The University of Texas at Austin
Aerospace Engineering
and Engineering Mechanics
Cockrell School of Engineering

ASE 479W Aerial Robotics
Section 14190

Tuesday, Thursday: 8:00 - 9:30

Final Report

Andrew Doty, Jacob Hands, Aaron Pandian
Date: 05/01/2024

Contents

1 Introduction

During the previous five labs, the framework for building a robust digital twin was developed, from attitude estimation and kinematics, the introduction of estimation and control, sensor inputs, path planning, and balloon computer-vision image recognition. These aspects were all combined together for completion of the tournament. The following report details the strategies, implementation, and results of the `student_autonomy_protocol` and `balloon_finder` implementation used to complete the tournament.

2 Theoretical Development

2.1 Path Planning

The following questions cover the path planning aspect of the tournament:

2.1.1 Path-Planning Algorithm

The A* algorithm is used to find the path. The first iteration was not adapted to 3D at all, and instead used Andrew's `Astar2D()` implementation from Lab 4. This sped up runtime because the dimensions of the grid to navigate became O^2 instead of O^3. To get the graph required for A*, a function called `TwoDtoThreeD()` was developed that flattened the 3D grid into a 2D grid by omitting the z-axis. The function ran the A* 2D algorithm, then converted the 2D grid back to a 3D grid by adding the pathing vector at the height of the first obstacle. Then in `UpdateTrajectories()`, those pathing values were offset linearly by simply taking the velocity of the drone and adding a z-component that was the difference between the current z-value and the target z-value. The major downsides of this implementation included a lack of obstacle avoidance in the z-direction, and a limit of the quad's maximum speed to compensate for the changes in the z-direction.

After only a couple days of testing, obstacles and targets of different heights were added, leading to the development of a 3D Astar implementation. This was a very simple modification, swapping out `Node2D` objects to `Node3D` objects, and helped to simplify the code by removing `TwoDtoThreeD()`. This new implementation did not use the `unordered_set` datatype, because for the simpler maps, A* computation time was not a significant contributor to slow runtime.

The next major change was adopting the `unordered_set`. This only changed three main lines of code in the Astar3D function, detailed below.
Old:

1. `std::vector<NodeWrapperPtr> explored;`
2. `if (exists(node_to_explore, explored))  continue;`
3. `else explored.push_back(node_to_explore); ...`

New:

1. `std::unordered_set<NodeWrapperPtr, std::hash<NodeWrapperPtr>, std::equal_to<NodeWrapperPtr>> explored;`
2. `if (explored.find(node_to_explore) != explored.end())  continue;`
3. `else  explored.insert(node_to_explore); ...`

This change was made because the `unordered_set` has a constant time complexity for insertion and search, while the vector has a linear time complexity for search. This change was made to improve the runtime of the A* algorithm. Once more complicated maps with more obstacles and wind were introduced into the, the previous implementation became the main bottleneck in the code.

In addition, path pruning and batching methods were added to improve runtime further. The path pruning method was based on the Ramer-Douglas-Peucker algorithm and modififed for use with the `waypoints` output by the path planner. The algorithm steps are as follows:

1. Start with a curve defined by a sequence of points.
2. Mark the first and last points of the curve to be saved.
3. Create a tolerance value that measures the perpendicular distance from a point to a line segment.

4. Select two points along the line with at least one point in-between. Create a line segment between these two points.
5. Find the perpendicular distance from that point to the line segment.
6. If the point distance is greater than this tolerance value, it is removed.
7. If the point distance is less than this tolerance value, the point is kept
8. Repeat this process until all points have been considered.

The batching method was post-processing that could be applied to a full or pruned trajectory. The functions `WaypointSplitter` and `TimeSplitter` take a vector as input and a chunk size integer. They then loop over the vector and create a vector of vectors with that chunk size. This allows the trajectory planner to work with a small chunk size of the pruned or full trajectory. In the implementation for the tournament, the first batch of points were applied to the trajectory planner, and the quad would move through that batch of points. Once the quad reached the last point in the batch, the next batch of points would be applied to the trajectory planner. While sending batched and pruned points would work, under heavy wind conditions, a small batch of the pruned path would still lead to the quad running into obstacles. This was mitigated by the `ClosetoTarget` function, which is detailed in the next section.

2.1.2 Safety Margin Definition

How did you choose the safety margin to ensure obstacle-free movement despite wind disturbance?

There were two main safety margin definitions defined in the code. A global safety margin was set in the headerfile, which was determined by trial and error. A value of around 0.6 was large enough to avoid any obstacles and small enough to allow the quad to still hit the balloons. This was mitigated by a function called `ClosetoTarget` which is detailed below. The function creates a raycast from the current position to the target position. If this raycast has a length less than a certain tolerance, then the distance and a boolean value are returned true. In the `UpdateTrajectories` function, if this flag is true, the safety margin is set to 0.2 and the quad paths straight to the balloon. This was a simple implementation that worked well in testing, and was not changed for the final tournament.

```
std::pair<bool, double> StudentAutonomyProtocol::ClosetoTarget(const Eigen::Vector3d
↪ &current_pos, const Eigen::Vector3d &target_pos, double tolerance)
{
    // Calculate the distance between the current position and the target position
    double distance = (current_pos - target_pos).norm();

    // If the distance is within the tolerance
    if (distance <= tolerance)
    {
    return std::make_pair(true, distance);
    }

    // If the distance is not within the tolerance or the path is empty, return false
    return std::make_pair(false, distance);
}
```

This check was performed every iteration, as after testing it added less than a millisecond to each iteration's compute time. The tolerance detailed above was set to 0.6 or 0.4, depending on the wind conditions.

2.1.3 Multi-target Path Planning

How did you handle the multi-target nature of the challenge (first balloon, second balloon, home)? Did you re-calculate an optimal path after each balloon was popped?

A function called `WhichisFaster` was developed to determine which path is optimal. This function takes in the current position, red balloon position, blue balloon position, and the goal position, as well as the occupancy grid and graph of the arena. The function outputs a boolean value and a set of waypoints. It calls the `FullTrajectory` function and counts the length of the waypoints vector from current-red-blue-goal and current-blue-red-goal. The function then returns the boolean value of which path is shorter, true for blue and false for red, and the waypoints vector corresponding to the shorter path. This function was called

in the `UpdateTrajectories` function.

The `WhichisFaster` function was not called every iteration. A boolean flag was established in the headerfile called `first_time_`. This function and some other boolean checks, such as setting the start position to the current position, and setting the old balloon positions to the current balloon positions. This function is called once again if the balloons teleport or move at any point during the run. Then the quad will calculate the entire trajectory to determine if there is a new optimal path.

2.1.4 Timestamp Calculation

> What was your approach to setting the time stamps of way points in your trajectory? Do you believe your approach is optimal or nearly so?

Two timestamp approaches were used in the development of the code. The one currently embedded into the system takes into account the known minimal time = $\frac{dx}{dv(max\_speed)}$. The minimal time value is then multiplied by a tolerance value to increase the travel time it takes for the quad to travel a distance of dx. This method allows the parameters of maximum speed and tolerance to be modified for complete path segment optimization.

Before this implementation, something similar was used that was given in `example_autonomy_protocol`. This method was very faulty as it prescribed the same amount of time for every dx. It was observed to be a major component in the acceleration constraint prevett error constantly being thrown out.

The tolerance method method is not optimal, but it is close. Using that method, an "optimal" time would assume infinite jerk when acceleration goes from 0 to max acceleration, as well as infinite jerk when the acceleration goes from max acceleration to 0. This instantaneous movement is not possible in the real world, so the tolerance added to the timestamps (scaling up or down by a given percent, $1/0.8 = 1.25 * t_{min}$ as an example of making the quad slower) is a good approximation. The current method is also not optimal because it does not full take the wind into account. The quad will attempt to move at a constant speed but can be blown off course with strong winds or slowed down by strong headwinds.

2.1.5 Path Replanning

> What was your strategy for re-planning in cases where the quad departed from your proposed path, e.g., due to wind disturbance?

The strategy for re-planning the proposed path due to wind disturbances was using the batching path-planning method and the `ClosetoTarget` function. The batching method was detailed in 2.1.1. The `ClosetoTarget` function was detailed in 2.1.2. The batching method allowed the quad to move through the map in small, manageable chunks and not travel too far off course from the desired path. The `ClosetoTarget` function enabled the quad to move immediately to the startpoint of the trajectory if it was blown off course. This was a simple implementation that worked well in testing, and was not changed for the final tournament.

For instance, the quads would occasionally overshoot the balloon by going too far to the left or right because of wind drift and a high speed. Increasing the safety margin and lowering the speed assisted in the quads completing a path segment with mild, stiff, or intense wind, but did not resolve the issue. The best solution was to have the quad perform a check from its current position to the starting position of the trajectory, and perform a `ClosetoTarget` path to the current trajectory.

2.1.6 Teleporting Balloons Mitigation

> What was your strategy for dealing with "teleporting" balloons?

To deal with the "teleporting" balloons parameter, there was an initalization of the red and blue balloon position by introducing a pointer to their object, under the `old_red_balloon_pos` and `old_blue_balloon_pos`. Next, a simple check was placed into the main while loop, where if the values of the initialized red and blue position changed, a boolean of `blue_balloon_moved` and `red_balloon_moved` would be set to true. This

Cell Size	.3	.4	.5	.6
Safety Margin	.6	.6	.6	.6
Chrono Time (s)	.8	1.05	1.63	2.05

Table 1: Constant Safety Margin vs Varied Cell Size

Cell Size	.4	.4	.4	.4
Safety Margin	.5	.6	.7	.8
Chrono Time (s)	1.28	1.05	.8	.6

Table 2: Constant Cell Size vs Varied Safety Margin

boolean would be passed into the waypoints generation code block, and trigger an entire recalculation of the optimal path with `WhichisFaster`.

2.1.7 Design Tradeoffs

Make a table of the design tradeoffs you confronted; e.g., 3D map cell size vs. safety (in obstacle avoidance) and computational efficiency.

The map used for testing was `temple`.
Observed above is the Chrono time reactions to an increased safety margin or cellsize. The chrono time reactions reflect the total time it takes for the trajectory path to be calculated. When the cell size is decreased, there is a lower processing time per tajectory calculation, but the overall computational efficiency drops due to the heavily increased number of waypoinyts that are made then batched. For the varied safety margins, an increased safety margin lead to a decreased computational time cost per trajectory, but the overall computational efficiency increased as there were now higher constraints that our trajectory must follow. For either of the variables, it had an oppurtunity to break the simulation by causing the quad to follow obscure paths or it makes the quad jitter rapidly and causes errors.

2.2 Machine Vision

2.2.1 False Balloon Mitigation

Did you avoid "false balloons" by employing sophisticated image processing or robust estimation (e.g., RANSAC)?

The false balloons within the images were avoided for estimation corruption using a robust estimation process. As opposed to varying the image processing technique where red and blue balloons- of specified color, size, shape, and distance from the edge- are detected, the results from said image processing are cleaned through a consensus-like estimation. Using this technique, the red and blue "false balloons" that were non-stationary are effectively ignored during the approximation of real 3D feature locations.

2.2.2 False Balloon Mitigation Strategy

Explain your strategy and the theory behind it.

The strategy implemented is a random sample consensus (RANSAC)-like method to detect and avoid outliers within the dataset. Much like traditional RANSAC implementations described in the notes and on online sources, the RANSAC used in the balloon-locator repository involved randomly selecting subsets of data. However, rather than fitting the data and counting the number of outliers within each subset, it uses known parameters to evaluate the error of data points. Through image processing, the algorithm identifies a number of images that it thinks are the balloons we want to estimate, alongside their projected balloon center locations in pixels and camera state information. With this, the estimation algorithm sets up to test multiple random samples within the data.

Done in groups of five, each random set is tested on their agreeability, similar to the consensus platform.

Using the linear least squares technique, assuming a noise-free measurement, a 3D feature location is estimated using all of the provided images in the sample set. Then, for each image, the algorithm determines the agreeability between the original image and the derived estimation. This is done by backwards derivation of the projected (2D) balloon center locations in pixels, using the 3D feature estimation and each image's camera state information. The outcome of this for each of the five images is an estimated value of what the logic thinks the original balloon's center location was within the original image, in pixels. Given this backwards-derived pixel point, it is compared to the original projected balloon center location provided to the estimation method by the image processing algorithm. This is the measure of agreeability. The error between these values for each data point (image) are summed for each random sample.

As the estimator moves through these sample sets, the information from the lowest accumulated error is saved, then output. This method allows for a consensus-like system by ensuring that as many of the images provided "agree" with each other, in terms of their combined 3D feature estimation. What this means is that estimations with high variance in the original image's 2D balloon center location are not selected. Furthermore, in the event that all the images selected are of moving balloons, this will still elicit an error higher than if the majority of balloons are stationary (the intended target of estimation).

3 Implementation

3.0.1 Diagrams and Class Overview

The main additions to `student_autonomy_protocol` were not in other classes, but were instead embedded as functions within the main class. This helped to keep the code extremely organized and easy to modify values for testing. For example, speeding up the quad for a section of flight was accomplished by changing a single variable in a function call.

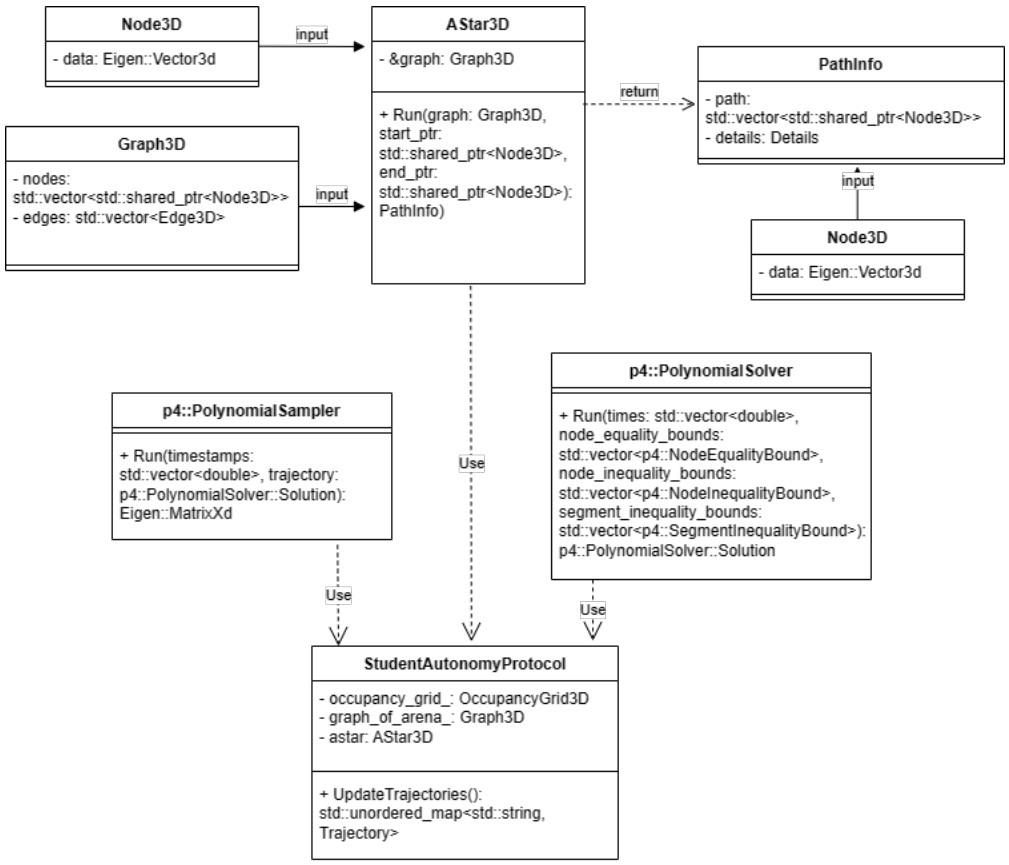

Figure 1: Class Diagram of interactions between classes

The above figure details the interactions between the different classes. `AStar3D` uses pointers to `Node3D` as inputs as well as a `Graph3D` map of the arena. This map is converted from an `OccupancyGrid3D` type into a `Graph3D` type in the headerfile using the `occupancy_grid_.AsGraph()` function.

More details for the `StudentAutonomyProtocol` function will be provided below. In the figure, all purple colored blocks are used for final trajectory generation. Orange colored blocks are path simplification, green colored blocks are for trajectory generation, and the grey colored blocks represent portions of the debug module. These debug module blocks were used for testing and the code.

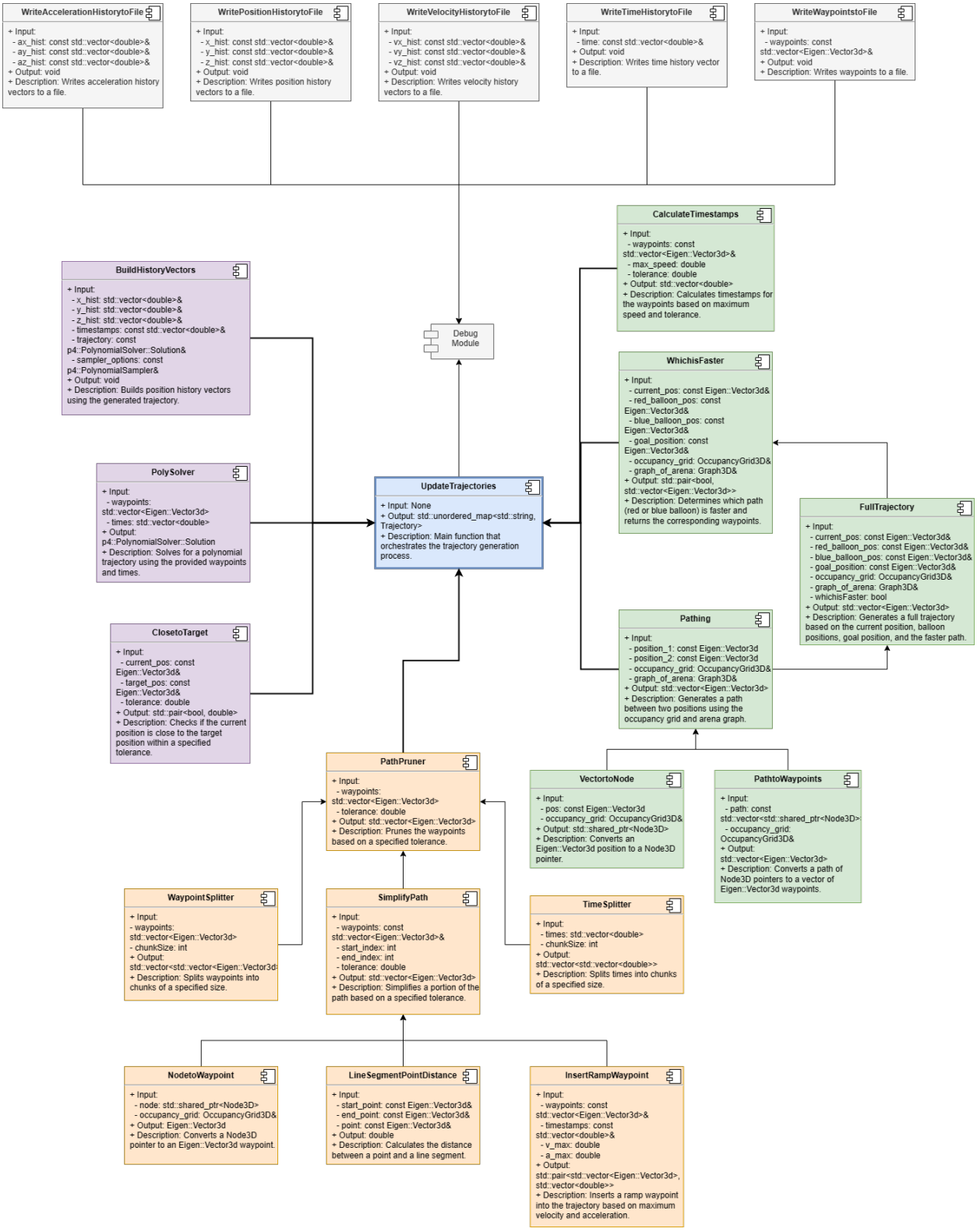

Figure 2: Function Diagram of Interactions between functions

3.0.2 Function Description

Describe the basic functionality of each class in the text of the report. You can embed snippets of code in the report and refer to these to elucidate a particular part of your algorithm.

The StudentAutonomyProtocol class inherits from the AutonomyProtocol class and overrides the `UpdateTrajectories()` function to implement custom trajectory generation logic. It contains several private member variables and functions that aid in trajectory generation. The main functionality is implemented in the UpdateTrajectories() function, which performs the following steps:

1. Retrieves the current quad position, balloon positions, and popped states from the game snapshot.
2. Checks for quad safety and performs safety checks based on balloon status and wind intensity.
3. Handles mediation layer trajectory codes and adjusts the trajectory accordingly.
4. Determines the remaining time for the trajectory based on the duration and current time.
5. Generates waypoints for the trajectory based on the balloon positions, goal position, and fastest path.
6. Prunes and simplifies the waypoints using the PathPruner() and SimplifyPath() functions.
7. Calculates timestamps for the waypoints using the CalculateTimestamps() function.
8. Solves for the optimal polynomial trajectory using the PolySolver() function.
9. Samples the trajectory to generate position, velocity, and acceleration vectors using the BuildHistoryVectors() function.
10. Writes the position, acceleration, and waypoint data to files for logging and visualization.
11. Constructs the final trajectory using the sampled data.
12. Validates the trajectory using the prevetter and handles any errors or constraints.
13. Adds the trajectory to the quad-to-trajectory map and returns it.

Here is a snippet of code that illustrates the waypoint generation logic.

```
    if (red_balloon_popped && blue_balloon_popped)
    {
        std::cout << "Both balloons are popped, pathing to goal" << std::endl;
        waypoints.clear();

        std::vector<Eigen::Vector3d> waypoints1 = Pathing(current_pos, goal_position_,
        ↪  occupancy_grid_, graph_of_arena_);
        waypoints.insert(waypoints.end(), waypoints1.begin(), waypoints1.end());
        timestamps = CalculateTimestamps(waypoints, .3, time_tolerance);
    }
    else if (red_balloon_popped == true && blue_balloon_popped == false)
    {
        std::cout << "Red Balloon is popped, pathing to blue" << std::endl;
        waypoints.clear();

        std::vector<Eigen::Vector3d> waypoints1 = Pathing(current_pos, blue_balloon_pos,
        ↪  occupancy_grid_, graph_of_arena_);
        std::vector<Eigen::Vector3d> waypoints2 = Pathing(blue_balloon_pos,
        ↪  goal_position_, occupancy_grid_, graph_of_arena_);
        waypoints.insert(waypoints.end(), waypoints1.begin(), waypoints1.end());
        waypoints.insert(waypoints.end(), waypoints2.begin(), waypoints2.end());
        timestamps = CalculateTimestamps(waypoints, .3, time_tolerance);
    }
```

PolySolver(): This function solves for the optimal polynomial trajectory using the provided waypoints and timestamps. It sets up the necessary constraints and bounds for the polynomial solver, such as node equality bounds, node inequality bounds, and segment inequality bounds. It then configures the polynomial solver options and runs the solver to obtain the optimal trajectory solution.

Here's a snippet of code from PolySolver() that sets up the node inequality bounds:

```
    // Set node inequality bounds to constrain the trajectory positions within the map
    ↪  bounds
```

```
    const double p_bound = 0.1;
    std::vector<p4::NodeInequalityBound> node_inequality_bounds;
    for (int w = 1; w < waypoints.size(); w++)
    {
        node_inequality_bounds.push_back(p4::NodeInequalityBound(
            0, w, 0, waypoints[w].x() - p_bound, waypoints[w].x() + p_bound));
        node_inequality_bounds.push_back(p4::NodeInequalityBound(
            1, w, 0, waypoints[w].y() - p_bound, waypoints[w].y() + p_bound));
        node_inequality_bounds.push_back(p4::NodeInequalityBound(
            2, w, 0, waypoints[w].z() - p_bound, waypoints[w].z() + p_bound));
    }
```

Pathing(): This function generates a path between two positions using the A* algorithm. It takes the start and end positions, occupancy grid, and arena graph as inputs. It converts the start and end positions to nodes using the VectortoNode() function, runs the A* algorithm using the AStar3D class, and converts the resulting path from nodes to waypoints using the PathtoWaypoints() function.

BuildHistoryVectors(): This function samples the trajectory to generate position, velocity, and acceleration vectors. It takes the timestamps, trajectory solution, and sampler options as inputs. It uses the PolynomialSampler class to sample the trajectory at the specified timestamps and builds the history vectors for position, velocity, and acceleration based on the derivative order specified in the sampler options.

WhichisFaster(): This function determines the faster path between the current position, balloon positions, and goal position. It calls the FullTrajectory() function twice, once for the path through the red balloon and once for the path through the blue balloon. It compares the sizes of the resulting waypoint vectors and returns the faster path along with a boolean indicating which path is faster.

FullTrajectory(): This function generates a full trajectory based on the current position, balloon positions, goal position, and the faster path determined by WhichisFaster(). It uses the Pathing() function to generate waypoints between the current position, balloon positions, and goal position, and concatenates them to form the full trajectory.

PathPruner() and SimplifyPath(): These functions are responsible for pruning and simplifying the waypoints based on a tolerance value. PathPruner() is the main entry point and calls SimplifyPath() recursively to simplify the path. SimplifyPath() uses the Ramer-Douglas-Peucker algorithm to remove redundant waypoints while maintaining the overall shape of the path within the specified tolerance.

CalculateTimestamps(): This function calculates timestamps for the waypoints based on a maximum speed and tolerance. It iterates through the waypoints, calculates the distance between consecutive waypoints, estimates the time based on the maximum speed, and adds the timestamps to a vector.

InsertRampWaypoint(): This function inserts additional waypoints to create a ramp-like trajectory. It takes the original waypoints, timestamps, maximum velocity, and maximum acceleration as inputs. It calculates the time and distance required to reach the maximum velocity and inserts new waypoints at the appropriate positions to create a smooth ramp-like trajectory.

ClosetoTarget(): This function checks if the current position is close to a target position within a specified tolerance. It calculates the Euclidean distance between the current position and the target position and returns a boolean indicating whether the distance is within the tolerance.

WriteAccelerationHistoryToFile(), WritePositionHistoryToFile(), WriteVelocityHistoryToFile(), WriteTimeHistoryToFile(), WriteWaypointstoFile(): These functions write the respective data (acceleration, position, velocity, time, and waypoints) to files for logging and visualization purposes. They take the corresponding history vectors as inputs and write them to text files in a specific format.

PathtoWaypoints(): This function converts a path represented as a vector of Node3D shared pointers to a vector of Eigen::Vector3d waypoints. It iterates through the nodes in the path, converts each node to a waypoint using the NodetoWaypoint() function, and adds the waypoints to a vector.

VectortoNode(): This function converts an Eigen::Vector3d position to a Node3D shared pointer in the occupancy grid. It uses the occupancy grid's mapToGridCoordinates() function to convert the position to grid coordinates and creates a new Node3D object with the corresponding coordinates.

NodetoWaypoint(): This function converts a Node3D shared pointer to an Eigen::Vector3d waypoint in the occupancy grid. It retrieves the grid coordinates from the Node3D object, multiplies them by the grid size, and adds the origin of the occupancy grid to obtain the waypoint coordinates.

WaypointSplitter() and TimeSplitter(): These functions split the waypoints and timestamps into chunks of a specified size. They take the waypoints or timestamps vector and the chunk size as inputs, iterate through the vector, and create sub-vectors of the specified size, adding them to a vector of vectors.

3.0.3 Testing and Debugging Strategy

The main method for testing and debugging was using the standard library package `fstream`. Fstream enabled the terminal to remain clear as debugging information such as position, velocity, time, and acceleration vectors were written to separate files in the bin folder. This method allowed for quick inspection along a built path in determining where acceleration constraint errors were commonly found. Additionally, these files were easily deleted and cleared through the `cstudio` package, so only the newest data was written.

By testing the functionality of the code with the `area_empty.map`, small bugs such as failed calculations of time steps or determining the safest number of chunks to allocate into p4 would become visibly clear through the quads reactions to new implementations.

Now moving onto the `balloonfinder` repository, the main method for testing and debugging was to use the `computeStructure()` function. The `computeStructure()` function within the `StructureComputer` class started off with the same code as Lab 5. Building on this working function, comments were placed throughout the script noting the steps the function should conduct, taking into consideration where specifically each step should occur. Once the outline was built, print statements were decorated across the function to assess where things were going wrong during the debugging process.

Once the code was working, an originally implemented threshold approach for the error was used. The function would break once an error was detected that passed the threshold requirements, in order to save computation time. Testing this required analyzing the summed errors of each sample set and their resulting estimation to determine what threshold could be low enough. However, with this approach, it was realized that the difference in time it took to find a threshold deemed low enough and completing an evaluation of all the samples was almost negligible. As a result, the `computeStructure` function was converted to use the 3D centerpoint estimation from the lowest error.

The number of data points or `CameraBundle` objects to be used in the random sample was also continued to be tested. After numerous iterations, it was found that using five or more provided the centimeter accuracy needed, so five was chosen as the sample size.

4 Results

4.1 Autonomy Protocol Results

4.1.1 Path-Planning Improvements

Show a table of how your path planning strategy improved over the pre-tournament (when operating on the same environment: map, wind, etc.). In the table, list the key innovation that led to each improvement.

Below is a table with the enviornment the implementations were evaluated on, following is the results of each path planning strategy and the major variables associated.

Map	**arena_empty.map**
Wind disturbances	0
Safety Margins	0.6

Table 3: Map and Conditions

The above table represents the map, wind disturbance, and safety margin that were used to evaluate the path planning strategy. The following table shows the results of each path planning strategy and the major variables associated with each strategy.

Paths	**AStar**	**AStar Pruning**	**AStar Batching**
First Path Generation Speeds m/s	1.1	1.1	1.1
Balloon Moved Speeds m/s	1.4	1.4	1.4
Blue to Red Speeds m/s	0.8	0.8	0.8
Red to Blue Speeds m/s	1.4	1.4	1.4
Current Position to Goal Speeds m/s	1.1	1.1	1.1
Completion time (s)	58.2736	48.6513	39.02

Table 4: Pathfinding Metrics Comparison

This represents the evolutions in pathplanning mentioned in 2.1.1.
It is observed that once AStar Batching was working, the path planning strategy was highly effective and it was a matter of optimizing the variables for each map. While the pruned trajectory worked, if the tolerance was set too high, the path sent to p4 would often fall close enough to an obstacle that it would trigger the prevetter. The batching strategy was the most effective, as it allowed the quad to move through the map in small, manageable chunks. In other testing, such as the `temple.map` with wind, the batching strategy was the only one that allowed the quad to complete the map.

4.1.2 Completion Time vs Wind Intensity

Indicate in a table or graph your obstacle course completion time as a function of increasing wind disturbance intensity. Push your strategy to failure and note the failure-causing wind intensity.

Because the team was not able to correctly maneuver the quad through this year's tournament map, the table will reflect data obtained from `params.yaml` pre-tournament day 9 map, temple:

Wind Intensity	**Completion Time (s)**
0 (zero)	36.8457
1 (mild)	35.641
2 (stiff)	40.2303
3 (intense)	38.2276
4 (ludicrous)	Cannot Complete

Table 5: Completion Time vs Wind Intensity

By changing the different wind disturbances, it was observed that the code ran very similarly throughout wind intensities 0 to 3. However, despite the efforts in changing safety margins and maximum speeds, wind intensity 4 was too much for the quad to handle. It would consistently sway around and be pushed into nearby obstacles.
Several elements of the path planning strategy attempted to account for this. Since paths were passed to A* in batches, and the quad only moves through a small batched trajectory at a time, the `ClosetoTarget` function enabled the quad to move immediately to the startpoint of the trajectory if it was blown off course. However, when encountering obstacles of different sizes, the quad would either not pathplan as a high safety margin would prevent it from finding an obstacle-free path, or the wind would push it along the trajectory and it would be blown into an obstacle directly.

4.2 Balloon Finding

4.2.1 Balloon Finding Performance

Show a table that summarizes your balloon finding performance on the tournament-train image set. Your table should indicate how the performance changes when you activate or deactivate your strategy for dealing with the "false balloons."

Item	Deactivated	Activated	Optimized
Red Ball. Est. Location Error (m)			
X	-0.239349	-0.240605	0.0455781
Y	-0.181367	-0.0247017	-0.00988785
Z	-0.0424167	-0.276065	0.00514032
Blue Ball. Est. Location Error (m)			
X	2.55677	1.34578	-0.044039
Y	-0.556624	-0.509228	-0.000954509
Z	-0.00996989	-0.0776651	-0.0240315

Table 6: Performance results with various benchmarks of outlier mitigation strategy

In the table above, results from different stages of the computeStructure function are displayed. As evident, similar results are apparent without the use of outlier mitigation and an unoptimized version of the strategy implementation. In this unoptimized version, a threshold strategy was utilized where an arbitrary number was chosen. This threshold, like previously described, meant the function returned the first instance where a random sample of images had a summed error lower than the threshold value of five. At this stage, the sample size was two.

Upon optimizing, it was found that finding the lowest error from numerous samples, of a size of five, allowed an estimation far below the 10 cm requirement. This was in contrast to the unoptimized version which utilized an arbitrary threshold.

4.2.2 Challenging Images

Show a few images that were particularly challenging and explain why.

(a) Image 1

(b) Image 2

(c) Image 3

(d) Image 4

(e) Image 5

(f) Image 6

Figure 3: Various experimental setups with balloons

In image 2, the effect of the TA moving in front of the target balloon caused the image processing algorithm to null the image due to the fact that the shape of the balloon contour was not within the specified range. Even given the distractions placed in image 1, the stationary balloon centerpoint still passed the verification. However, once the TA began to block a portion of the balloon, the measurement was no longer valid.

In image 3, the effect of the distracting balloons becomes evident. Unlike image 2 where the TA is blocking a portion of the balloon, in image 4, the TA seems to be adding to the balloon size with the distracting balloons. Due to the fact that in the image the red balloons, stationary and moving, are about the same color, moving the distracting balloon behind the stationary balloon causes the image processing algorithm to count that as one big balloon. This not only throws off the centerpoint measurement but sets the balloon size out of range to pass the image as a CameraBundle object. With a little more separation between the two balloons, as in image 3, the stationary red balloon shows a more accurate centerpoint and passes all requirements for a valid estimation.

Lastly, both image 5 and 6 present images with a high density of distractions. These images are partic-

ularly challenging because in some cases, these distractions pass as valid, especially when the distracting balloons are in a better position, in terms of lighting and distance from the camera, than the stationary balloons. However, in the case of image 5 and 6, the image processing algorithm was able to successfully estimate and pass the stationary balloon centerpoints while neglecting the distractions.

5 Contributions

Andrew wrote the 3D a-star implementation, `WaypointSplitter, TimeSplitter, PolySolver` and the `NodetoWaypoint, PathtoWaypoints, VectortoNode, Pathing` and `FullTrajectory` functions, and edited the report for accuracy and style. Andrew also developed a partial RANSAC implementation for the balloonfinder repository without optimization, just representing the Deactivated version. Jacob and Andrew both developed the `PathPruner, SimplifyPath` functions, switch cases for the prevetter and wind conditions, the `WhichisFaster` and `ClosetoTarget` functions, and the trajectory generation code. Jacob developed the "print-to-file" functions and the `InsertRampWaypoint` function. Jacob performed most of the testing and debugging for the autonomy protocol and the additional testing needed for results in the report. Aaron developed the acceleration and velocity prevetter functions, the `CalculateTimestamps` and `LineSegmentPointDistance` functions, debugged the `PolySolver` function, built a final the balloon-finder repository, and worked on the balloon-finder portions of the final report.

6 Conclusion

Over five labs and the development of a robust digital twin, this twin was tested over the pre-tournament and final tournament. Although the quad did not determine the optimal path on the final tournament map, this project represented a significant milestone in tying the various aspects of the course together. It was a great and humbling learning experience to work with pre-existing code modules and integrate functionalities learned in class. With additional time, future work for the project would include fixing the optimization on the `tournament` map, breaking out the functions within StudentAutonomyProtocol into other classes, and optimizing the overall functionality of the program. Future proposed structural changes include switching from VMWare's distribution to a Docker container and integrating some basic documentation into a navigatable and visually interesting wiki.