

Autonomous Quadcopter

Design Review

Andrew Martin, Baobao Lu, Cindy Xin Ting

Group 37

TA: Katherine O'Kane

September 28th, 2015

Table of Contents

1. Abstract	2
1.1 Statement of Purpose	2
2. Design	3
2.1 Block Diagram	3
2.2 Block Descriptions	4
2.2.1 Motors	4
2.2.2 Electronic Speed Controllers (ESC)	4
2.2.3 Flight controller	4
2.2.4 LiPo Battery	4
2.2.5 ARM Protoboard	4
2.2.6 Ultrasonic Sensor	5
3. Requirements and Verification	6
3.1 High Level Requirement Points Allocation	6
3.2 Requirements and Verification	6
3.3 Tolerance Analysis	8
4. Cost and Schedule	9
4.1 Cost Analysis	9
4.1.1 Labor	8
4.1.2 Parts	9
4.1.3 Grand Total	9
4.2 Schedule	10
5. Calculations and Simulations	11
5.1 Ultrasonic Sensor Simulations	11

5.2 MATLAB Simulation	12
5.3 Quadcopter Kinematics Calculations	13
6. Safety Statement	13
6.1 Mechanical Safety	13
6.2 Electrical Safety	14
7. Ethical Assessments	14
7.1 Relation to IEEE code of ethics	14
7.2 Relation to ECE 445 course ethics guidelines	15
8. References	16
9. Appendix.	17
9.1 Sensor Codes	17
9.2 MATLAB Codes	22

1. Abstract

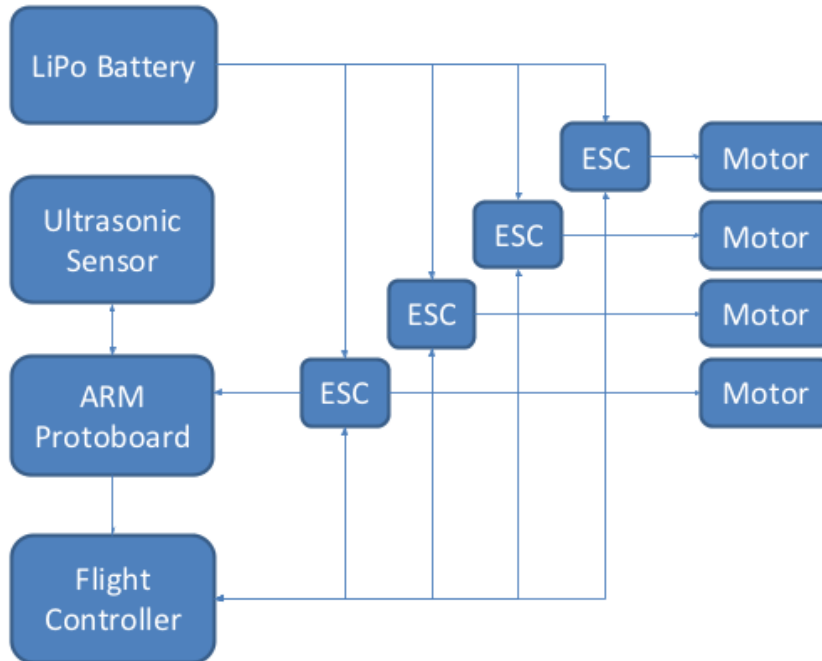
1.1 Statement of Purpose

Quadcopters are a popular project for hobbyists due to their relative low complexity and cost to build. However, due to their growing popularity, there is also an increase in number of new quadcopter builders and flyers. Thus, accidents are more likely to occur due to their inexperience in handling or controlling a quadcopter during the building phase or the test flying phase.

In order to help prevent such accidents, our goal is to design an autonomous quadcopter, including control and mechanical system, capable of balancing on it's own and flying from an origin to a goal using preset paths and directions. Our quadcopter will also have collision prevention system to further reduce the number of accidents while flying a quadcopter.

2. Design

2.1 Block Diagram



Note: ESC is short for Electronic Speed Controller

Figure 1: Block Diagram

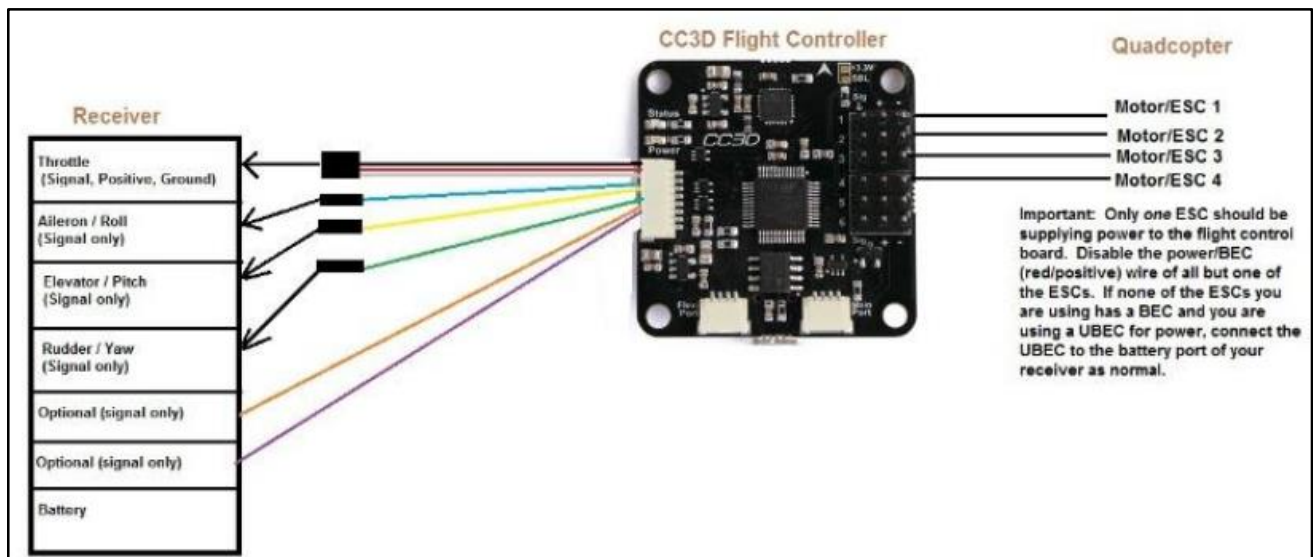


Figure 2: CC3D Flight Controller Inputs and Outputs

2.2 Block Descriptions

2.2.1 Motors

HT-450 Motors

The motors receive power from ESCs and are used to drive the propellers.

2.2.2 Electronic Speed Controllers (ESC)

F-20A Fire Red Series Simonk (RAPIDESC)

These controllers supply power to the motors by translating signals from the flight controller to electrical energy. The amount of electrical energy is based on the input signals from the flight controller. These controllers also provide power to the ARM protoboard and flight controller. They are powered directly by the LiPo Battery.

2.2.3 Flight controller

The CC3D flight controller enables aircraft stabilization and flight. the CC3D board is a stabilization hardware flight controller which runs the OpenPilot firmware. It can fly any airframe and is configured and monitored using a variant of OpenPilot software called Cleanflight. The flight controller controls the ESC's transmission of power to the motors.

2.2.4 LiPo Battery

Source of power. A Turnigy 3 cell 3300mAh LiPo battery with voltage 11.1V that routes power directly to the ESCs; which divert power to the motors, flight controller, and ARM board.

2.2.5 ARM Protoboard

This is a replacement for the traditional receiver. It's a development platform that takes input from ultrasonic sensor and charts an autonomous flight path by replicating receiver signals. It alters course if input from sensors indicates a nearby object that the quadcopter may collide with.

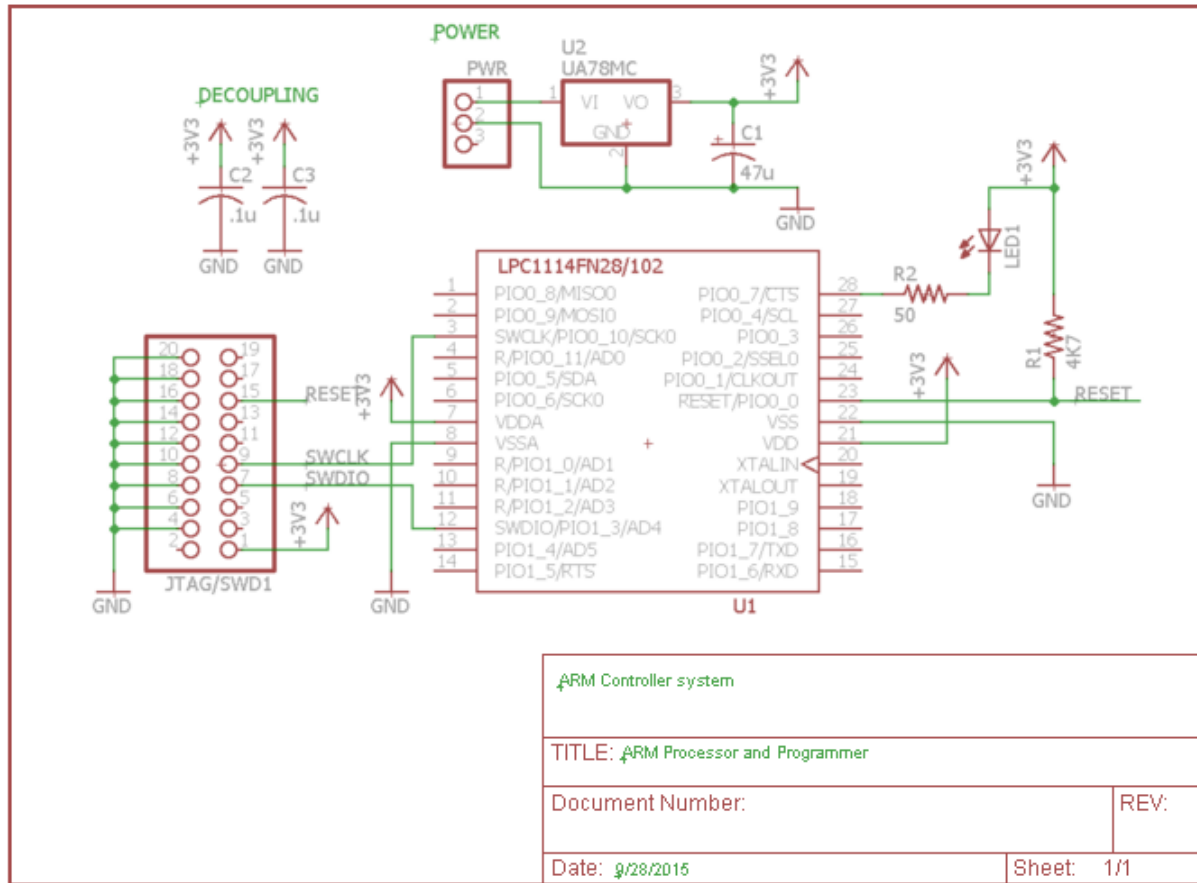


Figure 3: Circuit schematic for ARM processor and programmer

2.2.6 Ultrasonic Sensor

The HC-SR04 sensor is in charge of detecting obstacles and outputting distance data to the ARM. It is triggered by a pulse input from the ARM. It has ranging accuracy of up to 3mm. The sensor has wire connections that consist of 5V supply, ground, trigger pulse input and echo pulse output. The working current is 15mA, and the working frequency is 40Hz.

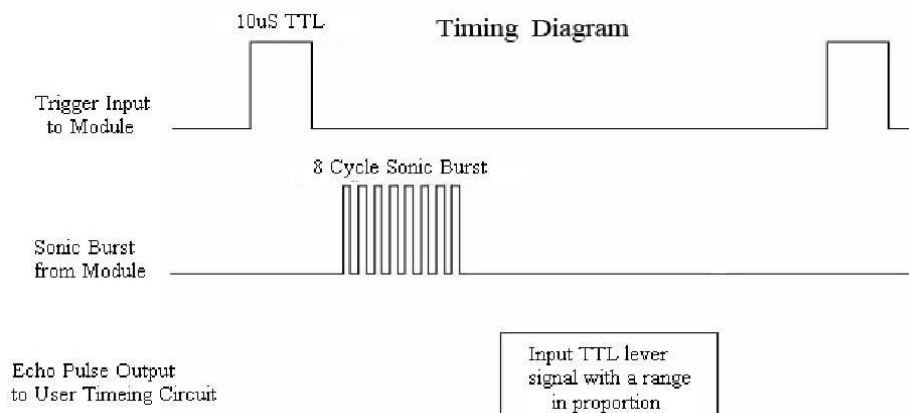


Figure 4: TTL timing diagram for HC-SR04 ultrasonic sensor

3. Requirements and Verification

3.1 High Level Requirement Points Allocation

Module Name	High Level Requirement(s)	Points
Power/ESC Module	<ul style="list-style-type: none">• Must route correct power at stable rate to all components on quadcopter including protoboard, CC3D, and motors	15
ARM Protoboard Module	<ul style="list-style-type: none">• Must duplicate receiver signals normally sent by receiver and pass to flight controller• Must receive data from ultrasonic sensors by supplying a pulse and modify flight path if proximity conditions are met	35
Flight Controller Module	<ul style="list-style-type: none">• Must correctly calibrate• Must run flashed version of Cleanflight software	15
Sensor Module	<ul style="list-style-type: none">• Must send ultrasonic distance data to ARM Protoboard once triggered via a TTL pulse	25
Motor Module	<ul style="list-style-type: none">• Must be individually calibrated to turn and propel aircraft according to flight path set by ARM Protoboard Module	10
Total		100

3.2 Requirements and Verification

Requirements	Verification
Power: Must charge to and provide 11.1V±1.5V and 12A continuous, 16A burst to ESCs	Power: <ol style="list-style-type: none">1. Monitor initial charge with IMax B6 Digital Battery charger2. When plugged in to ESCs, monitor voltage with multimeter by measuring from LiPo-ESC connector in parallel, measure amperage by connecting multimeter to same connector, but in series

<p>ARM Protoboard:</p> <ol style="list-style-type: none"> 1. Must duplicate receiver signals normally sent to flight controller and send to flight controller 2. Must receive data from ultrasonic sensors by supplying a pulse and modify flight path if proximity conditions are met 	<p>ARM Protoboard:</p> <ol style="list-style-type: none"> 1. if receiver signals are correctly duplicated, motors will turn in the same way as if they had been controlled directly by the receiver 2. If ultrasonic sensor data is correctly received and processed the flight path will change and an LED indicator will light up on the protoboard. Sending of a pulse can be verified by hooking the appropriate output pin to an oscilloscope.
<p>Flight Controller:</p> <p>Must calibrate correctly, process receiver or ARM signals correctly and send appropriate signals to ESCs.</p>	<p>Flight Controller:</p> <p>Cleanflight software will indicate if flight controller is calibrated correctly. Motors will turn when signals are sent to the flight controller if functional.</p>
<p>Sensors:</p> <p>Must send ultrasonic distance data to ARM Protoboard once triggered via a TTL pulse. TTL pulse must be $10\text{ }\mu\text{S} \pm 2\text{ }\mu\text{S}$. In order to prevent trigger signal conflicting with echo signal, measurement cycle must have a delay between pulses of at least 60 ms.</p>	<p>Sensors:</p> <p>Externally powering the sensor via power supply and looking at the echo output on the sensor via oscilloscope once a pulse trigger is supplied should show the signal generated by the return of the ultrasonic pulse to the sensor. The ARM will supply this pulse once attached to the sensor.</p> <p>Formula for calculating range based on sensor output: $\mu\text{S} / 58 = \text{cm}$ or $\mu\text{S} / 148 = \text{inch}$; or: the range = high level time * velocity (340M/S) / 2;</p>
<p>ESC:</p> <p>Must provide up to $5\text{V} \pm 0.5\text{ V}$ to flight controller and ultrasonic sensor and 10-12 A to motors</p> <p>Must output the right electrical energy that corresponds to the input signals.</p>	<p>ESC:</p> <p>Voltage and current outputs can be tested by hooking up a multimeter in parallel and series respectively. Motors will turn if system is fully assembled and controlled via ARM processor or receiver.</p>
<p>Motors:</p> <p>Must turn when powered from ESC</p>	<p>Motors:</p> <p>It's unsafe to directly power motors via a power supply for testing, so verification can only occur after full assembly and ESCs are connected to the motor leads.</p>

Table 1: Requirements and Verification

3.3 Tolerance Analysis

It's essential to determine the maximum reliable distance at which the ultrasonic sensor can detect objects. Object detection among most ultrasonic sensors is said to extend to 450 cm \approx 15 ft, but this is likely only true under ideal conditions. Since mounting such a sensor to a quadcopter will produce conditions for detection that are far from ideal, we must test the extremes to which the ultrasonic sensor can detect objects.

We must first test the fastest rate at which we can gather data from the ultrasonic sensor. The faster we can gather data, the quicker we can make adjustments in flight if the quadcopter gets too close to an object. We can accomplish this by first testing the maximum rate of triggering for which we can get useful output from the ultrasonic sensor. According to the datasheet for the ultrasonic sensor we're interested in using, we only need send out a pulse of 10 μ s to trigger a response. the datasheet suggests a delay of 60ms between pulses to prevent interference between the outgoing and returning signals, so we intend to test up to and past that rate of pulses for various distance measurements.

Our second set of tolerance analysis will be the maximum distance at which we can get reasonable results. This low-end ultrasonic sensor is said to be quite noisy, so we'll have to verify if we can get reasonable results from the sensor when it's subject to turbulence when attached to the quadcopter. We can check it's ability to operate in unstable conditions by physically moving the sensor around while it takes measurements. By placing the sensor on a flat surface next to a ruler length and pushing it down the length of a ruler away from an object while taking measurements, we can test the limits of the sensor's accuracy in taking distance measurements while in motion. We can also see the exact distance at which the sensor becomes unreliable in taking measurements.

4. Cost and Schedule

4.1 Cost Analysis

4.1.1 Labor

Name	Hourly Rate	Total Hours Invested	Total Labor = Hourly Rate x 2.5 x Total Hours Invested
Andrew Martin	40.00	100	10,000.00
Baobao Lu	40.00	100	10,000.00
Cindy Xin Ting Lee	40.00	100	10,000.00
		Total	\$30,000.00

Table 2: Labor

4.1.2 Parts

Parts	Unit Cost (USD)	Quantity	Total
Flip Sport Quadcopter Frame	65.00	1	65.00
HT-450 Motor - House	15.00	4	60.00
F-20A Fire Red Series Simonk-(RapidEsc)	8.00	4	32.00
HQ 9X4.5 MM Fiberglass Composite Propeller	3.98	4	15.92
Open Pilot CC3D	15.53	1	15.53
Turnigy 3300mAh 3S 30C Lipo Pack	25.80	1	25.80
IMax B6 Digital RC Lipo NiMh Battery Charger	22.72	1	22.72
Turnigy TGY-i6 Transmitter and 6CH Receiver	49.00	1	49.00
Breadboard PCB	8.50	1	8.50
LPC1114FN28	5.28	1	5.28
USB MINIJTAG Debugger/Emulator	13.00	1	13.00
UA78MC-UIC	1.00	1	1.00
R4.7/25 47uF Capacitor	0.15	1	0.15
COM-08375 0.1uF Capacitor	0.25	2	0.50
COM-00533 3mm Red LED	0.35	2	0.70`
CF1/2W472JRC 4.7k resistor	0.09	2	0.18
HC-SR04 Ranging Distance Sensor	10.00	1	10.00
		Total	\$326.28

Table 3: Parts

4.1.3 Grand Total

Labor	30,000.00
Parts	326.28
Total	\$30,326.28

Table 4: Grand Total

4.2 Schedule

Week	Andrew	Baobao	Cindy	Notes
9/13-9/19	Investigate spare Receiver/ Transmitter systems	Research	Finish Proposal	Project Proposal due
9/20-9/26	Purchase final parts/ Assemble Quadcopter	Draft ARM program	Investigate Reciever Signal Duplication	Eagle Assignment due
9/27-10/3	Assemble quadcopter	Design review, perform Matlab simulation	Design review, test ultrasonic sensor function	Design Review due
10/4-10/10	Investigate Reciever Signal Duplication	Draft ARM program	Investigate Reciever Signal Duplication/ Flash Cleanflight	
10/11-10/17	Assemble ARM Board	Investigate/Test Ultrasonic Sensors	Test code on assembled ARM program board	
10/18-10/24	Test ARM program board/ Individual Project Report	Investigate/Test Ultrasonic Sensors	Test ARM program board/ Individual Project Report	
10/25-10/31	Individual Project Report/ prepare mock demo	Individual Project Report/ prepare mock demo	Individual Project Report/ prepare mock demo	Individual Project Reports due
11/1-11/7	Integrate Ultrasonic Sensors with ARM board	Integrate Ultrasonic Sensors with ARM board	Refine ARM program board code	Mock Demo due
11/8-11/14	Integrate Ultrasonic Sensors with ARM board	Integrate Ultrasonic Sensors with ARM board	Refine ARM program board code	
11/15-11/21	Finalize Design/ Write final Paper	Finalize Design/ Write final Paper	Finalize Design/ Write final Paper	
11/29-12/5				Final Papers due

Table 5: Schedule

5. Calculations, Codes, and Simulations

5.1 Ultrasonic Sensor Simulations

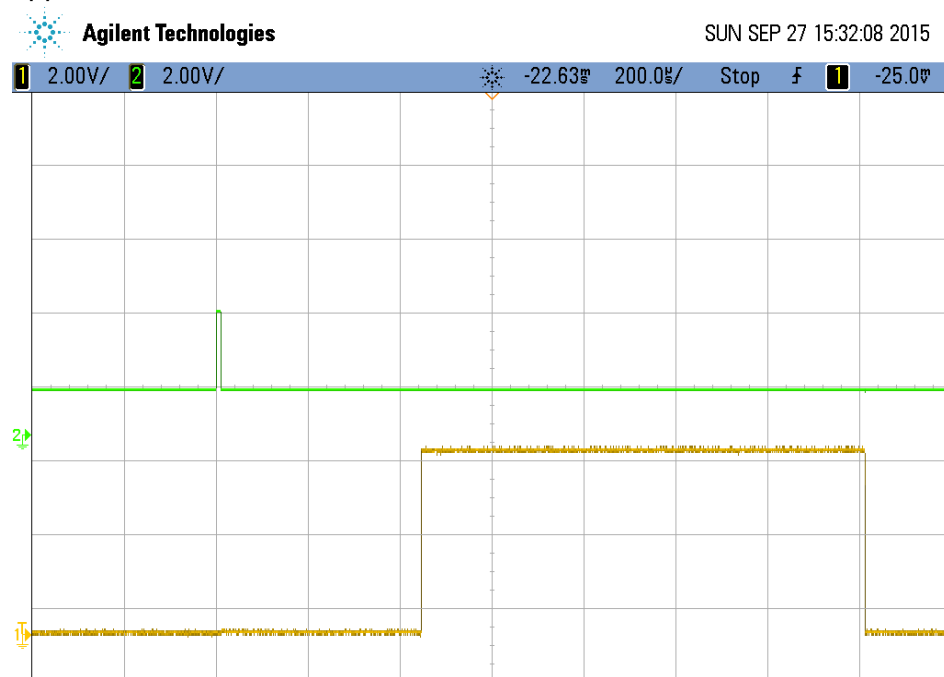
As part of our testing, we wanted to determine whether it was possible to receive correct output from the HC-SR04 sensor given ideal input and power conditions from a benchtop setting.

These considerations included powering the sensor at Vcc of 5 volts and sending a 10uS TTL pulse to the trig pin. The following settings were used to produce a 10uS TTL pulse from the signal generator:

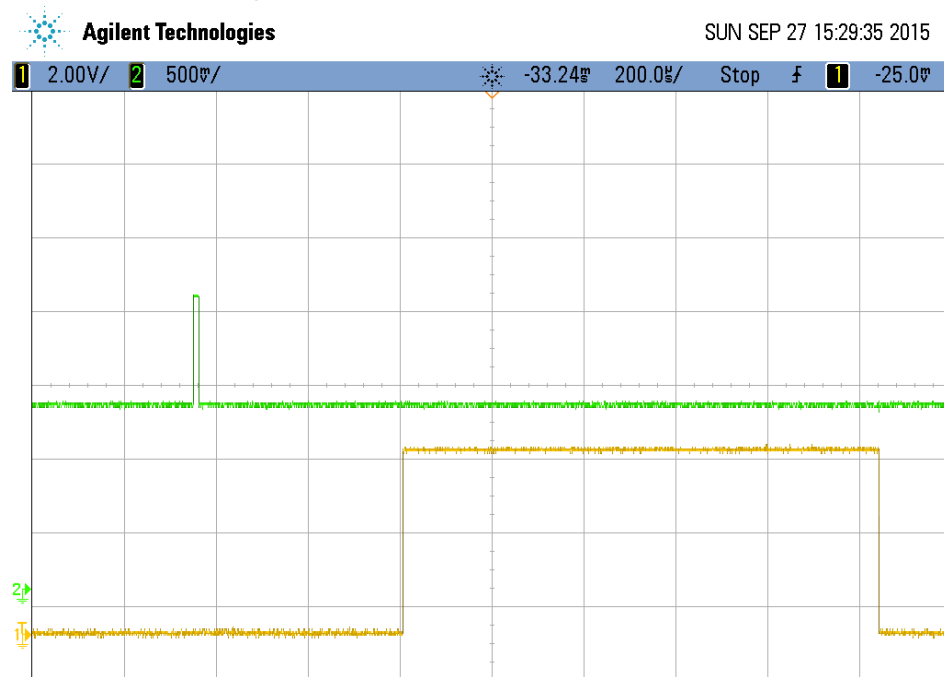
- Freq: 20kHz
- Amplitude: 1.25 Vpp
- Waveform: Square
- Offset: 1.25 Volts
- Duty Cycle: 25%
- Burst: On
- Trigger: Single

By implementing these settings, we can produce a 10uS TTL pulse that can be triggered by pressing the trigger button on the signal generator.

The resultant output for two scenarios can be seen below. The green line in both cases indicates the length of the trigger input, while the yellow line indicates the output from the echo signal. In the first graph, the object being detected is placed close to the sensor, about 1 ft. The second graph shows the results of putting an object roughly 6 inches further away. The first graph shows an echo result which is shorter in length than the second signal. This correctly indicates that the echo takes less time to travel in the first case versus the second case, because the object is closer in the first case. Further tolerance analysis must be conducted to test the limits of the sensor. Additional past arduino code for the sensor can be found in the appendix.



Plot 1: Waveform plot for short distance case



Plot 2: Waveform plot for long distance case

5.2 MATLAB Simulation

This is the quadcopter kinematics simulation of MATLAB codes. It shows the effect of quadcopter positioning on angular velocity and displacement of the quadcopter. We are in the process of customizing the code to fit our planned predetermined flight paths. Code specific to the current operation of this simulation can be found in the appendix.

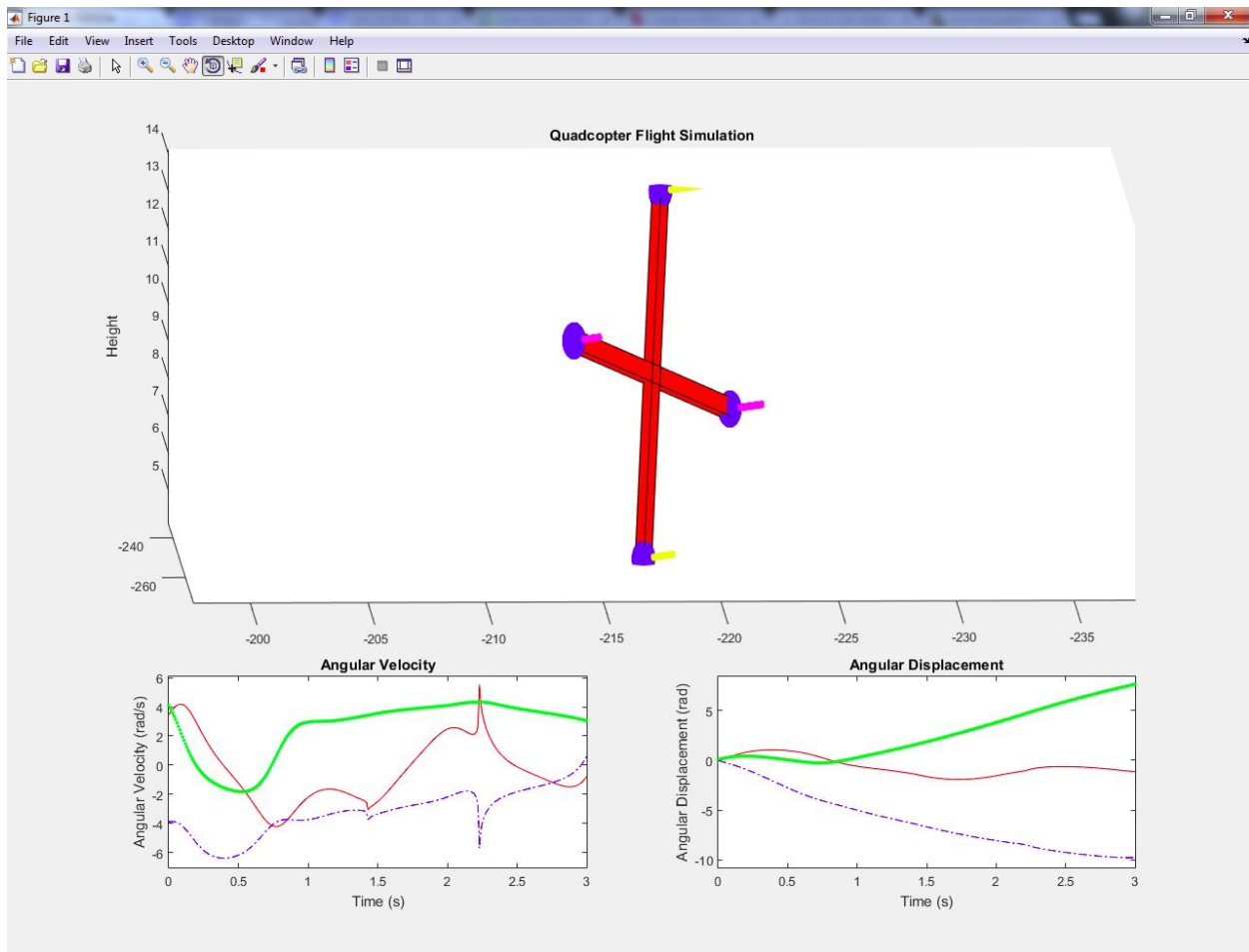


Figure 5: Matlab result screen capture.

5.3 Quadcopter Kinematics Calculations

$$F = ma$$

$$m = 43 \times 4 + 11 + 5.7 + 278 + 40 + 200 = 0.7067 \text{ kg}$$

$$F_{\text{gravity}} = m \cdot g = 0.7067 \times 9.8 \approx 6.9 \text{ N}$$

$$F > F_g \sim > 7 \text{ N}$$

$$W = F \cdot d = 7 \times 1 \text{ m hover}$$

$$P = W/t = 7/1 \text{ s} = 7 \text{ Joules/second}$$

6. Safety Statement

6.1 Mechanical Safety

To avoid accidents when testing our quadcopter, we need to perform checks on the quadcopter and the space it would be flying in. Failing to do so may cause the quadcopter to go in wrong directions or crash into a person or object. We need to examine for visible damages on the quadcopter such as loose screws, faulty connections and solder joints, and unbalanced or damaged propellers. Prior to flight, it's necessary to test the propellers to see if they spin smoothly and that they stay on the quadcopter. No objects should be near the propellers before

flight. During flight, the quadcopter should be tested in low altitudes to make sure the motors and sensors are working fine. Quadcopter must be in line of sight at all times. In laboratory testing, the propellers should be completely removed during testing to prevent any accidents.

6.2 Electrical Safety

Electrical safety largely centers around the charging and discharging of our battery - a Turnigy 3300mAh 3S 30C Lipo battery. This battery is capable of discharging 30 coulombs, or 30 amps in one second. Our electronic speed controllers each require 30 amps for normal operation, and can output a max of 3 amps to various control circuitry if necessary. Needless to say, this battery and the attached components can pose a serious safety hazard if mishandled.

During the process of charging a battery, it is necessary to use a specific LiPO charging station designed for the battery. This station monitors the charge of the battery and prevents overcharging. It is necessary to observe the battery charging during the entire duration of the charging process to check for swelling within the battery. Any such swelling of the battery would require the handler to immediately disconnect power from the battery and to reach a safe distance. Such a battery has a risk of catching fire and must be safely disposed of after waiting at least 15 minutes. The battery cannot be discharged below 9.9V during a test run without risking deterioration of battery quality, so calculations will need to be done on power drain to determine the maximum safe flight time for the quadcopter. During operation, proper matching connectors should be used between the battery and ESCs in order to minimize chance of shorts or electrical shocks.

7. Ethical Assessment

In order to adhere to both the IEEE code of ethics and general ethical guidelines set out for this course, we shall examine some of the ethical issues involved in design and the future production of our project.

7.1 Relation to IEEE Code of Ethics

The very first standard within the IEEE code is perhaps the most relevant standard to our project:

"To accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment"

Our goal in designing tools for making a safer quadcopter means that we have to rigorously hold to standards of safety in all aspects of design. If our quadcopter tools will truly make the quadcopter safer, we need to ensure that it will not make the quadcopter more dangerous to control or operate in other aspects. A quadcopter that flies safely by our design can have a host of secondary issues if not addressed. For example, the extra power drain could cause the quadcopter to stop working without proper warning. Poor documentation of our setup could cause unexpected errors or pose risks to users attempting to duplicate our work. If it happens that our design upon completion has the potential to introduce additional risk, it is our

duty to identify, document, publish, and address said risks. This issue falls closely in line with statements 3, 5, 7 and 9 of the IEEE code.

7.2 Relation to ECE 445 course ethics guidelines

The ECE 455 course ethics guidelines touch heavily on thinking outside the limits of the IEEE code, and on the wider impact our project may have on the world. It is important to address ethical concerns during design, build, and general use of our quadcopter project.

Would we be comfortable with the choices we've made in design of our project? The major concern of ethics in design is credit: have we given proper credit to the people and projects that have come before us, and the influence their work has had on us? It's safe to say that this is true: we have rigorously cited the works from which we have pulled information from and have explained in detail why we've chosen the parts and methodology we have. Our code sources and print sources are all documented according to MLA standards.

An additional question of ethics relates to the build of the quadcopter. It also falls in line closely with statement 6 of the IEEE code. Essentially, "are we qualified and competent enough to make sound decisions in the construction and design of this quadcopter?" Such a project comes with understandable risks: high voltage batteries that can hurt people and damage equipment, fast moving propellers that can injure students, and other such concerns. We feel we've thoroughly documented the risks that are posed in the building of the quadcopter, and that this suggests that we're capable of approaching the build with deliberation. Further, our diverse educational background in control systems, power electronics, and computer science gives us all the necessary background to implement the build successfully.

A final question of ethics relates to the general use and future of the quadcopter project. Will we be comfortable having our names attached to this project, and any legacy that it may have on our lives and careers? Ultimately, I'm proud of the work we're doing and the goals we set out to accomplish through this project. We want to make quadcopters safer through our work, and will work hard to make our project meet that goal. Even if we fail to deliver a safer solution to quadcopter flight, I know our research and work will support the work of others who will come after us with the same goal in mind. And I know that we won't push our product unless we're entirely sure its existence in the public domain will support our goal to make quadcopter use safer.

In conclusion, we feel that we've examined all aspects of quadcopter design, build, and future use and have found that we can appropriately answer any ethical questions, as well as further questions presented by our peers or advisors.

8. References

Cleanflight Flight Controller. Cleanflight. Web. Cleanflight. 25 Aug. 2015

Cytron Technologies. "HC-SR04 Ultrasonic Sensor." HC-SR04, May 2013. E-Book. 20 Sept. 2015.

GeeTech. "CC3D Flight Control Board." CC3D, May 2013. E-Book. 28 Mar. 2015. 25 Aug. 2015

Gibiansky, Andrew. "Quadcopter Dynamics and Simulation." Andrew Gibiansky: Math - Code. N.p., 23 Nov. 2012. Web. 20 Sept. 2015.

Nic. "Quadcopter." Mr Digital. N.p., 7 May 2011. Web. 5 Aug. 2015.

NXP Semiconductors. "32-bit ARM Cortex-M0 microcontroller; up to 64 kB flash and 8 kB SRAM." LPC1110/11/12/13/14/15, 16 Apr. 2010 [Revised Mar. 2014]. E-Book. 20 Sept. 2015.

Painless260. "(4/8) Naze32 Flight Controller – Adding Sonar (HC-SR04 Module)." YouTube. YouTube, 17 Feb. 2015. Web. 25 Sept. 2015.

"Unmanned Aircraft Systems (UAS) Frequently Asked Questions." Unmanned Aircraft Systems (UAS) Frequently Asked Questions. Federal Aviation Administration, 24 July 2015. Web. 9 Sept. 2015.

9. Appendix

9.1 Sensor codes

```
#include <Servo.h> // servo library
```

```
#include <AcceleroMMA7361.h>
```

```
AcceleroMMA7361 accelero;
```

```
int x;
```

```
int y;
```

```
int z;
```

```
int led = 4;
```

```
int led_state = 0;
```

```
Servo servo1; // servo control object
```

```
#define echoPin 7 // Echo Pin
```

```
#define trigPin 8 // Trigger Pin
```

```
#define LEDPin 13 // Onboard LED
```

```
int maximumRange = 50; // Maximum range needed
```

```
int minimumRange = 5; // Minimum range needed
```

```
long duration, distance; // Duration used to calculate distance
```

```
//Constants to name the pins
```

```
const int analogInPin = A0; //Analog input pin that the IR sensor is attached to
```

```
const int analogOutPinL = 6; //Analog output pin that goes to Left-side wheel drive
```

```
const int analogOutPinR = 11; //Analog output pin that goes to Right-side wheel drive
```

```
//Constants to ensure the angles in the turning functions work correctly
```

```
//These must be found through trial and error
```

```
const float turnDelayConstantR = 6.1;
```

```
const float turnDelayConstantL = 6.3;
```

```
//Initialize the input variable
```

```
int sensorValue = 0;
```

```
boolean trackingState = true;
```

```
void turnCarRight(int turnAngle);
```

```
void turnCarLeft(int turnAngle);
```

```
void ultra(int position);
```

```
void setup()
```

```
{
```

```

Serial.begin (9600);
pinMode(trigPin, OUTPUT);
pinMode(echoPin, INPUT);
pinMode(analogOutPinL, OUTPUT);
pinMode(analogOutPinR, OUTPUT);
servo1.attach(9);
Serial.begin(9600);
accelero.begin(5, 12, 10, 13, A1, A2, A3);
accelero.setARefVoltage(3.3);           //sets the AREF voltage to 3.3V
accelero.setSensitivity(LOW);           //sets the sensitivity to +/-6G
accelero.calibrate();
pinMode(led, OUTPUT);
}

void loop()
{
  int position;
  //servo1.write(180); // Tell servo to go to 180 degrees

  //ultra(position);    // Pause to get it time to move
  //servo1.write(0);    // Tell servo to go to 0 degrees
  //ultra(position);    // Pause to get it time to move
  // Change position at a slower speed:
  // To slow down the servo's motion, we'll use a for() loop
  // to give it a bunch of intermediate positions, with 20ms
  // delays between them. You can change the step size to make
  // the servo slow down or speed up. Note that the servo can't
  // move faster than its full speed, and you won't be able
  // to update it any faster than every 20ms.
  // Tell servo to go to 180 degrees, stepping by two degrees

  while(trackingState == false){
    analogWrite(analogOutPinR, 0);
    analogWrite(analogOutPinL, 0);
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
  }
}

```

```

digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);
duration = pulseIn(echoPin, HIGH);
//Calculate the distance (in cm) based on the speed of sound.
distance = duration/58.2;
if(distance<=5)
{
    trackingState = false;
}else{
    trackingState = true;
    digitalWrite(led, LOW);
    break;
}

}

if(trackingState==true){
for(position = 0; position < 180; position += 10)
{
    if(trackingState==false) {break;}
    servo1.write(position); // Move to next position
    delay(5);      // Short pause to allow it to move
    //Serial.println(position );
    ultra(position);

}
// Tell servo to go to 0 degrees, stepping by one degree
for(position = 180; position >= 0; position -= 10)
{
    if(trackingState==false) {break;}
    servo1.write(position); // Move to next position
    delay(5);      // Short pause to allow it to move
    //Serial.println(position );
    ultra(position);

}
}

```

```

}
void ultra(int position){
  do {
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
    duration = pulseIn(echoPin, HIGH);
    //Calculate the distance (in cm) based on the speed of sound.
    distance = duration/58.2;
    //Serial.println(distance);
    if (distance <= 5){
      digitalWrite (led, HIGH);
      trackingState = false;
    }else{
      digitalWrite (led, LOW);
      trackingState = true;
    }
  }
  if (distance >= maximumRange || distance <= minimumRange){
    //sensorValue = analogRead(analogInPin);
    //analogWrite(analogOutPinL, 170);
    //analogWrite(analogOutPinR, 175);

    //if(sensorValue > 700) {
      //turnCarLeft(180);
      //turnCarRight(180);
      //trackingState = true;
      //Serial.println(trackingState);
    //}

    /* Send a negative number to computer and Turn LED ON
    to indicate "out of range" */
    //Serial.println("-1 ");
    //Serial.println(distance);
  }
  else {

```

```

/* Send the distance to the computer using Serial protocol, and
turn LED OFF to indicate successful reading. */
// Serial.println(distance);
Serial.println(position);
if(position > 90)
{
    turnCarRight(position-90);
    Serial.println(position);
}
else if (position >= 80 && position <= 100){
    analogWrite(analogOutPinL, 170);
    analogWrite(analogOutPinR, 175);
}
else {
    turnCarLeft(90-position);
    Serial.println(90-position);
}

    //delay(1000);
}
delay(50);
}while (distance <= maximumRange && distance >= minimumRange);
return;
}

void turnCarRight(int turnAngle) {
    analogWrite(analogOutPinL, 200);
    analogWrite(analogOutPinR, 0);
    delay(turnDelayConstantR * turnAngle);
    analogWrite(analogOutPinL, 0);
}

//Turn the car to the left
void turnCarLeft(int turnAngle) {
    analogWrite(analogOutPinL, 0);
    analogWrite(analogOutPinR, 200);
    delay(turnDelayConstantL * turnAngle);
    analogWrite(analogOutPinR, 0);
}

```

```
}
```

9.2 MATLAB codes

Controller.m

% Create a controller based on it's name, using a look-up table.

```
function c = controller(name, Kd, Kp, Ki)
    % Use manually tuned parameters, unless arguments provide the parameters.
    if nargin == 1
        Kd = 4;
        Kp = 3;
        Ki = 5.5;
    elseif nargin == 2 || nargin > 4
        error('Incorrect number of parameters.');
```

end

```

    if strcmpi(name, 'pd')
        c = @(state, thetadot) pd_controller(state, thetadot, Kd, Kp);
    elseif strcmpi(name, 'pid')
        c = @(state, thetadot) pid_controller(state, thetadot, Kd, Kp, Ki);
    else
        error(sprintf('Unknown controller type "%s"', name));
    end
end
```

% Implement a PD controller. See simulate(controller).

```
function [input, state] = pd_controller(state, thetadot, Kd, Kp)
    % Initialize integral to zero when it doesn't exist.
    if ~isfield(state, 'integral')
        state.integral = zeros(3, 1);
    end
```

```

    % Compute total thrust.
    total = state.m * state.g / state.k / ...
        (cos(state.integral(1)) * cos(state.integral(2)));
```

```

    % Compute PD error and inputs.
    err = Kd * thetadot + Kp * state.integral;
    input = err2inputs(state, err, total);
```

```

    % Update controller state.
    state.integral = state.integral + state.dt .* thetadot;
```

end

% Implement a PID controller. See simulate(controller).

function [input, state] = pid_controller(state, thetadot, Kd, Kp, Ki)

 % Initialize integrals to zero when it doesn't exist.

 if ~isfield(state, 'integral')

 state.integral = zeros(3, 1);

 state.integral2 = zeros(3, 1);

 end

 % Prevent wind-up

 if max(abs(state.integral2)) > 0.01

 state.integral2(:) = 0;

 end

 % Compute total thrust.

 total = state.m * state.g / state.k / ...

 (cos(state.integral(1)) * cos(state.integral(2)));

 % Compute error and inputs.

 err = Kd * thetadot + Kp * state.integral - Ki * state.integral2;

 input = err2inputs(state, err, total);

 % Update controller state.

 state.integral = state.integral + state.dt .* thetadot;

 state.integral2 = state.integral2 + state.dt .* state.integral;

end

% Given desired torques, desired total thrust, and physical parameters,

% solve for required system inputs.

function inputs = err2inputs(state, err, total)

 e1 = err(1);

 e2 = err(2);

 e3 = err(3);

 lx = state.l(1, 1);

 ly = state.l(2, 2);

 lz = state.l(3, 3);

 k = state.k;

 L = state.L;

 b = state.b;

 inputs = zeros(4, 1);

 inputs(1) = total/4 -(2 * b * e1 * lx + e3 * lz * k * L)/(4 * b * k * L);

 inputs(2) = total/4 + e3 * lz/(4 * b) - (e2 * ly)/(2 * k * L);


```

        inputs(3) = total/4 - (-2 * b * e1 * lx + e3 * lz * k * L)/(4 * b * k * L);
        inputs(4) = total/4 + e3 * lz/(4 * b) + (e2 * ly)/(2 * k * L);
    end

```

quadcopter.m

```

pParam = 0.5; iParam = 0.01; dParam = 0.1;
control = controller('pid', pParam, iParam, dParam);

```

```

tstart = 0; tend = 3; dt = 0.01;
data = simulate(control, tstart, tend, dt);

```

```

visualize(data);

```

rotation.m

% Compute rotation matrix for a set of angles.

```

function R = rotation(angles)
    phi = angles(3);
    theta = angles(2);
    psi = angles(1);

    R = zeros(3);
    R(:, 1) = [
        cos(phi) * cos(theta)
        cos(theta) * sin(phi)
        - sin(theta)
    ];
    R(:, 2) = [
        cos(phi) * sin(theta) * sin(psi) - cos(psi) * sin(phi)
        cos(phi) * cos(psi) + sin(phi) * sin(theta) * sin(psi)
        cos(theta) * sin(psi)
    ];
    R(:, 3) = [
        sin(phi) * sin(psi) + cos(phi) * cos(psi) * sin(theta)
        cos(psi) * sin(phi) * sin(theta) - cos(phi) * sin(psi)
        cos(theta) * cos(psi)
    ];
end

```

simulate.m

```

% Perform a simulation of a quadcopter, from t = 0 through t = 10.
% As an argument, take a controller function. This function must accept

```

```

% a struct containing the physical parameters of the system and the current
% gyro readings. The controller may use the struct to store persistent state, and
% return this state as a second output value. If no controller is given,
% a simulation is run with some pre-determined inputs.
% The output of this function is a data struct with the following fields, recorded
% at each time-step during the simulation:
%
% data =
%
%     x: [3xN double]
%     theta: [3xN double]
%     vel: [3xN double]
%     angvel: [3xN double]
%     t: [1xN double]
%     input: [4xN double]
%     dt: 0.0050
function result = simulate(controller, tstart, tend, dt)
    % Physical constants.
    g = 9.81;
    m = 0.5;
    L = 0.25;
    k = 3e-6;
    b = 1e-7;
    I = diag([5e-3, 5e-3, 10e-3]);
    kd = 0.25;

    % Simulation times, in seconds.
    if nargin < 4
        tstart = 0;
        tend = 4;
        dt = 0.005;
    end
    ts = tstart:dt:tend;

    % Number of points in the simulation.
    N = numel(ts);

    % Output values, recorded as the simulation runs.
    xout = zeros(3, N);
    xdotout = zeros(3, N);
    thetaout = zeros(3, N);
    thetadotout = zeros(3, N);
    inputout = zeros(4, N);

```

```

% Struct given to the controller. Controller may store its persistent state in it.
controller_params = struct('dt', dt, 'l', l, 'k', k, 'L', L, 'b', b, 'm', m, 'g', g);

% Initial system state.
x = [0; 0; 10];
xdot = zeros(3, 1);
theta = zeros(3, 1);

% If we are running without a controller, do not disturb the system.
if nargin == 0
    thetadot = zeros(3, 1);
else
    % With a control, give a random deviation in the angular velocity.
    % Deviation is in degrees/sec.
    deviation = 300;
    thetadot = (2 * deviation * rand(3, 1) - deviation)*pi/180;
end

ind = 0;
for t = ts
    ind = ind + 1;

    % Get input from built-in input or controller.
    if nargin == 0
        i = input(t);
    else
        [i, controller_params] = controller(controller_params, thetadot);
    end

    % Compute forces, torques, and accelerations.
    omega = thetadot2omega(thetadot, theta);
    a = acceleration(i, theta, xdot, m, g, k, kd);
    omegadot = angular_acceleration(i, omega, l, L, b, k);

    % Advance system state.
    omega = omega + dt * omegadot;
    thetadot = omega2thetadot(omega, theta);
    theta = theta + dt * thetadot;
    xdot = xdot + dt * a;
    x = x + dt * xdot;

    % Store simulation state for output.
    xout(:, ind) = x;
    xdotout(:, ind) = xdot;
end

```

```

        thetaout(:, ind) = theta;
        thetadotout(:, ind) = thetadot;
        inputout(:, ind) = i;
    end

    % Put all simulation variables into an output struct.
    result = struct('x', xout, 'theta', thetaout, 'vel', xdotout, ...
        'angvel', thetadotout, 't', ts, 'dt', dt, 'input', inputout);
end

% Arbitrary test input.
function in = input(t)
    in = zeros(4, 1);
    in(:) = 700;
    in(1) = in(1) + 150;
    in(3) = in(3) + 150;
    in = in .^ 2;
end

% Compute thrust given current inputs and thrust coefficient.
function T = thrust(inputs, k)
    T = [0; 0; k * sum(inputs)];
end

% Compute torques, given current inputs, length, drag coefficient, and thrust coefficient.
function tau = torques(inputs, L, b, k)
    tau = [
        L * k * (inputs(1) - inputs(3))
        L * k * (inputs(2) - inputs(4))
        b * (inputs(1) - inputs(2) + inputs(3) - inputs(4))
    ];
end

% Compute acceleration in inertial reference frame
% Parameters:
% g: gravity acceleration
% m: mass of quadcopter
% k: thrust coefficient
% kd: global drag coefficient
function a = acceleration(inputs, angles, vels, m, g, k, kd)
    gravity = [0; 0; -g];
    R = rotation(angles);
    T = R * thrust(inputs, k);
    Fd = -kd * vels;

```

```

        a = gravity + 1 / m * T + Fd;
end

% Compute angular acceleration in body frame
% Parameters:
% I: inertia matrix
function omegadot = angular_acceleration(inputs, omega, I, L, b, k)
    tau = torques(inputs, L, b, k);
    omegadot = inv(I) * (tau - cross(omega, I * omega));
end

% Convert derivatives of roll, pitch, yaw to omega.
function omega = thetadot2omega(thetadot, angles)
    phi = angles(1);
    theta = angles(2);
    psi = angles(3);
    W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
    omega = W * thetadot;
end

% Convert omega to roll, pitch, yaw derivatives
function thetadot = omega2thetadot(omega, angles)
    phi = angles(1);
    theta = angles(2);
    psi = angles(3);
    W = [
        1, 0, -sin(theta)
        0, cos(phi), cos(theta)*sin(phi)
        0, -sin(phi), cos(theta)*cos(phi)
    ];
    thetadot = inv(W) * omega;
end

```

```

tune.m
% Compute an optimal set of PID parameters
% by initializing the parameters randomly,
% minimizing a cost function using a numerical gradient
% estimate, repeating this several times, and
% choosing the best result.

```

```

function theta = tune();
    % How many times should we repeat to try to get better results?
    attempts = 10;

    % Keep track of minimum cost so far, and best parameter set.
    min_cost = 1e10;
    best_theta = -1;

    for i = 1:attempts
        % Compute next set of parameters and their costs.
        [theta, costs] = minimize;

        % If this is the best we've seen so far, store it.
        if costs(end) < min_cost
            min_cost = costs(end);
            best_theta = theta;
        end
    end

    % Return best parameters.
    theta = best_theta;
end

% Minimize a cost function by estimating the gradient
% numerically and using modified gradient descent to
% choose the best set of parameters.
function [theta, costs] = minimize()
    % Initialize weights randomly.
    theta = 1 * rand(1, 3);

    % Use a small step size  $\alpha$ .
    alpha = 0.03;

    % Maximum number of iterations to use.
    % In general, we should not reach this,
    % as we should get to a steady-state earlier.
    max_iterations = 500;

    % Each time we compute cost and gradient, we are actually
    % computing them using different disturbances. In order to make our
    % control parameters general, we take the average of several costs and gradients
    % in order to obtain our estimates for a given parameter set. This variable
    % indicates how many different measurements we average. As we iterate longer,
    % we may want to increase this in order to make our gradients more precise.

```

```

average_length = 3;

for iteration = 1:max_iterations
    disp(sprintf('Iteration %d...', iteration));

    % Compute costs (with averaging) for the current parameters.
    costs(iteration) = mean_value(@cost, theta, average_length);

    % Check if we can stop. We stop if we have reached a steady-state.
    % In order to decide whether a steady-state has been reached, we
    % look at the previous fifty costs. We fit a line to the graph of
    % costs vs iterations, and if the slope of that line is statistically
    % insignificant (the 99% confidence interval includes zero), we
    % say that we have reached a steady state, and stop iterating.
    num_costs = 50;
    if iteration > num_costs + 5
        % Previous fifty costs.
        recent_costs = costs(end - num_costs + 1:end);

        % Compute linear regression, with a bias term b(1) and a slope b(2).
        % Also, compute 99% confidence intervals for the bias and slope.
        [b, int] = regress(recent_costs', [ones(num_costs, 1) (1:num_costs)'], 0.99);

        % Find the boundaries of the slope confidence interval.
        % If zero is in-between them, our slope is negligible, and
        % further training is unnecessary. Stop iterating.
        larger = max(int(2, :));
        smaller = min(int(2, :));
        if 0 < larger && 0 > smaller
            break;
        end
    end

    % Change step size and averaging to adjust for training duration.
    % After longer training times, we may want to decrease step size and
    % increase the number of averaged samples, so that our algorithm may
    % be more sensitive to small changes and avoid overshooting the minimum.
    if iteration > 100
        alpha = 0.001;
        average_length = 8;
    elseif iteration > 200
        average_length = 15;
        alpha = 0.0005;
    end
end

```

```

    % Compute gradient for our parameters (with averaging).
    grad = mean_value(@gradient, theta, average_length);

    % Adjust parameters using step size and gradient.
    theta = theta - alpha * grad;
end

end

% Given a function that has some random component and
% may return different values for the same input, as well
% as an input for that function, compute N values for
% that function and return their average.
function value = mean_value(func, input, n)
    % Compute first one out of loop, to determine size(value).
    value = func(input);

    for i = 2:n
        value = value + func(input);
    end
    value = value / n;
end

end

% Numerically estimate the gradient of the cost function
% at a particular point in PID parameter space.
function grad = gradient(theta)
    % Use a very small displacement to estimate the limit.
    delta = 0.001;

    % Store random seed, so that all simulations are using the same disturbance.
    % Although different gradients may use different disturbances, we want the different
    % components of the gradient to be computed using the same simulation.
    s = rng;

    for i = 1:length(theta)
        var = zeros(size(theta));
        var(i) = 1;

        % Restore the random seed for each cost computation.
        % This way, the simulation that's done is the same every time.
        rng(s); left_cost = cost(theta + delta * var);
        rng(s); right_cost = cost(theta - delta * var);

        % Compute gradient with respect to ith parameter.

```



```

        grad(i) = (left_cost - right_cost) / (2 * delta);
    end
end

% Compute the cost function for a given parameter set.
% The cost function is defined as:
%  $J(\theta) = \frac{1}{t_f - t_0} \int_{t_0}^{t_f} e(t, \theta)^2 dt$ 
% where  $e(t, \theta)$  is the error at time  $t$ .
function J = cost(theta)
    % Create a controller using the given gains.
    control = controller('pid', theta(1), theta(2), theta(3));

    % Perform a simulation. Only simulate the first second, and
    % use a relatively large time-step. We do many simulations for
    % each iteration of the tuning, so we need each simulation to be quite fast.
    data = simulate(control, 0, 1, 0.05);

    % Compute the integral,  $\frac{1}{t_f - t_0} \int_{t_0}^{t_f} e(t)^2 dt$ 
    errors = sqrt(sum(data.theta.^2));
    J = sum(errors.^2) * data.dt;
end

```

visualize.m

```

% Visualize the quadcopter simulation as an animation of a 3D quadcopter.
function h = visualize_test(data)
    % Create a figure with three parts. One part is for a 3D visualization,
    % and the other two are for running graphs of angular velocity and displacement.
    figure; plots = [subplot(3, 2, 1:4), subplot(3, 2, 5), subplot(3, 2, 6)];
    subplot(plots(1));
    pause;

    % Create the quadcopter object. Returns a handle to
    % the quadcopter itself as well as the thrust-display cylinders.
    [t thrusts] = quadcopter;

    % Set axis scale and labels.
    axis([-10 30 -20 20 5 15]);
    zlabel('Height');
    title('Quadcopter Flight Simulation');

    % Animate the quadcopter with data from the simulation.
    animate(data, t, thrusts, plots);
end

```

```

% Animate a quadcopter in flight, using data from the simulation.
function animate(data, model, thrusts, plots)
    % Show frames from the animation. However, in the interest of speed,
    % skip some frames to make the animation more visually appealing.
    for t = 1:2:length(data.t)
        % The first, main part, is for the 3D visualization.
        subplot(plots(1));

        % Compute translation to correct linear position coordinates.
        dx = data.x(:, t);
        move = makehgtform('translate', dx);

        % Compute rotation to correct angles. Then, turn this rotation
        % into a 4x4 matrix representing this affine transformation.
        angles = data.theta(:, t);
        rotate = rotation(angles);
        rotate = [rotate zeros(3, 1); zeros(1, 3) 1];

        % Move the quadcopter to the right place, after putting it in the correct orientation.
        set(model,'Matrix', move * rotate);

        % Compute scaling for the thrust cylinders. The lengths should represent relative
        % strength of the thrust at each propeller, and this is just a heuristic that seems
        % to give a good visual indication of thrusts.
        scales = exp(data.input(:, t) / min(abs(data.input(:, t))) + 5) - exp(6) + 1.5;
        for i = 1:4
            % Scale each cylinder. For negative scales, we need to flip the cylinder
            % using a rotation, because makehgtform does not understand negative scaling.
            s = scales(i);
            if s < 0
                scalez = makehgtform('yrotate', pi) * makehgtform('scale', [1, 1, abs(s)]);
            elseif s > 0
                scalez = makehgtform('scale', [1, 1, s]);
            end

            % Scale the cylinder as appropriate, then move it to
            % be at the same place as the quadcopter propeller.
            set(thrusts(i), 'Matrix', move * rotate * scalez);
        end

        % Update the drawing.
        xmin = data.x(1,t)-20;
        xmax = data.x(1,t)+20;
    end
end

```

```

    ymin = data.x(2,t)-20;
    ymax = data.x(2,t)+20;
    zmin = data.x(3,t)-5;
    zmax = data.x(3,t)+5;
    axis([xmin xmax ymin ymax zmin zmax]);
    drawnow;

    % Use the bottom two parts for angular velocity and displacement.
    subplot(plots(2));
    multiplot(data, data.angvel, t);
    xlabel('Time (s)');
    ylabel('Angular Velocity (rad/s)');
    title('Angular Velocity');

    subplot(plots(3));
    multiplot(data, data.theta, t);
    xlabel('Time (s)');
    ylabel('Angular Displacement (rad)');
    title('Angular Displacement');
    end
end

% Plot three components of a vector in RGB.
function multiplot(data, values, ind)
    % Select the parts of the data to plot.
    times = data.t(:, 1:ind);
    values = values(:, 1:ind);

    % Plot in RGB, with different markers for different components.
    plot(times, values(1, :), 'r-', times, values(2, :), 'g.', times, values(3, :), 'b-.');

    % Set axes to remain constant throughout plotting.
    xmin = min(data.t);
    xmax = max(data.t);
    ymin = 1.1 * min(min(values));
    ymax = 1.1 * max(max(values));
    axis([xmin xmax ymin ymax]);
end

% Draw a quadcopter. Return a handle to the quadcopter object
% and an array of handles to the thrust display cylinders.
% These will be transformed during the animation to display
% relative thrust forces.
function [h thrusts] = quadcopter()

```

```

% Draw arms.
h(1) = prism(-5, -0.25, -0.25, 10, 0.5, 0.5);
h(2) = prism(-0.25, -5, -0.25, 0.5, 10, 0.5);

% Draw bulbs representing propellers at the end of each arm.
[x y z] = sphere;
x = 0.5 * x;
y = 0.5 * y;
z = 0.5 * z;
h(3) = surf(x - 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'b');
h(4) = surf(x + 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'b');
h(5) = surf(x, y - 5, z, 'EdgeColor', 'none', 'FaceColor', 'b');
h(6) = surf(x, y + 5, z, 'EdgeColor', 'none', 'FaceColor', 'b');

% Draw thrust cylinders.
[x y z] = cylinder(0.1, 7);
thrusts(1) = surf(x, y + 5, z, 'EdgeColor', 'none', 'FaceColor', 'm');
thrusts(2) = surf(x + 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'y');
thrusts(3) = surf(x, y - 5, z, 'EdgeColor', 'none', 'FaceColor', 'm');
thrusts(4) = surf(x - 5, y, z, 'EdgeColor', 'none', 'FaceColor', 'y');

% Create handles for each of the thrust cylinders.
for i = 1:4
    x = hgtransform;
    set(thrusts(i), 'Parent', x);
    thrusts(i) = x;
end

% Conjoin all quadcopter parts into one object.
t = hgtransform;
set(h, 'Parent', t);
h = t;
end

% Draw a 3D prism at (x, y, z) with width w,
% length l, and height h. Return a handle to
% the prism object.
function h = prism(x, y, z, w, l, h)
    [X Y Z] = prism_faces(x, y, z, w, l, h);

    faces(1, :) = [4 2 1 3];
    faces(2, :) = [4 2 1 3] + 4;
    faces(3, :) = [4 2 6 8];
    faces(4, :) = [4 2 6 8] - 1;

```

```

faces(5, :) = [1 2 6 5];
faces(6, :) = [1 2 6 5] + 2;

for i = 1:size(faces, 1)
h(i) = fill3(X(faces(i, :)), Y(faces(i, :)), Z(faces(i, :)), 'r'); hold on;
end

% Conjoin all prism faces into one object.
t = hgtransform;
set(h, 'Parent', t);
h = t;
end

% Compute the points on the edge of a prism at
% location (x, y, z) with width w, length l, and height h.
function [X Y Z] = prism_faces(x, y, z, w, l, h)
    X = [x x x x x+w x+w x+w x+w];
    Y = [y y y+l y+l y y y+l y+l];
    Z = [z z+h z z+h z z+h z z+h];
end

```