= B11110000;                                       //Set port 4, 5, 6 and 7 high at once
  timer_channel_1 = esc_1 + zero_timer;                     //Calculate the time when digital port 4 is set low.
  timer_channel_2 = esc_2 + zero_timer;                     //Calculate the time when digital port 5 is set low.
  timer_channel_3 = esc_3 + zero_timer;                     //Calculate the time when digital port 6 is set low.
  timer_channel_4 = esc_4 + zero_timer;                     //Calculate the time when digital port 7 is set low.

  while(PORTD >= 16){                                       //Execute the loop until digital port 4 to 7 is low.
    esc_loop_timer = micros();                              //Check the current time.
    if(timer_channel_1 <= esc_loop_timer)PORTD &= B11101111;     //When the delay time is expired, digital port 4 is set low.
    if(timer_channel_2 <= esc_loop_timer)PORTD &= B11011111;     //When the delay time is expired, digital port 5 is set low.
    if(timer_channel_3 <= esc_loop_timer)PORTD &= B10111111;     //When the delay time is expired, digital port 6 is set low.
    if(timer_channel_4 <= esc_loop_timer)PORTD &= B01111111;     //When the delay time is expired, digital port 7 is set low.
  }
}
```

Figure 6: ESC output function contained in Arduino flight controller sketch. The timer values are obtained from the pulse signals from the receiver recorded via the interrupt sequence.

2.5 Internal Measurement Unit

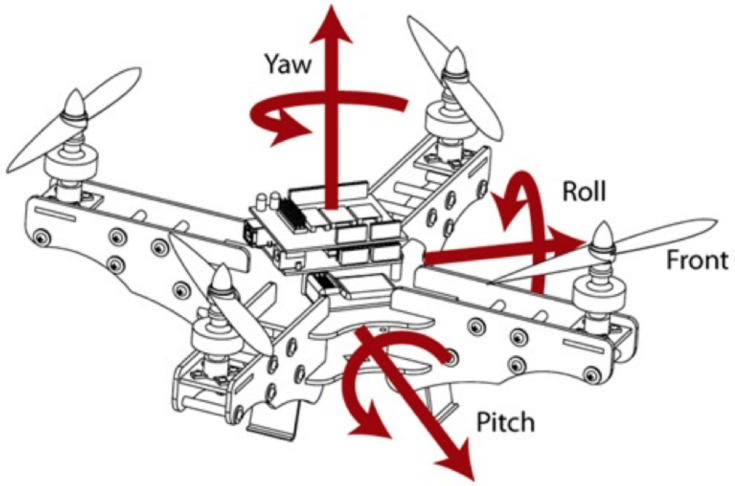


Figure 7: Quadcopter axes corresponding to the roll, pitch and yaw movements [7].

The internal measurement unit (IMU) is the eyes and ears of the quadcopter. The IMU is composed of two key hardware parts the accelerometer and gyroscope. The IMU essentially measures the current orientation of the quadcopter as angles from the yaw, roll and pitch motions (see figure 7. This angle calculation is done from the raw data output of the gyroscope and accelerometer readings. The gyroscope measures the angles as a rate and represents that rate as a raw integer value (depending on the sensitivity setting of the gyro the max rate in degrees per second is $2000°/s$. With the full scale gyro range set to $\pm 500°/s$ the base rate of one degree per second rotation will yield a value of 65.5 for each axis. Thus to scale the raw gyro output reading to be in degrees per second the raw output is divided by 65.5. This angle reading is still insufficient for a quadcopter IMU unit however, this is due to the initial noise and offset present in the first gyroscope readings. To correct for this the axis outputs are averaged over 2000 readings and these averages are subtracted from the gyroscope measurements. To obtain a traveled angle reading from a rate measurement, the rates must be summed consecutively (analogous to taking an integral of velocity to obtain position). The rates are summed at a sample rate of 250Hz (one sample every four milliseconds) and multiplied by a factor to give a travelled angle measurement. This factor is $1.66 \times 10^{-5} s^{-1}$ which accounts for the gyros full scale measurement (65.5) and the sample rate of 250Hz. The last requirement is that the axis of the gyro be coupled together. This means that rotation involving more than one axis can be measured

by the quadcopter. The equation for the coupled axis are:

$$\text{Angle Roll Coupled} = \text{Angle Roll} - \text{Angle Pitch} \times \sin \text{Angle Yaw}$$

$$\text{Angle Pitch Coupled} = \text{Angle Pitch} + \text{Angle Roll} \times \sin \text{Angle Yaw}$$

The coupled angle readings for the travelled angle are now usable for a short period of time. Because this is a real world system there is an drift factor present in the angle measurement from just the gyroscope, thus for use on a quadcopter the a gyroscope IMU is not useful.

The accelerometer is used to correct for this drift term and provide a starting angle position for the IMU. The accelerometer works very similarly to the gyroscope, however it measures the acceleration present rather than the velocity. The measurement comes out as a raw integer value and is converted to a quantity in g's (1g being the base force on the surface of the Earth). The vector direction is used to obtain a measured angle on the pitch and roll axis. The following equations show how the angle is calculated:

$$a_{total} = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

$$\text{angle}_{\text{pitch}} = \alpha \times \text{Angle Roll} \times \arcsin \frac{a_y}{a_{\text{total}}}$$

$$\text{angle}_{\text{pitch}} = -\alpha \times \text{Angle Roll} \times \arcsin \frac{a_x}{a_{\text{total}}}$$

$$\alpha = 57.3^\circ$$

Note that is a correction term dependent on the sensitivity full scale range of the accelerometer, in our case we utilized a sensitivity range of +/-8g. While the accelerometer angles work in theory the vibrational acceleration due to the motors provides too much noise for practical use. The final IMU utilizes a combination of the gyroscope and accelerometer angle calculations.

2.6 Flight Controller

The flight controller is the brain of the quadcopter. We designed out flight controller on the ATmega328p MCU using an Arduino development board [8]. The purpose of the flight controller is connecting the pilot's movements on the transmitter with the desired movement on the quadcopter. Since the quadcopter is inherently an unstable system due to vibrations from the motors, wind and other factors, it is necessary to have some feedback control system that regulates the flight control output to the motors with the desired user input from the transmitter. For this reason, we utilze a proportional-integral-derivative (PID) controller that is integrated into our flight controller.

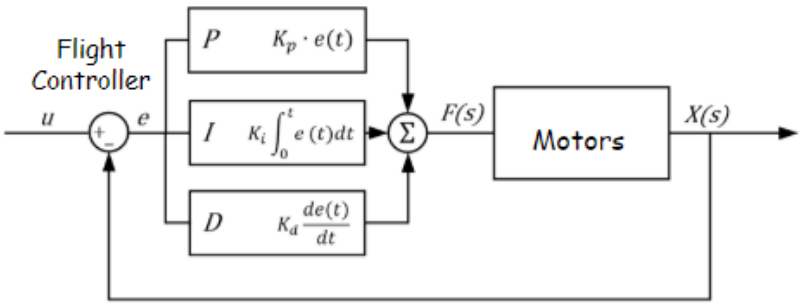


Figure 8: Block diagram of PID feedback control within flight controller

The PID loop control mechanism works by considering a proportional, integral and derivative aspect of the error between the user inputted output set-point and actual output [9]. . Consider the following equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Here, $e(t)$ denotes the error, $u(t)$ denotes the output, and K_p, K_i, K_d denote the proportional, integral and derivative gain constants respectively. It is clear to see why we would need a proportional gain to reduce the error and reach the user set-point - the larger the error, the larger the output response to counteract this error. We can see two problems with this kind of error though: firstly, there is a possibility for the output response to overshoot the set-point, and cause oscillations about this point. We can solve this problem by considering a time derivative term that reacts to how the error is changing with time. Secondly, for there to be any output response from the system there must always be an error, causing the final steady-state output to be above or below the user set-point. To solve this problem we use an integral component which sums the error up over time, adding up to fix this steady-state error.

```
void calculate_pid(){
  //Roll calculations
  pid_error_temp = gyro_roll_input - pid_roll_setpoint;
  pid_i_mem_roll += pid_i_gain_roll * pid_error_temp;
  if(pid_i_mem_roll > pid_max_roll)pid_i_mem_roll = pid_max_roll;
  else if(pid_i_mem_roll < pid_max_roll * -1)pid_i_mem_roll = pid_max_roll * -1;

  pid_output_roll = pid_p_gain_roll * pid_error_temp + pid_i_mem_roll + pid_d_gain_roll * (pid_error_temp - pid_last_roll_d_error);
  if(pid_output_roll > pid_max_roll)pid_output_roll = pid_max_roll;
  else if(pid_output_roll < pid_max_roll * -1)pid_output_roll = pid_max_roll * -1;

  pid_last_roll_d_error = pid_error_temp;

  //Pitch calculations
  pid_error_temp = gyro_pitch_input - pid_pitch_setpoint;
  pid_i_mem_pitch += pid_i_gain_pitch * pid_error_temp;
  if(pid_i_mem_pitch > pid_max_pitch)pid_i_mem_pitch = pid_max_pitch;
  else if(pid_i_mem_pitch < pid_max_pitch * -1)pid_i_mem_pitch = pid_max_pitch * -1;

  pid_output_pitch = pid_p_gain_pitch * pid_error_temp + pid_i_mem_pitch + pid_d_gain_pitch * (pid_error_temp - pid_last_pitch_d_error);
  if(pid_output_pitch > pid_max_pitch)pid_output_pitch = pid_max_pitch;
  else if(pid_output_pitch < pid_max_pitch * -1)pid_output_pitch = pid_max_pitch * -1;

  pid_last_pitch_d_error = pid_error_temp;

  //Yaw calculations
  pid_error_temp = gyro_yaw_input - pid_yaw_setpoint;
  pid_i_mem_yaw += pid_i_gain_yaw * pid_error_temp;
  if(pid_i_mem_yaw > pid_max_yaw)pid_i_mem_yaw = pid_max_yaw;
  else if(pid_i_mem_yaw < pid_max_yaw * -1)pid_i_mem_yaw = pid_max_yaw * -1;

  pid_output_yaw = pid_p_gain_yaw * pid_error_temp + pid_i_mem_yaw + pid_d_gain_yaw * (pid_error_temp - pid_last_yaw_d_error);
  if(pid_output_yaw > pid_max_yaw)pid_output_yaw = pid_max_yaw;
  else if(pid_output_yaw < pid_max_yaw * -1)pid_output_yaw = pid_max_yaw * -1;

  pid_last_yaw_d_error = pid_error_temp;
}
```

Figure 9: Complete PID calculation function contained within Arduino flight controller sketch

For our quadcopter, we required three PID controllers each for 3 movements: pitch, roll and yaw. The full code method that implements equation 2.6 for these movements is seen in figure 9. You can also refer to figure 8 to see a block diagram version of how the PID controller is used in our project. We tuned our gain parameters manually, through flight and testing. Unfortunately, we do not quite have the best gain settings as of yet.

```
throttle = receiver_input_channel_3;                                    //We need the throttle signal as a base signal.

if (start == 2){                                                        //The motors are started.
  if (throttle > 1800) throttle = 1800;                                 //We need some room to keep full control at full throttle.
  esc_1 = throttle - pid_output_pitch + pid_output_roll - pid_output_yaw; //Calculate the pulse for esc 1 (front-right - CCW)
  esc_2 = throttle + pid_output_pitch + pid_output_roll + pid_output_yaw; //Calculate the pulse for esc 2 (rear-right - CW)
  esc_3 = throttle + pid_output_pitch - pid_output_roll - pid_output_yaw; //Calculate the pulse for esc 3 (rear-left - CCW)
  esc_4 = throttle - pid_output_pitch - pid_output_roll + pid_output_yaw; //Calculate the pulse for esc 4 (front-left - CW)
```

Figure 10: Snippet of Arduino flight controller sketch that calculates the necessary ESC output

The last aspect of the flight controller is to communicate these PID outputs to the ESCs. This can be seen using the code snippet in figure 10. Here, the outputs of the pitch, yaw and roll PID calculations are either subtracted or added to the throttle value. To determine whether the value should be added or subtracted, you simply have to consider what the motor response should be for a given movement. To pitch the nose up, you need the front two motors to be spinning faster than the rear two motors. To roll right, you need the left two motors spinning faster than the right two motors. Finally, for yawing right, you need the front right motor and left rear motor spinning faster than the other two motors. For the opposite movement, the vice versa is true. This completes the primary objective flight controller.

3 Parts List

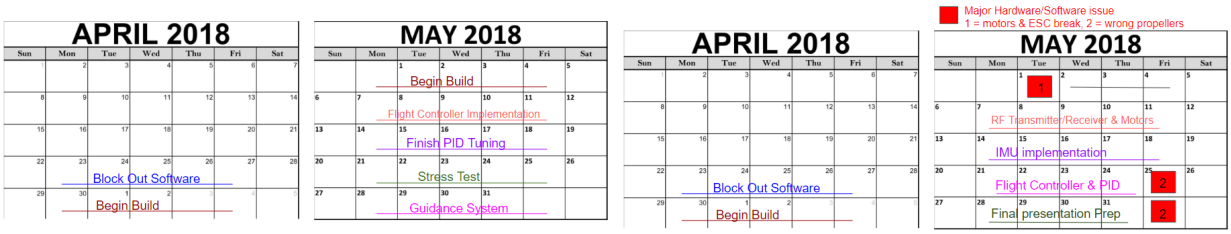
PARTS	VENDOR	UNITS	COST/UNIT	TOTAL
Drone Frame				
YoungRC F450 Quadcopter MultiCopter Frame Kit	Amazon	1	$18.99	$18.99
Propellers				
Hausbell 2x CW+CCW 9450 Self-tightening Propellers	Amazon	2	$9.99	$19.98
RAYCorp® 1045 10x4.5 Propellers. 8 Pieces(4 CW, 4 CCW)	Amazon	1	$9.99	$9.99
Motors+ESC				
4x 920KV Brushless Motor (CW/CCW) + 4x 30A ESC	Amazon	2	$63.00	$126.00
Battery				
3S 11.1V 2800mAh 35C Li-Po Battery Pack	Amazon	1	$20.99	$20.99
Li-PO/Li-Fe Balance Charger	Amazon	1	$19.99	$19.99
Internal Measurement Unit (IMU)				
GY-521 MPU-6050 6DOF	Amazon	1	$5.98	$5.98
Radio Transmitter & Receiver				
Flysky FS-T6 6-CH TX Transmitter + Radio Control System	Amazon	1	$48.00	$48.00
Miscellaneous Items				
Glarks 635Pcs Connector, Headers & Wire Cable Assortment Kit	Amazon	1	$13.97	$13.97
Secondary Objective Parts				
HC-SR04 Ultrasonic Sensor Distance Module (5pcs)	Amazon	1	$9.97	$9.97
Google AIY Voice Kit	Amazon	1	$15.94	$15.94
CanaKit Raspberry Pi 3 Complete Starter Kit - 32 GB Edition	Amazon	1	$74.99	$74.99
Elegoo For Arduino Nano V3.0 (X3)	Amazon	1	$13.86	$13.86
				$398.65

Table 1: Finalized parts list for the project. **Please note that we had ordered some DigiKey parts, which never arrived. We have not included these parts as they were replaced by similar parts from Amazon.*

Table 1 shows our finalized parts list for the project. Unfortunately, we had gone slightly above budget. This is largely due to the fact that we had damaged 2 motors and 2 ESCs during the motor testing stage. Furthermore, we had preemptively ordered our secondary objective items thinking that our flight controller

would work on first attempt smoothly, and that the parts would take about 2 weeks to arrive.

4 Timeline



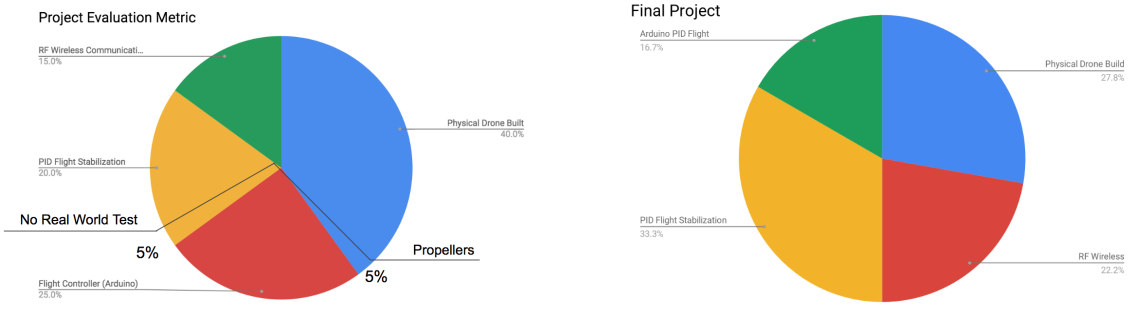
(a) Initial timeline

(b) Final timeline

Figure 11: The original and final timelines

The initially proposed timeline (figure 11a) was definitely an ambitious estimate of the projects progress. Originally, we had planned to have a week to stress test the quadcopter and a final week to implement the ultrasonic guidance control system. These bonus features had to be cut for the sake of time. The PID flight stabilization took longer to realize than expected and due to hardware ordering issues we had to postpone our first flight until the end of tenth week. This was namely due to the confusion surrounding the propeller fitting that is connected to the motor. We found that every company sells a different propeller fitting for use with their proprietary motors, this set us back about a week during which we had finished the PID feedback but we couldn't test the flight capabilities. Another hardware mishap occurred in the first batch of ESC's that were delivered, in which one of the ESC's had a mosfet that was shorted. When this ESC was connected to a motor, it immediately blew the motor. The shorted ESC destroyed two motors in total which pushed us back by roughly a week in the hardware goals for the project. Finally, the extra two weeks alloted for stress testing and bonus features were very useful for extending the original flight demonstration timeline.

5 Project Evaluation Metric



(a) Initial project evaluation metric

(b) Final project evaluation metric

Figure 12: The two project evaluation metrics are show as pie charts in the figure above.

The original proposed project evaluation metric shown in ,the figure above definitely, had a few flaws. Firstly, the physical drone build took up too much space on the metric this was corrected in the final metric. Assembling the drone was much easier than anticipated and should not have constituted nearly half of the

evaluation metric. Also the PID flight stabilization was increased in the final metric, as this was the core of the project. Finally, when considering the difficulty present in implementing the RF control into the flight controller this category was chosen to be increased.

6 Conclusion

As stated in the timeline section above, we were overly ambitious for this project and should have focused solely on our primary objectives rather than considering the secondary objectives. Nevertheless, we learned a lot about programming the ATmega328p specifically rather than relying on Arduino based functions, coding in C/C++, and understanding how ESC and BLDC motors, I2C communication, PID feedback loop control, and, obviously, quadcopters work. The final setback of ordering the wrong propellers was tough and cost us a lot of time and testing.

The final result of our project was a quadcopter that could lift off and turn, but not to a full enough range where it could be considered flyable. Over the first week of summer, we plan to tweak our quadcopter to make it 'flyable', and perhaps begin adding our secondary objective features.

7 Acknowledgements

We would like to thank Professor Christian Schneider for all his guidance and teaching done for this class as it was greatly appreciated. Lastly, we would also like to thank Anthony Ransford and Jacob Saret for their help, support and motivation throughout the building of our project.

References

[1] J.M. Brokking. Project YMFC-AL - the arduino auto-level quadcopter, Apr 2017.

[2] LHI 4x 2212 920KV Brushless Motor (CW / CCW) + 4x SIMONK 30A ESC For DJI Phantom. `https://www.amazon.com/LHI-920KV-Brushless-SIMONK-Phantom/dp/B00XQYTZQ2/`.

[3] Lynxmotion. *SimonK ESC User Guide*. Revision 1.0.

[4] Brushless DC Motors Control - How it Works (Part 1 of 2). `https://www.youtube.com/watch?v=ZAY5JInyHXY`, June 2012.

[5] FLOUREON 3S 11.1V 2800mAh 35C Lipo Battery Pack with XT60 Plug for RC Airplane RC Helicopter RC Car RC Truck RC Boat, RC Hobby. `https://www.amazon.com/FLOUREON-2800mAh-Battery-Airplane-Helicopter/dp/B00RXA02QW`.

[6] Flysky. *FS-T6 Digital proportional radio control system, Instruction Manual*, 2012.

[7] Faisal Zaman. Nothing Beats a Clean Signal (especially for Drones/UAVs). `http://www.qualtre.com/2015/10/nothing-beats-a-clean-signal/`, October 2015.

[8] Atmel. *ATmega328/P Datasheet Complete*, November 2016.

[9] John Bechhoefer. Feedback for physicists: A tutorial essay on control. *Rev. Mod. Phys.*, 77:783–836, Aug 2005.