Feromone Flight Systems Manual

FeromoneRobotics

This manual includes a description of each system in the drone and goes into detail about the math, code, and hardware related to it and its overall responsibility/role in keeping the drone flying.

Link to main repository:
https://github.com/corbosiny/Quadcopters
(Nearly all code is fully commented and all of it is publicly available)

Authors:
-Corey Hulse

Full Team:
-Corey Hulse
-Alexander Opstad
-Shawn Victor

Sensors and Functions

Accelerometer - MPU6050, reads the acceleration and therefore the forces acting on the drone in each direction(X,Y,Z)

Gyroscope - MPU6050, the gyroscope part of the MPU reads the rotational motion of the drone in each direction(Pitch, Roll, Yaw)

Barometer - BMP080, reads the air pressure acting on the drone and has internal functions to compute the altitude of the drone.

Magnetometer - HMC5883L, reads our heading like a compass with regards to due north, this allows us to determine the rotation of our drone in degrees from straight north

Ultrasonic Sensor - HC-SR04, four are used on the drone for obstacle avoidance purposes. They sound out a sound wave pulse and can measure the distance of an object based upon how fast the pulse returns.

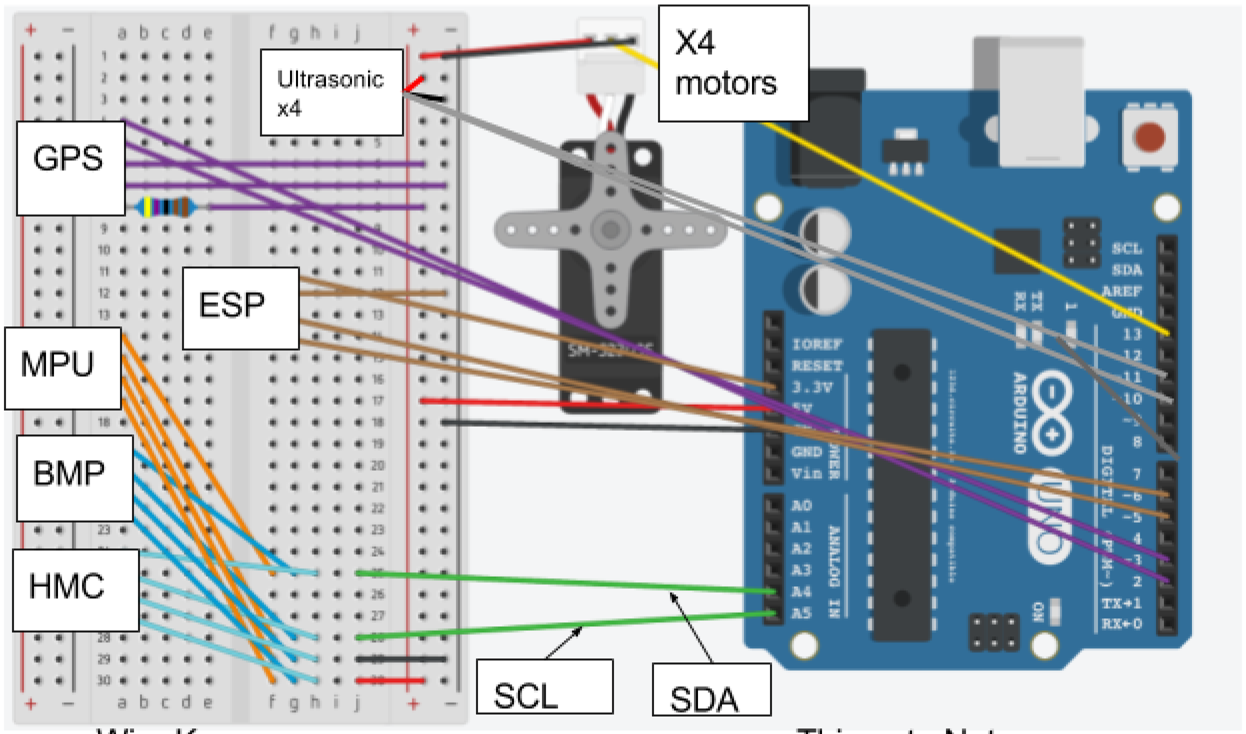
GPS - Adafruit Ultimate GPS module, this is used to give the drone and idea of its geographical location

ESP8266 - This module is a cheap and effective way to create an access point for the drone to connect to the internet, so the drones can communicate with us and with a main base to share information

Arduino - The microcontroller that controls the motors and receives the sensor readings from nearly all sensor modules except for the GPS(this is due to a slow refresh rate that would severely slow down the refresh rate of the Arduino)

Raspberry Pi - A microprocessor that controls the higher level functions such as Kalman filtering, GPS route navigation/route planning, machine learning, and more. It will talk to the Arduino through a software serial port.

Schematic



Wire Key:
Purple - Adafruit Ultimate GPS
Orange - MPU6050
Blue - BMP080/185
Turqoise - HMC5883L
Yellow - Motor Signal Wires
Grey - Ultrasonic Sensor

Things to Note:
-The GPS and ESP arduino Pins are serial connections.

-Both the Trig and Echo pin on the ultrasonic can be digital pins

** THE ESP MAY NEED AN EXTERNAL POWER SOURCE**
IT SOMETIMES CAN DRAW TOO MUCH CURRENT

This schematic is mainly for an overview of what modules we are using and how many arduino pins we need. Look at each module’s page for more specific pin outs and wiring. Pin outs for each module are very easy to find with a quick google search.

Control Flow of One Frame

1. **IMU Update**
    a. Read accelerometer + gyro readings
    b. Perform integral and vector calculations
    c. Perform complementary filter and predict new state
2. **PID Update**
    a. Read new state from IMU
    b. Calculate error from desired state for each axis
    c. Apply PID equations to error and calculate motor adjust
    d. Calculate proximity to nearby objects ← ***Handled by Obstacle Avoidance Systems***
    e. Turn proximity reading into obstacle avoidance adjust
    f. Combine motor adjust and obstacle avoidance adjust
    g. Send adjust to Motor Controller
3. **Motor Controller Update**
    a. Receive adjust from PID controller
    b. Map each adjust to each motor
    c. If given command; calibrate motors
4. **GPS Update(Occurs on a timer to avoid slowing down the system)**
    a. Clear GPS Serial(as old data builds up whether we want it or not)
    b. Read NMEA sentences and parse them to get lat and lon
    c. If given a point, calculate its distance and heading from the drone

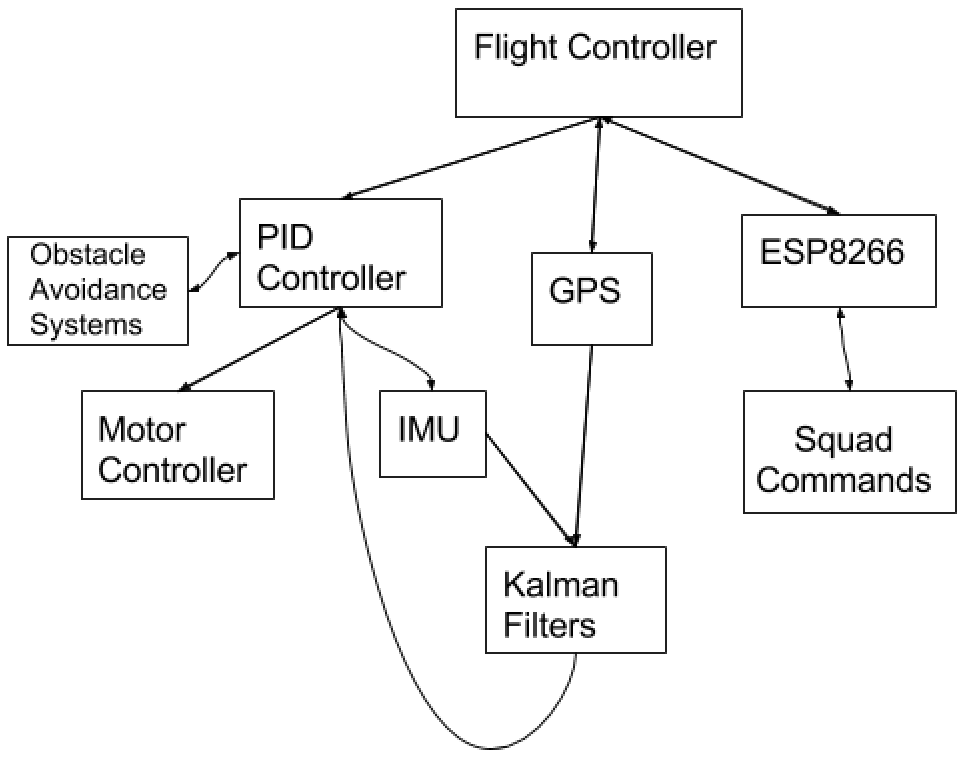
4.5 Kalman Filter Update(very complicated)
    a. Calculate the uncertainties of location estimate(based on previous readings and predictions) and new measurement
    b. Combine both values in a weighted average based off of uncertainties to create a new estimate of its location.
    c. Based on external movement make prediction of where it will be in the next time step and update state based on prediction

5. Flight Controller Update
    a. Calculates if it is close enough to its designated GPS location
    b. If not it will have the GPS calculate the distance and heading
    c. The flight Controller will calculate the adjust for each axis in order to move in that direction
    d. The flight Controller sends the new shifted desired states to the PIDs who in turn resets the integral and derivative terms

e. If the motors aren't working efficiently the flight controller will signal a re-calibration of the motors

END FRAME AND CONTINUE FROM THE TOP AGAIN

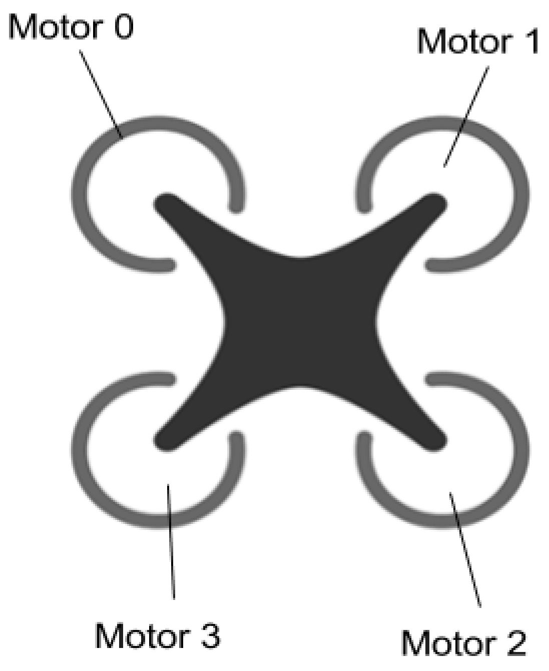
<u>Figure of System Hierarchy</u>



<u>**Note that since the kalman filters influence the state estimate that they influence the error calculations of the PID controller, and they affect the integral terms of the IMU; hence the feedback loop.**</u>

How we define motors and axises

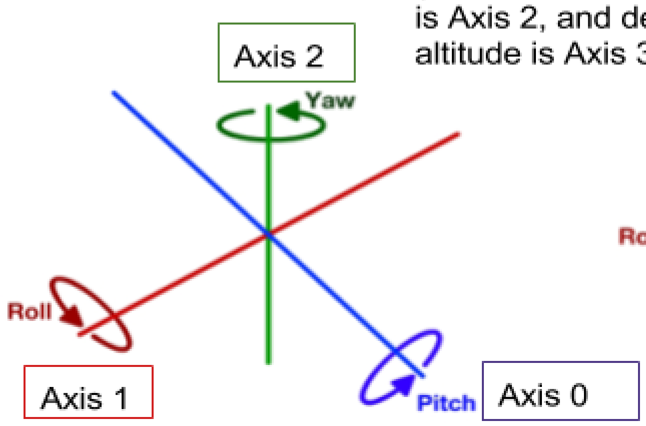
These numbers are used when these motors or axises are used in the code

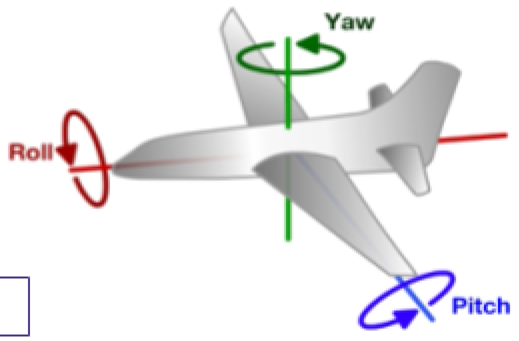


We start with the front left motor and call that motor: motor 0.

Then we continue clockwise around the drone, so front right is 1, back right is 2, and back left is 3

We start with pitch and label it: Axis 0, and then go down the list of axises. So Roll is Axis 1, Yaw is Axis 2, and despite not being in the picture: altitude is Axis 3





Inertial Measurement Unit

(IMU)

Description: This part of the drone measures the drones current rotations/states along the pitch, roll, and yaw axis. It also determines current altitude and heading with respect to north(directly facing north == 0). It uses an accelerometer and gyroscope readings combined in a complementary filter to estimate the current states. A barometer is used for altitude detection. A magnetometer is used for detecting heading in relation to north.

Extended Description(with lots of math!)**:** First off I just want to say that many of the constants you see in the code are simply from the data sheet of the part. Any constants in the PID equations are derived from experimental observation instead of rigorous mathematical proofs so don't feel intimidated. The constants simply turn accelerometer readings into actual measurements like degrees of rotation per second. The complementary filter is essentially a weighted average between the guesses at the current state of the accelerometer and the gyroscope. We weight the gyroscopes readings about 99% more. This is because the accelerometer is more susceptible to vibrations than the gyroscope when it comes to taking readings while vibrations are present. The gyroscope also sums all previous state estimates together to predict its current state so a couple bad readings don't affect the overall guess so much. So why do we need the accelerometer? That is because the gyroscope is susceptible to "drift", meaning that all of its minor bad readings will eventually over time effect its guess at our state and become large enough that we can't ignore it. Here is where the accelerometer comes in; when the quadcopter has a relative moment of stillness the accelerometers readings will become quite accurate. So with the gyro being slightly off, but the accelerometer being accurate, the weighted average will lean slightly toward the real angle. Now since the gyroscope is affected by the previous state estimates it will now read a bit closer to the real angle as well, and so the next time the weighted average is taken(assuming the accelerometer is still being accurate because of the relative stillness) it will get a little bit closer to the actual angle again, and so on. This way we get the best of both worlds, a integral system that can be robust when getting bad readings every once and awhile, and also a system that can correct its own drift. The gyro gets its guess at the current state by reading how fast it is moving in a certain direction, multiplying that by the time since the last measurement(which assumes constant velocity in that time, which is a large cause of drift I believe) and adds that to the previous state estimate. The accelerometer gets its guess at the state estimate from reading its accelerometer readings in the x and y directions, then using the inverse tangent of their ratios can calculate the angle that the accelerometer is off the ground. The accelerometer also has one more added benefit; it is what we use to calculate the initial angles of the drone up start up. This is because the drone will be motionless so the accelerometers readings will be very accurate and also because the gyroscope has no way of telling what angle it is at, only how fast it is moving in one direction and it has no previous states to draw from for it to predict its current state. So on startup the initial state is estimated from the accelerometers. The final part of the math to talk about is the pitch and roll transfer during yaw rotation. So say the drone is pitched forward and rotates along the yaw direction. Well we have no way with just the gyroscope and accelerometer to measure initial yaw so therefore the gyroscope never really knows where it is, just that is is rotating along the yaw direction. Say it rotates 90 degrees, well technically the drone is no longer tilted in the pitch direction, but in the roll direction. So we must add in code, that when the drone is rotating in the yaw direction, to transfer some of the roll and pitch between each other. If the actual transfer is plotted the transfer takes on a sin wave so we can use the built in sin function to accomplish this. We can do this in two lines:

```
1.) anglePitch += angleRoll * sin(gyroZ * 0.000001066);
2.) angleRoll -= anglePitch * sin(gyroZ * 0.000001066);
```

Now what are we doing here? Well let us take an example, so first off let me state that the constant just turns the output of the gyro into radians as that is what the sin function needs. Let's say we start 45 degrees rotated forward in the pitch direction. So we feed the amount, in radians, the drone has moved in the yaw direction(Z) since the last time step to calculate how much we must transfer. Now roll is initially zero so pitch is unaffected, but Roll is subtracted from and becomes negative! What? Well if you look at what we consider a positive roll direction you will see it is in the opposite of the positive yaw direction, so when we rotate along the yaw axis in a positive direction the roll is becoming negative, so we are essentially multiplying our transfer amount by negative one when we subtract. Now the next update the pitch has some of the roll added to it, but the roll is negative, so it essentially lowers the pitch value which is what we wanted because of the yaw rotation and the roll is further subtracted from but at a smaller amount because of the decrease in pitch. Eventually pitch will go to zero, and roll will take on all of the Pitch's value when we rotate 90 degrees. What if we kept going? Well, we first add some the roll to pitch, but roll is negative and the pitch is zero, so the pitch becomes negative! This is because we are starting to rotate to a point where we are 180 degrees flipped, and that means a negative pitch in relation to our initial pitch. And when we subtract the negative pitch to the negative roll it's like adding a positive and has the effect of lessening roll. Now if you continue with this line of logic you see that it will eventually return to its initial state once we have rotated 360 degrees and that no matter the state of the drones roll and pitch, these two lines will accurately predict the transfer between the two due to the circular properties of trigonometric functions such as sin. See support materials for another explanation of the math in this explanation, but this explanation goes a bit more in detail of why we use the math than the video does.

Hardware:

Gyro/Accel -	MPU6050
Barometer -	BMP085/BMP180, either will work with this code
Magnetometer -	HMC5883L

Pinouts(also in code):

MPU6050 PINOUT(I2C):

VCC - 5V
GND - GND
SDA - A4
SCL - A5

BMP085/BMP180 PINOUT:

VCC - 5V
GND - GND
SDA - A4
SCL - A5
3v3 - 3.3 volts out, can be used like an additional 3.3 volt supply
Don't worry about the other on board pins

HMC5883L PINOUT(I2C):

VCC - 5V
GND - GND
SDA - A4
SCL - A5
DRDY - Use only for faster data reading than 100 times a second
3v3 - 3.3 volts out, can be used like an additional 3.3 volt supply

Code: Under github repository quadcopter, look for IMU zip. All code is commented

Test Code: Same repository, but instead look at the IMU.ino file. The ino file allows you to wire up an IMU and read its outputs to the Serial monitor(small edits could be made for an LCD screen if you want to use one). Moving around the IMU unit will allow you to see how accurate its measurements are, how bad the drift is, if the transfer during yaw rotation is happening correctly, and whether or not you need to change the ratios of the complementary filter. If there is significant drift then the accelerometer should be given more weight in the complementary filter, although too much weight can cause the drone's angles to start to vary a bit wildly. A balance needs to be found for ideal state readings and estimates from the IMU

Support Materials: https://www.youtube.com/watch?v=4BoIE8YQwM8&t=1s
https://www.youtube.com/watch?v=j-kE0AMEWy4&t=261ss
^ IMU

https://www.youtube.com/watch?v=dZZynJLmTn8
^LCD tutorial for testing

Motor Controllers

Description: The motor controllers are the part of the drone that set and monitor the speed of the motors. This system is told what speeds to adjust the motors by the PID controllers in order for the drone to maintain a stable state and not wobble all around. It is optional but possible to have this system interface with the accelerometer of the IMU to calculate offsets for each motor that when applied keep all the motors running at relatively the same speed as to make the drone more stable.

Extended Description: It is important to keep track of the current motor speeds because of how this system interacts with the PIDs. The PIDs determine if the drone is not stable, say drifting slightly to the right or something, and will predict what motors need to change their speed in order to fix this. So if the PIDs tell the motor controller what to adjust the speeds by, and the motor controller doesn't know what the current speeds are, how does he know the total new speed to write the motors? The ESC's(electronic speed controllers) need one signal that tells it the total speed to run at, the motor controllers can't simply tell the ESCs the adjust and expect it to work, it needs to add that to the original signal then send it to the ESCs to adjust the speed. This is why it keeps track of the current motor speeds. For the calibration it works assuming there is at least a close to linear relationship between voltage signal and motor speeds. The drone is set up on the ground, and lands if it was flying, then slowly writes higher and higher voltages to motor one until it turns on, it can tell it turns on when the accelerometer readings fluctuate wildly due to vibrations caused by the motor spinning. It then continues this for each motor. Then it takes the highest voltage it took to turn on any one motor and subtracts each motor's start up voltage from it and multiplies it by negative one to get their offsets. We do this because when the motor controller sends out the start up voltage signal it will send out the highest start up voltage to ensure all motors turn on. Yet if we do this, the motors that turn on earlier will be at higher speeds, so we add the respective offsets to each individual motor's signals so that they too are just receiving their respective start up voltages and all motors will at least roughly be running at the same speed. Finally, in the code you may notice two odd things. We use the arduino Servo class to model our ESC's, this is because the ESC's take signals identical to those of continuous rotation Servo's(that is they take square waves) which the ESC's turn into a sin wave which the AC motors need to work. We use writeMicroSeconds instead of just write for one reason. Write can only go between 0 and 180, while writeMicroSeconds can go between 0 and over 2000 with a normal servo working anywhere in the ranges of 700 to 2300. The signals still equate to the same output speed of the motor, but writeMicroSeconds has a larger range and therefore a more precise amount of voltages that we can write to our motors for tighter speed control.

Hardware:

Motors x 4: https://hobbyking.com/en_us/turnigy-multistar-2216-800kv-14pole-multi-rotor-outrunner-v2.html

1. ESCs x 4: https://hobbyking.com/en_us/hobby-king-20a-esc-3a-ubec.html

Pinouts: To be determined, but each motor only needs one pin on the arduino

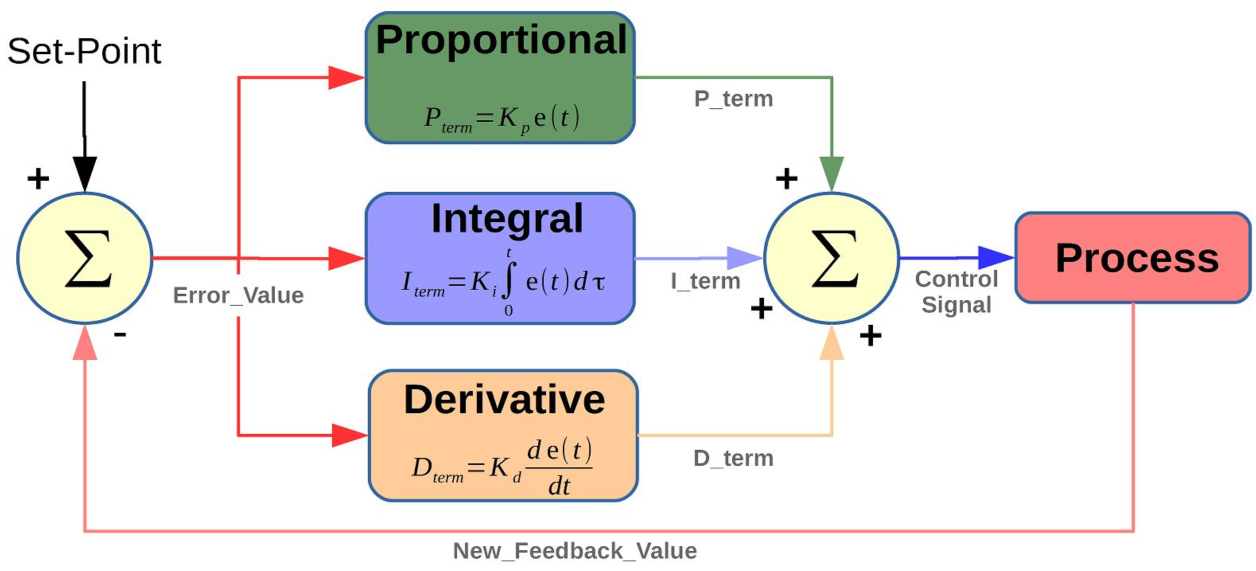
Code: Under the quadcopter repository, look for motorController.zip which contains the library for the motorController class. All code is commented

Test Code: Both the motorController library and quadcopterMotorController.ino file have a function called motor test to test if the motors are working and for the user to see around what signal strengths turn on the motors. The .ino file would be easier to use though for testing purposes.

Support Materials: http://www.instructables.com/id/ESC-Programming-on-Arduino-Hobbyking-ESC/

PID Controller

(Proportional Integral Derivative Controller)



Description: The PID controller is the part of the drone that works directly with both the IMU and the motor controller to try and keep the drone stable in a certain “state”. By state we mean we tell it a certain rotation amount on each of the axises and tell it to stay there. Without all of the PID equations our drone would be far more unstable in flight and it even helps us maintain a relatively straight flight path. It utilizes some fancy equations that watching the first video of the support materials section will give you a general understanding of why we use them but not what the equations are, or you can read the extended description.

Extended Description: The proportional part of the equations is the easiest to understand; the proportional term simply is a constant we multiply by how far off we are from our desired state. So the greater correction we apply in response to the greater we are straying from our desired state. Yet if proportional will get us back why do we need the other equations? Every term in the set of PID equations is crucial to balancing the drone. Just having the proportional part means it can wobble quite drastically. Imagine a spring the more you stretch the spring the more the restoring force will try and correct itself. However when the spring returns to its equilibrium state all the built up momentum isn’t countered and it continues in the other direction. This causes wobbles as it has to correct itself again and again until friction slows it to a stop. Now imagine a drone in a windy location, there are constant forces acting on the drone, and relying on the proportional part alone will make us a bit wobbly. The integral tries to prevent these wobbles by “solving” what amount of correction force is needed to counteract the constant force the winds are putting on our drone. It will see over time that the drone is over correcting due to the constant changing of state. The integral slowly but surely picks some amount of correction factor, like the proportional component does, then looks at how the state is changing. Over time it slowly but surely hones in on the correct amount of correction factor to get the drone steady and counteract the constant force on the drone. So to compare them, the integral first deduces if a constant correction force is needed from looking at error over time. It then slowly figures out what this constant force should be to perfectly balance out any force on the drone. The proportional term simply acts when acted upon, and does not use any prior memory of drone states, it only acts in proportion to how big the error is from the desired state and cannot tell whether its correction factor is truly being effective or causing some the wobbles it’s trying to correct. So the integral essentially solves the problem Think of driving a your car, and you want to drive to some stop lights and the closer you get the less you press on the gas(this is proportional), but we know this isn’t enough to stop a car because of built up momentum so you overshoot the lights. Then you go in reverse and do the same thing of slowly lifting off the gas, and over shoot again. These are the wobbles that occur from purely proportional control, the integral term sees these wobbles and tells you to use the brakes a bit more every time until you learn over time from your past experiences the best way to apply the brakes to stop correctly at the lights every time. Since the integral term is built from previous knowledge getting one bad reaction(say the car slipping on ice) won't completely change your understanding of where to apply the brakes, and you can be confident the integral is honing in on the right output. The only problem is that the integral portion of the output is very slow and takes a bit to hone in as each new reading only affects it a tiny bit, the solution to help with a fast changing state where its obvious the proportional and integral aren’t correcting fast enough is the derivative.The derivative term prevents undercorrection. An example of undercorrection being if a serious wind comes and our proportional term wasn’t designed to handle it our drone would fall out of the sky, but the derivative can read the drastic change in state and ramp up the correction factor. If the derivative and integral are so smart than how come we need the proportional at all? Great question, once the integral is very close to the desired state, it begins to work terribly terribly slow as it wants to hone in on the exact correction factor and doesn’t want to overstep the solution. So essentially it is going so slow that the error from the desired state is remaining constant, and what is the derivative of a constant? Zero. That means our derivative term won’t help us correct our small offset. Yet do we know a term that will always mindlessly correct any amount of error? Yes, its the proportional term! Together they combine to make a very stable self correcting system. So if I want my drone to stay still I tell the PIDs the desired state is zero pitch, zero roll, zero change in altitude and a yaw lined up with North(0 degrees heading) and the PIDs then reads the IMU to determine what its current state is, it calculates the error of the current state in relation to the desired state and applies the PID equations and sends the output to the motors to ensure that the proper adjustment is undergone to return to the desired state. The PID also splits up the adjustment between all motors to ensure they all see the same average amount of wear and tear. An example is say the drone is supposed to have zero pitch, but it is slightly rotating forward in the pitch direction. So the PID will read this from the IMU and calculate the proper adjustment in output. It will then split this adjustment in half, and feed half to the front two motors and the other half(but of opposite sign) to the back motors. Why opposite signs? As to adjust this specific rotation we would need either the front motors to have a higher output, or the back motors to have a lower output, we simply take the needed output, divide it in half and do both. We have the front motors increase their output and the back decrease there's by half the desired output so that the resultant output is equivalent of what the PID calculated we would need. This not only spreads the burden of adjustment around the motors, but also decreases range of speeds that the motors need to run at because of halving their individual adjustments avoiding running the motors at high or low speeds that may be less efficient than their rated speeds. We can't actually do integrals or derivatives on the arduino so we instead simulate them. For integrals we imitate riemann sums and rely on the fast update rate to secure some bit of accuracy with that approximation. The equation is simply the last integral term added with the timer interval since the last measurement multiplied by the integral constant and the error. The derivative is simply the instantaneous rate of change, so once again we rely on the fast refresh rate to get us some bit of accuracy here. We simply take the error minus the last error all multiplied by the derivative constant and then divide the result by the time since the last measurement. When the PID is given a new desired state, we reset the built up integral and we reset the derivative terms so that a change of base point doesn't make it look like we had a sudden change of movement off of the desired state which would cause a massive over correction from the derivative terms, and if we didn't reset the integral terms it could take awhile for this shift to take effect with how much we have built up the integral term.

Hardware: See IMU and Motor Controllers

Pinouts: See IMU and Motor Controllers

Code: See the PID.zip on the github library, it is a library for the PID class which has dependaices for the motorController and IMU libraries.

Test Code: The PIDsketch.ino is a file that has similar functions except for a few extra that tell you specifically what axis they are adjusting instead of having to enter in the axis number. These functions all use the serial monitor to show you the outputs they are calculating for the motors, so you can choose to hook them up to motors or just read the outputs to see if it is calculating them correctly. For readability and testability short delays are added into the code.

The PID simulator code, written in processing, in the simulations folder on the github can also be used to test the performance of the PID controller on a rigid body by varying the PID constants and forces acting on the body.

Support materials: https://www.youtube.com/watch?v=0vqWyramGy8&t=135s
https://www.youtube.com/watch?v=JEpWlTl95Tw
https://www.youtube.com/watch?v=XfAt6hNV8XM&t=29s

Obstacle Avoidance Systems

Description: The job of the obstacle avoidance systems is right in its name. It avoids obstacles. The drone has four ultrasonic sensors that work similar to how a bat would use echolocation to determine its proximity to nearby objects. The drone has certain tolerances in how close it can get and when it gets too close to a certain object it will look like the drone is bumped backwards by an invisible force. These systems also help prevent the drone from colliding with its neighbors when flying in formation. The four ultrasonic sensors look in front, back, left, and right of the drone. There is nothing on the top or bottom of the drone that allows the drone to avoid collisions in those directions.

Extended Description: The obstacle avoider is pretty easy to set up, all it needs is the distance it should start obstacle avoidance at, the pins of it's four sensors, the distance at which it should be spitting out it's max adjust value, and the maximum outputs it's allowed to spit out. The last parameter is optional and if not put in will assume that a duty cycle of 100% is the maximum it is allowed to spit out. The obstacle avoider takes the distance reading of an object on the scale of the maxDistance(the distance it starts adjustments at) and the minDistance(the closest an object is allowed to be) and maps that onto the scale of 0 to the maximum output force it is allowed. Several error catching statements are used when the occasional NaN value comes out if there ever is a divide by zero error within the map function call which turns out output forces to zero. The units of distance work in centimeters and assume the air temp is at room temp. In the future we may tap this into a temperature sensor that can adjust the distance calculation based off of the air temp.

Hardware: Four HC-SR04 Ultrasonic Sensors

Pinouts:

HC-SR04:

VCC - 5v
Trig - *Any digital Pin(TBD)
Echo - *Any digital Pin(TBD)
GND - GND

Code: See oAvoider zip in the quadcopters gitihub

Test Code: See oAvoider.ino file in the github, it is made to take four sensors and spits out their distance readings and their mapped force adjust values to the serial monitors. Use it to see if they are reading/calculating values correctly.

See the PID simulator and the Swarm Simulations code in the simulations folder on the github. Each has a version of the obstacle avoiding algorithms programmed into the rigid body drones. There you can play with different mapping functions and min/max distances.

Support Material: http://www.instructables.com/id/Simple-Arduino-and-HC-SR04-Example/

Flight Controller + GPS

(The code for the GPS wasn't enough to warrant its own library)

Description: The flight controller handles all of the high level stuff like keeping track of GPS location, telling the drone where it needs to go, and how it needs to get there. The flight controller will also be in charge of monitoring the data flow of the ESP8266 and making sure it's running as an internet access point.

Extended Description: The flight controller uses something called the shifting base point algorithm to accomplish movement. The base point is essentially the state that the PID controller tries to keep the drone at default parallel to the ground and stable. Yet if we want to move to a new location the drone needs to rotate in a specific direction, so what we do is "shift the basepoint" or basically the flightController figures out what direction and how far we need to rotate in a certain direction by using position vectors + our heading in relation to the desired point then tells the PIDs that the new base point is whatever rotation we calculated. The PIDs have no clue that will actually move us, just that they must keep us rotated like that. The result is stable and clean movement in the desired direction. The flight controller first takes in the desired location to move to, then it calculates is latitude difference and longitude difference with the desired point. It then uses the inverse tan of the x and y ratio to calculate the heading of the point in relation to the drone's current heading and calculates the magnitude of the distance. With this information it multiplies the max adjustment it is allowed to do by the sin of the angle to calculate its adjustment on the pitch axis, then it does the same calculation with the cos of the angle to calculate the roll adjustment, it then feeds these adjustments into the PIDs which then relay what motor outputs need to be changed in order to reach this state to the motor controllers and the movement is achieved. Each refresh of the flight controller updates this calculation if the drone's path has slightly strayed so that the drone is constantly adjusting and honing in on the correct location. The drone has a tolerance between 1 and 3 meters that once it gets within that distance from the point it will stop moving, but if say a breeze drifts it too far away it will return to that location.

Hardware: GPS - Adafruit Ultimate GPS Module

Pinouts:

Adafruit Ultimate GPS:

2 - Rx
3 - Tx
EN - Enable pin, pull high to 5 volts with a 10k resistor
VBAT - Battery backup, currently not used
Fix - Used to determine if the GPS has a fix or not, currently not used
Vin - 3 to 5 volts(clean power supply is important)
GND - GND
PPS - Used to synch up with the GPS if necessary, not used

Code: See flight Controller.ino on the github repository for the quadcopter

Test Code: The adafruitGPS.ino file allows you to test the accuracy of the GPS readings and the accuracy of the calculations of the positional vectors related to the drone to any point you enter. It uses the serial monitor to print out these calculations

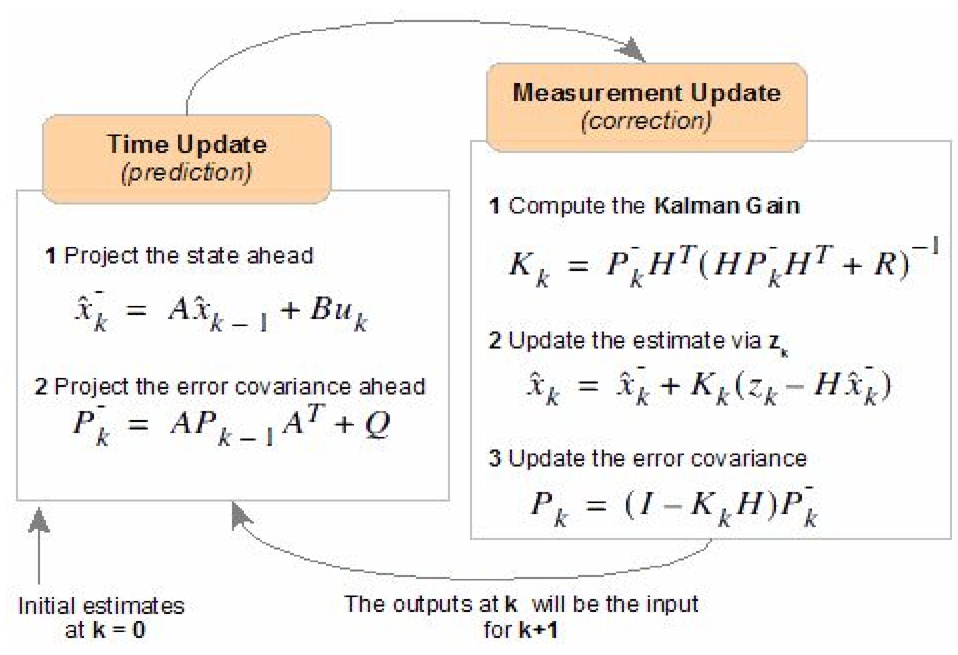
Support Materials: https://learn.adafruit.com/adafruit-ultimate-gps/pinouts
https://www.youtube.com/watch?v=QjgkfY8-gxM

Kalman Filter

Warning, at first glance this part of the drone may be overwhelming

Description: The kalman filter is an “ideal estimator”. It looks at a sensors readings and compares it with what the filter believes should be the right reading based off of previous values, and tries to make the best guess of the true reading. For example if our GPS keeps telling a robot that it’s at a point (3,0) for a long time, our robot will become quite confident that it is actually at (3,0); so what happens if we have a sudden GPS reading spike to (-15, 0) out of nowhere when our robot didn’t try and move anywhere. Our robot will then say I was quite convinced that I was at (3,0) and so the reading is probably some random spike, but I may have moved a slight bit due to wind or something. The robot then looks at the reliability of the sensor, if the sensor is known to be very random sometimes the robot will only adjust its estimate of where it is to maybe between (1.5,0) and (2.5,0) yet if the sensor is known to be rock solid in its measurements than the robot will concede and change its estimate to somewhere far closer to sensor’s readings like (-14, 0) to (-11,0) depending on its reliability. If the robot isn’t so confident in its own estimate yet the sensor isn’t the most reliable either it will pick somewhere in between, and over time look at the sensor reading patterns to refine its estimate to one it can be quite confident about.

Extended Description: I will explain the run through of a one dimensional kalman filter then two dimensional and how they work under the influence of control inputs(the movement the drone is telling the motors to go through). First here is a picture of the equations we will dissect:


Time Update
(prediction)
1 Project the state ahead
$\hat{x}^-_k = A\hat{x}_{k-1} + Bu_k$
2 Project the error covariance ahead
$P^-_k = AP_{k-1}A^T + Q$
Measurement Update
(correction)
1 Compute the Kalman Gain
$K_k = P^-_k H^T(HP^-_k H^T + R)^{-1}$
2 Update the estimate via z_k
$\hat{x}_k = \hat{x}^-_k + K_k(z_k - H\hat{x}^-_k)$
3 Update the error covariance
$P_k = (I - K_k H)P^-_k$
Initial estimates at k = 0
The outputs at k will be the input for k+1

Each variable here are N dimensional matrices, and everyone you have to design yourself based off of a physics model you create of the relationships between all of your state variables. Sounds intimidating, but let's start with a simple problem. I have a GPS sensor, and I know it's not 100% reliable but im also not 100% sure of where I am but I do have a general idea. So who do I trust, my sensor or my estimation? Well you may think, let's take an average of the two. A straight average has a few issues, what if the sensor is more unreliable than my estimate? If my sensor readings are telling me I'm on the other side of the earth, than my average will put me halfway around the world. Not good. So instead let's do a weighted average based off their uncertainty(also called error), the more uncertain, the less we put weight into its readings. The kalman gain is a measure of that, it measures how much of our estimates uncertainty is a percentage of the total uncertainty(our estimates uncertainty and our sensor uncertainty). Lets throw in some numbers now, say that I believe I am standing 2 meters down a path, and I get a reading from my GPS that says 2.5 meters. Well to start my calculations I need to know the uncertainty of each estimate(measurements and state estimates will be called estimates now), so how do I find this? Well the uncertainty usually comes from the variance of its readings, so if I plot the readings and measure the variance I will have my uncertainty for the sensor. Now the uncertainty for my estimate is simply based on the variance of how far I think I am off with my guess. So I say I could be on average 2 meters off(which is my standard deviation) so my error is 4, and let's say my sensor's variance is 2. Ok now let's calculate the kalman gain: It's simply my error divided by the sum of my error and the sensor error, so 4 / 6 = .667. Or basically my error accounts for two thirds of all the known errors in the overall estimate. Now what do we do with the kalman gain? Essentially we use it to tell us how much of the sensor to read, let's test some base cases. Say my sensors uncertainty was 0, then my reading would account for 100% of the error and basically I should always trust my sensor. What if my estimate had no uncertainty, well then the kalman gain would be 0 and I should ignore my sensor because I know where I'm at! If the kalman gain is .5 then we pick the middle of my estimate and the sensor reading to be our new estimate. Let's see the equation that satisfies this: $x_k = x_k + K(y - x_k)$. Where X_k is the current state estimate, K is the kalman gain, and y is the measurement. Let's dissect this, so the y - X_k is simply seeing the difference between our estimate and the sensor. It would make some intuitive sense to add that to the current state as we would be shifting our estimate to the measurement readings. Sadly know, we run into our problem of "can we trust the sensor"? That is why we multiply the kalman gain by this, we are saying how much should we shift our estimate toward the measurement. If the kalman gain is 1, our sensor is always right, then we fully shit over our estimate to the sensor reading. If our kalman gain is zero and we are confident in our estimate than we don't shift over at all as the zero nullifies the y - X_k term. So for us: X_k = 2 + .667(2.5 - 2) is what our

equation would look like, and running the numbers we now think we are at 2.334. We see this is a bit closer to our reading than our initial estimate. Great, now one more part to updating our estimate whenever we get a new measurement. Whenever we get more information, we should be getting more confident in our estimate of the true state. Makes sense, if over time we see more and more sensor readings hovering around a single point then we can become more and more confident that point is our true state, even if we never quite hit that point exactly with our sensor readings. Now be careful, the sensor's uncertainty never changes as it will still have the same noise in its readings no matter how sure we are of where we are at, it is only our estimate that we become more sure of. As a quick aside, we must go over how to merge "gaussian distributions", or more commonly known as bell curves, as that is what we are actually doing with these equations. A bell curve has a unique property. Say we have a sensor whose readings follow a bell curve pattern when graphed, this means that 66% of its readings fall within one standard distribution of the mean. Anything farther than 3 standard deviations of the man has a change of practically zero chance of occurring. Having a higher standard deviation means a longer range of values we will normally see in that distribution. Essentially we assume that our uncertainty of where we are is shaped like a gaussian because we are pretty sure we would know where we are, and are quite sure that guess that put us say a mile or even a continent away don't make sense. The sensor readings are assumed to be gaussian and most sensors are. The filter will still work even if the noise isn't gaussian but won't be as effective I have found out in personal research. Nevertheless it is still useful in refining our estimate even without Gaussian distributions. When combining gaussians the mean is somewhere between the two, and is closer the one skinnier Gaussian. Why? That is because a skinnier Gaussian means those values fall within a tighter range and can be relied upon more. Sounds similar to the reasoning behind the first equation we did right? Now how about the variance of the new Gaussian? Well it ends up being both variances multiplied together, divided by their sum. This always leads you to a smaller variance in the resultant gaussian. Now let's setup the equation:
$New\ Error\ =\ (Estimate\ Error\ *\ Sensor\ Error)\ /\ (Estimate\ Error\ +\ Sensor\ Error)$ What happens if I factor out our estimate Error on the right side:
$Estimate\ Error\ *\ (Sensor\ Error\ /\ (Estimate\ Error\ +\ Sensor\ Error))$ Does that right part inside the parenthesis look familiar? It should it's basically the kalman gain, if we instead measured what percentage of the error the sensor error made up. Since percentages add up to one, we know that this is equivalent to one minus the kalman gain. So we can say our new estimate is: $(1\ -\ K)\ *\ Estimate\ Error$, and that explains the second equation in the measurement update set. For us it would look like:
$New\ Estimate\ Error\ =\ (1\ -\ .667)\ *\ 4$, which comes out to be 1.332. No it looks like our estimate is getting a bit more accurate. Now we could keep doing this and adding in

new measurements until we get a very precise guess, but let's spice things up. Let's say that I am moving along the path at a speed of 1 meters per second. That means the true state is changing and therefore my estimate should change too, so this is when we enter the prediction set of equations. The only thing we have to cover before we delve into these equations is talk about how the covariance matrix and all of our uncertainties vary with time. If say I was say taking a golf shot, and I know I'm between 120 and 122 meters from the hole, I have a decent estimate of where I am, but what happens when I hit the ball? There's no way I can be 100% certain where I hit it, but I can guess I hit it between 40 and 50 meters. So depending on where I was before my ball could be anywhere from 82 to 70 meters from the hole. That's more than the range of the two separate uncertainties! So uncertainty builds up over time and the more actions I take. So when we predict ahead of time, our uncertainty from our previous estimate only adds to our prediction. Now with that in mind let's take a look at the first future prediction equation: $x_k = Ax_{k-1} + Bu$ where x_k is our future prediction, A is a physics model we will discuss in a second, x_k-1 is our current state estimate, u is control variables or basically external motion that we switch on and off, like motors or something and B is the physics model of how they affect the state. Multiplying the respective matrices together and adding them gives us our predicted future state. Yet we still have to update our compounded error, we do that with the equation: $A * P_{k-1} * A^T$. Here we multiply our past uncertainties by our physics model and its transpose, what this does is keep our covariance matrix symmetric and compounds the all the errors together in concurrence with our physics model, with regards to our example it adds the uncertainty of where the ball initially was to the uncertainty of how far we hit the ball. To be explicit, this step INCREASES the uncertainty. As long as our measurements come in enough we can combat this increasing uncertainty, if we go too long between our measurements than the uncertainty will go out of control. This should be taken into account when implementing a kalman filter onto the drone, and is why we try to keep the refresh rate of our sensor systems as fast as possible. As some final things to note, the key to a robust kalman filter is the covariance. Covariance helps us determine the relationship between the two variables, so if the error of our main variable's estimate is quite large and we see the covariance is quite large we can assume much of the uncertainty is coming from that variable it relates to so we adjust it and the more certain about our overall estimate the more we can be certain we know these accompanying variables because if they were still uncertain then we wouldn't be certain of our main estimate.

Code: See GPSKalmanFilter.py in the main github Quadcopter repository

Test Code: At the bottom of the python module is an if __name__ == "__main__" statement that contains basics test code for an example filter and can be changed to simulate several different conditions

Hardware: NONE - just takes in readings from things like the IMU and GPS, the computations will be done on a raspberry pi most likely

Pinouts: NONE ^see hardware just above

Support Material: https://www.youtube.com/watch?v=bm3cwEP2nUo

http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

https://home.wlu.edu/~levys/kalman_tutorial/

https://www.youtube.com/watch?v=CaCcOwJPytQ&list=PLX2gX-ftPVXU3oUFNATxGXY90AULiqnWT
https://www.youtube.com/watch?v=FkCT_LV9Syk

Formation Algorithms

Description: Our drones fly in squad formations. If we put a squad leader in charge of every group of drones and have him delegate the tasks to his squad mates, this limits the amount of drones our main server has to directly talk to. To make a squad more organized we have them fly in formation. Every squad can be given a special math function, unique to them if desired, that calculates their formation based on the number of drones in the squad. Whenever a new drone enters a squad it simply asks for its spot from the leader, who plugs in the number of the new member(EX: the fifth member is given number 5), and tells the new member his spot. Each member now only needs to remember its spot in relation to the leader, and not the whole shape. In fact not even the leader doesn't need to remember the whole shape, the function takes care of that when he inputs the new members number. Only the number of people in the squad need to be remembered. If designed right a drone formation function can expand to fit an infinite amount of drones in the squad.

Extended Description: Here I'll describe how are standard formation function works. I'll refer to it as the shell function. In order for it to work, two things need to be specified: the number of drones that will fit in the first shell and the first shell's radius in terms of a unit of distance from the leader. What it does is keep track of how many drones are in the current shell, and when a new drone enters the shell it maps its number on a scale of zero to the max number in the shell + 1(I'll explain why in a moment) to an angle of zero to two pi. Then using the angle and magnitude of the shell radius, it can turn this from polar coordinates to x and y coordinates the drone's PID controller understands. The reason we do zero to the max plus one is that when the final drone enters the shell, if our scale was o to max, then max would map to two pi which is the same as the angle zero; the same angle the first drone in the shell would get, and so the drones would overlap. Adding the plus one prevents this. When the shell is full, the function calculates the new shell radius by just adding the first shell's radius to itself again(so each shell's radius is a multiple of the first shell's radius). To calculate the new max that can fit in that shell we need to take the ratio of the circumferences of the shells; since we know the two radiuses of the shells, which is the only thing that has changed between the shells, we can just take the ratio of their radiuses and multiply that by the old max to get the new max(rounded down). For example, if the second shell was 40 meters wide and the first only 20, then the ratio of their radiuses is 2; so now we take 2 and multiply it by the original max number allowed in the first shell, let's say 8, to get the new max. Which would be 16 in this case. One problem that occurs is non-optimal spacing; for example when a shell is only half full the drones still only space themselves out in half the shell and we get an awkward half moon shape. The function addresses this by taking the current number in a shell, dividing 360 by that number and telling everyone to space out in multiples of that angle amount. This way even if a shell is only partially full, everyone in it spreads out to get the most coverage in that shell. All of these calculations are contained within the formation function. Any new custom formation functions must only keep track of the number of people in a squad, as that is the only thing passed into the function by the squad leader. If a function can be made to handle and work around this, it could be substituted for the squad leader's default shell algorithm. Remember that only the squad leader needs to know the formation algorithm, and each one can be given a different function tailored to a specific purpose.

Code: The main drone code for this is not yet written as the communication structure between the drones is still being fleshed out. For examples though check out the Swarm Simulations in the simulations folder on the github.

Test Code: See the Swarm Simulations in the simulations folder on the github. Here you can spawn drones to two different drone leaders to see how their formation functions differ slightly based on different starting parameters. The standard shell formation we have implemented only requires the radius of the first shell, and how many drones should fit in it. Any other formation algorithms really only need to adjust the variables that count how many are in the formation, and adjust the function that removes them from the squad.