Aerial Robotics Project Report – Team Azeem

Azeem Bhaiwala, George Braun, Justin Hart

May 1, 2023

1 Introduction

Throughout this course, various quadcopter topics have been studied and implemented in lab assignments. These include modeling quad dynamics, modeling sensor measurements, performing control based on position and attitude, performing estimation based on sensor measurements, planning a quad trajectory, and finally, enabling quad perception of the environment. In this project, all of the above topics are utilized in conjunction to complete two separate objectives. The first is commanding a quadcopter to pop two balloons and return to a goal position in the fastest time possible, in an obstacle-filled arena. The second is to estimate red and blue balloon center locations to within 10 cm based on a set of images and their corresponding metadata.

2 Theoretical Development

2.1 Path Planning

The high-level strategy for the quadcopter's trajectory planning follows the steps described below, choosing to recalculate the trajectory at the beginning of the flight, once a balloon has been popped, or a balloon teleports. In the case of a balloon teleporting, the current trajectory is halted while the quad recalculates.

First, the start and end points for the current trajectory are selected; the starting point will always be the quadcopter's current position, while the end point will be determined by how many balloons have been popped. If no balloons have been popped, the end point will be placed at the closer of the two balloons relative to the quad. If one balloon has been popped, the other balloon is selected as the endpoint, and finally the goal position is chosen if both balloons have been popped.

Next, the two points are mapped to a 3D occupancy grid discretizing the environment into occupied and unoccupied cells. The discretized environment analyzes whether cells are within a set safety distance from any obstacle to determine their vacancy. Because the tournament rules specify failure if the quad ventures within 0.4 meters of any obstacles, a safety distance of at least 0.5 meters was chosen to account for wind disturbances while allowing the quad to traverse narrow passageways. The A* method is used to quickly find the shortest path of adjacent cells between the start and end points, and the heuristic chosen for A* is the 3-dimensional Euclidean distance from the current cell to the end cell.

This list of waypoint cells is then fed into the pruning method, which uses Bresenham's line algorithm applied to a 3D grid to determine whether non-adjacent waypoints can be directly traversed to, meaning the intermediate cells can be discarded. Usually, this drastically reduces the number of waypoints the quad must travel and smooths the later-generated trajectory.

To use the piecewise polynomial path planning software to generate a trajectory, times for these waypoints must also first be determined. The approach to setting time stamps of waypoints in the quad's trajectory is an iterative one. This problem entails setting the time stamps so that the piecewise polynomial path planning trajectory is admissible (respects the maximum velocity and acceleration constraints set and avoids obstacles) but is also minimum time among the admissible trajectories. This is accomplished by first determining each waypoint time by assuming travel at the max velocity constraint (2.0m/s) over the distance between each waypoint.

Finally, the trajectory is computed based on the waypoints and times assigned. If the trajectory violates any of the maximum constraints, then the timing for that segment containing the point of violation is relaxed and the trajectory is calculated again. This process is repeated until an admissible trajectory is found. We believe the approach used for generating times is nearly optimal. Although not solved as a combined optimization problem since the problem is non-convex, in theory, the described approach should find a near-time-optimal trajectory by iterating on the trajectory starting at the maximum velocity constraint. This approach could be improved upon by shrinking the segment of the trajectory near the point of violation in which the timing is relaxed.

2.2 Vision Test

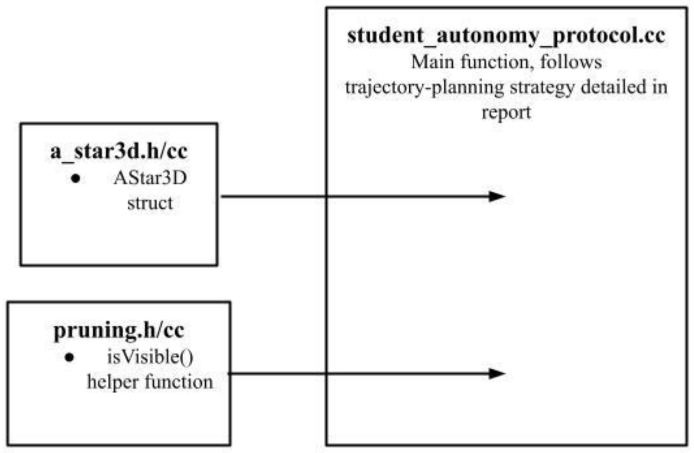
False balloons are avoided by employing random sample consensus (RANSAC), a robust estimation technique first published in [2]. The need for RANSAC stems from the fact that, in practice, measurement noise is not always Gaussian in nature and may contain outliers (possibly due to an error in a decision made during the measurement process). RANSAC takes observed data containing outliers and estimates the parameters of a mathematical model from this set of data. The outliers are to have no influence on the outcome of the estimation. The RANSAC methodology is as follows. First a minimal subset of data points is randomly selected. Then the least squares estimate is solved using this subset of data and the solution's support is calculated. The support is the number of data points lying within a distance threshold from the estimate – these points form the consensus set for this estimate. These first three steps are repeated a large number of times (as shown on slide 68 of [1]) and the estimated solution with the largest support is determined. The measurements comprising the consensus set of this estimated solution are used to solve the least squares problem – this is the robust solution.

RANSAC is implemented on the set of balloon images for the vision test using the following steps. First, two camera bundles are taken at a time (chosen randomly with replacement) and the balloon location is solved using structure computation for each pair – this is done 100 times. The candidate 3D balloon location for each pair is re-projected onto each image in the data set with a camera bundle object associated with it. The distance between the measured balloon center to the re-projected center is measured. The camera bundle for that image offers its support to the candidate location if this measured distance is less than 100 pixels. The candidate solution with the highest support is chosen and the camera bundles in this consensus set are pushed into the structure computer object and solved for using least squares. This is the robust estimated 3D balloon location.

3 Implementation

3.1 Path Planning

To facilitate the development of the trajectory planning logic, the first few steps were organized into separate files:

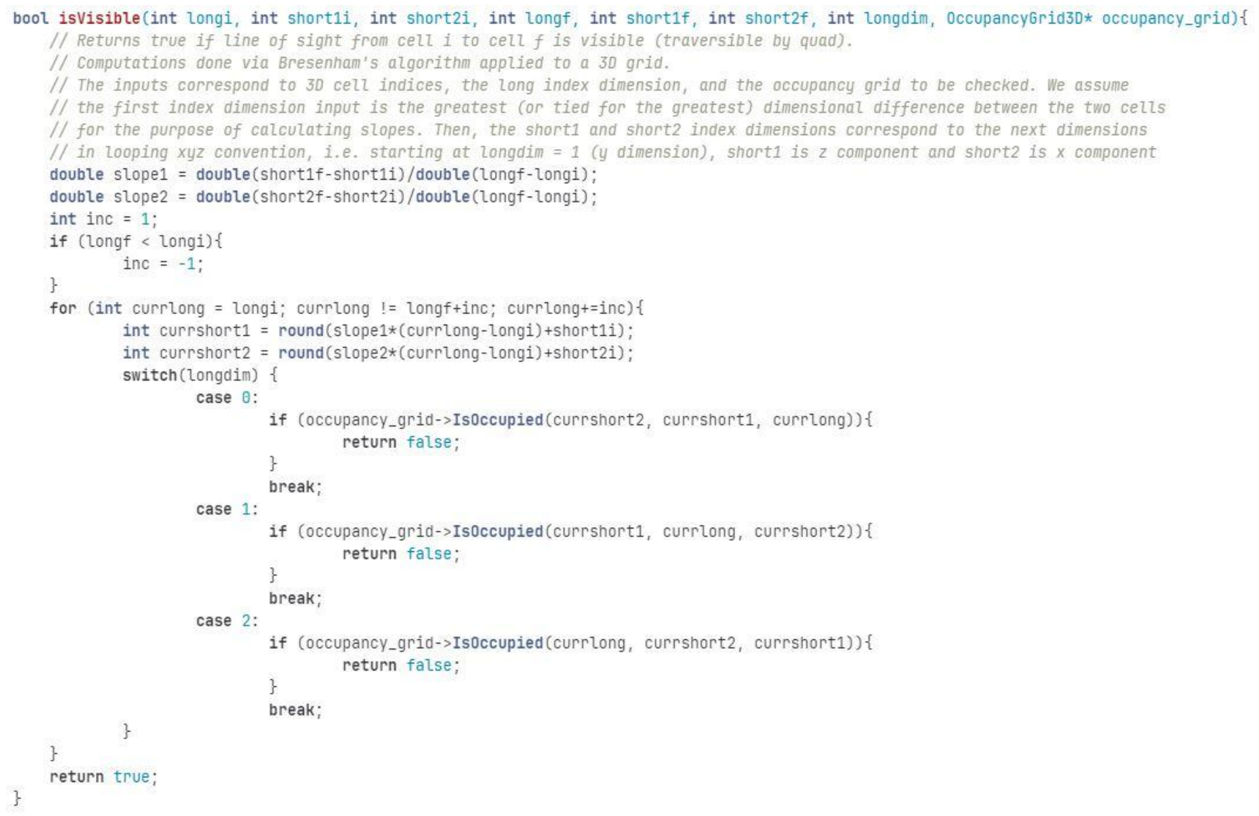


student_autonomy_protocol.cc
Main function, follows trajectory-planning strategy detailed in report
a_star3d.h/cc
● AStar3D struct
pruning.h/cc
● isVisible() helper function

Figure 3.1.1: Vision test strategy implementation in software.

a_star3d.cc follows from the 2D A* code used in lab 4. It contains a NodeWrapper structure that holds a node, its parent node, the cost to reach the node, and the heuristic cost. The heuristic cost is the 3D euclidean norm from the current node to the end node. a_star3d.cc also contains various helper functions to compare nodes and to calculate the heuristic. The main portion of the file is the function AStar3D::Run, which takes a graph of the arena, and pointers to the start and end nodes. The function runs through the A* algorithm, comparing node costs and building a PathInfo structure that contains the nodes that make up the path and other information about the path such as the length and total cost. The PathInfo structure is then used in student_autonomy_protocol.cc to feed into the polynomial planner, after the node numbers are converted into coordinates.

pruning.cc contains the isVisible helper function, which takes in the integer coordinates of two indexed cells and an occupancy grid (class defined in the game engine's built in occupancyGrid3D.cc) to determine whether a direct path between the cells is feasible. To best use Bresenham's algorithm, the largest (or tied for largest) dimensional displacement in the x, y, or z directions must first be determined, which is done just before calling the helper function. This is done to ensure that all slopes calculated are <= 1, which is vital for checking as many cells along the direct path as possible with this algorithm. The helper function itself is pictured below:



```
bool isVisible(int longi, int short1i, int short2i, int longf, int short1f, int short2f, int longdim, OccupancyGrid3D* occupancy_grid){
    // Returns true if line of sight from cell i to cell f is visible (traversible by quad).
    // Computations done via Bresenham's algorithm applied to a 3D grid.
    // The inputs correspond to 3D cell indices, the long index dimension, and the occupancy grid to be checked. We assume
    // the first index dimension input is the greatest (or tied for the greatest) dimensional difference between the two cells
    // for the purpose of calculating slopes. Then, the short1 and short2 index dimensions correspond to the next dimensions
    // in looping xyz convention, i.e. starting at longdim = 1 (y dimension), short1 is z component and short2 is x component
    double slope1 = double(short1f-short1i)/double(longf-longi);
    double slope2 = double(short2f-short2i)/double(longf-longi);
    int inc = 1;
    if (longf < longi){
            inc = -1;
    }
    for (int currlong = longi; currlong != longf+inc; currlong+=inc){
            int currshort1 = round(slope1*(currlong-longi)+short1i);
            int currshort2 = round(slope2*(currlong-longi)+short2i);
            switch(longdim) {
                    case 0:
                            if (occupancy_grid->IsOccupied(currshort2, currshort1, currlong)){
                                    return false;
                            }
                            break;
                    case 1:
                            if (occupancy_grid->IsOccupied(currshort1, currlong, currshort2)){
                                    return false;
                            }
                            break;
                    case 2:
                            if (occupancy_grid->IsOccupied(currlong, currshort2, currshort1)){
                                    return false;
                            }
                            break;
            }
    }
    return true;
}
```

Figure 3.1.2: Start and end cell indices are used with Bresenham's algorithm to determine traversability.

student_autonomy_protocol.cc calls a_star3d.cc and pruning.cc in sequence (following the previously described logic) before running the waypoint time generation and iterative pre-vetting procedure itself.

To more easily debug the procedure, the lists of waypoint node indices, coordinates, and times were formatted and printed to the terminal after each step of the process, as well as statements commenting on the current stage of the trajectory planning strategy to track its status. Most of these print commands are simply commented out in the final version of the code, leaving only targeting comments and the few initially pruned waypoints as outputs to the terminal.

3.2 Vision Test

No new C++ classes were developed for the vision test portion of the project. Fig. 3.2.1 below shows how the vision test strategy is implemented in software.

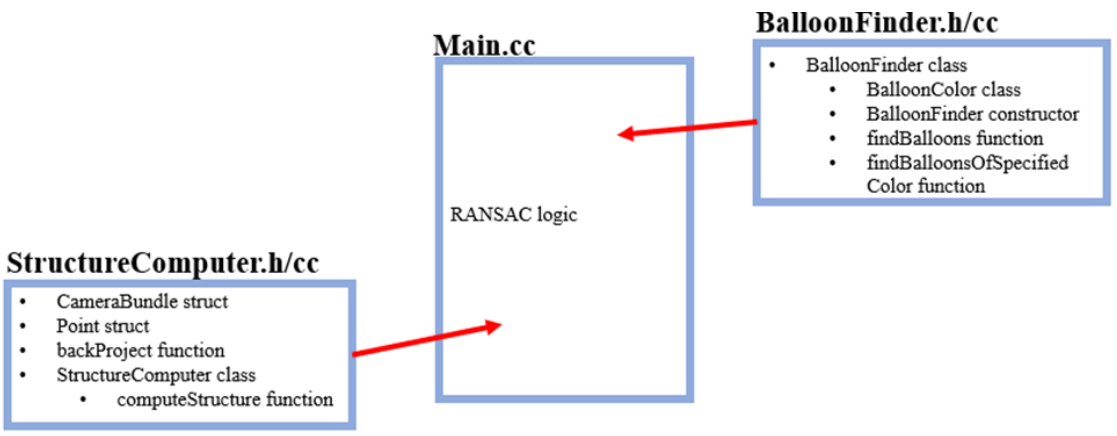
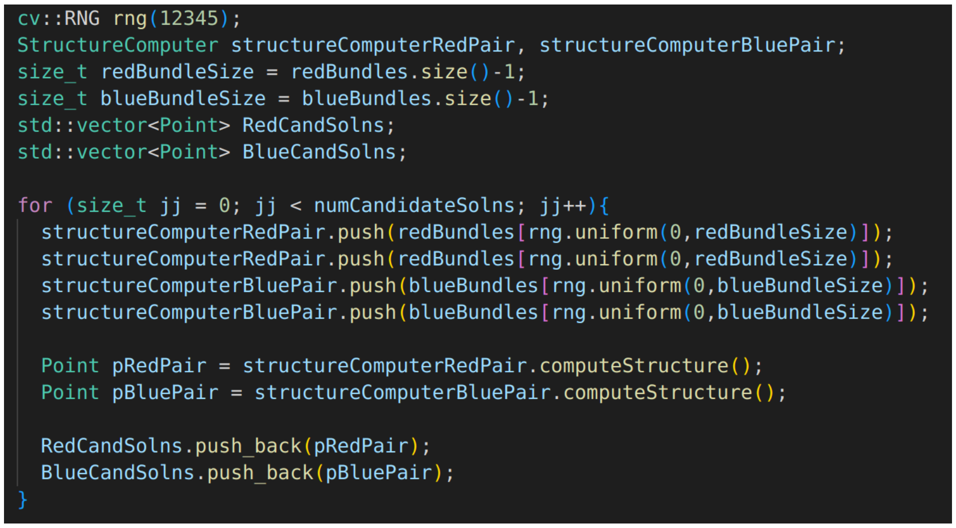


Figure 3.2.1: Vision test strategy implementation in software.

Within the main.cc file, a BalloonFinder object, StructureComputer objects for each balloon color, and a vector of CameraBundles for each color are instantiated. The metadata for each image within the tournament-train image set is retrieved, and each image is sent to the BalloonFinder object. The findBalloons function (which also utilizes the findBalloonsOfSpecifiedColor function) is used on the BalloonFinder object, and the CameraBundle objects found from the images are pushed into their respective vectors of CameraBundle objects based on the BalloonColor.

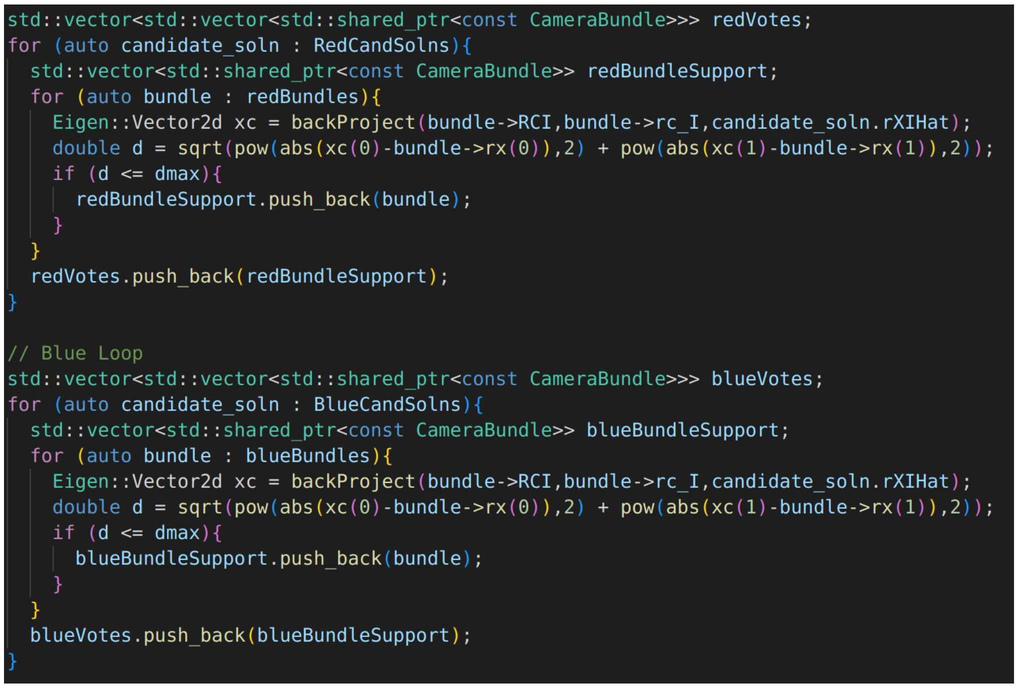
At this point, the RANSAC logic is implemented within main.cc. First, two parameters are set - the number of candidate solutions and the distance, dmax, the CameraBundle support threshold. Next, pairs of CameraBundle objects from the previous vector are pulled at random and pushed into a StructureComputer object for each BalloonColor. The Point structure solution is calculated for each StructureComputer object using the computeStructure function. These Points are pushed into vectors of Points for each BalloonColor. This code is shown in Fig. 3.2.2:



```
cv::RNG rng(12345);
StructureComputer structureComputerRedPair, structureComputerBluePair;
size_t redBundleSize = redBundles.size()-1;
size_t blueBundleSize = blueBundles.size()-1;
std::vector<Point> RedCandSolns;
std::vector<Point> BlueCandSolns;

for (size_t jj = 0; jj < numCandidateSolns; jj++){
  structureComputerRedPair.push(redBundles[rng.uniform(0,redBundleSize)]);
  structureComputerRedPair.push(redBundles[rng.uniform(0,redBundleSize)]);
  structureComputerBluePair.push(blueBundles[rng.uniform(0,blueBundleSize)]);
  structureComputerBluePair.push(blueBundles[rng.uniform(0,blueBundleSize)]);

  Point pRedPair = structureComputerRedPair.computeStructure();
  Point pBluePair = structureComputerBluePair.computeStructure();

  RedCandSolns.push_back(pRedPair);
  BlueCandSolns.push_back(pBluePair);
}
```

Figure 3.2.2: pairs of CameraBundle objects are pulled randomly and the estimated location is calculated

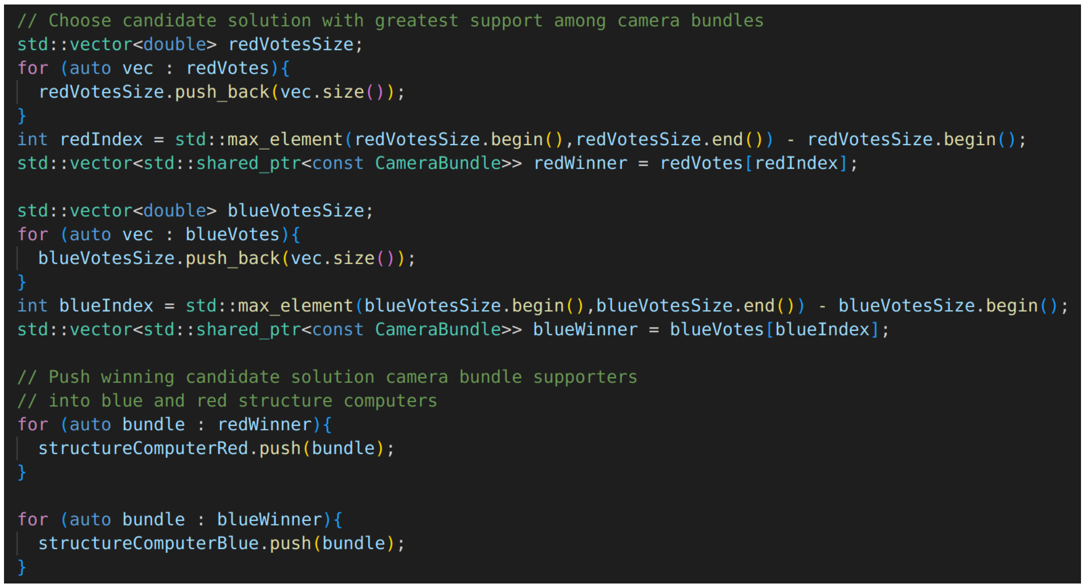
Next, the 3D location of each candidate solution is re-projected onto the images corresponding to the initial vector of CameraBundles found. The distance from the candidate solution to the measured balloon center of the image is calculated. If this distance is below the support threshold, then the CameraBundle object associated with that image offers its support for the candidate solution being tested. If the CameraBundle offers its support, then it is pushed back into a vector of CameraBundle shared pointers. After the iteration is complete for each individual candidate solution, the vector of CameraBundle shared pointers offering support is pushed into a vector of vectors of CameraBundle shared pointers. This is shown in Fig. 3.2.3.



```
std::vector<std::vector<std::shared_ptr<const CameraBundle>>> redVotes;
for (auto candidate_soln : RedCandSolns){
  std::vector<std::shared_ptr<const CameraBundle>> redBundleSupport;
  for (auto bundle : redBundles){
    Eigen::Vector2d xc = backProject(bundle->RCI,bundle->rc_I,candidate_soln.rXIHat);
    double d = sqrt(pow(abs(xc(0)-bundle->rx(0)),2) + pow(abs(xc(1)-bundle->rx(1)),2));
    if (d <= dmax){
      redBundleSupport.push_back(bundle);
    }
  }
  redVotes.push_back(redBundleSupport);
}

// Blue Loop
std::vector<std::vector<std::shared_ptr<const CameraBundle>>> blueVotes;
for (auto candidate_soln : BlueCandSolns){
  std::vector<std::shared_ptr<const CameraBundle>> blueBundleSupport;
  for (auto bundle : blueBundles){
    Eigen::Vector2d xc = backProject(bundle->RCI,bundle->rc_I,candidate_soln.rXIHat);
    double d = sqrt(pow(abs(xc(0)-bundle->rx(0)),2) + pow(abs(xc(1)-bundle->rx(1)),2));
    if (d <= dmax){
      blueBundleSupport.push_back(bundle);
    }
  }
  blueVotes.push_back(blueBundleSupport);
}
```

Figure 3.2.3: Candidate solution 3D locations are re-projected onto each image corresponding to the image CameraBundle objects for each BalloonColor.

Lastly, the candidate solution with the greatest support among CameraBundles is chosen. This is done by creating a vector of the sizes of each vector in the vector of vectors of CameraBundle shared pointers. The index of the largest sized vector is determined, and this is the vector of CameraBundle objects to be pushed into the red and blue StructureComputer objects. This is shown in Fig. 3.2.4:



```cpp
// Choose candidate solution with greatest support among camera bundles
std::vector<double> redVotesSize;
for (auto vec : redVotes){
  redVotesSize.push_back(vec.size());
}
int redIndex = std::max_element(redVotesSize.begin(),redVotesSize.end()) - redVotesSize.begin();
std::vector<std::shared_ptr<const CameraBundle>> redWinner = redVotes[redIndex];

std::vector<double> blueVotesSize;
for (auto vec : blueVotes){
  blueVotesSize.push_back(vec.size());
}
int blueIndex = std::max_element(blueVotesSize.begin(),blueVotesSize.end()) - blueVotesSize.begin();
std::vector<std::shared_ptr<const CameraBundle>> blueWinner = blueVotes[blueIndex];

// Push winning candidate solution camera bundle supporters
// into blue and red structure computers
for (auto bundle : redWinner){
  structureComputerRed.push(bundle);
}

for (auto bundle : blueWinner){
  structureComputerBlue.push(bundle);
}
```

Figure 3.2.4: candidate solution with the greatest CameraBundle support is chosen and pushed into red and blue StructureComputer objects.

The computeStructure function is used on each of these StructureComputer objects to determine the robust, estimated 3D location of the red and blue balloons.

Testing and debugging was performed using a built in debugging structure of the boost program options library. When executing the vision test code, a debugging mode could be enabled. When this mode was enabled, each image was displayed showing all contours found of the specified color. If a balloon was found, this was stated and the found center location, aspect ratio, radius, and true center location was output to the terminal. A circle was drawn around the contour that identified the balloon as well as a smaller circle around the found center of the balloon. The true center location of the balloon was also projected onto the image. All of this information allowed for the detailed analysis of the performance of the BalloonFinder functions on the tournament-train image set. Parameters within the BalloonFinder functions were adjusted until adequate performance was achieved.

The tournament-train image set was used for dialing in the RANSAC parameters to be set – the support threshold distance as well as the number of candidate solution pairs to be sampled. These were iterated on until the estimated 3D balloon locations were less than 10cm from their true locations.

Debugging was completed by printing out different structure values at specified points in the code to see if the values were close to what was expected at that point. This helped in locating and correcting any bugs.

4 Results and Analysis

4.1 Path Planning

When initially developing our path planning strategy, we opted to simply fly the quadcopter directly to each of the balloons and the goal location in a set order with generous arrival times to allow for velocity and acceleration limits. While this strategy was relatively quick to implement and execute, it failed whenever there was an obstacle impeding the direct path. To fix this, A* was then implemented along with the waypoint timing prevetter in order to generate trajectories around objects, and this resulted in a functional but slow path, averaging around 200 seconds without balloon teleportation to complete the final tournament “temple map” configuration. We determined that this slow speed was mainly due to the quad’s acceleration limits on the many sharp turns it needed to make from each cell waypoint to the next, so we then implemented the pruning algorithm to remove many of these waypoints. Because the quad was able to take smoother, more direct paths, the travel time shortened dramatically to an average of around 50 seconds with balloon teleportation and around 30 seconds without teleportation to complete the final tournament configuration. The above iteration process is outlined in Table 4.1.1 below.

Strategy	**Use case**	**Failure case**	**Key innovation**
Straight line with generous arrival times	At the beginning of the pre-tournament, when the arena contained no obstacles and speed was less of a concern	Failed when obstacles were introduced	Understanding the given structures and how to feed trajectories
A* implementation and automatic targeting priority	In the middle of the pre-tournament, when the arena contained some obstacles, but speed was still of less importance, and when inputting by hand which balloon to target first became unfeasible due to teleporting balloons	Failed when considering the goal was to have the fastest time	Creating and implementing A* algorithm, integrating with polynomial planner and trajectory structure.
Trajectory pruning and trajectory timing optimizations	Towards the end of the pre-tournament, when speed became a priority	Fails at high wind speeds, does not use an unordered map for further speed optimization	Creating and implementing pruning and timing algorithms

Table 4.1.1: Outline of sequential improvements over the course of the pre-tournament.

The final code was tested for the five different wind intensities, ranging from no wind to hurricane-speed winds. All trials were run with the temple map, and with no balloon teleportation, as the random timing of the balloon teleportation was not conducive to comparing times for various wind speeds. The results are shown in Table 4.1.2 below.

Wind Disturbance Level (0-4)	Elapsed Time (s)
0 - No wind	32.5752
1 - Mild	32.6144
2 - Stiff	32.5873
3 - Intense	47.1922, 33.1583, no completion
4 - Ludicrous	No completion

Table 4.1.2: Elapsed times for various wind disturbance levels.

From the table, it can be seen that the difference between levels zero through two were negligible, and the quad was able to successfully navigate the course, popping both balloons and arriving at the goal position. However, for level three, the results varied based on the randomness of the wind itself. For one trial, the quad successfully completed the mission in a time comparable to the previous wind levels. For another trial, however, the quad was not lucky in terms of the wind direction or gusts, and ran into some obstacles, causing it to freeze for a period of time, delaying the completion to roughly 47 seconds. For yet another trial, the quad did not complete the course at all, as it was stuck against an obstacle, and every time it would get free from the mediation layer penalty, it would be blown into the obstacle again in an endless loop, leading to a result of no completion. For wind level four, the quad was consistently overpowered by the wind and was blown against obstacles, unable to escape, leading to no completions.

4.2 Vision Test

Table 4.2.1 highlights the ability of the vision test code to find balloons using the tournament-train image set.

	Red Balloon		Blue Balloon	
	Without RANSAC	RANSAC	Without RANSAC	RANSAC
Estimated X error (m)	-0.29582	0.00082	0.52451	0.00112
Estimated Y error (m)	-0.11231	0.03110	-0.05302	-0.01051
Estimated Z error (m)	-0.09537	-0.00261	-0.02037	-0.03095
X Sqrt Diagonal Covariance Matrix (m)	0.00151	0.00175	0.00246	0.00274
Y Sqrt Diagonal Covariance Matrix (m)	0.00126	0.00144	0.00163	0.00184
Z Sqrt Diagonal Covariance Matrix (m)	0.00091	0.00104	0.00127	0.00140

Table 4.2.1: Balloon finding performance on tournament train image set with and without RANSAC.

As seen in the table, the RANSAC implementation decreases the error in the estimate in all but the Z location for the blue balloon. In these cases where the error in the estimate is decreased, all are decreased by at least an order of magnitude, except for the Y location of the blue balloon (which is still five times smaller than without RANSAC). Without RANSAC, only the estimated location error in Z for the red balloon and the estimated location error in Y and Z for the blue balloon were under 10cm. With RANSAC, the estimated 3D location error for both balloon colors along each axis were under the 10cm threshold.

Certain images within the tournament-train image set were especially difficult to process. A few of these images are highlighted below with the debugging output displayed on them. Image 00580 shown in Fig. 4.2.1 was challenging because only a red mat was displayed. The color fell within the HSV limits for red set in the findBalloonsOfSpecifiedColor function. The contour and radius found for the mat also within the allowed, causing the mat to be identified as a balloon and contributing to a false identification or outlier in this case.

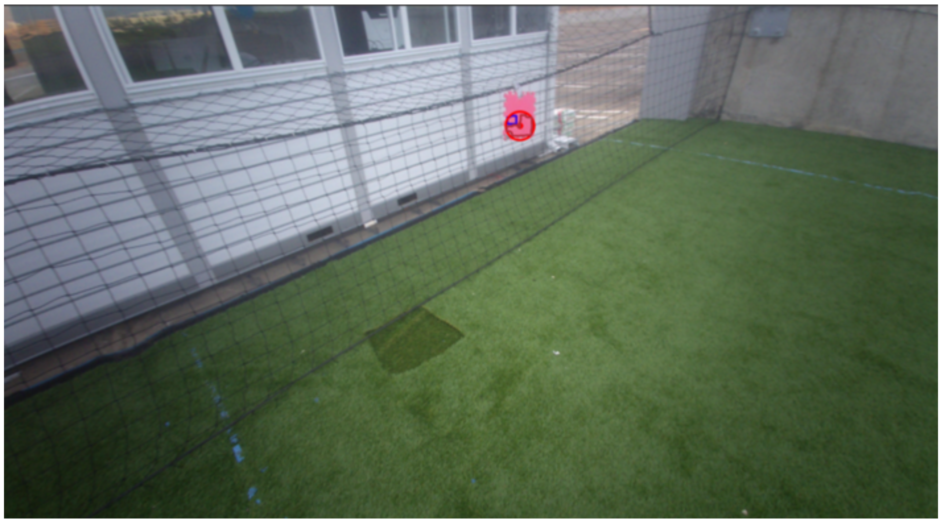


Figure 4.2.1: tournament-train mage 00580

Image 00602 displayed in Fig. 4.2.2 below was tough because there are two red balloons, but the true balloon location is not identified because it sits along the edge of the image. The second red balloon in the incorrect location is identified and contributes an outlier to the measurement data.

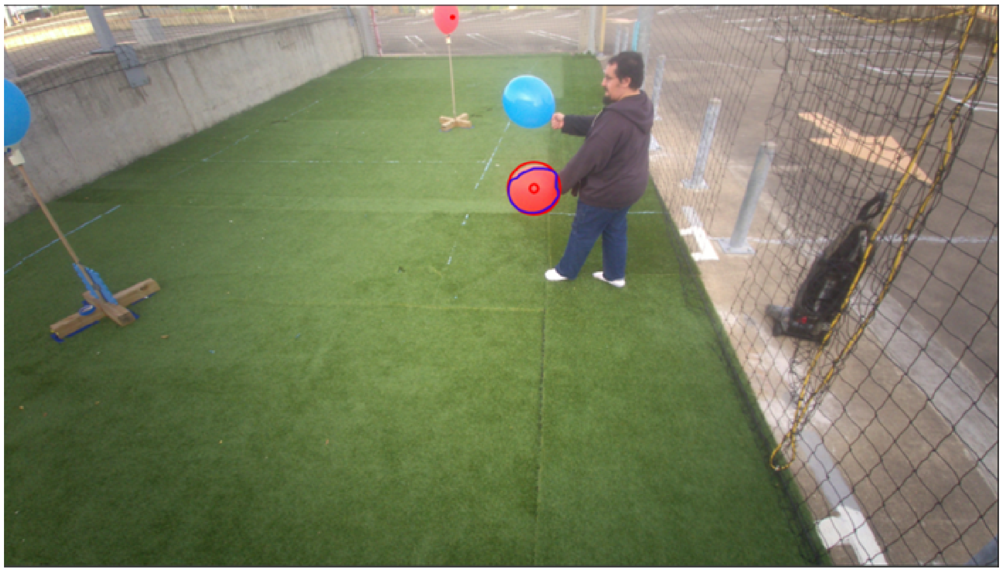


Figure 4.2.2: tournament-train image 00602

Image 00616 in Fig. 4.2.3 is similar to 00602, but the true red balloon does not sit along the edge of the image. It was still not identified as a red balloon though, because the contour of the red color identified within the balloon does not possess an aspect ratio and radius within the accepted limits. The second red balloon was identified and contributed an outlier to the measurement.

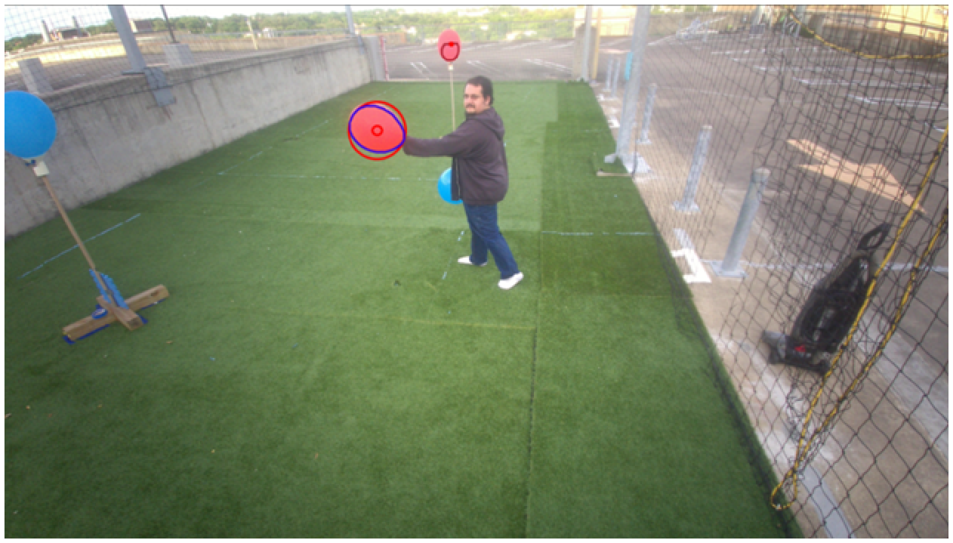


Figure 4.2.3: tournament-train image 00616

Image 00664 in Figure 4.2.4 was challenging because there are two blue balloons within the image. The location of the true balloon was not identified because the contour is obstructed by the TA's head. This distorts the aspect ratio of the contour, and the balloon was not found. The second blue balloon in the incorrect location is unobstructed and therefore identified. This contributed a blue balloon location outlier to the data set.

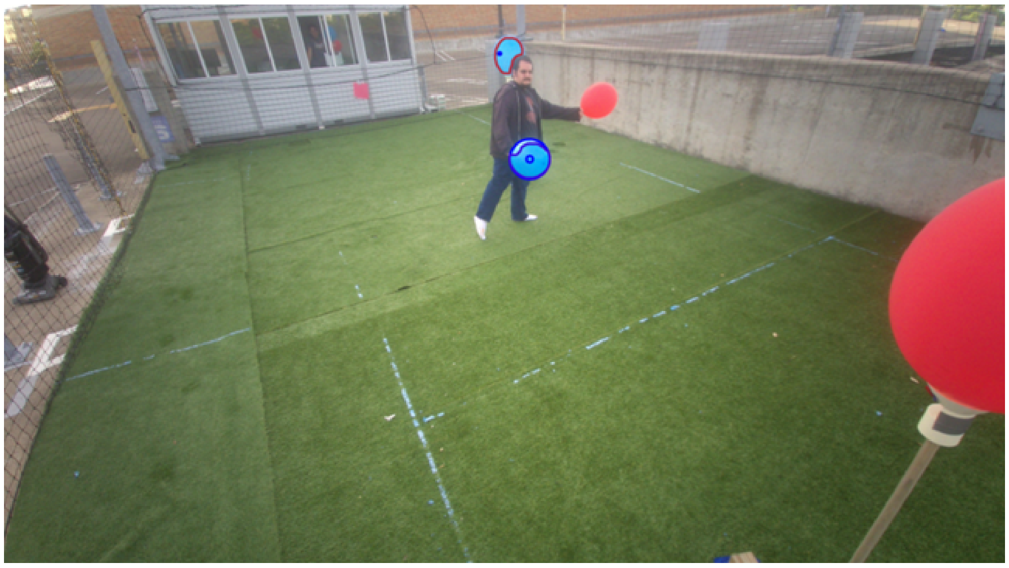


Figure 4.2.4: tournament-train image 00664

Image 00800 shown in Fig. 4.2.5 was difficult because there are two blue balloons within the image. Both balloons are identified, but since one of the balloons is not the true balloon, this adds an outlier to the measurements.

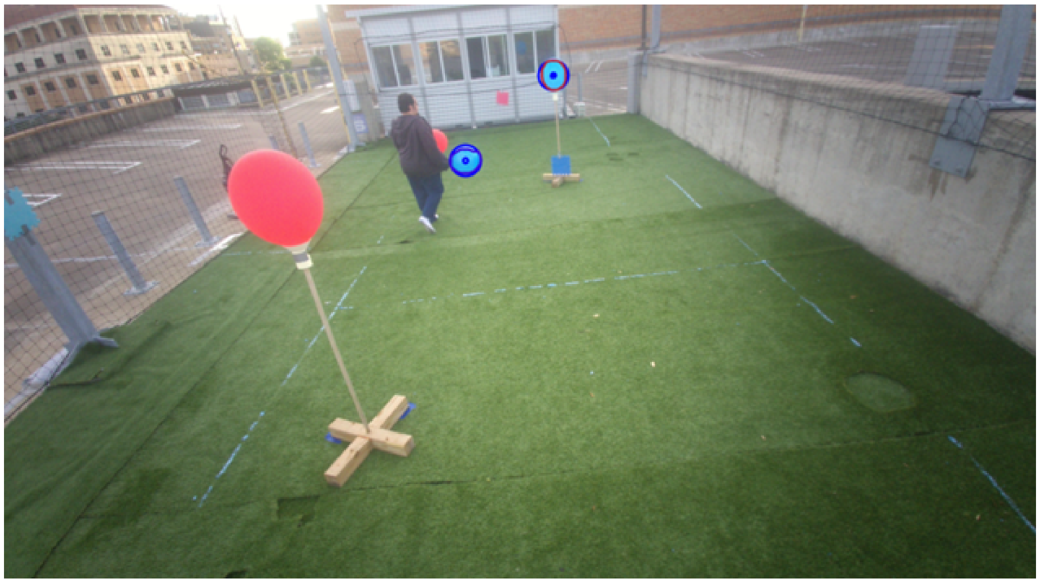


Figure 4.2.5: tournament-train image 0800

5 Conclusion

One goal of the project was to program a quadcopter such that it could pop two balloons and return to a specified end position in a timely manner, while avoiding obstacles and accounting for potential teleportation of the balloons. This goal was successfully achieved, with the quad completing the course reliably for reasonable wind values, within a reasonably fast time. The path planning strategy was discussed and the implementation in software was presented. The gradual improvement of the path planning strategy over time was shown and the performance in varying wind intensities were compared.

A second goal of the project was to identify red and blue balloons from a series of images. This goal was also achieved. Robust estimation to outliers was achieved using RANSAC to estimate the red and blue balloon center locations to within 10 cm based on the tournament-train image data set and its metadata. The theory behind RANSAC was described, and the implementation in code was summarized. A comparison to estimation without robust methods was presented and challenging images for the estimation task were highlighted.

6 Contributions

Azeem wrote and integrated the A* implementation and the targeting logic. George wrote the pruning method and worked on strategy organization and code integration for the trajectory planner. Justin wrote the vision test code and helped with the trajectory timing implementation for the trajectory planner. All three team members contributed equally to testing and debugging as well as writing the final report.

References

[1] *Robots that know what they do - uzh*. (n.d.). Retrieved April 30, 2023, from https://rpg.ifi.uzh.ch/docs/Visual_Odometry_Tutorial.pdf

[2] Fischler, M. A., & Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM, 24(6), 381-395.