# QuadQWOPter

Ken Katagiri, Jeffrey Lu, Lawrence Ng
EE 149 — Fall 2014— December 19, 2014

## Abstract

The goal of this project was to create a user-controllable quadcopter that would receive user reference variables for x, y, and z velocities via radio and respond by moving in the appropriate direction. The project incorporated two major concepts introduced in EE149: Modeling of physical dynamics and simulation strategies. To find reasonable values for quadcopter motor outputs, a simulation was crucial — with no simulation, the quadcopter would most likely fail to perform any kind of aerodynamic feet. In order to simulate the physical dynamics of the quadcopter, though, a mathematical model of the physical dynamics was needed. Once equations describing the state of the quadcopter over time was obtained, Matlab tools such as Simulink was applied to obtain theoretical evolution of the state of the quadcopter.

## Physical Dynamics and Model

In this project, we implemented the quadcopter configured to fly in the x-orientation, following the right-hand rule 3-dimensional coordinate system. Much of the mathematical derivations and physical assumptions used in this project were derived in parallel to Andrew Gibiansky's paper, "Quadcopter Dynamics, Simulation, and Control"; differences in quadcopter configuration and coordinate systems were incorporated into the original calculations performed in that paper. Such differences manifested itself in the equations of quadcopter torque, as well as the rotation matrix utilized to transform the derived quadcopter dynamics from the quadcopter body frame to the inertial frame. The final rederived equations successfully described the 9 state variables of interest as differential equations. The equations that describe the evolution of linear acceleration, angular velocity, and the time derivative of roll, pitch, and yaw follow:

$$\ddot{x} = \frac{-k_d}{m}\dot{x} + \frac{k}{m}(s_\phi s_\psi + c_\psi s_\theta c_\phi)(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2)$$

$$\ddot{y} = \frac{-k_d}{m}\dot{y} + \frac{k}{m}(-c_\psi s_\phi + s_\psi s_\theta c_\phi)(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2)$$

$$\ddot{z} = -g - \frac{k_d}{m}\dot{z} + \frac{k}{m}(c_\psi c_\phi)(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2)$$

$$\dot{\omega_x} = \frac{I_{yy} - I_{zz} w_y w_z + Lk(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2)}{I_{xx}}$$

$$\dot{\omega_y} = \frac{I_{zz} - I_{xx} w_x w_z + Lk(-\omega_1^2 + \omega_2^2 + \omega_3^2 - \omega_4^2)}{I_{yy}}$$

$$\dot{\omega_z} = \frac{I_{xx} - I_{yy} w_x w_y + b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2)}{I_{zz}}$$

$$\dot{\phi} = \omega_x + s_\phi t_\theta \omega_y + c_\phi t_\theta \omega_z$$

$$\dot{\theta} = c_\phi \omega_y - s_\phi \omega_z$$

$$\dot{\psi} = \frac{s_\phi}{c_\theta}\omega_y + \frac{c_\phi}{c_\theta}\omega_z$$

In the above equations, $k$, $k_d$, and $b$ are physical constants that represent forces that act upon the quadcopter. We were able to measure the $k$-value by measuring the minimum voltage required to have the quadcopter take off from an on-ground position. The drag coefficient, kd, was estimated by approximating the quadcopter as two overlapping rectangles. Finally, a rough estimate of $b$ was obtained by experimentally measuring the change in yaw while forcing the quadcopter into a horizontal orientation. Other constants such as $L$ (length from center of quadcopter to any of its propellers) and $m$ (mass of quadcopter) were easily measured through conventional measuring methods.

## Materials

We arbitrarily decided that we would like a medium sized quadcopter. We decided upon the HobbyKing X525 V3 glass fiber frame for 2 main reasons. The first was that HobbyKing had many of the required components in stock and reasonable shipping time.

The second was the cushion the shock absorbing landing legs provided. As we were novice pilots, this cushion prevented some damage to the frame from rough landings. We predicted a loaded weight of roughly 1.5 kg. Since the recommended upper voltage limit for the Arduino is 12V, we required a 3 cell 11.1V LiPo battery, an industry standard. 4 1200kv brushless outriders combined with 8 inch propellers at 11.1V was sufficient to not only provide enough thrust to hover, but also left us enough available thrust to ascend. Since the maximum current draw of each motor was listed at 17A, we opted to use 25A ESC to mitigate the likelihood of heat issues. Estimating a current draw of up to 10A per motor, we chose to purchase a 4000 mAh battery for a minimum flight time of 5 minutes.

## Controller

Based on the mathematical model of the quadcopters physical dynamics, Gibiansky provides derivations and equations for desired voltage outputs to obtain a constant height through a simple unity feedback mechanism while converging to a user-specified roll, pitch, and yaw. Again, by altering these derivations to suit our system, we were able to obtain appropriate controller equations for our system, as shown below:

$$\alpha = \frac{I_{xx}e_\phi}{Lk}$$
$$\beta = \frac{I_{yy}e_\theta}{Lk}$$
$$\gamma = \frac{I_{zz}e_\psi}{b}$$
$$\delta = \frac{(throttle+1)mg}{kc_\theta c_\phi}$$
$$motor1_voltage = \frac{-\alpha-\beta+\gamma+\delta}{4}$$
$$motor2_voltage = \frac{-\alpha+\beta-\gamma+\delta}{4}$$
$$motor3_voltage = \frac{\alpha+\beta+\gamma+\delta}{4}$$
$$motor4_voltage = \frac{\alpha-\beta-\gamma+\delta}{4}$$

Here, $e_\phi$, $e_\theta$, $e_\psi$ are the error in system angles passed through a standard PID controller. The proportional gain, derivative gain, and integral gain for each angle is the parameters fine-tuned during simulation to allow for flight without instability.

## Simulation

By modeling the mathematical equations of the plant and controller, a unity feedback Simulink model was created. Using this model, different values of proportional, derivative, and integral gains in each angle were simulated. Each iteration of simulation was improved upon by following guidelines given in Oscar Liangs article, Quadcopter PID Explained, and Tuning. After a number of iterations, a good combination of gain values was obtained, resulting in a theoretically stable, self-correcting quadcopter. These gains, as well as plots of the variables over simulated time, are shown below in response to an initial pitch and roll offset of 20 degrees and 30 degrees, respectively, and initial roll and pitch angular velocities of 70 degrees per second, and 60 degrees per second, respectively. These initial conditions were a rough upper bound on the angular parameters the quadcopter would be operating under, and correcting.

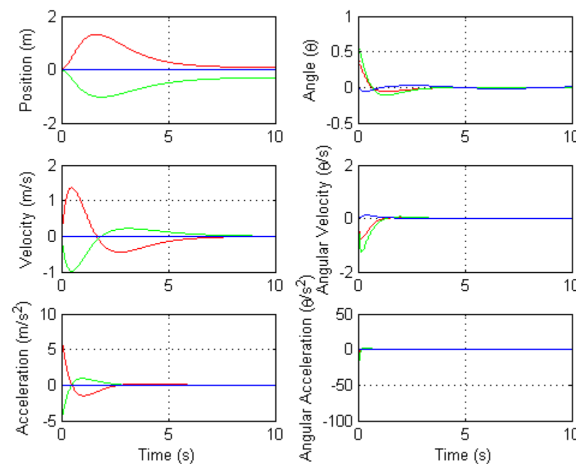


Figure 1: State Variable Values from Simulation

A link to the video representing a similar simulation can be found in the appendix.

# Implementation

For implementation of our quadcopter, we programmed the Arduino board with ease of testing and modification during the deployment phase as our priority. All state variables were made to global, so that we could peek into our system state, and not need to rely just on qualitative data. Also since we may like to test individual aspects of our quadcopter separately, we also wanted to be able to selectively turn on and off ISRs, as well as tune the frequencies of each ISR, and even add other ISRs with ease.

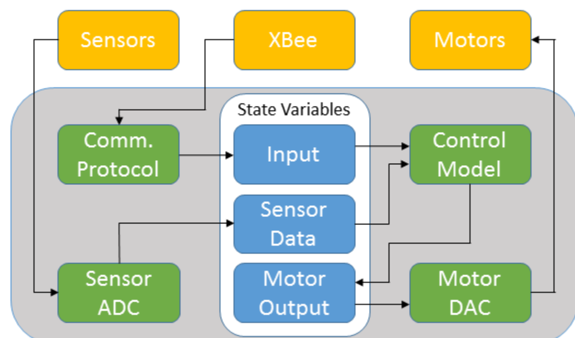


Figure 2: Overview of Quadcopter System

## Timing

Since the sensors and controllers needed to be updated at a fixed periodic rate in order to be useful, we needed to use a timed interrupt to coordinate timing. Using the Timer1 Arduino library, we set up a timed interrupt to occur every 4 milliseconds, which would then call an ISR that would take measure and interpret sensor readings, and calculate motor outputs using the derived control model. Our servo motors required a PPM signal, whose width also had to be timed, so we used the ServoTimer2 Arduino library to handle it.

As a safety net to prevent erroneous behavior from overlapping interrupt service routines, our timed ISR contained a static variable that would be set to true whenever the processor entered the service routine, and set back to false when it exited. This flag acted as lock, which would prevent further invocations of the same interrupt while it is being serviced, except rather than waiting for the previous ISR to complete, it would opt to drop the most recent interrupt completely. However, since even this behavior is suboptimal, we also made sure to keep track of the mean and maximum CPU utilization rates of the ISR, so that during testing, we could see how much of the CPU time the ISR was taking, and adjust the interrupt frequency if necessary.

## Sensor Measurements

To ensure accurate sensor readings, we needed to calibrate our sensors. We opted to calibrate by hand, rather than have the board calibrate sensors on startup, since sensor calibrations will not lose significant accuracy over time, and since we would like to be able to start our quadcopter in any arbitrary configuration, rather than necessitating our quadcopter to start with zero-degree angle offsets for pitch and yaw.

We used an affine sensor model to calibrate our sensors, where the reading is determined by the sensitivity and offset of the sensor. We found the offset experimentally, by taking the average sensor reading when each sensor value should be 0. For sensitivity, we were able to determine this experimentally as well, since the normal force exerted on the accelerometer when it is stationary is know to be 9.81 m/s$^2$. However, we did not have tools to measure the gyro sensitivity very accurately, so we used the value as reported by the IMU3000 data sheet, which was 131 bits per deg/s.

We also needed to reduce noise, so we implemented an exponential lowpass filter. The exponential lowpass filter worked by taking a new sensor measurement, $s_t$ and taking a weighted sum with the old filtered sensor value, $x_{t-1}$ with some weight $0 < \alpha < 1$, such that:

$$x_t = (1 - \alpha)x_{t-1} + \alpha s_t$$

Higher values of alpha would cause more recent measurements to be weighed more heavily, and thus cause the filtered sensor value to converge to changes in sensor readings much more quickly, while lower values will keep sensor readings stable, and cause it to be more resilient to noise. The exponential lowpass filter proved to be sufficient when we were filtering the accelerometer measurements, as the amount of noise even when the motors were on were small, and the time that the accelerometer took to converge to sudden changes in acceleration was only a fraction of a second.

However for the gyroscope, we could not find an $\alpha$ value that would allow the angular velocity readings to both be resilient to noise and converge quickly to changes in sensor readings while being resilient

enough to noise. We decided then to use a Kalman filter as implemented by Kristian Lauszus, which would try to predict what the next state will be, take measurements and compare them to the predicted state to determine its accuracy, and then adjust values accordingly.

### Communication

For the user to control the quadcopter, they would need to be able to communicate four input variables: desired velocities in the $x$, $y$, and $z$ directions, and yaw. However, we could not simply send all the values continuously in a sequence, since in a byte stream, it would be near impossible to determine where a command begins and where it ends. Also, we would like to ensure that the data has not been corrupted in transition, so we would like to be able to verify that what we received is correct. To do this, we decided to create an abstraction for packets which would encapsulate the data containing the inputs to our quadcopter. There would be a header, which would indicate where the packet began and end, and a checksum to validate the packet received.

We first established that the user could only send values from a range of 0 to 100. This meant that the byte for each input would always have its most significant bit set to 0. Using this property, we defined a header as a byte whose most significant byte (bit 0) would be set to 1. Bit 1 would be a flag for the emergency kill switch, while bits 2-3 would be flags reserved for other special options that we have not decided to implement, and bits 4-7 would be used to indicate the length of the packet. So for packet with 4 inputs, our header byte would look like `0x86`. If we wanted to send a kill signal, we could just send a two-byte packet with the header `0xc2`, which would indicate that we want to kill all motor outputs immediately.

To ensure a degree of data integrity, we also include a simple checksum in the packet as well. The checksum is simply calculated by taking the exclusive or of all other bytes in the packet, using an initial value of `0x80`. The sequence of bytes is then sent to the quadcopter, and the onboard processor checks that the XOR of all the bytes in the packet are equals `0x80`, or else it simply drops the packet and attempts to find the next header. By using an initial value of `0x80`, this would retain the property that our header is the only byte that has its most significant bit set to 1, and so would allow us to easily identify which byte is the header. So if we wanted to send a packet commanding the quadcopter to hover (the inputs would be 50, 50, 50, 50), then our packet with checksum would be: `0x86 32 32 32 32 06`.

## Deployment and Testing

For our earlier attempts to test our quadcopter, we wanted to make sure we could keep the quadcopter under control. We suspended the quadcopter with cord from the top, and also anchored it down from the bottom, effectively limiting the change in position the quadcopter could take. Plotting the accelerometer and gyro data along with the derived values for angular position, we were able to verify that quadcopter was indeed attempting to correct itself in the direction we wanted. However, our harness restrained the quadcopter too much. The quadcopter bounced off the restraints and was thrown into instability in our tests.

We then attempted to test without the harness in a room with more space. We were able to see that the quadcopter maintained orientation and flew in straight lines, but did not hover exactly as we expected, due to a steady-state error in angle. After tuning our parameters, we were able to get it to hover somewhat more reasonably, although not perfectly.

## Next Steps

As of this writing, we have an Xbox controller nominally setup to control the quadcopter. However, the range of inputs given by the gamepad still needs to be tuned to give finer control, since the large ranges that the user can currently input to the quadcopter too move too quickly in the given direction.

A simple process to improve performance and stability would be to properly balance the propellers. This drastically reduces the vibrations and consequently reduces the noise in our sensor readings and also makes flight smoother. Furthermore, we have not yet tested our quadcopter outdoors. It still needs to be tuned to resist disturbances such as wind.

Once our quadcopter is capable of stable flight, we can add a payload to it. We still have plenty of available thrust to work with and carry cargo. We can also add safety features such as self-landing upon loss of user input via radio, and if we're feeling particularly ambitious, basic collision avoidance using IR sensors.

# Appendix

Quadcopter Simulation with pure PD control: https://www.youtube.com/watch?v=GyPl5mLGbUY

Project Video: http://youtu.be/sWGtMhMRC5k

# References

Gibiansky, Andrew. "Quadcopter Dynamics, Simulation, and Control." 23 Nov. 2012. Web. 19 Dec. 2014. <http://andrew.gibiansky.com/downloads/pdf/Quadcopter Dynamics, Simulation, and Control.pdf>.

Lauszus, Kristian. "TKJElectronics/KalmanFilter." 10 Sept. 2012. Web. 19 Dec. 2014. <https://github.com/TKJElectronics/KalmanFilter>.

Liang, Oscar. "Quadcopter PID Explained and Tuning." OscarLiang.net. 13 Oct. 2013. Web. 19 Dec. 2014. <http://blog.oscarliang.net/quadcopter-pid-explained-tuning>.

Margolis, Michael. "nabontra/ServoTimer2." 2008. Web. 19 Dec. 2014. <https://github.com/nabontra/ServoTimer2>.

Stoffregen, Paul. "PaulStoffregen/TimerOne." 4 Jun. 2014. Web. 19 Dec. 2014. <https://github.com/PaulStoffregen/TimerOne>.