# Team 1: Balsa Steel

## Final Report

MAE 157A — Aerospace Design Laboratory

prepared for

Dr. Brett Lopez, Brandon Dee, Aaron John Sabu — University of California, Los Angeles

by


| | |
|---|---|
| Lucas Beyer | Manufacturing |
| Kyle Brown | Controls |
| Bennett Ji | Design |
| Robert Lee | Guidance |
| Daniel Paul | Electronics |
| Sophie Ye | Simulation |

## Table of Contents

## Abstract

In this report, the various processes that our team, Balsa Steel, went through to make our micro-air vehicle (MAV), are detailed. The initial step involved was the design and manufacturing of the hardware for the drone. The MAV's components were mounted on a carbon fiber and balsa wood composite frame manufactured using a waterjet. The motors and propellers were characterized through testing allowing us to optimize our selection. With the components assembled onto the frame, the electronics were soldered together to complete the hardware side of the MAV. On the software side, a dynamic programming solution was developed to efficiently and optimally sequence the race waypoints and the gains of the MAV's outer loop PD controller were chosen and validated through simulation. The telemetry data from the MAV race are analyzed and our takeaways from this project and this class are discussed.

## Vehicle Design & Manufacturing

### Conceptualization

The earliest stage of MAV design was presented during the concept design review (CoDR) and consisted of a baseline organization of the flight deck. This was a rough sketch, seen in Figure 1, based on understandings of weight distribution and component requirements in the stack. The battery and lidar are the two heaviest components weighing in at 460 and 267 grams, respectively. These were separated to be on either side of the chassis with the lidar being mounted at the very top to allow for an unobstructed field of view. The flight computer was mounted on the opposite side of the battery to take advantage of the fact that carbon fiber is not RF transparent and block some noise generated by the battery. That aside, the remaining components were seated below the chassis based on convenience.

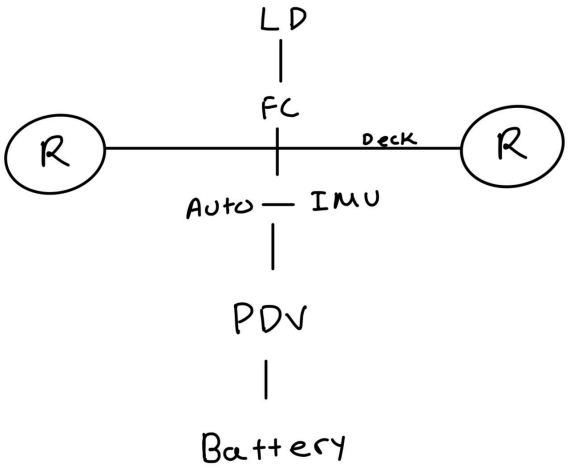


**Figure 1:** Conceptual design review MAV flight stack.

## Preliminary Design

This layout remained fairly consistent throughout the design process, though following the preliminary design review (PDR) some minor changes were made. The PDR included a detailed vehicle CAD with the flight deck configuration presented in the CoDR. This can be found in Figures 2 and 3 below. The overall design closely mirrors the sketched flight stack design, though the power distribution board (PDB), inertial measurement unit (IMU), flight controller, and electric speed controller (ESC) were all mounted at the same level on the upper battery mount. Notably the motors are mounted below the chassis to allow for unobstructed airflow being pushed down by the propellers.

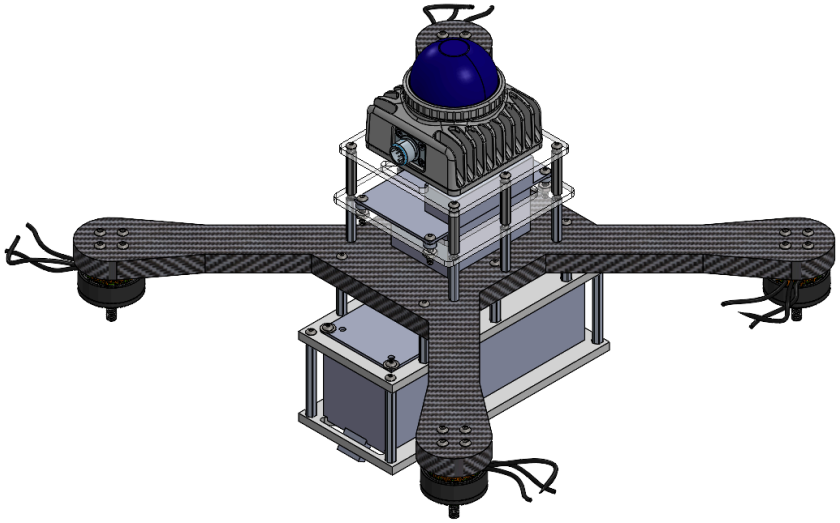

**Figure 2:** Preliminary design review vehicle design isometric view.

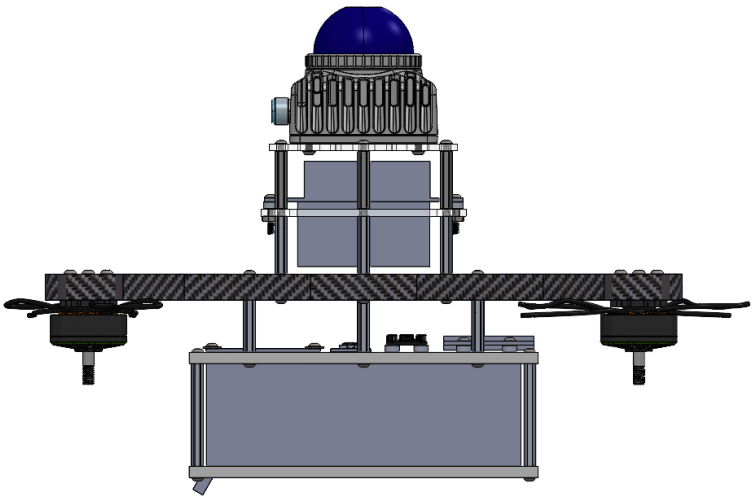

**Figure 3:** Preliminary design review vehicle design side view.

## Final Design

Following the PDR, a few minor changes were made based on feedback received from the instructional team. In the vehicle design for the PDR, the PDB, IMU, flight controller, and ESC were all mounted using fasteners; however, this was substituted for foam adhesive in the final design as the additional strength offered by fasteners not worth the weight and added integration complexity for such lightweight components. The other change was swapping the location of the flight controller and IMU in order to better center the IMU at the center of gravity of the vehicle. The last addition to the vehicle was attaching legs to the bottom battery housing bracket so that the vehicle has something to land on; this was not yet CADed in the PDR but accounted for in terms of design. Renders of the final design can be found below in Figures 4 and 5. The colors in the render are matched to the corresponding PLA and acrylic colors used in manufacturing.

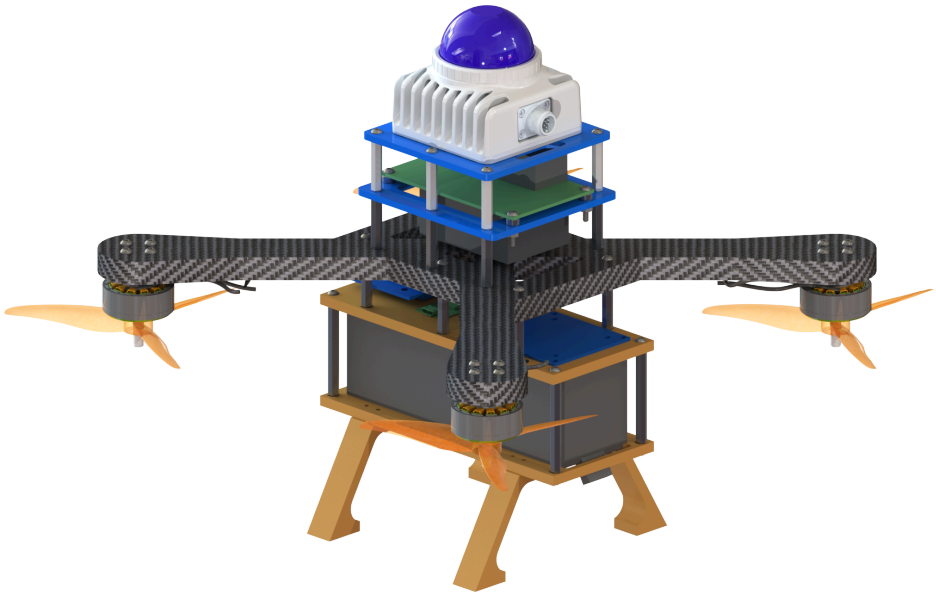

**Figure 4:** Final vehicle design render isometric view.

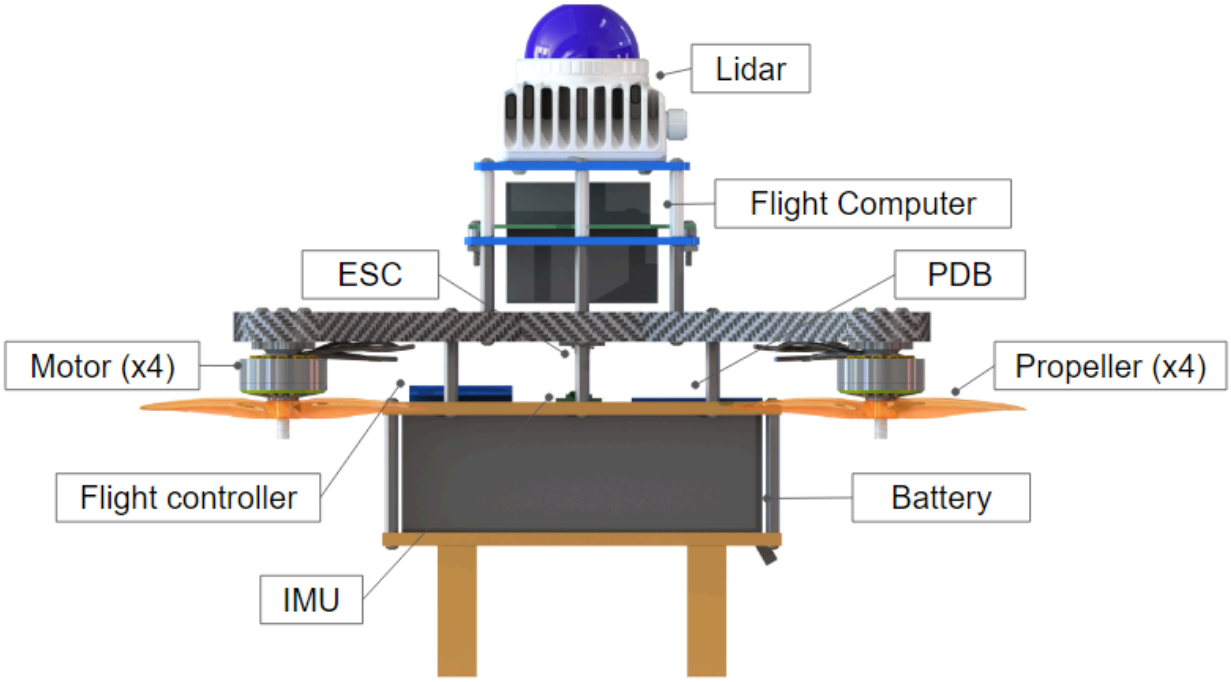


**Figure 5:** Final vehicle design render labeled side view.

Looking a bit into the design philosophy of various components, all components below the chassis are designed for 3D printing while parts above use laser cutting. The components below the chassis either have non-planar geometries or are too thick to be laser cut. The PLA battery brackets are designed with indents so the battery is secure in all axial frames during flight; the top and bottom brackets are separate pieces to allow for easy adjustment to any battery spec inaccuracies. Above the chassis, all mounting plates use laser cut acrylic pieces which allow for quick and precise manufacturing; since the lidar and flight computer are fastened with screws, indenting is not necessary. In general, standoffs are used to space out components since they are standard easy to source COTS parts.

The final vehicle weighed 1388 grams compared to a preliminary estimate of 1308 grams; though a few fasteners were removed from the design, additional PLA legs were added as well as additional harnessing which likely accounts for the difference. Figures 6 and 7 show a top and a side view of the vehicle showing the location of the center of mass—denoted by the alternating black and white circle. The manufacturing methods used for in-house components are tabulated in Table 1.

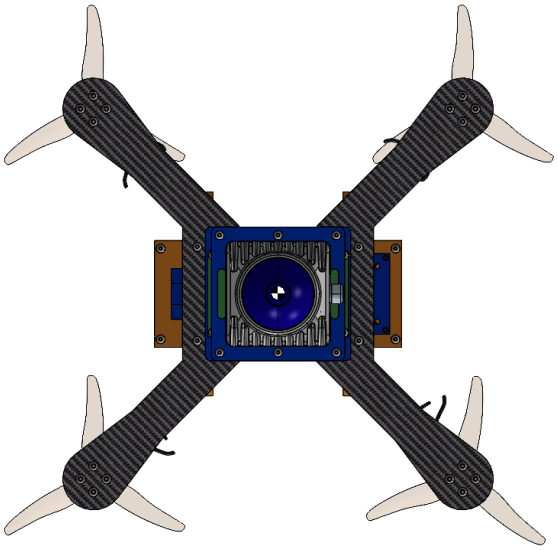

**Figure 6:** Vehicle top view with center of mass.

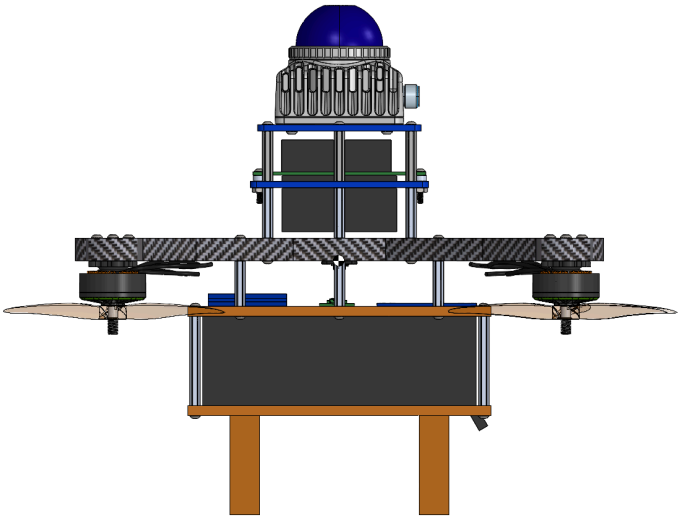

**Figure 7:** Vehicle side view with center of mass.

**Table 1:** Manufacturing methods for each in-house fabricated component.

| **Component** | Balsa Core CF Frame | Acrylic Stages | PLA Stages |
|---|---|---|---|
| **Fabrication Method** | Waterjet | Laser Cutter | 3D Printer |

## Theoretical vs Experimental Thrust Profile

The analysis aimed to compare the theoretical thrust generated by a MAV with the experimental thrust observed during flight. Understanding the differences between these thrust profiles helps in assessing the accuracy of the MAV's control systems and identifying potential areas for improvement.

The experimental and theoretical thrust profiles were plotted over time in Figure 8, allowing for a visual comparison of the MAV's performance. The experimental thrust was estimated using a hypothetical relationship based on the square of the velocity components. The graph shows how closely the experimental thrust matches the theoretical thrust over the duration of the flight. Generally, the two lines should follow a similar trend if the MAV's control systems are performing well. Deviations between the two lines indicate discrepancies between the expected and actual thrust values. Smaller deviations suggest accurate control and prediction of thrust, while larger deviations highlight potential issues or disturbances affecting the MAV's performance. The theoretical thrust was calculated using assumed motor and propeller characteristics, specifically a motor constant and propeller efficiency factor.

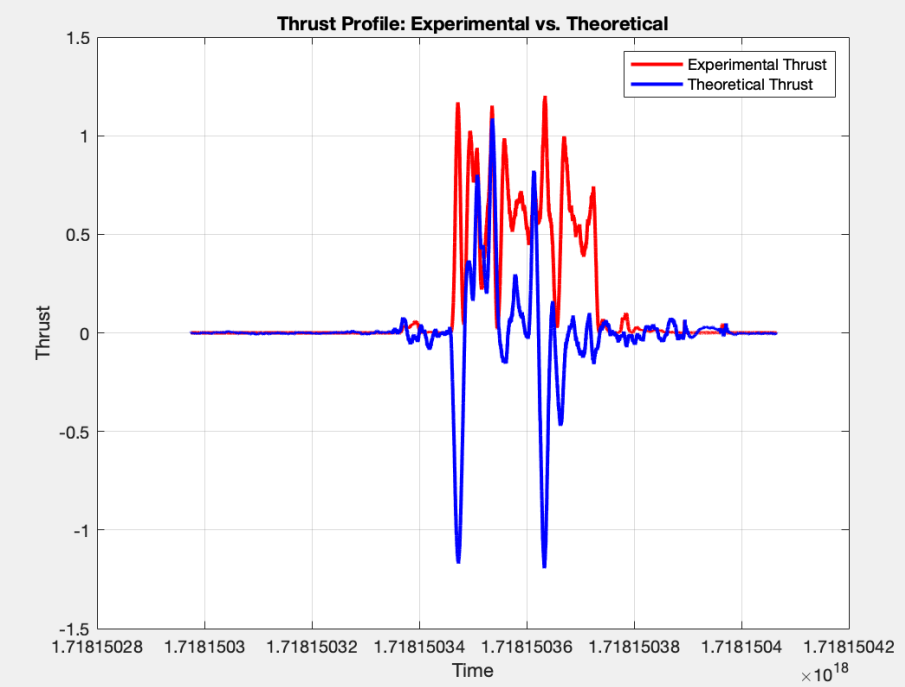


**Figure 8:** Thrust profiles over time for both the experimental (red) and theoretical (blue) thrust values.

The absolute error between the experimental and theoretical thrust values was calculated and plotted to quantify the deviations as seen in Figure 9. The error graph provides a clear visualization of the differences between the experimental and theoretical thrust at each time point. Lower values indicate that the MAV's actual thrust is closely aligned with the theoretical predictions. Peaks in the error graph represent instances where the MAV's thrust deviated significantly from the expected values. These peaks can be critical for identifying specific moments during the flight when the MAV experienced control challenges or external disturbances. Analyzing the error graph helps in understanding the overall performance of the MAV and identifying periods where improvements in control algorithms or system calibration may be necessary.

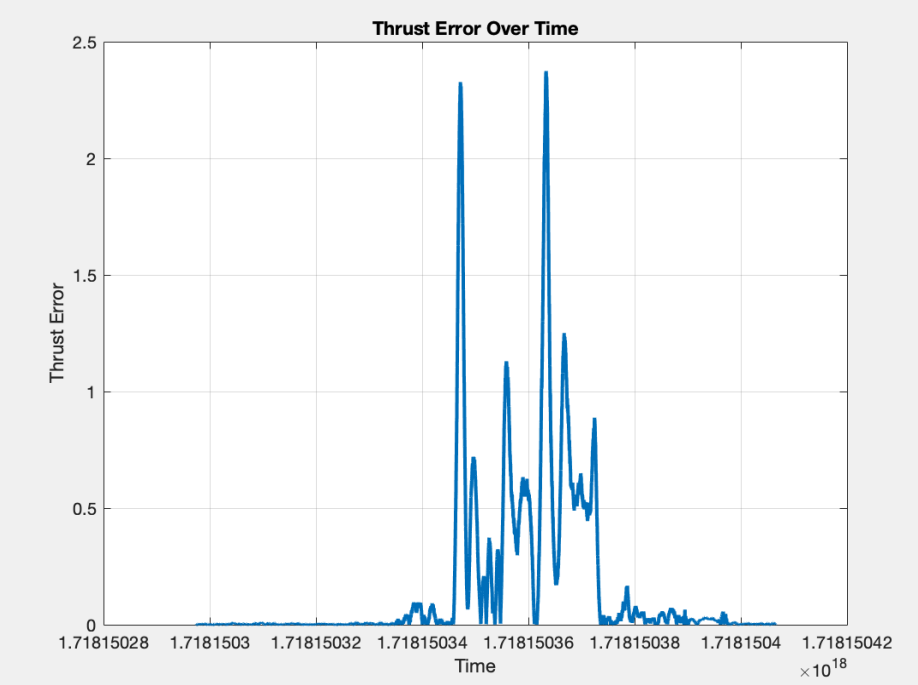


**Figure 9:** The absolute error between the experimental and theoretical thrust values over time.

The results highlight several important aspects of the MAV's performance. The mean thrust error of 0.167 suggests that, on average, the MAV's actual thrust closely matches the theoretical predictions. This indicates that the control systems are generally effective in maintaining the desired thrust levels. However, the maximum thrust error of 2.374 points to specific instances where the MAV's thrust significantly deviated from theoretical expectations as seen in Figure 9. These deviations could be attributed to various factors, such as sudden changes in flight dynamics, environmental disturbances, or limitations in the control algorithms.

Overall, the analysis demonstrates that the MAV performs well in terms of maintaining the desired thrust, with relatively low average deviations. The instances of larger deviations, as indicated by the maximum thrust error, suggest that there are still areas for improvement in the MAV's control systems. Future work should focus on refining the control algorithms and addressing potential sources of disturbance to minimize these deviations and enhance overall flight stability.

## Electronics Overview

### Components

Our electronics system consists of several components. First is the battery, which in our case is a 5000 mAh 4S 14.8V LiPo battery. It has a hard case to protect it from drops. We then have our 4S speed controller that controls the 4 motors, which are F60 Pro V-LV 1950Kv motors. Our Flight controller that controls the ESC and the motors is a Teensy 4.0. The flight controller is connected to the IMU which is an Adafruit MPU 9250. Our flight computer and LiDAR are the Jetson Orin Nano and the Livox Mid-360.

A basic power/data block diagram is shown in Figure 10. The battery is connected to the Power Distribution Board (PDB). The PDB supplies power to everything on the drone. It powers the Electronic Speed Controller (ESC), which powers each of the 4 motors through a PWM connection. The PDB also powers the voltage regulator, which changes the voltage so that it can be used for the flight computer and the LiDAR. The flight computer powers the flight controller which it sends data to and from, while also receiving data from the LiDAR. THe flight controller sends power and receives data from the IMU. The flight controller also controls the speed controller via a data connection.

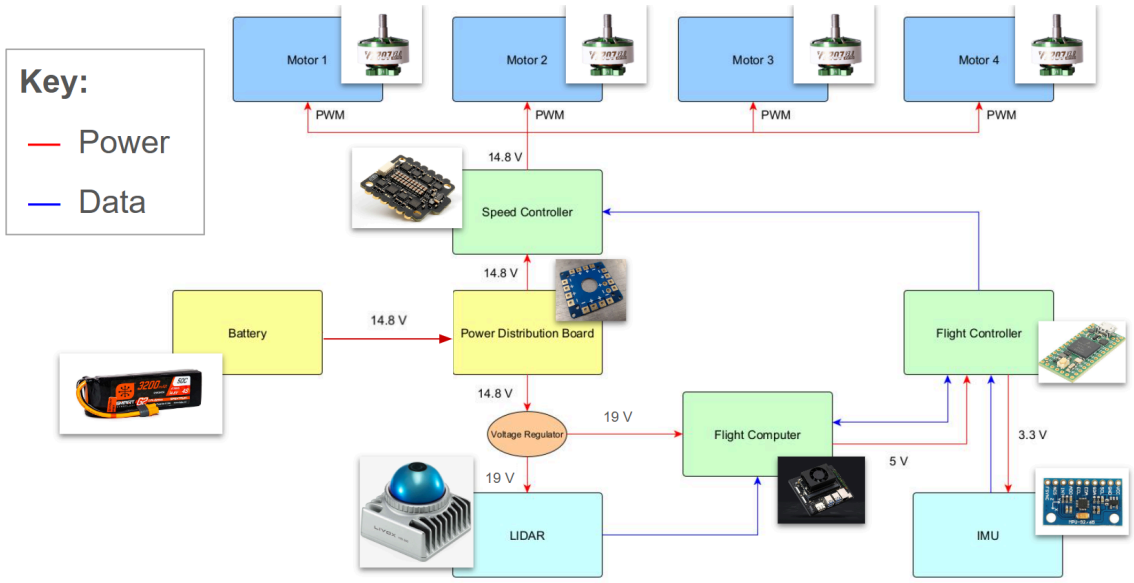


**Figure 10:** Power (red) and data (blue) block diagram

## Design Changes

In Figure 11, the flight computer, ESC, and flight controller are shown. This design was changed as the ESC's pins to the flight controller broke frequently, and so the plug was used instead of the pins present. Breakout pins were soldered onto the flight controller to connect it to the various components. Besides that, and some wires being modified to be longer, our electronics that we worked on mostly stayed the same throughout the design process.

When designing our electronics layout, we initially wanted to have our own Voltage Regulator Modules and a capacitor in parallel with the battery to allow a smoother flow of current in case of any abrupt voltage fluctuations. However, we did not end up implementing them ourselves, and we ended up using the VRMs for the flight computer and LiDAR that everyone else in the class used.

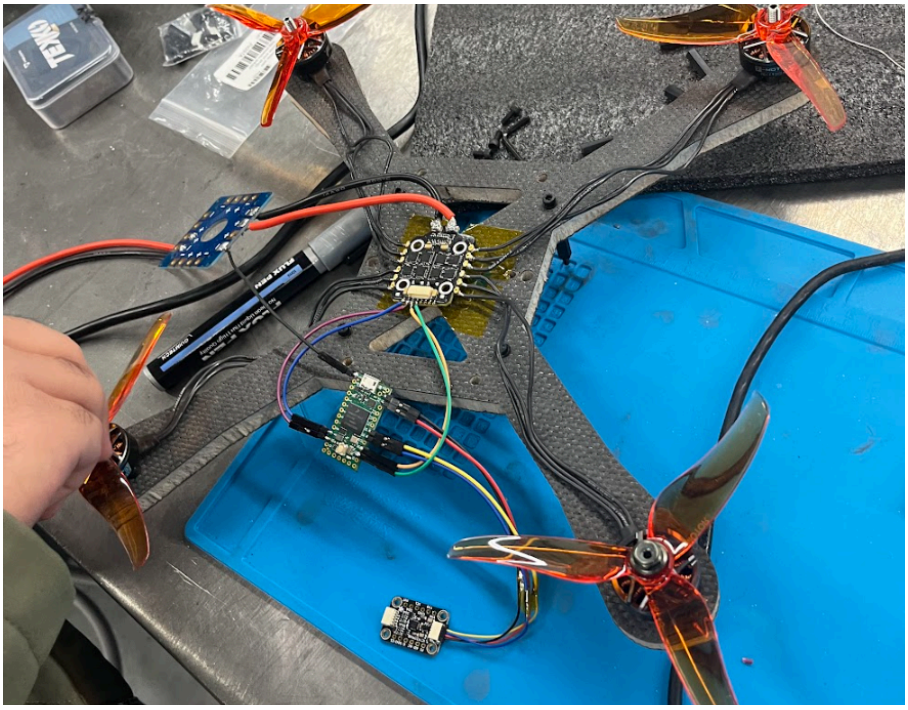

**Figure 11:** Photo of the MAV electronics during assembly.

## Manufacturing

Our parts were all soldered together to create a lasting bond between the electronic components, as seen in Figure 12. Plugs and pins were used on places like the flight controller or the power for the LiDAR/Flight Computer where we didn't want to have a permanent connection with solder. We frequently soldered and resoldered parts whenever we moved the hardware around in order to have a good fit.

**Figure 12:** Photo of soldering process.

## Guidance Design & Analysis

### Background

The MAV race consists of up to 11 predefined waypoints, the first and last of which are [0, 0, 1]. The drone must visit all waypoints starting at the first waypoint and ending at the last. The remaining "free" waypoints can be visited in any order. The goal of the MAV race is to minimize the time it takes to visit all waypoints, which suggests the optimization of the waypoint visit sequence.

To approach this optimization problem, we first reduce the problem from a minimum-time optimization to a minimum-distance optimization by assuming the MAV travels at roughly a constant velocity near the maximum speed of 1 m/s. The waypoints can then be represented as an undirected fully-connected graph, where each node represents a waypoint and each edge represents the Euclidean distance between the two nodes it connects. By choosing this representation, the optimization problem reduces to finding the shortest path that visits all nodes, starting at the first node and ending at the last node. This problem is classically known as the shortest hamiltonian path problem (SHP) with constraints on the end location or the traveling salesman problem (TSP)—given the fact that the start and end waypoints are at the same location.

### Shortest Hamiltonian Path Algorithms

There are many algorithms and heuristics for the SHP. In this project, we consider three primary methods—nearest neighbor, brute force, and dynamic programming—and compare their relative strengths and weaknesses.

The nearest neighbor method is a heuristic that greedily chooses the nearest unvisited node from the current node, starting from the designated start node. Though this heuristic has a low runtime with a time complexity of $O(n^2)$ allowing it to run quickly for up to n = 5000 waypoints, it does not guarantee to return the optimal path.

The brute force method is an algorithm that tries all possible paths by checking every permutation of the free nodes. The MATLAB built-in method "perms" is used to generate all permutations. It guarantees correctness but has a high time complexity of $O(n \times (n-2)!)$[1] which means it can run quickly for up to n = 13 waypoints.

Finally, the dynamic programming method is an algorithm similar to the brute force method, but uses dynamic programming concepts to reduce the time complexity. The general idea is to use the solution to smaller subproblems to solve larger problems. The subproblem dp[subset][i] is the shortest hamiltonian path over a subset of the nodes that ends at the i-th node. This way, each subproblem can be solved by smaller subproblems by the following formula:

$$dp[subset][i] = \min_{s \text{ in subset}, s \,!= i} (dp[subset - i][s] + w[s][i]).$$

[1] There are n-2 free nodes and it takes O(n) operations to compute the path length.

This method has a time complexity of $O(n^2 \times 2^n)$ which means it can run quickly for up to n = 24 waypoints.

Another method that was briefly considered is ant colony optimization (ACO), a heuristic that takes a biological approach to finding the shortest path. In this method, each edge in the graph is initially weighted equally. Then, it repeatedly sends out an "ant", which walks from one node to the next at random, taking edges with greater weight with a higher probability. Based on the final path length of the ant, the weights of the edges that it took are adjusted inverse proportionally. This ultimately promotes paths with shorter path lengths. ACO is also a heuristic and is typically applied to large-scale problems (n > 5000), so it was not investigated in depth for this project.

## Theoretical Performance

We evaluate the performance of these three methods by considering their runtime (Figure 13) and the relative path length of the nearest neighbor method against the optimal path (Figure 14). The growth of the time complexity for each method is as expected, with polynomial growth for nearest neighbor, exponential growth for dynamic programming, and super-exponential (factorial) growth for brute force. Even though the nearest neighbor method is significantly faster than the other two methods, on average for 11 randomly chosen waypoints, it returns a solution 8% longer than the optimal solution. Figure 13 also demonstrates the average path length excess from the nearest neighbor method grows with the number of waypoints. Since the MAV race will only consist of up to 11 waypoints, the dynamic programming method was chosen due to the optimality of its solution and its speed.

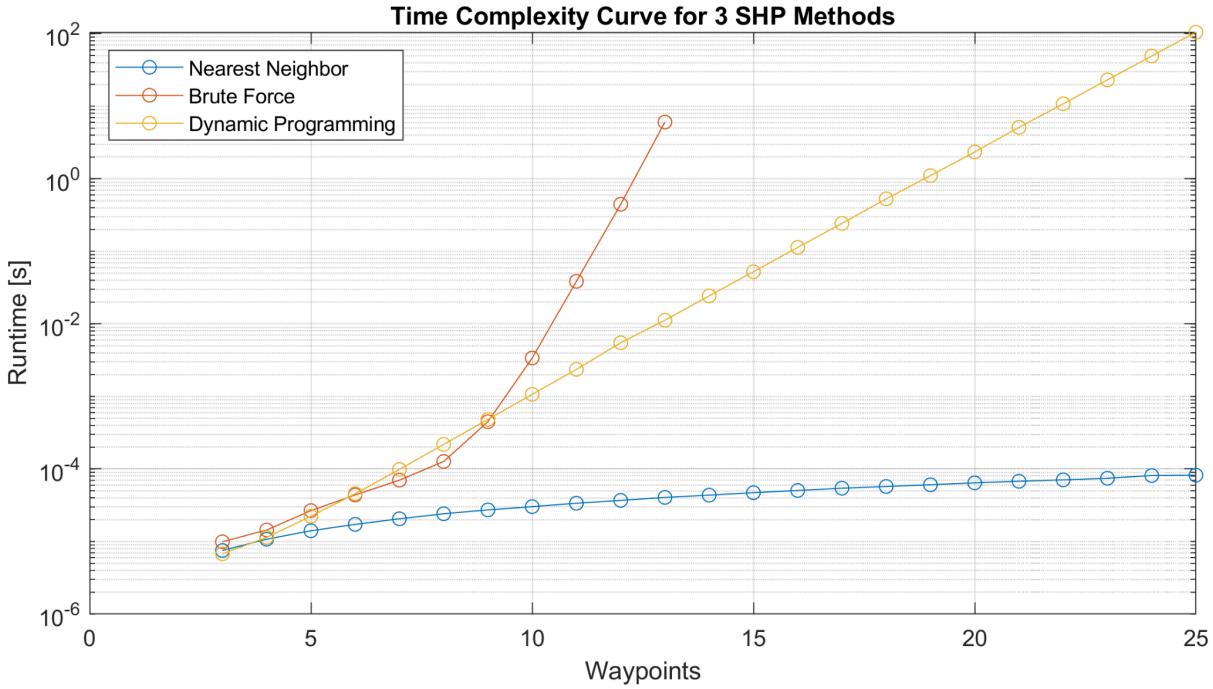


**Figure 13:** Runtime of the nearest neighbor (blue), brute force (orange), and dynamic programming (yellow) SHP methods for 3 to 25 waypoints. Brute force was only run for 13 or less waypoints due to memory limits. For timing accuracy, methods were run for 10,000 iterations or at least 1 second, whichever occurs first.


**Figure 14:** Relative path length excess of solution returned by the nearest neighbor method compared to the optimal path. For each number of waypoints, 1000 randomly generated waypoint sets were evaluated. Error bars represent the first and third quartile of the error.

## Flight Course Results

The MAV race consisted of three flight courses, shown in Figures 15–17, with the optimal and suboptimal (flown in the MAV race) sequences listed in Table 2. An error in implementation resulted in a suboptimal sequence being selected for the flight courses. This was due to the fact that the SHP algorithm did not consider the length between the second-to-last and last waypoint, which has been corrected in the final algorithm. Assuming a time-distance proportionality, if the optimal paths were flown, 1.52 s could have been shaved off our average flight time.

**Table 2:** Optimal and suboptimal (flight) waypoint sequences, along with their path lengths and relative path error, for each flight course.

| Flight Course | Optimal Sequence | Optimal Length | Suboptimal (Flight) Sequence | Suboptimal Length | Path Increase [%] |
|---|---|---|---|---|---|
| 1 | 1, 2, 5, 6, 3, 4, 7 | 22.75 | 1, 2, 4, 3, 6, 5, 7 | 25.35 | 11.43 |
| 2 | 1, 9, 10, 5, 7, 4, 2, 8, 3, 6, 11 | 27.42 | 1, 6, 3, 8, 2, 9, 10, 4, 7, 5, 11 | 27.68 | 0.94 |
| 3 | 1, 5, 8, 7, 4, 2, 6, 3, 9 | 18.68 | 1, 3, 5, 8, 6, 2, 4, 7, 9 | 19.56 | 4.75 |

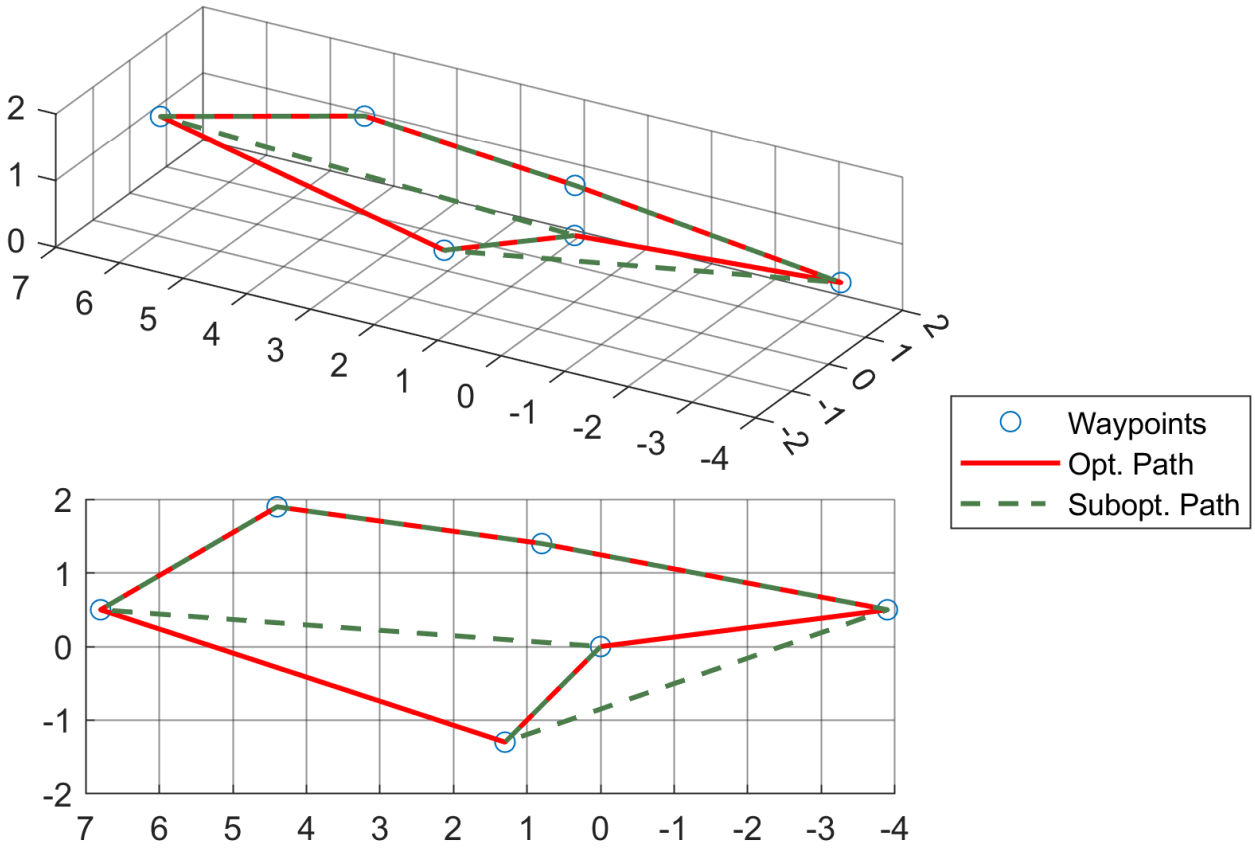


**Figure 15:** Isometric view (top) and top-down view (bottom) of flight course 1. Optimal and suboptimal flight paths are overlaid on the waypoints.

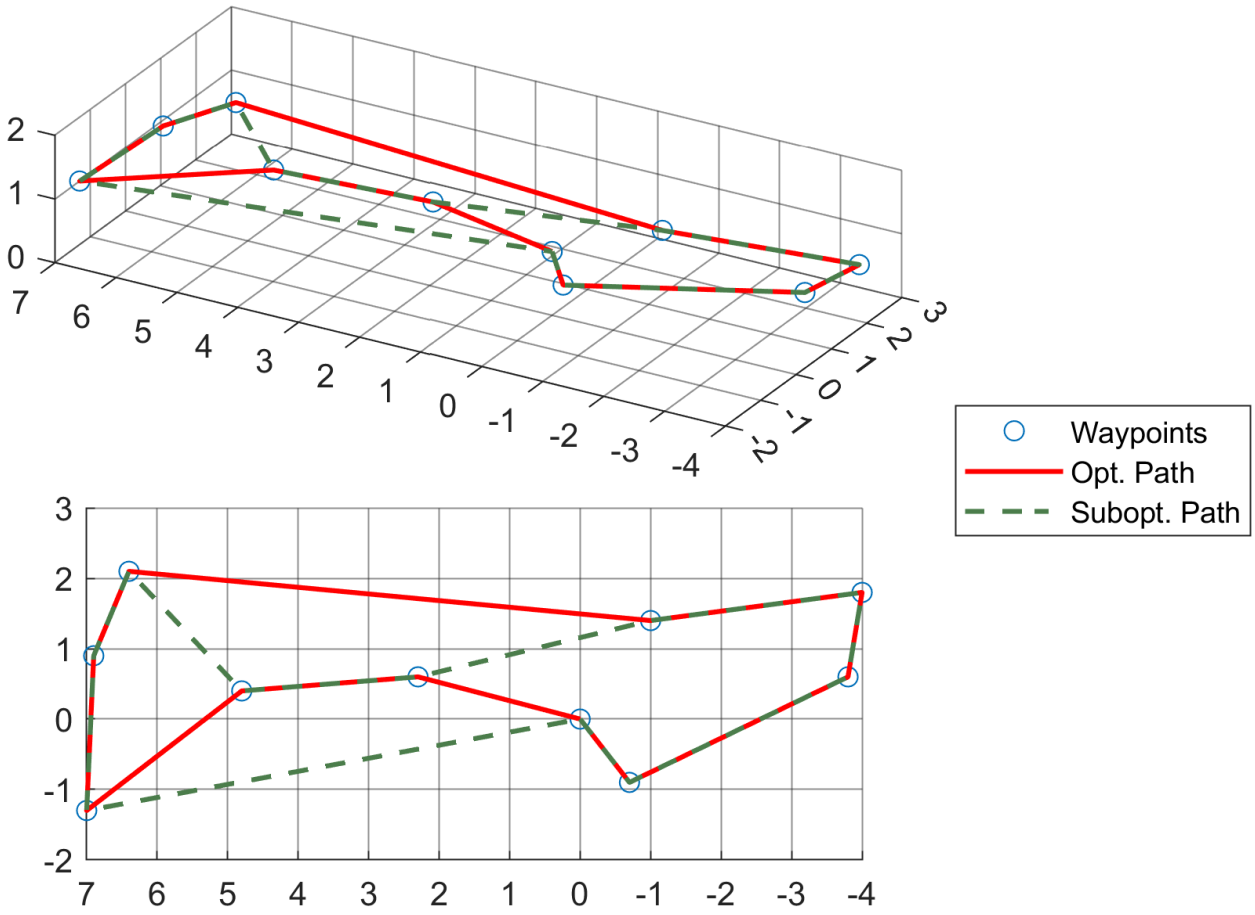


**Figure 16:** Isometric view (top) and top-down view (bottom) of flight course 2. Optimal and suboptimal flight paths are overlaid on the waypoints.

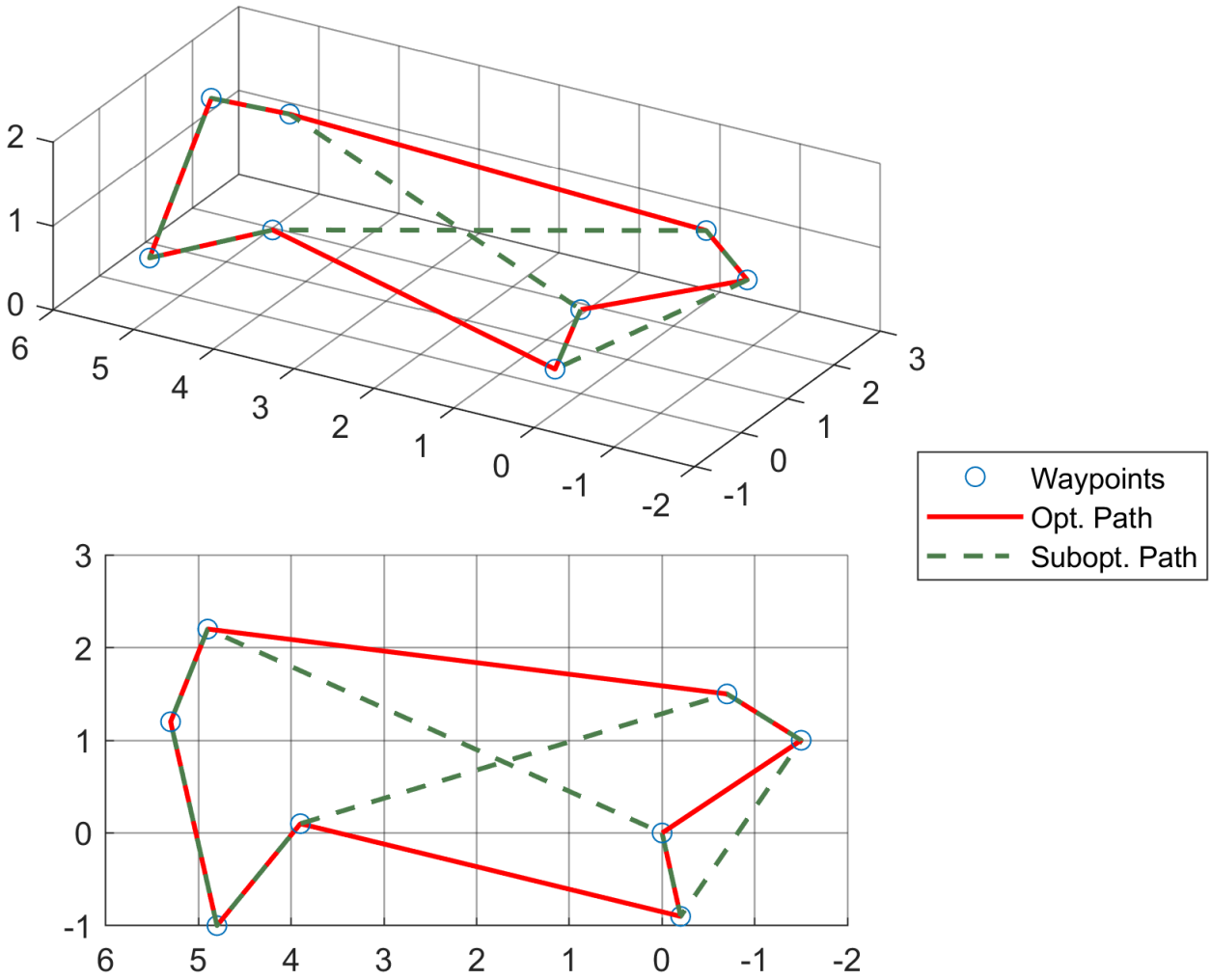


**Figure 17:** Isometric view (top) and top-down view (bottom) of flight course 3. Optimal and suboptimal flight paths are overlaid on the waypoints.

## Control & Simulation Analysis

### Simulation Structure and Pseudocode

The simulation code models the motion of the drone tracking a target trajectory path and velocity between a given set of waypoints; the block diagram is given in Figure 18. The drone is modeled as a point mass, and target trajectories and velocities transition in an abrupt, non-physical way when a waypoint is reached to simplify trajectory calculations.

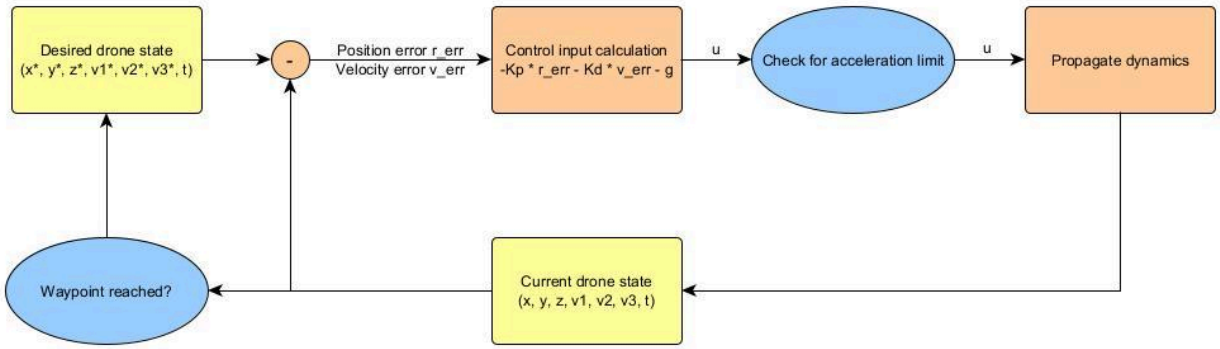


**Figure 18:** A block diagram overview of the simulation structure.

The simulation code was based on pseudocode provided by the instructional team. Some minor modifications were added to improve simulation performance and adaptability to the characteristics of different waypoint sets. All simulation code is attached in the appendix.

### Trajectory Calculation and Correction

The trajectory is calculated by computing the straight line between the current waypoint and next waypoint. The calculated trajectory always assumes that the drone is traveling at its maximum speed, and uses this assumption to estimate the time needed to travel between two waypoints.

Initially, at each timestep, the ideal position was calculated using kinematic equations and the time passed since the drone last crossed a waypoint. The ideal velocity was assumed to be the maximum allowed velocity of the drone. Using this implementation, if the drone missed a waypoint, the simulation could not recover, and the drone would not finish visiting all waypoints. This most commonly occurred in waypoint orderings that involved abrupt changes in direction; for example, if the third waypoint is [-4.3, 2.3, 1.2], and the fourth waypoint is [3.4, -1.2, 1.5], it was likely that the drone would miss the fourth waypoint and not visit all waypoints.

This issue arose because the "ideal" position computed by the simulation did not appropriately account for the position of the drone relative to the target waypoint. To address this, an additional check was added to see if the drone's position was within a 0.5 m threshold of the target waypoint's coordinates. If yes, then the ideal location would be set to the target waypoint's position in that coordinate axis, and the ideal velocity would be set to zero. An overview of this control flow is shown in Figure 19.

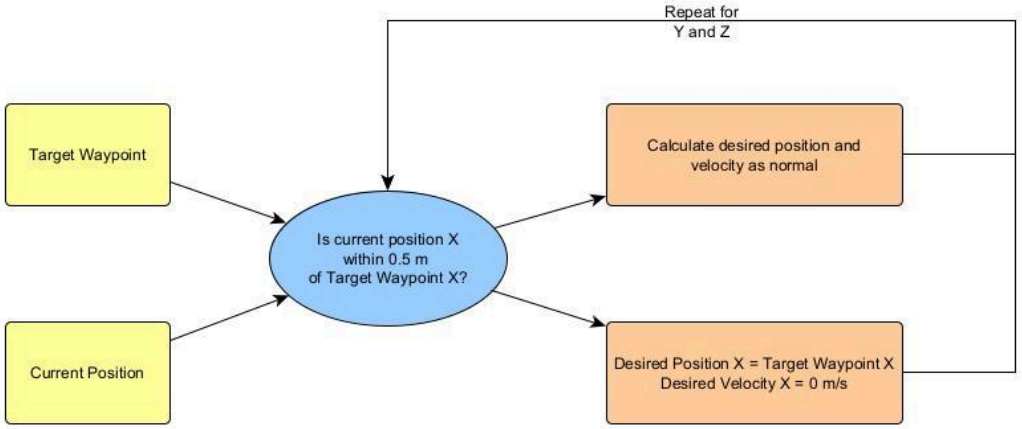


**Figure 19:** The control flow used to account for the drone's current position relative to the target waypoint.

An even better way to address this issue would be to use a smarter trajectory computation that accounts for the drone's velocity when it passes a waypoint, as well as its current position relative to the position of the target waypoint. This would lead to the computation of physically realistic trajectories that can appropriately account for position and velocity overshoot.

## Gain Optimization

Simulation gains are intended to minimize the total "distance error at waypoints" (i.e. the position error computed when a waypoint is considered "reached"), and were found via Monte Carlo trials over randomly generated waypoint sets. A key assumption in the implementation of the Monte Carlo trials as done here is that the waypoint distance error as a function of $K_p$ and $K_d$ has a singular, global minimum for a given set of waypoints, and i.e. the function $e_{wp}(K_p, K_d)$ will look similar to Figure 20. Note that the location of this minimum is not necessarily the same for each set of waypoints.

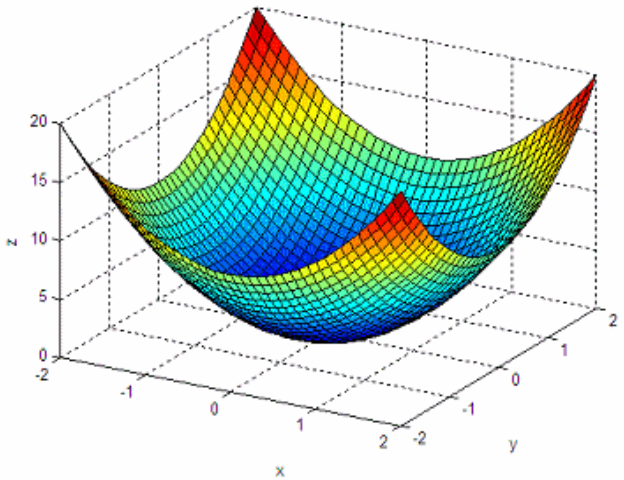


**Figure 20:** The gain optimization process assumes that the function $e_{wp}(K_p, K_d)$, which represents the total distance error at waypoints, has a global minimum for a given set of waypoints. In this plot, $e_{wp}$ would be the $z$-axis, and $(K_p, K_d)$ would be the $x$ and $y$ axes. Image source.

Each Monte Carlo trial involved a brute-force grid search, in which ($K_p$, $K_d$) were varied over some range at some increment. The full Monte Carlo simulation was run twice, first with a coarser grid of $K_p$ = [1, 25] and $K_d$ = [1, 25] with gains incremented between integers. The gains were limited to this range as a consideration for whether or not the control system actuators (motors) would realistically be able to produce the inputs that would result from a higher gain. The results from this first Monte Carlo simulation were then evaluated, and the search space for ($K_p$, $K_d$) was refined to $K_p$ = [20, 25] and $K_d$ = [15, 18] based on what ranges of values were most frequently "optimal" across all Monte Carlo trials in the simulation. This second search space was then searched at increments of 0.1.

## Simulation and Gain Optimization Results

After running both Monte Carlo simulations, it was found that the gains of ($K_p$, $K_d$) = (24.4, 15.1) best addressed a wide variety of waypoint combinations.

A few example plots from the same Monte Carlo trial (i.e. same waypoint set) are shown in Figures 21–23. The results using ($K_p$, $K_d$) = (24.4, 15.1) are not included here, since this selection is merely intended to demonstrate how gain variation affects the resulting trajectory. A few things to note:

1. Different gain pairings result in different amounts of overshoot, both in position and velocity. Overshoot was not considered in this particular optimization process, but is generally a useful parameter to consider in gain selection.
2. Different gain pairings result in different "recovery times" after the drone passes and overshoots a waypoint; it may take much longer to begin tracking the desired trajectory again.
3. The control input can vary very significantly between different ($K_p$, $K_d$) combinations. In some combinations, the control input is frequently limited to ensure that the net acceleration of the drone does not exceed 4 m/s$^2$.

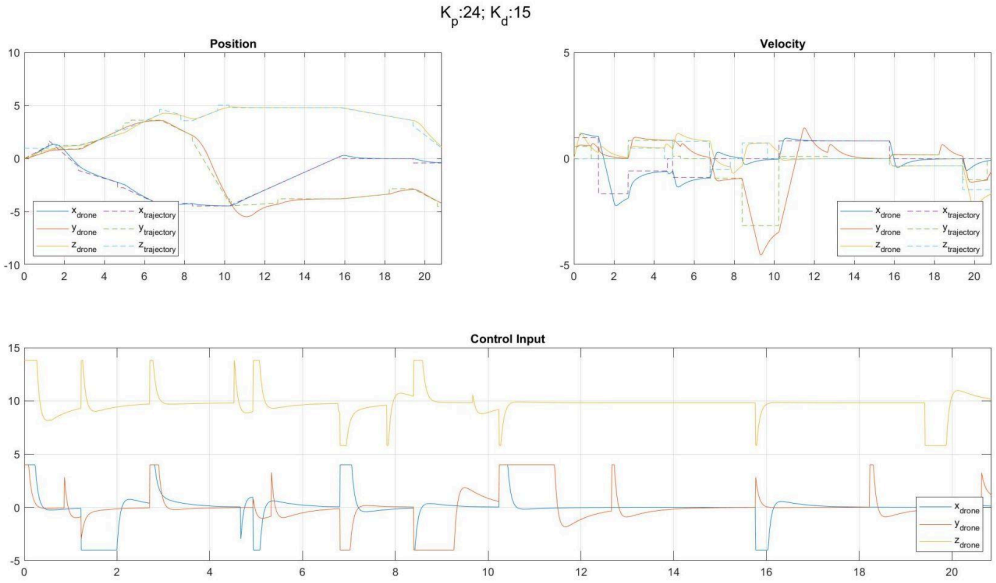


**Figure 21:** The position, velocity, and control input produced by the trajectory simulation using gains of ($K_p$, $K_d$) = (24, 15).

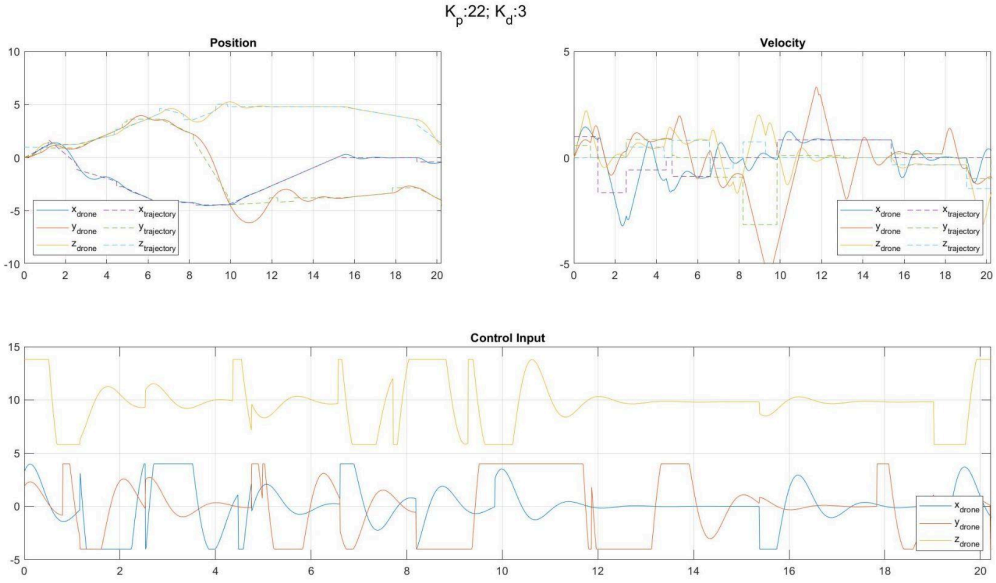


**Figure 22:** The position, velocity, and control input produced by the trajectory simulation using gains of $(K_p, K_d) = (22, 3)$.

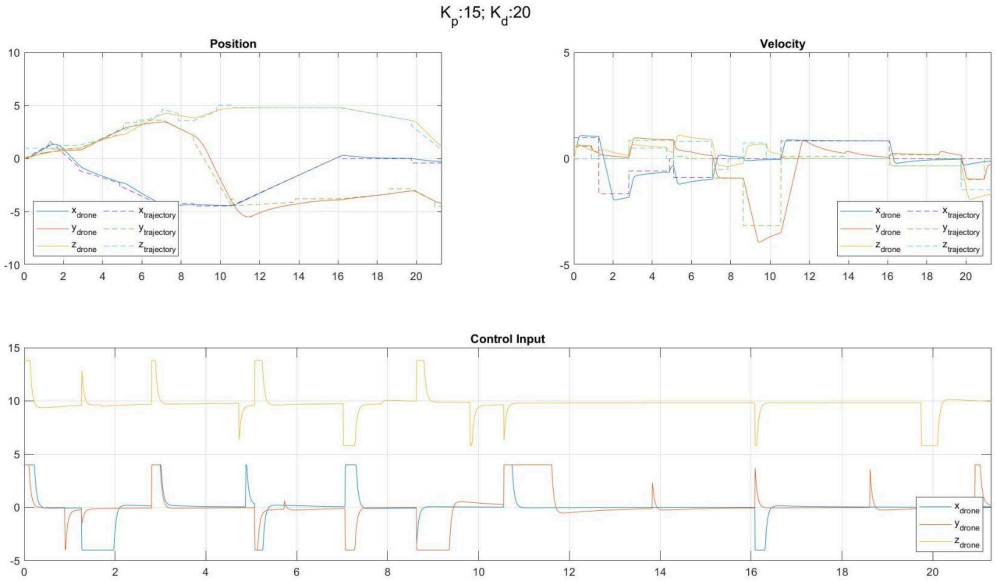


**Figure 23:** The position, velocity, and control input produced by the trajectory simulation using gains of $(K_p, K_d) = (15, 20)$.

## Flight Analysis

Ensuring that MAVs follow desired trajectories and maintain stable flight conditions is crucial for their effective operation. This analysis aims to evaluate the performance of an MAV across three different flight courses by comparing actual flight data with desired flight data. To do this we analyzed multiple subjects of the flight patterns for each course including: Trajectory Adherence as seen in Figures 24–26, Velocity Matching as seen in Figures 27–29, Error Quantification as seen in Figures 30–32, and Stability/ Control. The analysis involves plotting and interpreting several key metrics derived from the MAV's flight data. The data includes actual and desired values for positions, velocities, and orientations across three different courses.

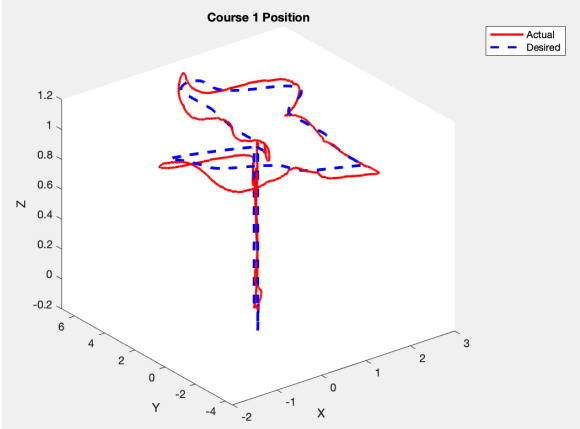


**Figure 24:** The 3D position trajectory of the MAV for Course 1. The red line represents the actual position of the MAV, while the blue dashed line represents the desired position. It helps visualize how closely the MAV follows the desired trajectory in three-dimensional space.

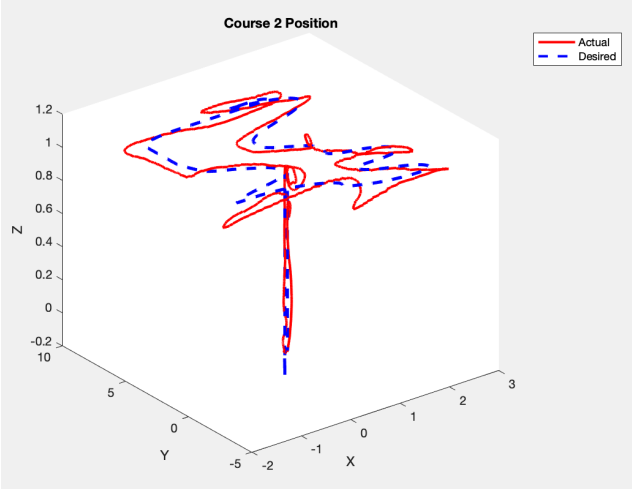


**Figure 25:** The 3D position trajectory of the MAV for Course 2. The red line indicates the actual position, and the blue dashed line indicates the desired position. This comparison highlights the accuracy of the MAV's path following for Course 2.

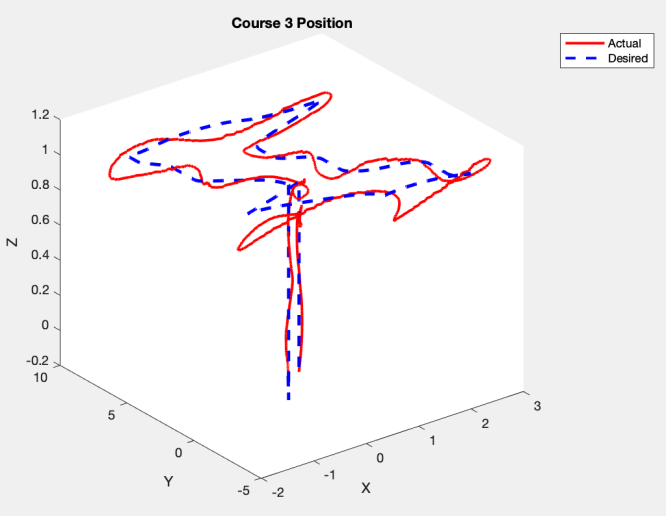


**Figure 26:** The 3D position trajectory of the MAV for Course 3. The red line is the actual position, and the blue dashed line is the desired position. This helps in understanding how well the MAV maintains its intended flight path for Course 3.

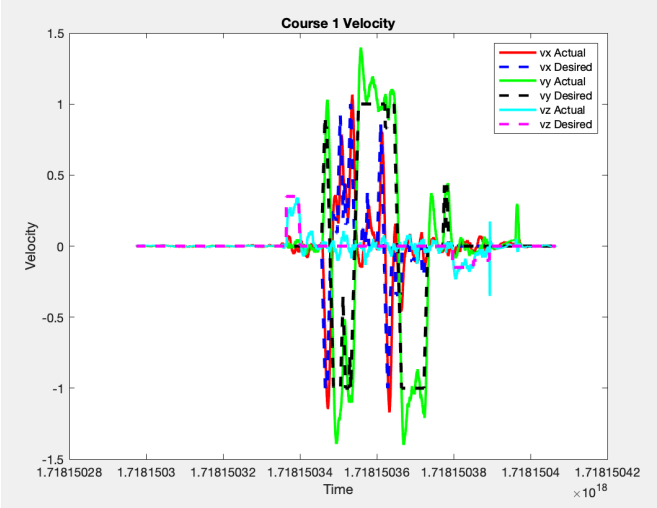


**Figure 27:** The velocities of the MAV along the x, y, and z axes over time for Course 1. The red, green, and cyan lines represent the actual velocities along the x, y, and z axes, respectively. The blue, black, and magenta dashed lines represent the desired velocities. This graph is useful for analyzing how well the MAV's velocity matches the desired values.

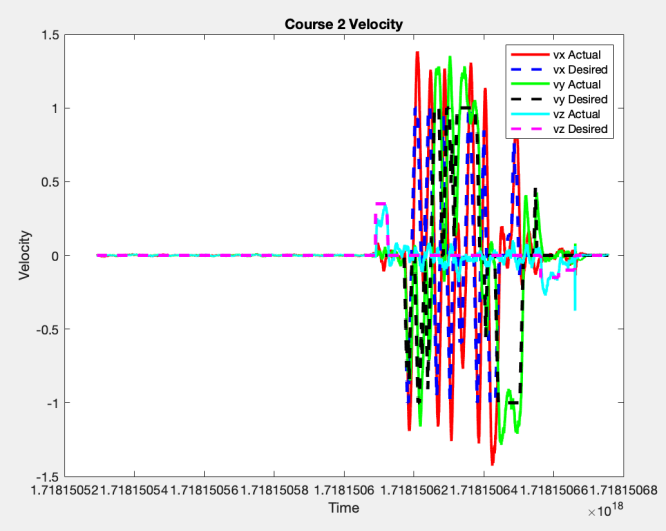


**Figure 28:** The velocities of the MAV along the x, y, and z axes over time for Course 2. The comparison between actual (solid lines) and desired (dashed lines) velocities provides insights into the MAV's performance in achieving desired speeds.

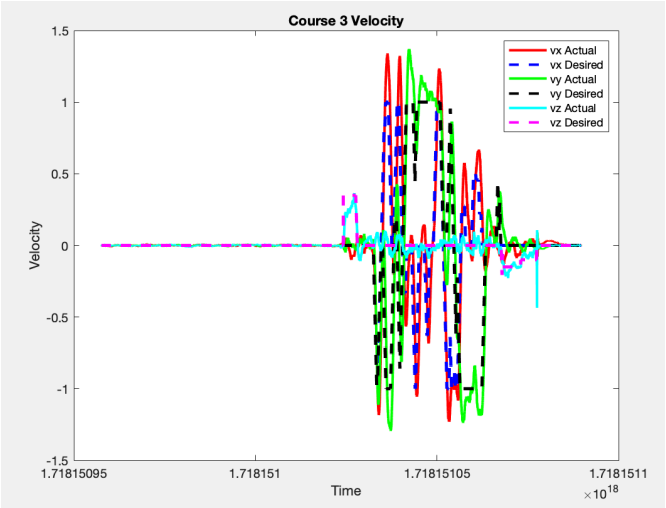


**Figure 29:** The velocities of the MAV along the x, y, and z axes over time for Course 3. The actual velocities are shown as solid lines, and the desired velocities are shown as dashed lines. This graph helps evaluate the MAV's ability to maintain desired velocities throughout the flight.

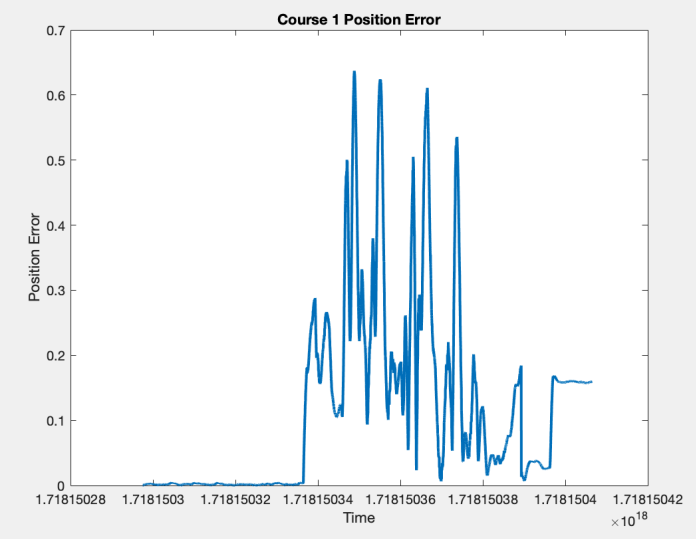


**Figure 30:** The position error over time for Course 1. The position error is calculated as the Euclidean distance between the actual and desired positions at each time step. A smaller error indicates better performance in following the desired trajectory.

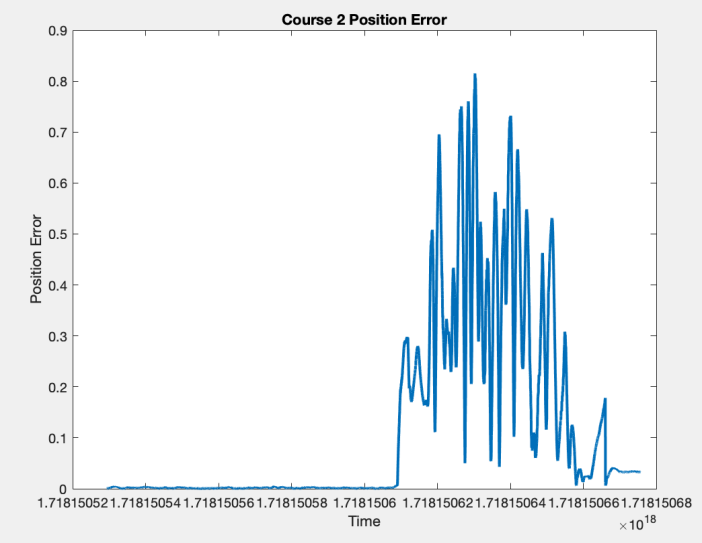


**Figure 31:** The position error over time for Course 2. It helps in assessing the accuracy of the MAV's path following for this specific course.

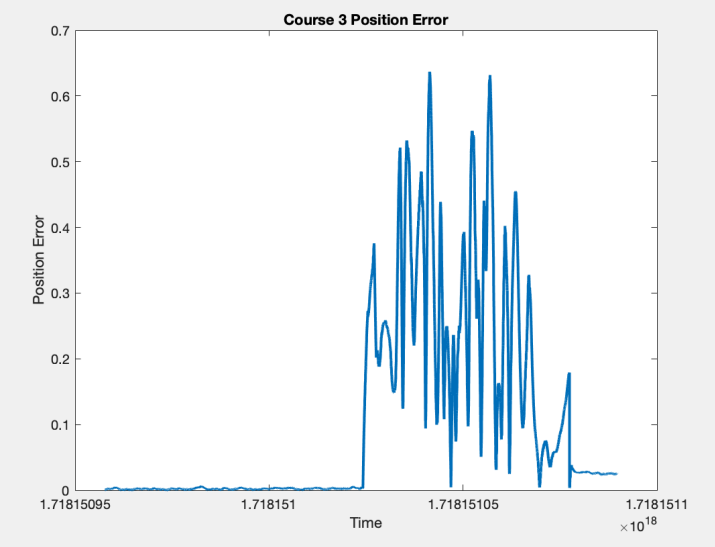


**Figure 32:** The position error over time for Course 3. It provides a measure of how closely the MAV adheres to the desired trajectory during the flight for this course.

By analyzing the MAV's flight data through these graphs, we can identify strengths and weaknesses in its performance. The insights gained from this analysis are critical for improving MAV control algorithms, ensuring more accurate path following, and enhancing overall flight stability. This comprehensive approach helps in developing more reliable and efficient MAV systems for various applications.

One useful metric beyond the telemetry to consider is the average speed of the MAV with respect to the intended path along the waypoints. Based on the flight waypoint sequence and the flight times, the average speed of the MAV is 0.9537 m/s, 0.9326 m/s, and 0.5741 m/s for courses 1, 2, and 3, respectively. We find that average speed is high for the two courses without change in altitude and closely matches the speed limit of 1 m/s of the MAV. On the other hand, average speed takes a drastic dip when altitude change is necessary due to the operational complexity for the pilot.

## Lessons Learned

- Vehicle Design & Manufacturing
    - Removing fasteners for lightweight components when not necessary and substituting for double sided tape or foam adhesive.
    - Move waterjet holes in balsa core carbon fiber further from edges as the abrasive and water eats away at the balsa core along the cutter path which results in very thin or exposed sections in the balsa core.
    - Solder with the components off of the board and unplugged if you can, always have extra wire when measuring it out as you can cut it off later.
- Thrust Profile
    - Data Synchronization and Calibration
        - The process of interpolating and comparing thrust data highlighted the importance of data synchronization and accurate calibration. Ensuring that the experimental data aligns correctly with the theoretical predictions is vital for meaningful analysis. This lesson emphasizes the need for meticulous data handling and precise calibration of sensors, especially the IMU, and control systems to achieve reliable performance assessments.
- Guidance Design
    - Incorrect formulation of the problem led to a suboptimal solution. This has been fixed in the final guidance code, but future efforts should verify these requirements more rigorously.
    - Turning angle between waypoints significantly impacts flight time due to change in direction, which violates the constant-velocity assumption. Future guidance methods may need to include higher fidelity considerations, such as patching time-optimal trajectories (similar to racing lines).
- Control & Simulation
    - Simplifying calculations is a balance between accuracy and performance
        - Hardware capabilities on the system being simulated should be kept in mine when writing the simulation or designing the control algorithm
        - Small details can make a large difference; figuring out what is worth including or excluding is part of the iterative process for simulation design.
- Flight Analysis
    - Through the evaluation of trajectory adherence, velocity profiles, and position errors, it became evident that even minor deviations in any of these aspects can significantly impact the MAV's stability and effectiveness. Therefore, consistently fine-tuning control algorithms and addressing external disturbances are crucial for maintaining optimal performance. This experience underscored that achieving high reliability and efficiency in MAV operations requires an ongoing commitment to data-driven improvements and adaptive strategies, ensuring the MAV can successfully meet the demands of various complex applications.

## Team Contributions

| Section | Contributor(s) |
|---|---|
| Abstract | Daniel, Robert |
| Vehicle Design & Manufacturing | Bennett, Lucas |
| Theoretical vs. Experimental Thrust Profile | Kyle |
| Electronics Overview | Daniel |
| Guidance Design & Analysis | Robert |
| Control & Simulation Analysis | Sophie |
| Flight Analysis | Kyle, Robert |
| Lessons Learned | Everyone |

# Appendix

## Flight Analysis Code

```
% Load the data
data1_state = readtable('course1_state.csv');
data1_desired = readtable('course1_desired.csv');
data2_state = readtable('course2_state.csv');
data2_desired = readtable('course2_desired.csv');
data3_state = readtable('course3_state.csv');
data3_desired = readtable('course3_desired.csv');
% Interpolate desired data to match state timestamps
data1_desired_interp.xd = interp1(data1_desired.t, data1_desired.xd, data1_state.t);
data1_desired_interp.yd = interp1(data1_desired.t, data1_desired.yd, data1_state.t);
data1_desired_interp.zd = interp1(data1_desired.t, data1_desired.zd, data1_state.t);
data1_desired_interp.vxd = interp1(data1_desired.t, data1_desired.vxd, data1_state.t);
data1_desired_interp.vyd = interp1(data1_desired.t, data1_desired.vyd, data1_state.t);
data1_desired_interp.vzd = interp1(data1_desired.t, data1_desired.vzd, data1_state.t);
data2_desired_interp.xd = interp1(data2_desired.t, data2_desired.xd, data2_state.t);
data2_desired_interp.yd = interp1(data2_desired.t, data2_desired.yd, data2_state.t);
data2_desired_interp.zd = interp1(data2_desired.t, data2_desired.zd, data2_state.t);
data2_desired_interp.vxd = interp1(data2_desired.t, data2_desired.vxd, data2_state.t);
data2_desired_interp.vyd = interp1(data2_desired.t, data2_desired.vyd, data2_state.t);
data2_desired_interp.vzd = interp1(data2_desired.t, data2_desired.vzd, data2_state.t);
data3_desired_interp.xd = interp1(data3_desired.t, data3_desired.xd, data3_state.t);
data3_desired_interp.yd = interp1(data3_desired.t, data3_desired.yd, data3_state.t);
data3_desired_interp.zd = interp1(data3_desired.t, data3_desired.zd, data3_state.t);
data3_desired_interp.vxd = interp1(data3_desired.t, data3_desired.vxd, data3_state.t);
data3_desired_interp.vyd = interp1(data3_desired.t, data3_desired.vyd, data3_state.t);
data3_desired_interp.vzd = interp1(data3_desired.t, data3_desired.vzd, data3_state.t);
% Plot positions for Course 1
figure;
plot3(data1_state.x, data1_state.y, data1_state.z, 'r', 'LineWidth', 2);
hold on;
plot3(data1_desired_interp.xd, data1_desired_interp.yd, data1_desired_interp.zd, 'b--', 'LineWidth', 2);
title('Course 1 Position');
xlabel('X'); ylabel('Y'); zlabel('Z');
legend('Actual', 'Desired');
grid on;
% Plot positions for Course 2
figure;
plot3(data2_state.x, data2_state.y, data2_state.z, 'r', 'LineWidth', 2);
hold on;
plot3(data2_desired_interp.xd, data2_desired_interp.yd, data2_desired_interp.zd, 'b--', 'LineWidth', 2);
title('Course 2 Position');
xlabel('X'); ylabel('Y'); zlabel('Z');
legend('Actual', 'Desired');
grid on;
```

```
% Plot positions for Course 3
figure;
plot3(data3_state.x, data3_state.y, data3_state.z, 'r', 'LineWidth', 2);
hold on;
plot3(data3_desired_interp.xd, data3_desired_interp.yd, data3_desired_interp.zd, 'b--', 'LineWidth', 2);
title('Course 3 Position');
xlabel('X'); ylabel('Y'); zlabel('Z');
legend('Actual', 'Desired');
grid on;
% Plot velocities for Course 1
figure;
plot(data1_state.t, data1_state.vx, 'r', 'LineWidth', 2, 'DisplayName', 'vx Actual');
hold on;
plot(data1_state.t, data1_desired_interp.vxd, 'b--', 'LineWidth', 2, 'DisplayName', 'vx Desired');
plot(data1_state.t, data1_state.vy, 'g', 'LineWidth', 2, 'DisplayName', 'vy Actual');
plot(data1_state.t, data1_desired_interp.vyd, 'k--', 'LineWidth', 2, 'DisplayName', 'vy Desired');
plot(data1_state.t, data1_state.vz, 'c', 'LineWidth', 2, 'DisplayName', 'vz Actual');
plot(data1_state.t, data1_desired_interp.vzd, 'm--', 'LineWidth', 2, 'DisplayName', 'vz Desired');
title('Course 1 Velocity');
xlabel('Time');
ylabel('Velocity');
legend;
grid on;
% Plot velocities for Course 2
figure;
plot(data2_state.t, data2_state.vx, 'r', 'LineWidth', 2, 'DisplayName', 'vx Actual');
hold on;
plot(data2_state.t, data2_desired_interp.vxd, 'b--', 'LineWidth', 2, 'DisplayName', 'vx Desired');
plot(data2_state.t, data2_state.vy, 'g', 'LineWidth', 2, 'DisplayName', 'vy Actual');
plot(data2_state.t, data2_desired_interp.vyd, 'k--', 'LineWidth', 2, 'DisplayName', 'vy Desired');
plot(data2_state.t, data2_state.vz, 'c', 'LineWidth', 2, 'DisplayName', 'vz Actual');
plot(data2_state.t, data2_desired_interp.vzd, 'm--', 'LineWidth', 2, 'DisplayName', 'vz Desired');
title('Course 2 Velocity');
xlabel('Time');
ylabel('Velocity');
legend;
grid on;
% Plot velocities for Course 3
figure;
plot(data3_state.t, data3_state.vx, 'r', 'LineWidth', 2, 'DisplayName', 'vx Actual');
hold on;
plot(data3_state.t, data3_desired_interp.vxd, 'b--', 'LineWidth', 2, 'DisplayName', 'vx Desired');
plot(data3_state.t, data3_state.vy, 'g', 'LineWidth', 2, 'DisplayName', 'vy Actual');
plot(data3_state.t, data3_desired_interp.vyd, 'k--', 'LineWidth', 2, 'DisplayName', 'vy Desired');
plot(data3_state.t, data3_state.vz, 'c', 'LineWidth', 2, 'DisplayName', 'vz Actual');
plot(data3_state.t, data3_desired_interp.vzd, 'm--', 'LineWidth', 2, 'DisplayName', 'vz Desired');
title('Course 3 Velocity');
xlabel('Time');
ylabel('Velocity');
```

```
legend;
grid on;
% Calculate and plot position errors for Course 1
figure;
position_error1 = sqrt((data1_state.x - data1_desired_interp.xd).^2 + (data1_state.y - data1_desired_interp.yd).^2 +
(data1_state.z - data1_desired_interp.zd).^2);
plot(data1_state.t, position_error1, 'LineWidth', 2);
title('Course 1 Position Error');
xlabel('Time');
ylabel('Position Error');
grid on;
% Calculate and plot position errors for Course 2
figure;
position_error2 = sqrt((data2_state.x - data2_desired_interp.xd).^2 + (data2_state.y - data2_desired_interp.yd).^2 +
(data2_state.z - data2_desired_interp.zd).^2);
plot(data2_state.t, position_error2, 'LineWidth', 2);
title('Course 2 Position Error');
xlabel('Time');
ylabel('Position Error');
grid on;
% Calculate and plot position errors for Course 3
figure;
position_error3 = sqrt((data3_state.x - data3_desired_interp.xd).^2 + (data3_state.y - data3_desired_interp.yd).^2 +
(data3_state.z - data3_desired_interp.zd).^2);
plot(data3_state.t, position_error3, 'LineWidth', 2);
title('Course 3 Position Error');
xlabel('Time');
ylabel('Position Error');
grid on;
```

## Theoretical vs. Experimental Thrust Profile Code

```
% Load the data
data1_state = readtable('course1_state.csv');
data1_desired = readtable('course1_desired.csv');
% Extract time and experimental thrust data (assuming velocity can be used for estimation)
time = data1_state.t;
% Calculate an estimate of experimental thrust
% Assuming a hypothetical relationship (this is just for demonstration, replace with actual method if known)
% Example: Thrust = k * velocity^2 (where k is a proportional constant)
k = 0.5; % example constant, replace with actual if known
experimental_thrust = k * (data1_state.vx.^2 + data1_state.vy.^2 + data1_state.vz.^2);
% Theoretical thrust calculation
% Assuming theoretical thrust can be calculated using some known parameters
% Replace the following with the actual calculation method
% Example parameters (replace with actual values or calculation method)
motor_constant = 1.2; % example motor constant (N/A)
propeller_efficiency = 0.85; % example efficiency factor
% Example theoretical thrust calculation
theoretical_thrust = motor_constant * propeller_efficiency * data1_state.vx;  % This is just an example formula
% Plotting the thrust profiles
figure;
plot(time, experimental_thrust, 'r', 'LineWidth', 2);
hold on;
plot(time, theoretical_thrust, 'b', 'LineWidth', 2);
title('Thrust Profile: Experimental vs. Theoretical');
xlabel('Time');
ylabel('Thrust');
legend('Experimental Thrust', 'Theoretical Thrust');
grid on;
% Calculate and plot the thrust error
thrust_error = abs(experimental_thrust - theoretical_thrust);
figure;
plot(time, thrust_error, 'LineWidth', 2);
title('Thrust Error Over Time');
xlabel('Time');
ylabel('Thrust Error');
grid on;
% Display summary statistics
mean_error = mean(thrust_error);
max_error = max(thrust_error);
fprintf('Mean Thrust Error: %.3f\n', mean_error);
fprintf('Max Thrust Error: %.3f\n', max_error);
```

## Simulation and Guidance Code

Attached to submission as a zip file due to its total length and large number of files.