ASE 479W Tournament
The Grid Explorers

Kumar Aditya, Aryaman Singh Samyal, Jianhua Liang

May 16, 2022

1 Introduction

A self-sufficient quadcopter is one that can detect and respond to changes to its dynamics and kinematics using a variety of tools such as Kalman filters to estimate its state and PID controllers to respond properly to those estimates to follow a desired trajectory. However, to be truly autonomous, a quadcopter must not only be able to perform those activities, but also choose its own desired trajectory rather than having one given to it to follow. To make such a system, it is important for the vehicle to integrate several components such as computer vision for perception, path-planning and accurate state-estimation and correction in real-time.
For the final project of the class, we implemented the path-planning and navigation module of an aerial vehicle's autonomy stack, to enable it to successfully go through an obstacle course to several target locations which are identified by colored balloons. The project also involved a separate portion in which we implemented a modified version of Lab 4, that improves the precision of the triangulation method that we used by integrating it with RANSAC, thus reducing the effect of outliers on the predictions. Furthermore, for the path-planning component, we implemented the A* algorithm on a 3D grid and feed the trajectory to the drone.

2 Theoretical Background

In this section, we explain the theory behind the different methods that were implemented for our project.

2.1 A* search algorithm

A* search. Algorithm 1 overviews the implementation of A* search that was employed for the project. It explores nodes to find a shortest path P by iterating between (1) selecting the most promising node from the list of candidates for constructing a shortest path and (2) expanding neighbors of the selected node to update the candidate list, until the goal v_{g} is selected. More specifically, the node selection (Line 3 of Algorithm 1) is done based on the following criterion:

$$v^* = \arg\min_{v \in \mathcal{O}} \left(g(v) + h(v) \right), \tag{1}$$

where $\mathcal{O} \subset \mathcal{V}$ is an *open list* that manages candidate nodes for the node selection. $g(v)$ refers to the actual total cost accumulating $c(v')$ for the nodes v' along the current best path from v_{s} to v, which is updated incrementally during the search. On the other hand, $h(v)$ is a heuristic function estimating the total cost from v to v_{g}, for which the straight-line distance between v and v_{g} is often

Algorithm 1 A* Search

```
Input: Graph $\mathcal{G}$, movement cost $c$, start $v_\mathrm{s}$, and goal $v_\mathrm{g}$
Output: Shortest path $P$
 1: Initialize $\mathcal{O} \leftarrow v_\mathrm{s}$, $\mathcal{C} \leftarrow \emptyset$, $\texttt{Parent}(v_\mathrm{s}) \leftarrow \emptyset$.
 2: while $v_\mathrm{g} \notin \mathcal{C}$ do
 3:     Select $v^* \in \mathcal{O}$ based on Eq. (1).
 4:     Update $O \leftarrow O \setminus v^*$, $\mathcal{C} \leftarrow \mathcal{C} \cup v^*$.
 5:     Extract $\mathcal{V}_\mathrm{nbr} \subset \mathcal{V}$ based on Eq. (2).
 6:     for each $v' \in \mathcal{V}_\mathrm{nbr}$ do
 7:         Update $\mathcal{O} \leftarrow \mathcal{O} \cup v'$, $\texttt{Parent}(v') \leftarrow v^*$.
 8:     end for
 9: end while
10: $P \leftarrow \texttt{Backtrack}(\texttt{Parent}, v_\mathrm{g})$.
```

used in the grid world. All the selected nodes are stored in another list of search histories called *closed list*, $\mathcal{C} \subseteq \mathcal{V}$, as done in Line 4.

In Line 5 of Algorithm 1, we expand the neighboring nodes of v^* as $\mathcal{V}_\mathrm{nbr} \subset \mathcal{V}$ based on the following criterion:

$$\mathcal{V}_\mathrm{nbr} = \{v' \mid v' \in \mathcal{N}(v^*) \land v' \notin \mathcal{O} \land v' \notin \mathcal{C}\}. \tag{2}$$

The neighbor nodes $\mathcal{V}_\mathrm{nbr}$ are then added to $\mathcal{O}$ in Line 7 to propose new selection candidates in the next iteration. The search is terminated once v_g is selected in Line 3 and stored in $\mathcal{C}$, followed by `Backtrack` that traces the parent nodes $\texttt{Parent}(v)$ from v_g to v_s to obtain a solution path P. For further information, please refer to original journal paper [1].

2.2 Piecewise Polynomial Path Planner (P4)

Given a set of waypoints and associated arrival times, the P4 solver finds a set of piece-wise polynomials that smoothly fit through the points and maintain continuity up to a specified derivative. At the core of the polynomial solver is a cost-minimization loop formulated by as a quadratic programming problem and solved with OSQP (Operator Splitting QP Solver). The solver fits the polynomials while minimizing the squared norm of a specified derivative and adhering to specified, derivative-based path constraints. This approach follows the minimum snap problem formulation and solution published by [2].

The formulation of the problem statement can be given as follows:

Suppose you have a set of waypoints (which in our use-case is determined by the A^* search algorithm) defined by 3 along with a set of time stamps at which you need to arrive at them given by 4.

$$\mathbf{W} = \begin{bmatrix}\vec{w}_0 & \vec{w}_1 & \vec{w}_2 & \dots & \vec{w}_m\end{bmatrix} \in \mathcal{R}^a \tag{3}$$

$$\vec{T} = \begin{bmatrix}t_0 & t_1 & t_2 & \dots & t_m\end{bmatrix} \tag{4}$$

The solver finds a solution to determine a set of n'th-order piecewise-continuous polynomials that smoothly connect these waypoints given by 5

$$\vec{p}(t) = \begin{cases} p_0(t) = \sum_{k=0}^{n} p_{0,k} \cdot (\frac{1}{k!}t^k) & t_0 \leq t < t_1 \\ p_1(t) = \sum_{k=0}^{n} p_{1,k} \cdot (\frac{1}{k!}t^k) & t_1 \leq t < t_2 \\ \vdots \\ p_m(t) = \sum_{k=0}^{n} p_{m,k} \cdot (\frac{1}{k!}t^k) & t_{m-1} \leq t < t_m \end{cases} \tag{5}$$

while minimizing the squared norm of the rth derivative

$$\min \int_{t_0}^{t_m} \frac{d^r}{dt^r} ||\vec{p}(t)||^2 dt \tag{6}$$

being subject to path and continuity constraints. For further information, please refer to original journal paper [2].

2.3 Structure Computer

The position finding algorithm was based on the image sensor projection model described in Lab 3. To summarize this model,

$$\begin{bmatrix} fX_\mathrm{c} \\ fY_\mathrm{c} \\ Z_\mathrm{c} \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_\mathrm{CI} & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

This can be further simplified to

$$\mathbf{x} = K \begin{bmatrix} R_\mathrm{CI} & t \end{bmatrix} \mathbf{X} = P\mathbf{X}$$

where $\mathbf{x}$ is the coordinates of a feature on the image sensor, P is a projection matrix that is dependent on the camera's intrinsic parameters such as K that contains its focal length, and extrinsic parameters such as $\mathrm{R_{CI}}$ and t that contain the camera's pose and a translation to handle the non-coincident centers of the camera and quadcopter, and $\mathbf{X}$ is the original inertial coordinates of the feature detected.

To solve for the original inertial coordinates, this model was set up as a linear least squares problem based on the observation that the cross product of $\mathbf{x}$ with P$\mathbf{X}$ is equal to 0. The solution was found as presented below.

$$X = ({H_\mathrm{r}}^\mathrm{T} R^{\text{-}1} H_\mathrm{r})^{\text{-}1} {H_\mathrm{r}}^\mathrm{T} R^{\text{-}1} z$$

where $\mathrm{H_r}$ and z are the first three and last columns of the matrix H(shown below) respectively and R is a diagonal matrix with the pixel level covariances filling the diagonal multiplied by the pixel size square.

$$H = \begin{bmatrix} x_1 {p_1}^{3\mathrm{T}} - {p_1}^{1\mathrm{T}} \\ y_1 {p_1}^{3\mathrm{T}} - {p_1}^{2\mathrm{T}} \\ \dots \\ x_\mathrm{n} {p_\mathrm{n}}^{3\mathrm{T}} - {p_\mathrm{n}}^{1\mathrm{T}} \\ y_\mathrm{n} {p_\mathrm{n}}^{3\mathrm{T}} - {p_\mathrm{n}}^{2\mathrm{T}} \end{bmatrix}$$

H is filled vectors with the x or y projected locations of the feature in meters multiplied by the third row from that camera's projection matrix and subtracted by either the first(for x) or second(for y) row from the projection matrix from the first to the nth camera. The solution's error covariance matrix can be calculated as shown below.

$$P_{\mathrm{x}} = (H_{\mathrm{r}}{}^{\mathrm{T}} R^{-1} H_{\mathrm{r}})^{-1}$$

2.4 RANSAC

Despite the precision of the structure computer aforementioned, it still assumes that measurement noise is zero mean with a Gaussian distribution and known error covariance. However, in practice this may not be the case in many situations and thus the performance of this linear least squares solution must be improved for real world use. One method of doing so is RANSAC, otherwise known as Random Sample Consensus. In this method outlier data points that have a negligible probability of being drawn from the measurement error distribution that is being modeled can be discarded so as to improve the robustness of the model used to estimate the solution. RANSAC is a simple voting-based algorithm that randomly samples the population of points and aims to find a subset of those lines which appear to conform to a model. It essentially performs the least squares method on subsets of the entire dataset and converges to the model which has the highest number of inliers (determined by the distance threshold) for the hypothesized line.

The algorithm at its simplest is as follows.

1. Select a minimal subset of data points containing barely sufficient data to make the least squares problem solvable.
2. Solve the least squares problem.
3. Calculate the support of the solution. The support is defined as the number of data points lying a threshold distance from the line generated by the solution.
4. Repeat steps 1 to 3 many times trying to find the solution with the largest support.
5. If the largest support exceeds some support threshold, solve the problem based on the data points for this largest support and use it as the robust solution.

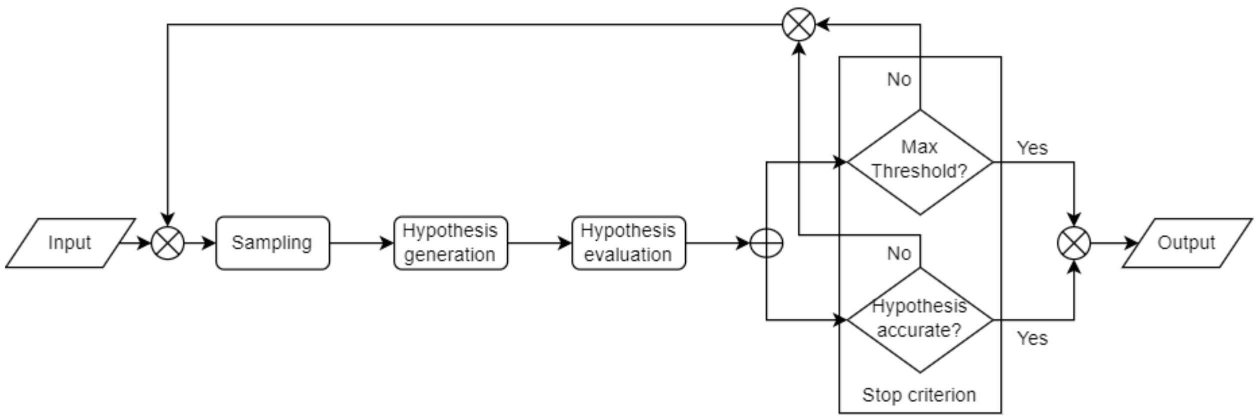


Figure 1: RANSAC pipeline

3 Implementation

The entire implementation of the project was done in C++, as being a statically-typed, pre-compiled programming language, it is much faster and efficient that other languages (Python) for robotic utilization. Further, apart from the basic, essential C++ libraries, we used OpenCV to use it's morphological and image transformation functions for easier localization of the target balloons. We also used Eigen for various matrix operations due to its ease of use compared to native C++ arrays and the lack of need to implement various matrix operations on our part.
In the upcoming sections, we give a comprehensive explanation of all the components of our autonomy stack.

3.1 Path-planning

For this project, we started off using the same A* algorithm implemented in Lab 4 in `a_star2d.cc` but modified for 3 dimensions rather than only 2 dimensions. To do this, we started by modifying `a_star2d.cc` and its header filer `a_star2d.h` to have the `Run` function accept a `Graph3D` object and two node pointers that each contained a `Node3D` object. Additionally, we changed the Heuristic function so it was compatible with the additional z data value each node contained.

3.2 Computer Vision

For the vision test, we started off using the same OpenCV algorithm implemented in Lab 5 in `balloonfinder.cc`. This algorithm essentially consists of converting a given image from RGB to HSV so that color would be constrained to one channel. This converted image is then filtered for values of red and blue for each respective balloon color. These filtered image is then eroded and dilated so as to remove most noise with the filtered color. Next, contours are generated for any objects of the desired color within the frame. These contours are then assessed for aspect ratio, number of vertices within the contour, and other factors to determine if the object represented by said contour is a balloon of the desired filtered color.
The other part of the vision test was implementing a triangulation method given by structure computer to take in a set of balloon image coordinates along with the respective metadata for each image including camera inertial location and pose, and determining the inertial location of the balloon. This also was started off by using a previously implemented algorithm (described in Section 2.3 above) from Lab 5 in `structurecomputer.cc`. To further enhance this algorithm's process, a form of RANSAC was implemented as well so as to make the structure computer's estimation of the balloon's inertial position more robust. Due to the nature of this least squares problem, the algorithm started of with a minimal subset of 2 images. It then ran throughout the full data trying to eliminate any exceptional outliers. Additionally, this implementation used 15 pixels as its distance threshold and 5 as its support threshold.
Another critical aspect of the implementation was tuning our `balloonfinder.cc` to accurately localize the balloon targets. As the dataset involved multiple objects such as fake balloon and colored foam boards, our task involved a combination of both RANSAC and relevant image transformation functions for precise detection. We took the following steps to tune our `balloonfinder.cc`:

- We implemented OpenCVs `approxPolyDP` function to calculate the number of edges of the detected objects. This allowed us to filter out the square foam boards as the balloons, being elliptical has a large number of computed edges.

- We used the `contourArea` function to calculate area enclosed by the contours. By setting a minimum threshold for the detection area, we reduced the number of false positives, which were mostly in the form of smaller circles.
- We used the `fitEllipse` function to generate the rotated rectangle of the contour. Then using this, we calculate the aspect ratio (AR) of the detection. The reason we did this was we noticed that the fake balloon targets that the TA was holding in the dataset were usually parallel and horizontal to the ground. This resulted in a different aspect ratio of these fake targets when compared to the actual target balloons which were perpendicular to the ground. Therefore, by setting a threshold for detection's AR, we were successfully able to increase our algorithms precision by making it more robust to the fake targets.

A summary of our computer vision pipeline is given by Figure 2.

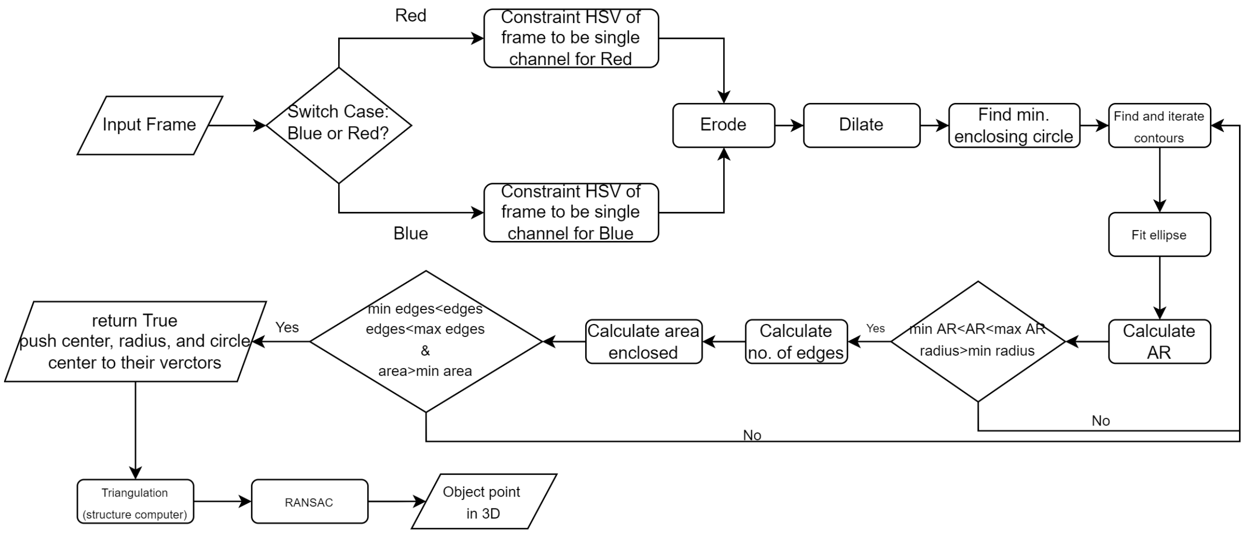

Input Frame
Switch Case: Blue or Red?
Red
Blue
Constraint HSV of frame to be single channel for Red
Constraint HSV of frame to be single channel for Blue
Erode
Dilate
Find min. enclosing circle
Find and iterate contours
Fit ellipse
Calculate AR
min AR<AR<max AR radius>min radius
Yes
No
Calculate no. of edges
Calculate area enclosed
min edges<edges edges<max edges & area>min area
Yes
No
return True push center, radius, and circle center to their verctors
Triangulation (structure computer)
RANSAC
Object point in 3D

Figure 2: Computer vision pipeline

3.3 Autonomy Protocol

Our autonomy protocol utilized the A^* search algorithm for finding the appropriate path to get to the target balloons. In the `student_autonomy_protocol.cc` we feed the graph of the arena, which is a `OccupancyGrid3D` object along with two pointers pointing towards values of the position (in the form of `Node3D` of the vehicle and the current target. When initializing, the current target is set as the closest balloon. When the first balloon is burst, the target is set to the second balloons, and after that the initial home position. A critical step that is done before calling A^* is converting the physical grid to the computational grid using the `mapToGridCoordinates` method. The opposite conversion is required after A^* return the object.
Further, we feed the output from A^* to the $P4$ solver. Here we use a custom variable `P4_inflate` which essentially helps us reduce the number of nodes that are received from A^*. We do this to further smoothen our trajectory and try and make it as similar to a straight line as possible, due to the fact that a straight is the shortest distance between two points. Our P4 solver then takes in this reduced set of points and generates a polynomial spline fitting them, under the specified constraints.

After we receive the trajectory, we call the `PreSubmissionTrajectoryVetter` to check if out trajectory is satisfying all constraints or not. If it is not, then we update parameters of our code depending upon `prevetter_response.code`. Most of the violations can be corrected by calling the `UpdateTrajectories` function again by updating the `dt_factor` value. This value essentially specifies the time stamp between each P4 solver sample. Therefore, by increasing it, we directly affect acceleration and velocity of the vehicle. Other violations can be corrected by updating the `P4_inflate` values which increases the resolution of the trajectory generation.
Finally we check if the new trajectory is faster than our previous trajectory. If this condition is affirmative, we send the trajectory the mediation layer for further processing.
The overview of our autonomy protocol is given by Figure 3.

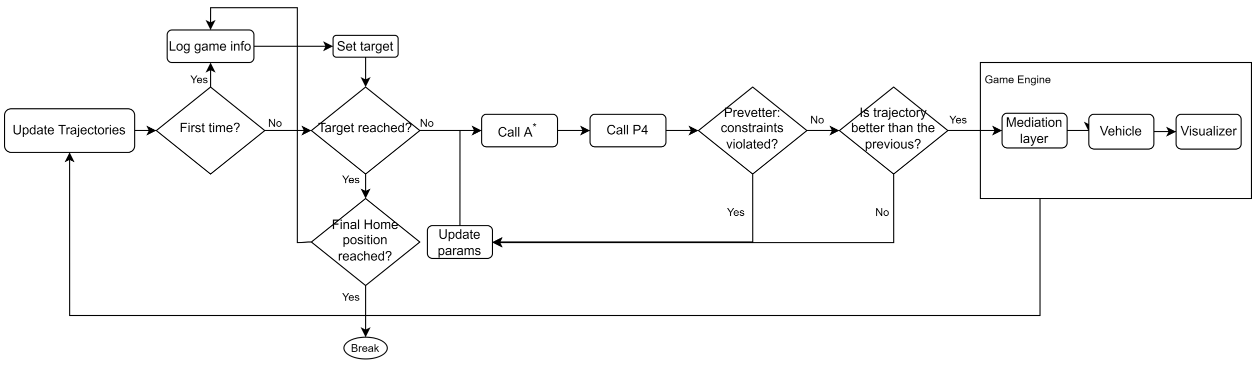


Figure 3: Autonomy protocol pipeline

4 Results and Analysis

4.1 Vision Test

For the vision test, the computer vision code from Lab 5 was first tested on the `tournament-train` dataset to check its precision. Next, we implemented RANSAC to check if the precision could be enhanced to be below the required threshold limit of $\pm 10cm$. However, the resultant algorithm did not totally achieve that threshold for particularly the blue balloon, which motivated us to tune and apply the various in-build OpenCV image processing functions to further increase the accuracy and finally achieve the desired error threshold. The summary of our computer vision testing results is given by Table 1.

Location error	Barebone	RANSAC	RANSAC+tuning
Blue	1.76588	3.54573	-0.0311681
	0.768879	-4.6995	-0.0450833
	0.0223811	-0.182959	-0.0147187
Red	0.383411	-0.0215964	-0.024622
	0.768879	-0.017276	0.00447609
	0.0223811	0.00790593	0.00661045

Table 1: Location error (top to bottom - x,y,z axis) for the CV pipeline

As shown below (Figure 4), despite the OpenCV optimizations, there are still certain errors that were able to be avoided such as the image on the right where the blue foam square is mistakenly marked as a balloon. This is where RANSAC helped avoid outliers due to it not counting outliers as a support, most data subsets generated through RANSAC would avoid outliers as they lead to lower support counts.

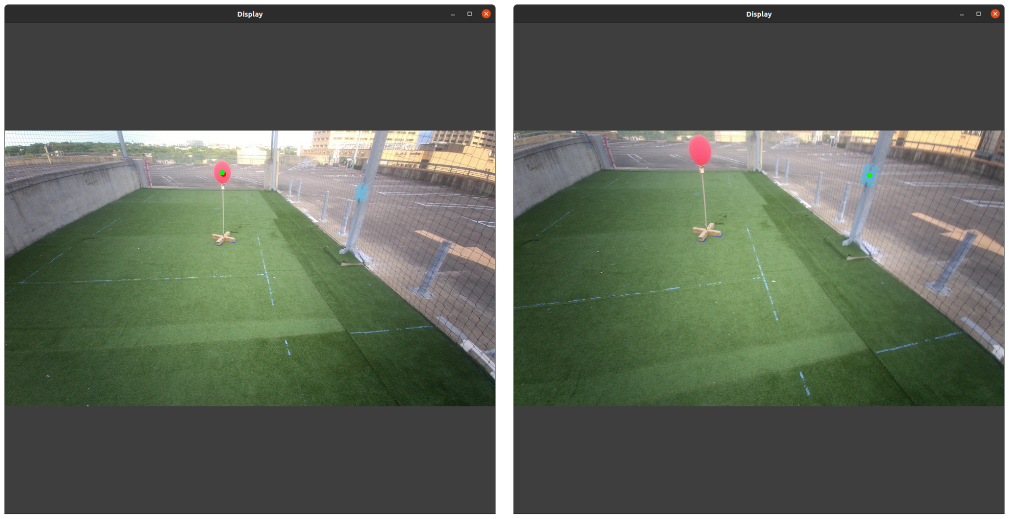


Figure 4: Example of inliers and outliers that RANSAC helps to select between. Left - After integrating RANSAC and tuning; Right - Without RANSAC

4.2 Obstacle Course

For the obstacle course, we tested our autonomy protocol by running it on the tournament map as shown below.

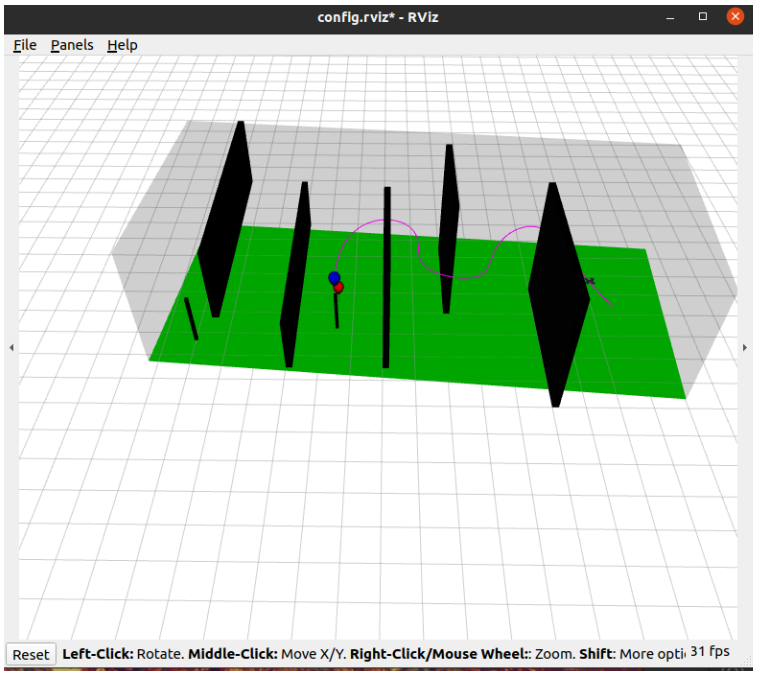

config.rviz* - RViz
File
Panels
Help
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More opti
31 fps

Figure 5: Autonomy Protocol running on Tournament Map towards closest balloon

As can be seen above (Figure 5), the protocol successfully finds a path around the various obstacles and generates a viable trajectory to the closest balloon for the quadcopter to follow. After popping both balloons, it also successfully finds a path back to its initial position as shown below (Figure 6) while respecting the mediation layer's rules on getting too close to the obstacles so as to not collide or get stuck in place by the mediation layer.

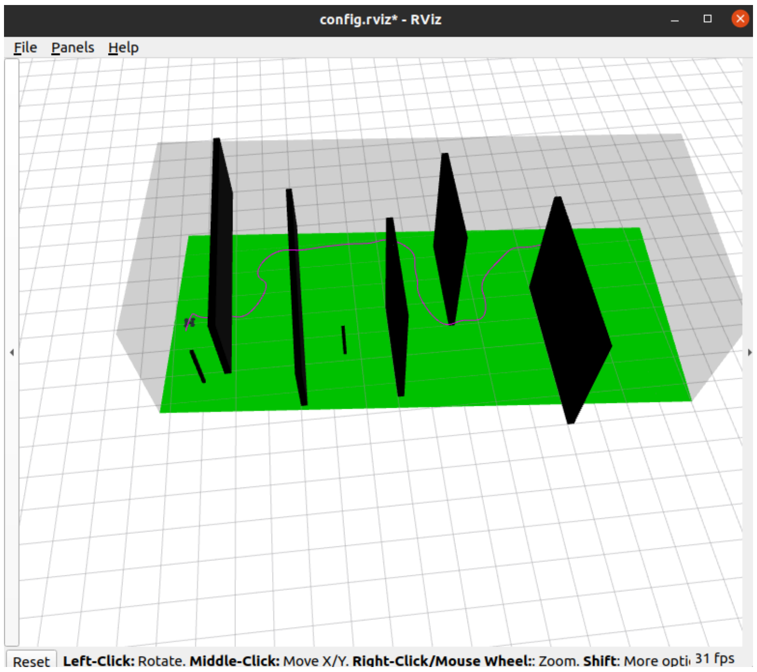
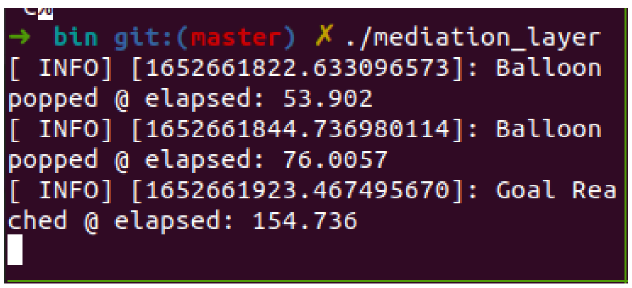

config.rviz* - RViz
File Panels Help
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More opti
31 fps

Figure 6: Autonomy Protocol running on Tournament Map back home

The timing of the autonomy protocol is presented below (Figure 7) as reported by the mediation layer. As shown below, it falls beneath the maximum allowed time limit of 300 seconds and finishes the whole trip in just slightly over half that time.



```
➜  bin git:(master) ✗ ./mediation_layer
[ INFO] [1652661822.633096573]: Balloon
popped @ elapsed: 53.902
[ INFO] [1652661844.736980114]: Balloon
popped @ elapsed: 76.0057
[ INFO] [1652661923.467495670]: Goal Rea
ched @ elapsed: 154.736
```

Figure 7: Autonomy Protocol Results according to Mediation Layer

5 Conclusion

For this final project, two computer vision and a path-finding algorithms were implemented in order to guide a quadcopter to determine the locations of two teleporting red and blue balloons, fly around obstacles to reach and pop them, and return back to its initial position. The computer vision algorithm was then tested and verified to be accurate within less than $\pm 5cm$. Meanwhile, the path-finding algorithm was integrated into an autonomy protocol for the quadcopter to determine the exact trajectory towards its targets and back home. This autonomy protocol proved to succeed in its task in a relatively decent time of about 155 seconds.

6 Contributions

Jianhua worked with the autonomy protocol. Aryaman worked on the OpenCV section of the vision test. Kumar worked on implementing RANSAC for the vision test. Everyone worked together on testing and debugging the code. Both Aryaman and Kumar worked on the report. All three members contributed equally to the project.

References

[1] P. Hart N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, 1968, pp. 100-107, doi: 10.1109/TSSC.1968.300136.

[2] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," 2011 IEEE International Conference on Robotics and Automation, 2011, pp. 2520-2525, doi: 10.1109/ICRA.2011.5980409.