Drone Design Project

Giovanni Zarich, Chris Imasdounian, Chris McCormick, Hunter Streeter, Michael Moussa, & Vidal Reynoso

1. Abstract

Micro aerial vehicles (MAVs) have been an important advancement in human engineering. They can be used as tools for transportation and photography to instruments that venture into dangerous areas, as they do not require a pilot onboard. In this class, the engineering design process was implemented in order to build a quadcopter that would navigate a set number of waypoints in the most efficient manner. Several areas of research were looked at in order to optimize and create the best viable configuration.

After building the drone and competing in the class competition, we discovered a variety of things. For manufacturing, developing and creating a simple design allowed for more flexibility when assembling our drone. Being able to iterate components and swap out parts proved to be useful when changes needed to be made. For autonomy, different optimization methods were evaluated to find the fastest path through the given waypoints. On the electronics front, we learned about the anatomy of a drone and how each component works together to achieve stable, controlled flight. For controls, gain values were varied between 1 and 50, and each combination of gain values was run in the simulation to determine what value for Kp and Kd would produce the least amount of error. The best gain values for an acceleration cap of 4 m/s^2 were found to be 28 for Kd and 10 for Kp. The simulation allowed us to compute the optimized flight time using our waypoint sequencing and controller. It also allowed us to predict flight data such as the thrust, throttle, and power usage of the system before we physically tested it.

2. Vehicle Design & Manufacturing

Vehicle Design

To begin the design process of the drone, a decision was made regarding the shape of the drone frame. An “X Frame” design was decided upon as it allows an even distribution of thrust across the airframe. This allows the drone to be stable yet maneuverable. The next design decision was to implement cut-outs on the frame to reduce the weight of the frame. This design can be shown in Figure 1.

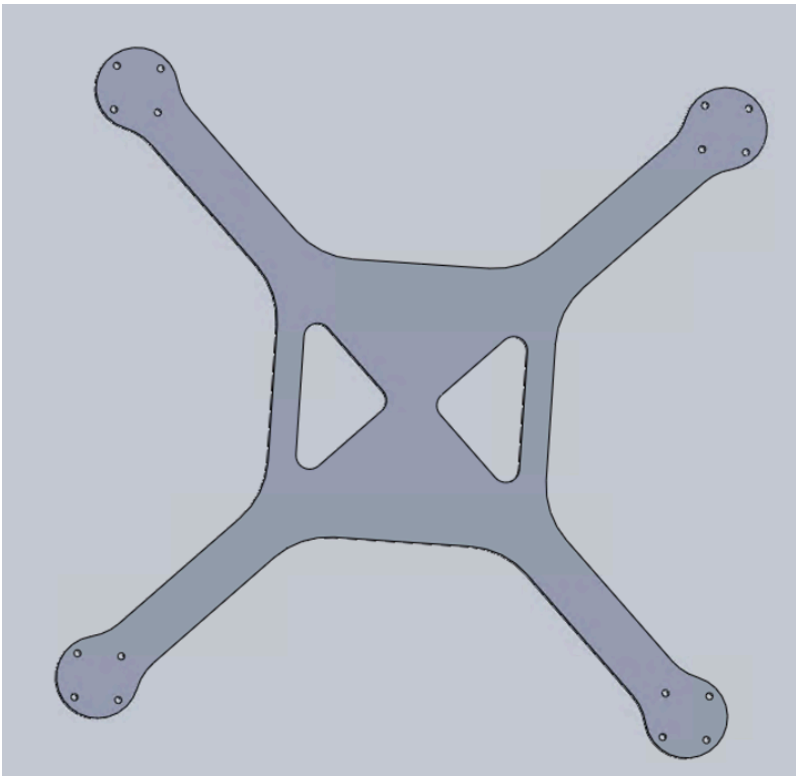


Figure 1: Original Drone Frame Design

The next major design choices included the drone legs and a cage for the LIDAR. The drone legs for the initial design were undersized but were used as placeholders for presentation. The LIDAR cage was designed to ensure the security of the LIDAR while also allowing vision. Both of these components can be seen in Figure 2.

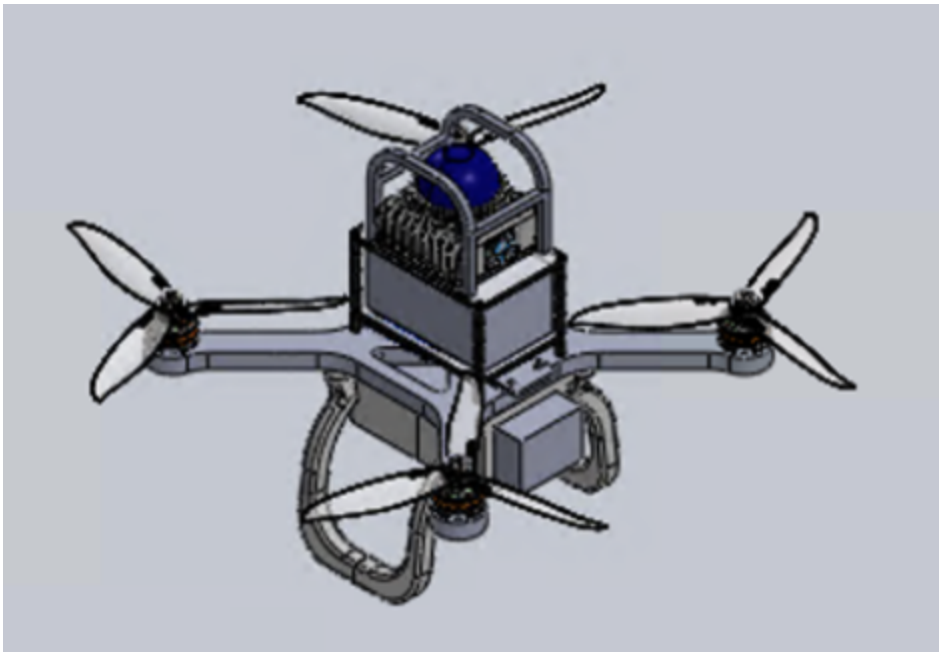


Figure 2: Complete Initial Drone Design

Following the preliminary design review (PDR), it was recommended to remove the cut-outs from the drone frame as it would result in little weight reduction. Another recommendation was to allow for more clearance between the propellers and the electronics stack which would be positioned in the middle of the drone frame. The last two recommendations proposed during the PDR include wider drone legs and increased protection on the top of the LIDAR cage. These changes can be seen in Figure 4 which showcases the final drone design.

After modifications were made following the PDR, further design changes were made in an effort to optimize the aesthetics and performance of the drone. The aesthetics of the drone were improved by rotating the motor mount holes. This allowed for the motor wires to smoothly run down the arms of the drone frame; preventing dangling wires. This can be visualized in Figure 3. A conversation with Professor Lopez provided insight into the improved performance of the drone if the motors are mounted on the bottom of the frame. This efficiency is a result of the propeller thrust not being obstructed by the frame as can be seen in Figure 4. The transition from motors being mounted on the top side to the bottom side of the drone was allowed due to the added clearance implemented following the PDR.

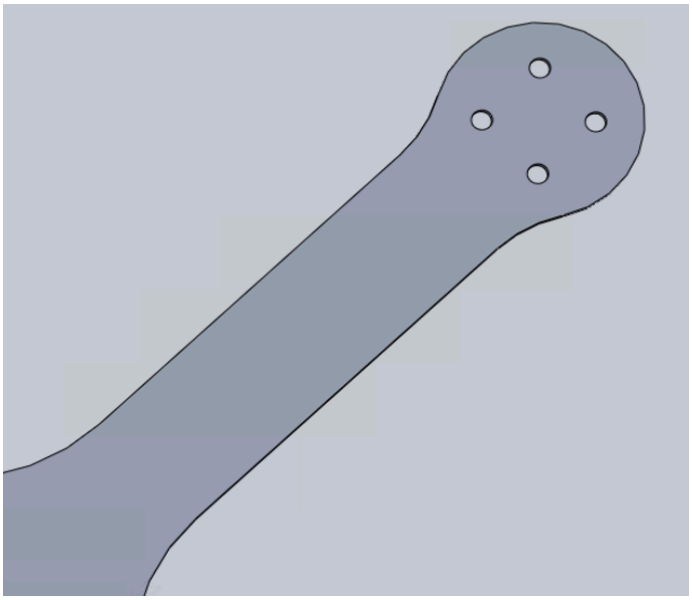


Figure 3: Rotated Motor Mount Holes

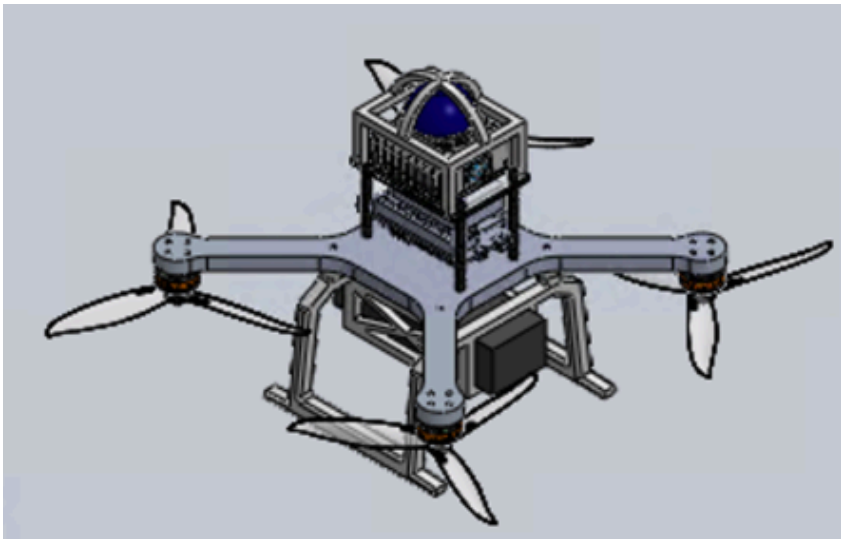


Figure 4: Final Drone Design with Improved LIDAR Cage, Wider Legs, and Bottom Mounted Motors.

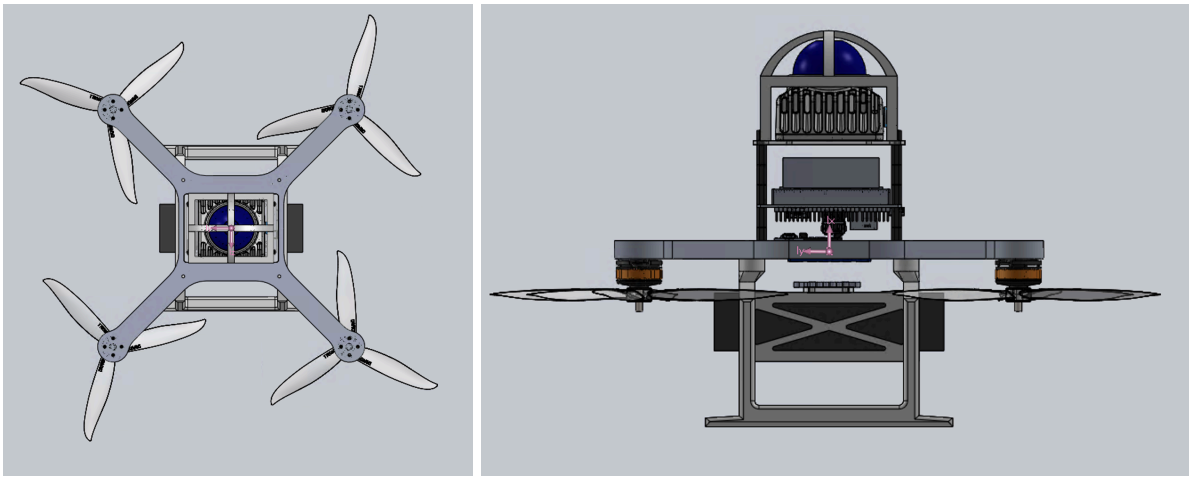


Figure 5: Final CAD Design featuring center of mass shown in pink.

As we continued to modify and tweak our drone design, we made sure to keep track of our center of mass and how it moved. In our initial design the center of mass was about a centimeter above the top of the frame. We made the decision to modify our drone legs in a way that would lower the battery slightly and try to bring down the center of mass to make it align closer to where we were going to place our IMU. Although in the final design the IMU was moved underneath the LiDAR, it was important to keep the center of mass as close to the center of the frame as possible to avoid uneven weight distribution during flight.

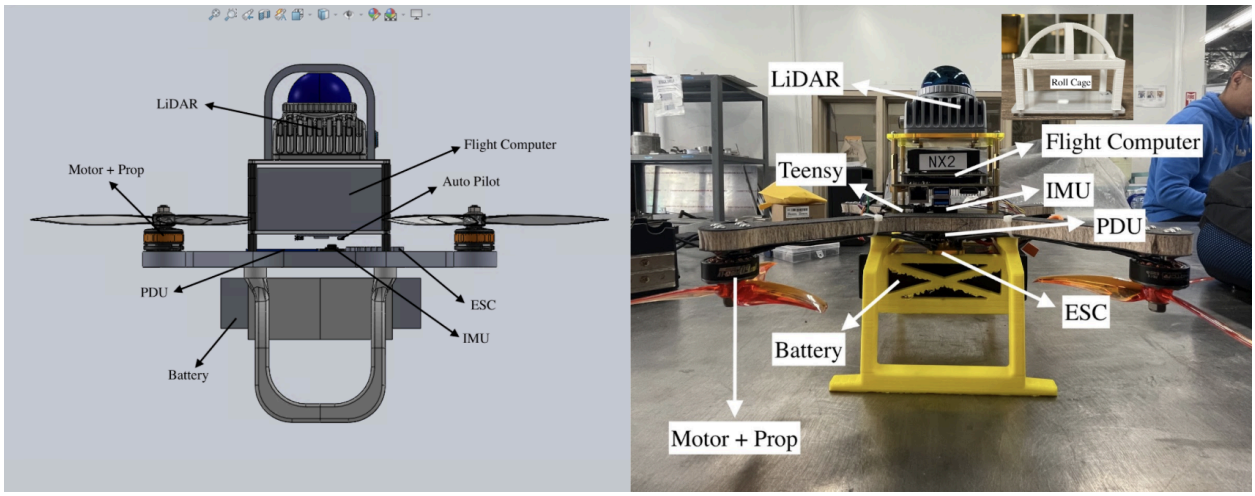


Figure 6: Original sensor stack design, shown left. Final sensor stack design, shown right.

Our initial instrument stack design was primitive and did not successfully take into account ease of assembly or simplicity. It featured two acrylic ‘base plates’ that were separated using standoffs. This design was meant to hold the fight computer in the middle, sitting on the first base plate. The LiDAR then would sit on the second base plate above it. Although this design did allow for more room underneath the flight computer for our other components, it did not make good use of the built in assembly holes of the flight computer. It also had all the other instruments simply placed on the top of the frame, which was simple but messy.

Our final design was much cleaner and made much better use of the space above and below the frame. We moved the PDU underneath the frame and the ESC sat on top of the battery cage. This cleared a lot of room on the top of our frame to ensure that our IMU could sit at the center of mass. We also made use of the built in holes of the flight computer by fixing the flight computer in place using our standoffs through them, this meant we only needed one base plate on top to hold our LiDAR, and roll cage. I also added a smaller acrylic plate on top of the battery cage to secure the ESC in place. This new design was far more efficient and easier to take apart and reassemble if need be.

Manufacturing

Item	Quantity
3m Screws	36
Acrylic plates	2
Standoffs	20
3D Printed Parts	2

Table 1: Manufacturing Parts List

The table above shows the minimal amount of parts needed to build our drone. When designing our drone we had a main goal to keep our design simple and easy to modify. As with

any project, as you begin building there is always something that does not work out. By designing our drone to be easy to modify we were able to move our motors underneath the frame, as well as move our PDU and ESC underneath. The simple design allowed us to create multiple iterations of parts and apply them to our drone the next lab session. In total we had 2 iterations of our drone legs, 4 iterations of our top base plate, and 2 iterations of our ESC holder plate.

Figure 7: 3d printed legs.

The image above shows two iterations of our 3d printed legs. Due to some measurement issues our first iteration's battery cage was too small. Although unfortunate, it was easy to redesign and print our second iteration of the legs which worked perfectly.

During the manufacturing process, we used a number of tools and machines that we had at our disposal to create and assemble our drone. We used the UCLA maker space to 3d print our drone legs, as well as our roll cage that protects our lidar. We also used the UCLA maker space to laser cut our acrylic base plate and ESC holder.

Figure 8: First Drone Legs Prototype with Heat Set Inserts

The image above shows our first drone legs prototype. Although this was not our final design, it does showcase a very important part of our manufacturing process which was heat set inserts. Heat set inserts were used to secure our legs to the frame itself, secure the ESC holder to the battery cage, and secure the LiDAR cage to the top base plate. Our only concern using these was that they would not be strong enough to secure the legs and battery the main frame, especially when the drone was flying. After some stress testing and trying to pull the inserts out using pliers it became apparent that they would be more than strong enough to accomplish their job.

The process to insert them began with drilling out a 3mm hole (or incorporating a 3mm hole into the cad design before printing) this was done to avoid too much plastic getting pushed up into the threads during the heating process. After placing the inserts on top of the hole, a soldering iron was placed inside the insert, which would then heat up the plastic and slide into

the hole. After removing the iron and allowing the plastic to cool it was ready to use. The inserts themselves were slightly wider than 3mm which allowed them to grab and take hold of the plastic as it melted into it.

3. Thrust Profile & Energy Calculations

After conducting Lab 2., we constructed an experimental thrust profile for our motors. This was done by running the motors at varying current values, measuring the resulting thrust, and then plotting the results. Below is displayed this plot, with the equation for the corresponding best-fit curve shown.

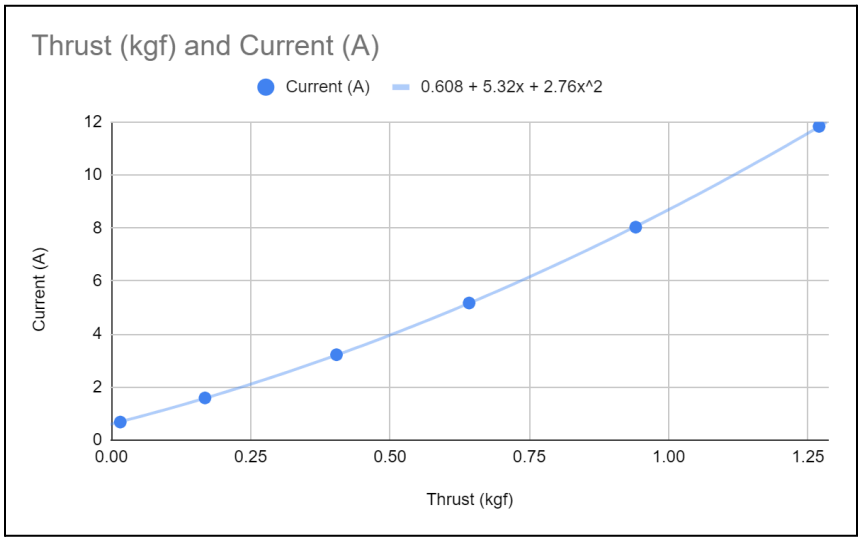


Figure 9: Lab 2 Thrust Curve

After creating this curve, using the predicted accelerations from one of our waypoint trajectory simulations, we calculated the thrust on the drone and mapped it to the current draw. Knowing the battery voltage, we then calculated the power draw at each of these accelerations, finding the max current and power for the entire trajectory.

Current (A)	Power (W)
17.87	264.45
17.16	254.05
…	…
Average Power:	182.14

Max. Current (A)	Max. Power (W)
17.87	264.45

Table 2: Current and Power Draw

We then calculated the total energy used through this simulated trajectory by integrating the power over time. Displayed below is the used battery capacity compared to the total battery capacity.

Total Used Capacity (mAh)	Battery Capacity (mAh)
136.89	5000

Table 3: Battery Capacity

4. Electronics Overview

One of the best parts of designing a drone is picking out each subcomponent. In the following section, the role of each component will be discussed, the specific component used will be shown, and how they all work together will be analyzed.

The main brain of our drone is the flight computer. Based on the signals it receives from the measurement systems on board, the flight computer calculates and decides how to achieve a given positional state. It then signals to various other systems on the drone to achieve the desired position. The flight computer selected for this drone is the Jetson Orin Nano, which takes 5V as input, and it is displayed below.

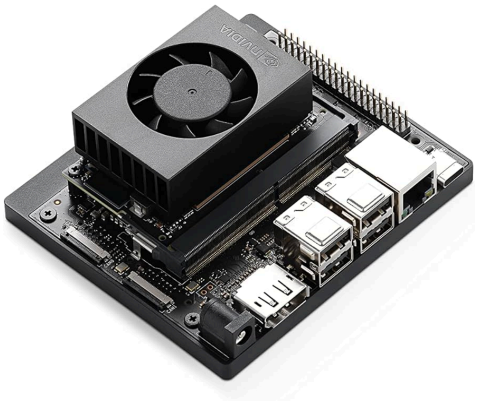


Figure 10: Jetson Orin Nano Flight Computer

Connected to our flight computer via a 3V connection is the autopilot, or flight controller. The flight controller receives attitude measurements from the IMU, compares them to the desired measurements, and then determines how fast each motor should spin to achieve the desired attitude. It then sends this signal to the ESC to be decoded and implemented. The autopilot used onboard our drone is the Teensy 4.0.

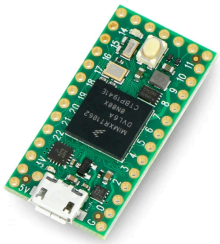


Figure 11: Teensy 4.0 Autopilot

Linked to the autopilot is the electronic speed controller (ESC). The ESC receives signals from the autopilot, telling it how to distribute power to the motors to achieve a desired attitude command. All power that goes to the motors first passes through the ESC.

Motors are rated according to their stator size and Kv rating. The stator size is essentially the size of the internal circle of coils inside the body of the motor. The Kv rating of a motor is basically the rotational speed of the motor (in RPM) when 1 volt is applied with no propellers attached. For heavier drones, a large stator and lower Kv are desired because these correspond to higher torques produced by the motor. For lighter drones, smaller stator sizes and a lower Kv rating are desired because not as much torque is necessary, and lower Kv corresponds to smoother, more responsive flight. On our drone, we used the Tekko32 F4 Mini 50A electronic speed controller with four 1950 Kv T-Motor F40 Pro V Motors.

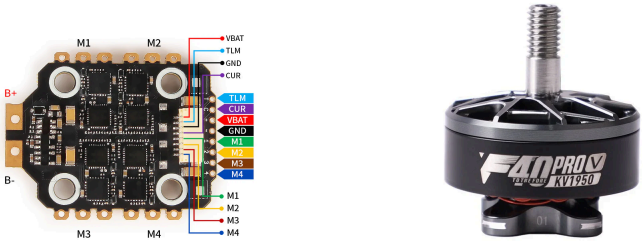


Figure 12: Tekko32 F4 Mini 50A ESC (left) and one 1950 Kv T-Motor F40 Pro V Motor (right)

For attitude measurements, we use the Adafruit MPU 9250 inertial measurement unit. As mentioned, this device measures and provides our flight controller with the x-, y-, and z-components of the drone's linear and rotational velocity and acceleration. The Adafruit IMU connects to our Teensy autopilot and is displayed in the figure below.

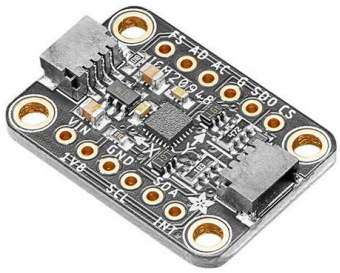


Figure 13: Adafruit MPU 9250 IMU

The positioning system on board our drone is the Livox Mid-360 LiDAR. LiDAR is a method of remote sensing the environment around the drone using lasers, to ensure the drone does not crash into the environment. This system relays position measurements to the flight computer, allowing the computer to make informed decisions about its next movement. The Livox Mid-360 LiDAR used on our drone takes 9V as input directly from the battery.

Figure 14: Livox Mid-360 LiDAR

Lastly, to supply all these components with energy, we use a 5000 mAh 4S battery. 5000 mAh represents the capacity of the battery, which is more than enough to complete our competition. 4S relates to the number of battery cells and how they are configured. 4 represents the number of cells, and S stands for series, meaning the cells are connected in series (rather than

P, which stands for parallel). Our battery connects to a power distribution board, which then connects to each of the various components on board to supply them all with power.

Figure 15: 5000 mAh 4S Battery

Lastly, it is valuable to have a visual representation of how each component is linked together. Therefore, displayed below is a schematic of each component and how they are all connected.

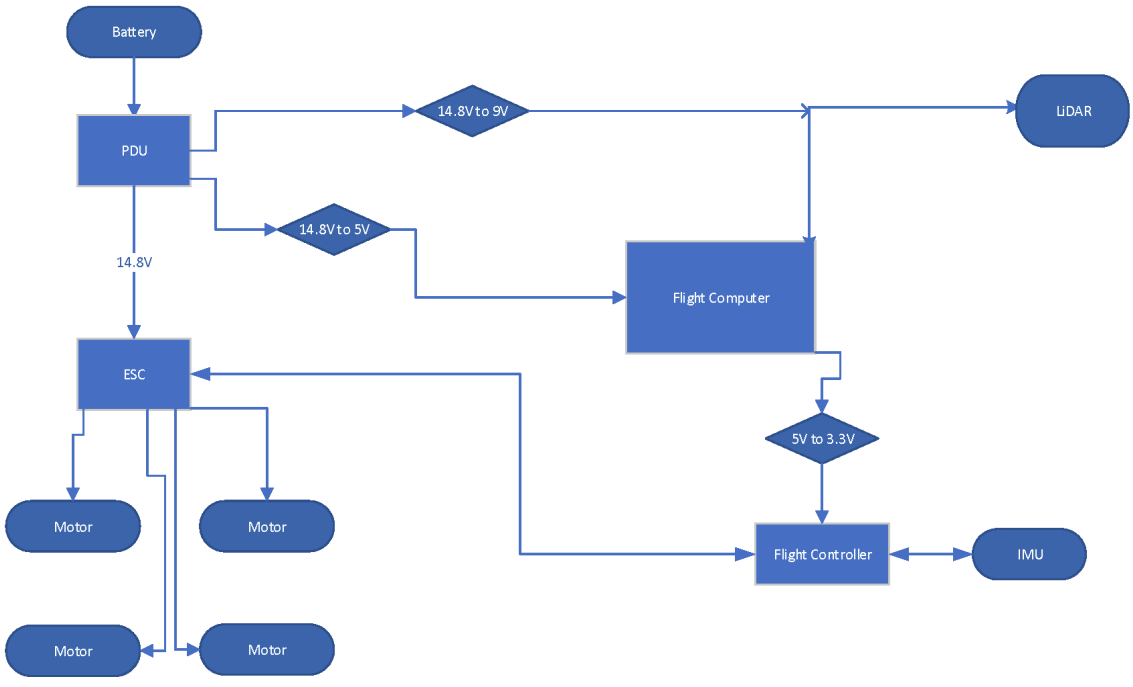


Figure 16: Drone Electronics Schematic

5. Guidance Design & Analysis

The role of the Autonomy lead is to calculate the optimal path between waypoints in order for the MAV to fly through the course as fast as possible. This is a version of the Travelling Salesman problem, one of the most famous optimization problems. In the Travelling Salesman problem, the salesman needs to visit a given number of cities, visiting each once and returning to the starting city in a way to minimize the distance traveled. The problem has been studied for almost 100 years, and over that time several popular algorithms have been developed and used to solve it. This problem has large real world applications, especially due to the rise in prevalence of online shopping and food delivery.

The most basic method to solve this problem is the nearest neighbor approach. In this approach, the salesman, or drone in our case, travels to the closest city to his current location. The following code excerpt shows the basis of the method:

```
% Compute the Euclidean distances from the current position to each waypoint
   distances = sqrt((waypoints_X - current_position(1)).^2 + ...
                    (waypoints_Y - current_position(2)).^2 + ...
                    (waypoints_Z - current_position(3)).^2);

   % Find the closest unvisited waypoint
   [min_distance, min_index] = min(distances);
```

Although being very intuitive, the nearest neighbor method does not always result in the path of least distance.

The next method is the brute force method, in which all possible paths are calculated. The path with the shortest distance is then chosen. This method is the most accurate of the optimization methods. However, with large numbers of nodes the method takes immense computing power and time, leading to the development of different iterative methods. This method was found to be the most effective for paths with under 11 nodes(cite).

The k-opt family of algorithms are heuristic algorithms. They start with an arbitrary path, and replace a number k of the legs of the path as long as they produce a shorter path. The most common k-values are 2 and 3, with 2 and 3 legs being checked and replaced respectively. This method was found to be the most effective for numbers of nodes between 11 and 50 (cite).

Markov's chain family of algorithms is most effective for over 50 nodes (cite). Markov's chain methods are stochastic methods and use probability based on current position to determine which node to go to next. One example of this is the Ant Colony optimization method, which seeks to use the idea of an ant leaving pheromones on its most traveled path to solve these problems.

In order to determine the ideal method, the nearest neighbor, brute force, and 2-opt methods were tested. Trials were conducted for a randomized series of waypoints with 8 waypoints each. The first three trials were conducted with constant z and the last two have changes in the z direction. The results are summarized in the table below:

Trials	1	2	3	4	5
Brute Force Distance (m)	37.7251	44.8706	38.5325	53.1183	70.0119
Nearest Neighbor Distance (m)	45.0744	44.874	55.5779	57.1227	73.0024
2-opt Distance (m)	37.7251	44.8706	38.5325	53.1183	70.0119
Brute Force comp. Time (s)	0.1531	0.129	0.1944	0.1646	0.1589
Nearest neighbor comp. Time (s)	0.075	0.0863	0.0886	0.0913	0.09
2-opt comp.	0.1141	0.1536	0.1507	0.1613	0.1156

Time (s)					

The table shows that the brute force and 2-opt methods gave the same path for each, with the brute force method generally taking a little longer to compute. The nearest neighbor method computed the fastest, but had a slower route in each trial. Due to its accuracy and intuitive process, the brute force method was chosen.

However, this method only accounts for distance. In the physical system, changing direction causes the craft to need to slow down. In order to attempt to simulate this, a penalty was given for changing direction, with different degrees of penalty depending on the magnitude of the change in direction. The value of the penalty was found empirically through comparison with the MAV simulation to be 1.05. A code snippet for the penalty section can be found below:

```
turn_penalty_factor = 1.05;
% Calculate the direction vector
       current_direction = waypoint_position - current_position;
       if norm(current_direction) > 0
           current_direction = current_direction / norm(current_direction); %
Normalize the direction vector
       else
           current_direction = [0, 0, 0];
       end
       % Calculate the direction change penalty
       if wp_idx > 1
           direction_change = dot(previous_direction, current_direction);
           penalty = (1 - direction_change) * turn_penalty_factor; % Penalty is
proportional to the change in direction, multiplied by the factor
                else
           penalty = 0;
       end
```

The code defines the current direction of the MAV and applies a specified penalty for changing direction. Upon further testing, it was observed that for a variety of waypoint sequences, only a penalty factor greater than 3 would cause the ideal path to deviate from that of the smallest distance. However, in the vast majority of cases, there was no path change regardless of the penalty factor. The results are tabulated below:

Trials	Crossover Penalty Factor
1	none
2	none
3	3.06
4	none
5	3.025
6	none
7	none
8	none
9	none
10	none

In addition to the computation code, two separate functions were created. One function reads in the designated csv file and splits the table into arrays to be used as inputs in the computational code. The other outputs the calculated waypoint path in the same format as the example file.

6. Control & Simulation Analysis

6.1 Control

The goal of optimizing the controls was to have the drone navigate its waypoints with the least amount of error. Having a controller with minimal overshoot that adheres to the velocity and acceleration constraints is what was desired. Two methods were considered when it came to optimizing the controller. One method involved finding the system's root locus, and varying the gain K from zero to infinity to determine which Kp and Kd values produce the least amount of overshoot and also have the smallest steady state error. This method involved identifying poles

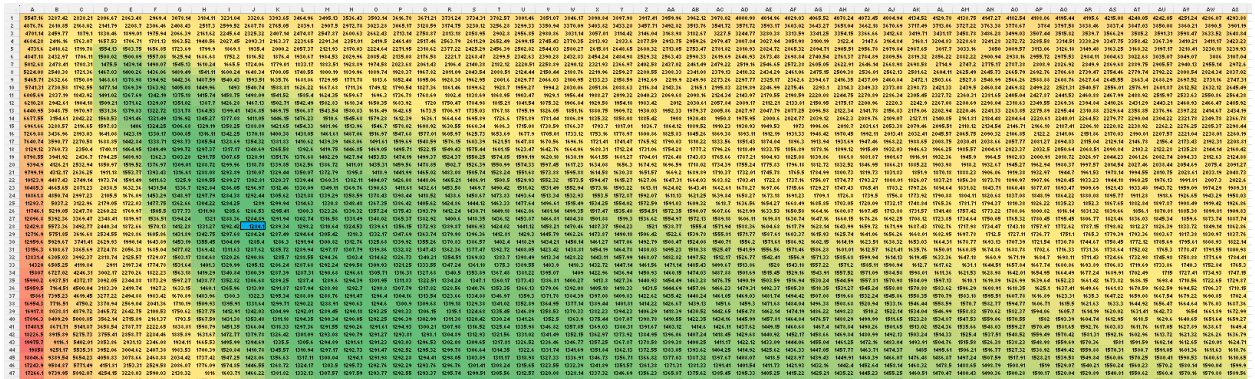
and zeros, drawing the paths of the poles when k varies from 0 to infinity, and utilizing stability criterion to make sure the system converges. The second method considered involved running the simulation n by n number of times where n is the maximum value of Kp and Kd, and observing how the controller reacts given a set of waypoints. Eventually the latter option, nicknamed the 'brute force method' was chosen for several reasons. First, it provided a better visual understanding of which Kp and Kd values would lead to the least amount of error and fastest time, however at the cost of understanding the theory and why those values were desirable. The brute force method relied on the sim, rather than theoretical equations, to tell us which controller gains would work the best. The brute force method also made it easier to adjust gain values given different waypoint information. Root locus cannot easily recalculate gain values when a change in waypoints is introduced.

The ideology behind the brute force method was to find the path that would produce the least amount of error. This involves position error, which is related to the proportional term Kp, and velocity error, which is related to the derivative term Kd. The simulation code was placed inside a double for loop. Each of the loops would iterate through integer numbers from 1 to n, n being the largest test value for Kp and Kd. In this fashion, the simulation is run n x n times, producing an error matrix that holds the total error for every combination of Kp and Kd up to n. The least amount of error corresponds to the most 'efficient' path. In order to calculate the error, the following equation was used

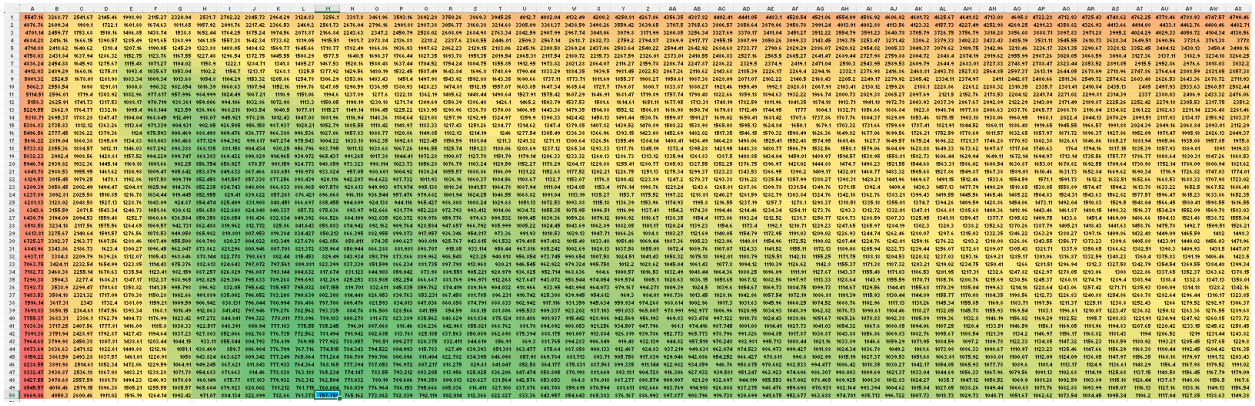
$$Error = \sum_{i=1}^{3} [\sum_{j=1}^{N} Error_{position} + \sum_{j=1}^{N} Error_{velocity}]$$

The outer summation is to sum errors in the x,y, and z components. The inner error is taking the sum of the errors of position and velocity at each time step. N is the total number of time steps. The location of the minimum value in the error matrix represents the desirable Kp and Kd

values. Kp values increase by the columns and Kd increases by the rows. The error matrix was exported into Excel to visualize which element held the smallest error. Below is a figure of that plot. Waypoint sequencing was taken from the guidance code.



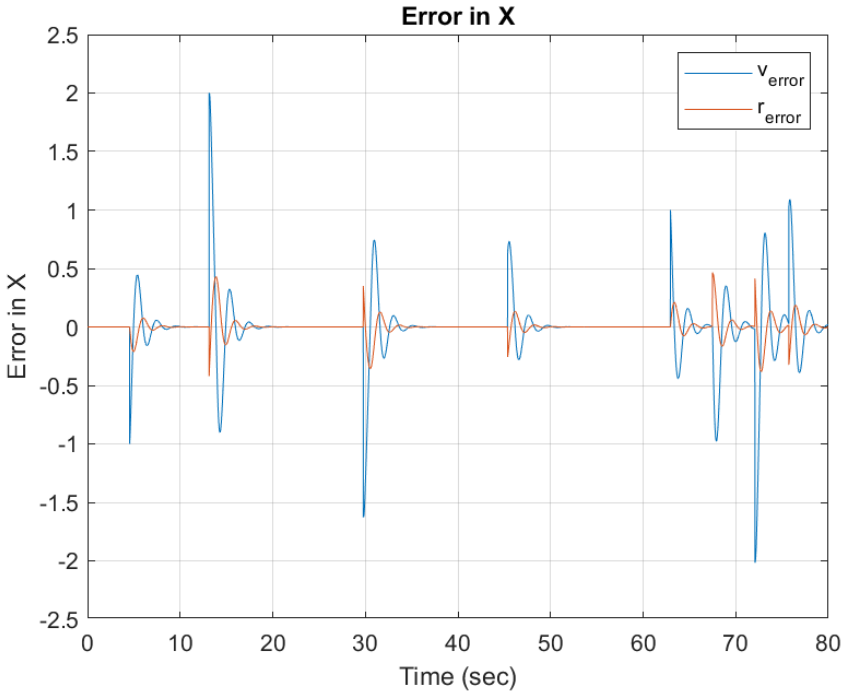
Highlighted in blue is the location of the minimum error. The minimal total error was found to be 1284.1 at a Kp value of 10 and Kd value of 28. While it was required that acceleration be capped at 4 meters per second, a secondary test was performed to observe the effects of removing the acceleration cap. This produced the following error matrix

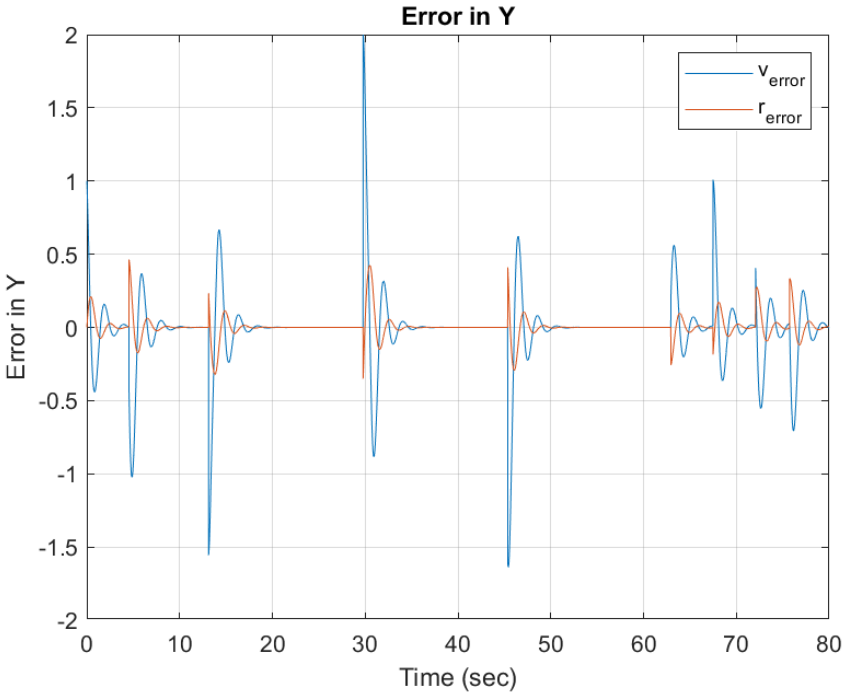


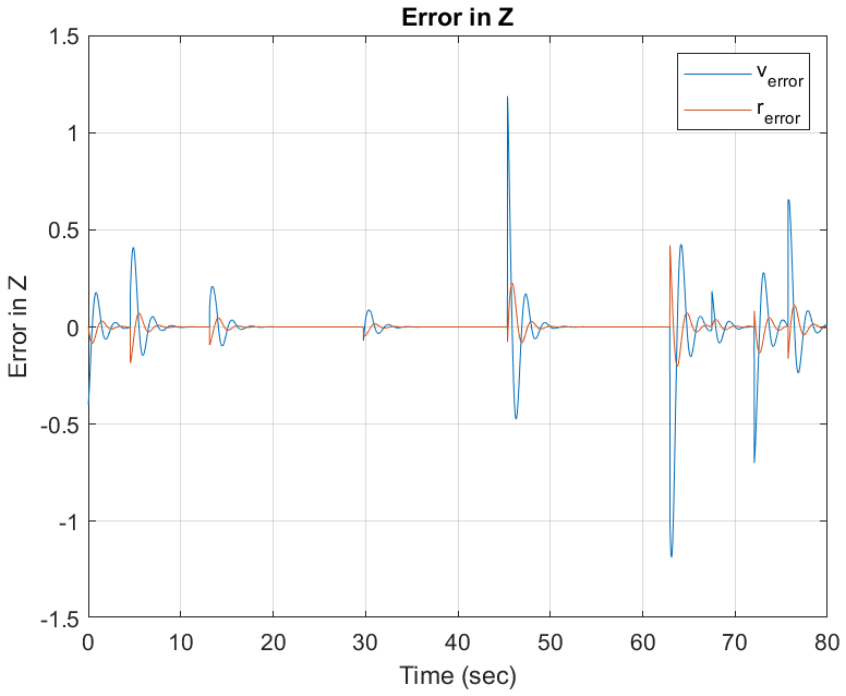
As we can see, Kd now becomes unbounded. The minimum total error was also less at 757.75. Kp increased slightly to 13. Larger and larger Kd values lead to better tracking due to the fact that larger accelerations can be demanded in order to get the desired velocity as quickly as possible. So increasing the gains in this case comes at no penalty. In reality, however, large

instantaneous accelerations demanded by having large gain values is not possible, and for safety reasons, decided to cap our acceleration input at 4 m/s^2.

Plots for position and velocity error in the x,y, and z components for the chosen Kp and Kd values were also produced to observe the respective demands after the simulation decides to move on to the next waypoint. Below are those plots.







It can be noted that large velocity spikes occur when moving from the current to the next waypoint. This is a result of the idealized case we are looking at in the simulation, where

acceleration is an impulse, and therefore velocity is a step function. This causes the velocity to take on large values as a result of an impulse to the system, which is the acceleration.

Further analysis was done to research how dependent waypoint data was in determining Kp and Kd values. Since the root locus method was not implemented, and reliance on the simulation was needed to find gain values, it was important to see how dependent gain values were to waypoint distances and directions. Waypoints were randomized within a -10 to 10 meter cubic space. 10 waypoints were generated, creating 10 error matrices. The following optimal Kp and Kd values were obtained.

	Random Waypoints 1	Random Waypoints 2	Random Waypoints 3	Random Waypoints 4	Random Waypoints 5	Random Waypoints 6	Random Waypoints 7	Random Waypoints 8	Random Waypoints 9	Random Waypoints 10
Kp	10	14	10	14	11	9	10	10	10	10
Kd	29	36	28	35	27	23	27	27	26	26

The standard error for Kp values was 0.5537 and the standard error for Kd values was 1.284. This data shows that, at least for the scope of the competition being held in the lab, Kp and Kd values will not vary significantly given different waypoints. Once distances and time scales begin to become larger, the dynamics may require new gain values, however, for this scale, Kp and Kd values are around 10 and 28 for most waypoints.

6.2 Simulation

The simulation works by making a reference path including position and velocity traversing through the waypoints. These values are calculated using polynomials from kinematic equations. Then it applies feed-forward or feedback controller loops to iterate the position, velocity, and acceleration of the drone. These calculations are done for each component of the position, velocity, and acceleration, and they take the form of:

$$u = -K_p r_e - K_d \frac{d}{dt} r_e$$

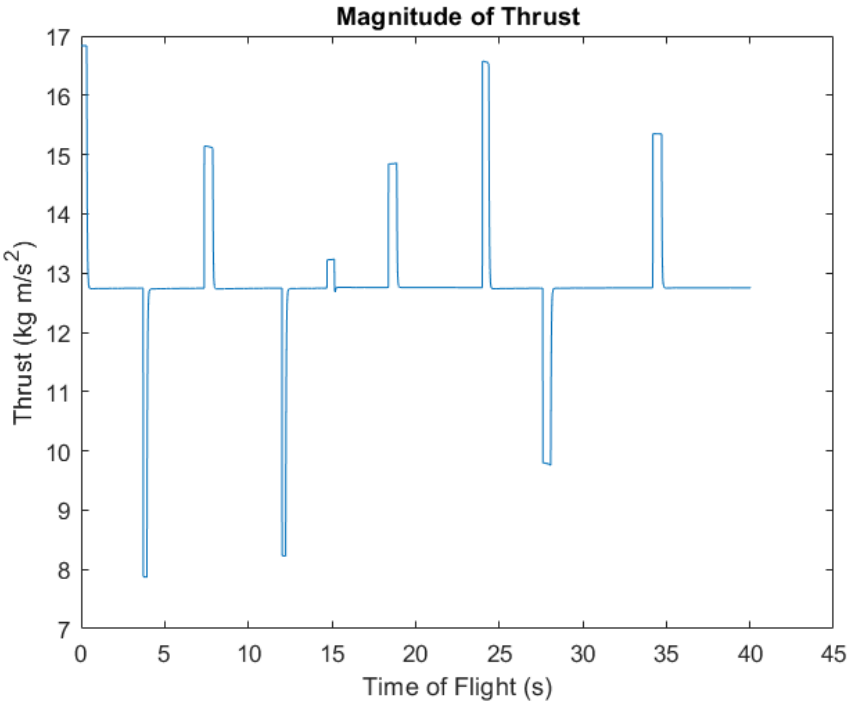
$$u = \frac{d^2}{dt^2} r_d - K_p r_e - K_d \frac{d}{dt} r_e + g$$

where r_d is equal to the reference position, and r_e is the difference between the current position and the reference position. The first controller is used for the x and y components where the reference (the waypoints) are not accelerating. However, in the z direction, acceleration due to gravity accelerates the reference, and a feed-forward loop is necessary to keep the controller from diverging off of its path. To simulate the time, discreet steps of dt = 0.01 are taken, and the control is run through a loop to find the u input at each time. We also implemented the physical constraint of the drone including a maximum net acceleration of less than or equal to $4 \frac{m}{s^2}$ to avoid overdrawing power from the battery and losing control.

The purpose of the simulation is to predict the path of the drone under ideal conditions. This was where we found the optimal flight time. The simulation takes the input of the autonomy system, which produces the optimal order of waypoints, and therefore the path. Then it applies the controller of the guidance system to control the path that the drone will take. Because the controls utilize the brute force method to find the optimal Kp and Kd coefficients, the simulation first runs the flight path under every combination of coefficients. Throughout every run, the error of the flight path and reference path is calculated and summed up at the end. Then, by comparing the errors, we chose the optimal controller coefficients and ran the simulation one more time using that controller to plot the path and extract the acceleration of the drone throughout the flight.

Data from the simulation is useful as it contains the flight time and acceleration of the drone. This allowed us to find the thrust profile, the power necessary for flight, and the energy

drawn from the battery to get through the whole course. The thrust profile for our test flight is given below.



In this profile, it can be seen that the hover thrust is equal to the acceleration due to gravity times the mass of the drone. It drops below that when the drone is traveling downwards. The maximum thrust during flight is 16.8364 kg m/s^2. The acceleration peaks at 12.95 m/s^2. When subtracting hover acceleration, we see the net acceleration is 3.14 m/s^2. It does not need to approach the maximum acceleration we set for safety. Using the thrust, our electronics subsystem is able to find the power draw throughout the flight as well as the energy draw from the battery.

One way to make the simulation more accurate in finding the optimal flight is to update our reference path with a higher-order kinematics equation. The reference path only finds the velocity and position between two points assuming the velocity is constant. If instead, we were to make v differentiable throughout the reference, the controller would have a more precise path

to follow. This would also help us find more optimal Kd and Kp values. This is because we determine the best Kd and Kp by measuring the total positional error throughout each trial flight. This makes the error unrealistic because the majority of it would come from the inability of a continuous velocity trajectory to follow the sharp turns of our current reference path.

Our current simulation runs according to these steps to find the reference.

```
for i = 1:N
        if j==1 || abs(norm(r(i,:) - wp(j,2:4)))<=ep && j < size(wp, 1)
            Tx = abs(wp(j+1 , 2) - wp(j,2))/v_max;
            Ty = abs(wp(j+1 , 3) - wp(j,3))/v_max;
            Tz = abs(wp(j+1 , 4) - wp(j,4))/v_max;

            T = max([Tx Ty Tz]);
            elapsedt_checkpoint(j) = i * dt;
            j = j+1;
            step(j) = i;
        end

          vd(i,1) = (wp(j,2) - wp(j-1,2)) / T;
          vd(i,2) = (wp(j,3) - wp(j-1,3)) / T;
          vd(i,3) = (wp(j,4) - wp(j-1,4)) / T;

          rd(i,1) = wp(j-1,2) + vd(i,1) * dt *k*(i-step(j));
          rd(i,2) = wp(j-1,3) + vd(i,2) * dt *k*(i-step(j));
          rd(i,3) = wp(j-1,4) + vd(i,3) * dt *k*(i-step(j));
```

And these steps to apply the controller:

```
u(i,1) = -Kp_opt*(r(i,1) - rd(i,1)) - Kd_opt*(v(i,1)-vd(i,1));
u(i,2) = -Kp_opt*(r(i,2) - rd(i,2)) - Kd_opt*(v(i,2)-vd(i,2));
u(i,3) = -Kp_opt*(r(i,3) - rd(i,3)) - Kd_opt*(v(i,3)-vd(i,3)) + g;

u_minus_g(i,1) = u(i,1);
u_minus_g(i,2) = u(i,2);
u_minus_g(i,3) = u(i,3) - g;

r(i+1 , 1) = r(i,1) + dt*v(i,1) + 0.5*dt^2*(u(i,1));
r(i+1 , 2) = r(i,2) + dt*v(i,2) + 0.5*dt^2*(u(i,2));
r(i+1 , 3) = r(i,3) + dt*v(i,3) + 0.5*dt^2*(u(i,3)-g);

v(i+1 , 1) = v(i,1) + dt*(u(i,1));
v(i+1 , 2) = v(i,2) + dt*(u(i,2));
v(i+1 , 3) = v(i,3) + dt*(u(i,3)-g);
```

In this model, we also assumed translational motion as the drone can navigate the path without having to rotate. In a future project, we could take on the task of designing the controls and simulation of an MAV that must rotate in the direction it will traverse. We could then analyze the power and time efficiency of the two methods of travel.

7 Flight Analysis

After competing in the class competition, we found that human error significantly influenced how well we performed. Though we had designed the best waypoint sequencing algorithm known to mankind, we placed last place in the class competition because of our poor ability to control the drone. Had we implemented an autopilot system to fly through the waypoints, we would have won the class competition with world-record-setting course times.

8 Lessons Learned

8.1 Vehicle Design & Manufacturing

Upon arrival for flight testing, it became known that our drone was used by the professor to gain insight on the class's drone controls. Due to the initial vehicle control testing, the drone endured many crash landings with the consequence of fractures developing in the drone legs; which later caused a complete break of the drone legs. To prevent this from occurring in the future and ensure the robustness of the drone, the infill of the 3D-printed legs should increased. This would produce more sturdy drone legs.

8.2 Controls

Many lessons were learned by creating a control system for this MAV. One important concept that was realized is that visualizing how acceleration, or the control input in this case,

affects velocity and position errors is not very simple. In the case where the simulation would move onto the next waypoint, large velocity errors were seen compared to the position. Position error makes a lot of sense, as it is the physical distance between the current MAV location and the next waypoint. However obtaining a velocity reference is dependent on the impulse in acceleration, which is less easy to visualize.

8.2 Electronics

From electronics, we learned how each component works together to achieve flight. We conducted research into each specific electronics component and learned what each component's role in the overall design was, and how it communicates with other subsystems. We also learned valuable technical skills through soldering, and how to apply concepts learned in the classroom to the real world by computing power, energy, and flight time calculations.

8.4 Autonomy

Following the competition, we learned that although all the groups stated to have used the brute force method, 4 of the 5 had differing routes for at least one set of waypoints. This could be attributed in our case to the inclusion of the penalty factor to penalize large turns. An additional factor that was not included in the code was the MAV returning to its original point, which could have been another explanation for the difference in paths.

8.5 Simulation

Throughout designing the simulation script of the drone, we learned how to use feedback and feed-forward loops to control and model an MAV trajectory. We also learned how to modify our model to accommodate multiple types of waypoint trajectories and physical constraints of

the flight. As we were developing our code, we started with a simple model in 2 dimensions and ignored gravity. Then, we implemented the vertical component, which was a level of complexity higher than the 2-dimensional path. Then we optimized the software to run faster, but we opted for a model that would sacrifice processing time for a more accurate Kd and Kp. Had the task been to optimize flight and processing time, a different method of finding the controls would have been implemented.

Team Contributions

Michael Moussa was the design lead and worked primarily with the CAD model. This included the design and optimization of the drone frame and other 3D CAD components. Hunter Streeter was the manufacturing lead and worked closely with Michael in redesigning and assembling the drone, this role included 3D printing parts, laser cutting components, and assembling the drone. Christopher McCormick was the electronics lead. Chris Imasdounian worked on obtaining the most efficient controller that would follow a set of waypoints. This included working with Vidal, the simulation lead, in order to code a simulator, then apply test conditions for different gain values and determine which gains were the most efficient. Chris McCormick handled all component compatibility, and soldering, as well as all the power, energy, and flight time calculations. Giovanni was responsible for writing the autonomy sections of the report, as well as creating the waypoint sequencing code. Vidal was responsible for writing the simulation script in order to test the waypoint sequencing and controller along with Giovanni and Chris. He also calculated the thrust profile and worked with Chris to find the power and energy profile of the drone throughout the flight.

7. Appendix

```
%% MAV Autonomy Final Code
%
% 105574974
% Giovanni Zarich
%
% This is the code to be used in the flight competition to determine the
% ideal path for the MAV.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code to read in waypoint file given
% Read the CSV file into a table, skipping the first row
filename = 'waypoints_example.csv';
data = readtable(filename);
% Convert the table to an array
dataArray = table2array(data);
% Separate the columns into individual arrays
sequenceNumber = dataArray(:, 1);
waypoints_X = dataArray(:, 2);
waypoints_Y = dataArray(:, 3);
waypoints_Z = dataArray(:, 4);
%% Post PDR Code
% this code attempts to solve for the time it takes to slow down / turn
% instead of just total distance
% Number of waypoints
num_waypoints = numel(waypoints_X);
% Start the cube at origin
```

```
current_position = [0, 0, 0];
% Keep track of visited waypoints
visited_waypoints = false(num_waypoints, 1);
% Output the X, Y, Z coordinates of the waypoints in the order the cube
navigates them
waypoints_path_bruteForce = zeros(num_waypoints, 3);
% Calculate all possible permutations of waypoints
waypoints_permutations = perms(1:num_waypoints);
% Iterate over each permutation
min_cost = Inf;
fastest_permutation = [];
fastest_perm_index = 0; % To store the index of the fastest permutation
for perm_idx = 1:size(waypoints_permutations, 1)
   % Initialize the current position to the starting point
   current_position = [0, 0, 0];
   total_cost = 0;
   previous_direction = [];

   % Iterate over each waypoint in the current path
   for wp_idx = 1:num_waypoints
       % Get the waypoint index from the current path
       wp_index = waypoints_permutations(perm_idx, wp_idx);

       % Calculate the distance from the current position to the current
waypoint
       waypoint_position = [waypoints_X(wp_index), waypoints_Y(wp_index),
waypoints_Z(wp_index)];
       distance_to_wp = sqrt(sum((waypoint_position - current_position).^2));
```

```
        % Calculate the direction vector
        current_direction = waypoint_position - current_position;
        if norm(current_direction) > 0
            current_direction = current_direction / norm(current_direction); %
Normalize the direction vector
        else
            current_direction = [0, 0, 0];
        end

       % Define penalty factor
        turn_penalty_factor = 1.05;
        % Calculate the direction change penalty
        if wp_idx > 1
            direction_change = dot(previous_direction, current_direction);
            penalty = (1 - direction_change) * turn_penalty_factor; % Penalty is
proportional to the change in direction
        else
            penalty = 0;
        end

        % Update the current position and total cost
        current_position = waypoint_position;
        total_cost = total_cost + distance_to_wp + penalty;

        % Update the previous direction
        previous_direction = current_direction;
    end

    % Update the minimum cost and waypoints path if needed
```

```
    if total_cost < min_cost
        min_cost = total_cost;
        fastest_permutation = waypoints_permutations(perm_idx, :);
        fastest_perm_index = perm_idx; % Store the index of the fastest
permutation
    end
end
% Check if the fastest_permutation is found
if isempty(fastest_permutation)
    error('No valid permutation found.');
else
    % Navigate the small cube through the fastest route
    for wp_idx = 1:num_waypoints
        % Get the waypoint index from the fastest permutation
        wp_index = fastest_permutation(wp_idx);

        % Update the current position to the current waypoint
        current_position = [waypoints_X(wp_index), waypoints_Y(wp_index),
waypoints_Z(wp_index)];
        waypoints_path_bruteForce(wp_idx,:) = current_position; % Store the
current position
    end
end
% Output the X, Y, Z coordinates of the waypoints in the order the cube
navigates them
disp('Waypoints Path:');
disp(waypoints_path_bruteForce);
%% Waypoint Writer Section
% % In case
```

```
%
% % Number of rows in the waypoints_path_BruteForce
% numRows = size(waypoints_path_bruteForce, 1);
%
% % Create a sequence number column
% sequenceNumber = (1:numRows)';
%
% % Combine the sequence number with the waypoints data
% dataWithSequence = [sequenceNumber, waypoints_path_bruteForce];
%
% % Convert the array to a table and add column titles
% columnTitles = {'seq num', 'x', 'y', 'z'};
% dataTable = array2table(dataWithSequence, 'VariableNames', columnTitles);
%
% % Define filename
% filename = 'waypoints_with_sequence_test.csv';
%
% % Write the table to a CSV file
% writetable(dataTable, filename);
%
% % Display a message indicating the file was written
% disp(['Data successfully written to ', filename]);


clc;
clear;
close all;
%% MAV Simulation
% Vidal Reynoso Jr
```

```
% 305622035
% MAE 157A
%% Initiate Variables
% Change the waypoint list columns to include sequence number
% Waypoint List Example
% [x y z] coordinates of waypoints within 10 x 10 m area
wp =  [1 1 1 1;
      2 1 5 5;
      3 2 9 5;
      4 4 4 8;
      5 4 1 7;
      6 8 3 5;
      7 2 7 5;
      8 3 8 9;
      9 1 1 6;
      10 2 7 8];
% Time step
dt = 0.01;
% Guess total steps
N=8000;
% Number of Kp and Kd values to check
n = 30;
Kp = 1:n;
Kd = 1:n;
r_error = zeros(N,3);
error = zeros(n,n);
time = zeros(n,n);
v_max = 1;
u_max = 4;
```

```
ep = 0.5;

g = 9.81;

iterations_ab = zeros(n,n);

for a = 1:n

   for b = 1:n

       % Distance and velocity from start node (0)

       r = zeros(N , 3);

       rd = zeros(N , 3);

       r(1,1) = wp(1,2);

       r(1,2) = wp(1,3);

       r(1,3) = wp(1,4);

       v = zeros(N , 3);

       vd = zeros(N, 3);

       ad = zeros(N,3);

       a = zeros(N,3);

       u = zeros(N,3);

       u_minus_g = zeros(N,3);

       j = 1; k = 1;

       elapsedt_checkpoint = zeros(size(wp,1),1);

       elapsedt_total = 0;

       step = zeros(1,size(wp,1));

       %% Loop

       % Point of confusion: is vmax for each velocity component or total

       % velocity?  If for total velocity,

       for i = 1:N

           if j==1 | abs(norm(r(i,:) - wp(j,2:4))) <=ep && j < size(wp, 1)

               Tx = abs(wp(j+1 , 2) - wp(j,2))/v_max;

               Ty = abs(wp(j+1 , 3) - wp(j,3))/v_max;

               Tz = abs(wp(j+1 , 4) - wp(j,4))/v_max;
```

```
    T = max([Tx Ty Tz]);
    j = j+1;
    step(j) = i;
end
if j ==size(wp,1) && abs(norm(r(i,:) - wp(j,2:4))) <= ep
    iterations_ab(a,b) = i;
end
vd(i,1) = (wp(j,2) - wp(j-1,2)) / T;
vd(i,2) = (wp(j,3) - wp(j-1,3)) / T;
vd(i,3) = (wp(j,4) - wp(j-1,4)) / T;
rd(i,1) = wp(j-1,2) + vd(i,1) * dt *k*(i-step(j));
rd(i,2) = wp(j-1,3) + vd(i,2) * dt *k*(i-step(j));
rd(i,3) = wp(j-1,4) + vd(i,3) * dt *k*(i-step(j));
u(i,1) = -Kp(a)*(r(i,1) - rd(i,1)) - Kd(b)*(v(i,1)-vd(i,1));
u(i,2) = -Kp(a)*(r(i,2) - rd(i,2)) - Kd(b)*(v(i,2)-vd(i,2));
u(i,3) = -Kp(a)*(r(i,3) - rd(i,3)) - Kd(b)*(v(i,3)-vd(i,3)) + g;
u_minus_g(i,1) = u(i,1);
u_minus_g(i,2) = u(i,2);
u_minus_g(i,3) = u(i,3) - g;
u1 = norm(u_minus_g(i,:));
if  abs(u1) > u_max
    u(i,:)= u_minus_g(i,:)/abs(u1) *u_max;
    u(i,3) = u(i,3) + g;
end
r(i+1 , 1) = r(i,1) + dt*v(i,1) + 0.5*dt^2*(u(i,1));
r(i+1 , 2) = r(i,2) + dt*v(i,2) + 0.5*dt^2*(u(i,2));
r(i+1 , 3) = r(i,3) + dt*v(i,3) + 0.5*dt^2*(u(i,3)-g);
r_error(i,1) = r(i,1) - rd (i,1);
r_error(i,2) = r(i,2) - rd (i,2);
```

```
            r_error(i,3) = r(i,3) - rd (i,3);
            v(i+1 , 1) = v(i,1) + dt*(u(i,1));
            v(i+1 , 2) = v(i,2) + dt*(u(i,2));
            v(i+1 , 3) = v(i,3) + dt*(u(i,3)-g);
        end
        % Find which Kp and Kd value produces least error
        error(a,b) = sum(abs(norm(r_error)));
        time(a,b) = error(a , b)/v_max;
        j = 1;
    end
end
%% Find optimal Kp and Kd
t_fastest = min(time,[],"all");
[row,col] = find(time==min(time(:)));
Kp_opt = Kp(col);
Kd_opt = Kd(row);
N = iterations_ab(row,col);
r = zeros(N , 3);
r(1,1) = wp(1,2);
r(1,2) = wp(1,3);
r(1,3) = wp(1,4);
rd = zeros(N , 3);
r_error = zeros(N,3);
v = zeros(N , 3);
vd = zeros(N, 3);
u = zeros(N,3);
u_minus_g = zeros(N,3);
%% Find path of optimal controls
for i = 1:N
```

```
    if j==1 || abs(norm(r(i,:) - wp(j,2:4)))<=ep && j < size(wp, 1)
        Tx = abs(wp(j+1 , 2) - wp(j,2))/v_max;
        Ty = abs(wp(j+1 , 3) - wp(j,3))/v_max;
        Tz = abs(wp(j+1 , 4) - wp(j,4))/v_max;
        T = max([Tx Ty Tz]);
        elapsedt_checkpoint(j) = i * dt;
        j = j+1;
        step(j) = i;
    end
    if j ==size(wp,1) && abs(norm(r(i,:) - wp(j,2:4))) <= ep
        elapsedt_total = i*dt;
        iterations = i;
    end
vd(i,1) = (wp(j,2) - wp(j-1,2)) / T;
vd(i,2) = (wp(j,3) - wp(j-1,3)) / T;
vd(i,3) = (wp(j,4) - wp(j-1,4)) / T;
rd(i,1) = wp(j-1,2) + vd(i,1) * dt *k*(i-step(j));
rd(i,2) = wp(j-1,3) + vd(i,2) * dt *k*(i-step(j));
rd(i,3) = wp(j-1,4) + vd(i,3) * dt *k*(i-step(j));
u(i,1) = -Kp_opt*(r(i,1) - rd(i,1)) - Kd_opt*(v(i,1)-vd(i,1));
u(i,2) = -Kp_opt*(r(i,2) - rd(i,2)) - Kd_opt*(v(i,2)-vd(i,2));
u(i,3) = -Kp_opt*(r(i,3) - rd(i,3)) - Kd_opt*(v(i,3)-vd(i,3)) + g;
u_minus_g(i,1) = u(i,1);
u_minus_g(i,2) = u(i,2);
u_minus_g(i,3) = u(i,3) - g;
u1 = norm(u_minus_g(i,:));
if  abs(u1) > u_max
    u(i,:) = u_minus_g(i,:)/abs(u1) *u_max;
    u(i,3) = u(i,3) + g;
```

```
    end
        r(i+1 , 1) = r(i,1) + dt*v(i,1) + 0.5*dt^2*(u(i,1));
        r(i+1 , 2) = r(i,2) + dt*v(i,2) + 0.5*dt^2*(u(i,2));
        r(i+1 , 3) = r(i,3) + dt*v(i,3) + 0.5*dt^2*(u(i,3)-g);
        r_error(i,1) = r(i,1) - rd (i,1);
        r_error(i,2) = r(i,2) - rd (i,2);
        r_error(i,3) = r(i,3) - rd (i,3);
        v(i+1 , 1) = v(i,1) + dt*(u(i,1));
        v(i+1 , 2) = v(i,2) + dt*(u(i,2));
        v(i+1 , 3) = v(i,3) + dt*(u(i,3)-g);

end
%% Plotting
v_mag = vecnorm(v');
v_norm = (v_mag- min(v_mag))/(max(v_mag) - min(v_mag));
figure(1)
scatter3(wp(:,2), wp(:,3),wp(:,4))
grid on
hold on
plot3(rd(:,1),rd(:,2),rd(:,3),'blue','LineWidth',2)
h = plot3(r(:,1),r(:,2),r(:,3),'Red','LineWidth',2);
% Initialize video writer
V = VideoWriter('animated1.avi');
open (V);
for k = 2:N
   set(h, 'XData',r(1:k,1),'YData',r(1:k,2),'ZData',r(1:k,3));
   pause(0.00005 / v_norm(k));
   frame = getframe(gcf);
   writeVideo(V,frame);
```

```
end
close(V);
hold off
xlim([0 10])
ylim([0 10])
%% Optional Visual (Kinda Slow)
cmap = jet(256);
figure(2)
scatter3(wp(:,2), wp(:,3),wp(:,4))
hold on
for k = 1:size(r,1)-1
   colorIndex = round(v_norm(k) *255) + 1;
   plot3(r(k:k+1,1), r(k:k+1,2),r(k:k+1,3),
'Color',cmap(colorIndex,:),'LineWidth',2);
end
grid on
xlim([0 10])
ylim([0 10])
colormap(cmap);
clim([min(v_mag) max(v_mag)]);
colorbar;
xlabel('X')
ylabel('Y')
zlabel('Z')
title('Path of the object with velocity gradient');
```