Mini Drone Race-USING 1 LATE DAY

Mrinalgouda Patil
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: mpcsdspa@gmail.com

Curtis Merrill
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: curtism@umd.edu

Ravi Lumba
Alfred Gessow Center of Excellence
University of Maryland
College Park, Maryland 20742
Email: rlumba@umd.edu

***Abstract*—This paper presents an approach for target identification and navigation. First, a Gaussian Mixture Model was created to detect a yellow window. Next, a closed loop controller was created and tuned such that any arbitrary trajectory could be followed. Finally, these two methods were integrated such that a yellow window could be detected and flown through.**

I. Introduction/Problem Statement

The goal of this project was to navigate through a colored window (Figure 1) of known size but unknown position and orientation. It involves the estimation of the window pose in 3D and implementation of trajectory planner and control algorithm to go through the window.

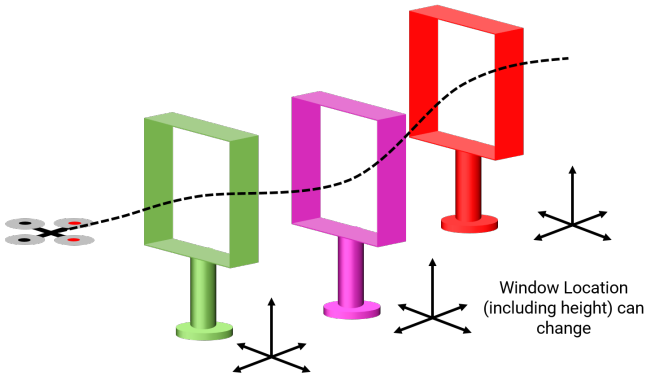

Fig. 1. The Project goal is to fly through a colored gate.

II. Camera Calibration

The first step in the project was to calibrate the camera provided. Both color and intrinsic calibration were performed.

A. Color Calibration

The camera was color calibrated using by tuning the bcorrect, gcorrect and rcorrect values. The calibration was performed for 800 $\times$ 460 resolution as this was used for the subsequent tests.

B. Intrinsic Calibration

Camera Intrinsic calibration entails with estimating the camera calibration matrix K which includes the focal length and the principal point and the distortion parameters. We used the calibration package **Kalibr** developed by ETHZ. An april grid was used to calibrate the camera. The following is the K matrix after calibration.

$$K = \begin{pmatrix} 702.932 & 0 & 427.86 \\ 0 & 700.346 & 227.57 \\ 0 & 0 & 1 \end{pmatrix} \quad (1)$$

III. Color Identification

Talk about how broken into different components. Color thresholding, single Guassian, and finally Guassian Mixture Model

A. Color Thresholding

Color thresholding is a process where colors are selected for simply by keeping pixels that fall within a certain treshold of values and throwing out the rest. Color thresholding worked really well when it was tuned to a particular instance. However, once the lighting conditions were varied even a small amount, the thresholding became really unreliable. The lack of robustness in different lighting conditions limited the usefullness of color thresholding.

B. Single Gaussian

For efficiency, the Single Gaussian was implemented with the GMM and setting k=1. The Single Gaussian was more robust than the color thresholding, but still struggled with lighting variations that were very significant.

C. Gaussian Mixture Model

Gaussian Mixture Model (GMM) was implemented to detect the window. The model was trained with camera images obtained from different lighting conditions and orientations. It was also trained with different number of clusters. The prediction results obtained from GMM were good. If the model was trained with some low light images, it would even show the violet color in the test set. This caused problems, hence the model was not trained with very low light images. The below figures show the GMM model predictions obtained for two cases namely, low lighting and high lighting for different number of clusters: 2, 3 and, 6.

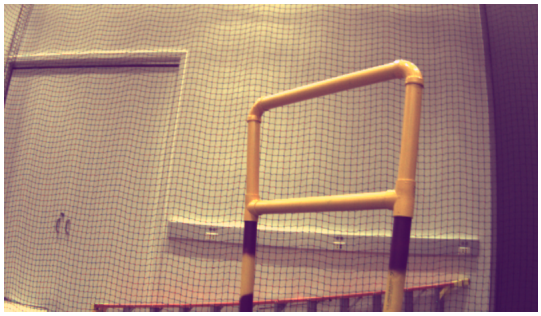

Fig. 2. Camera image with high lighting

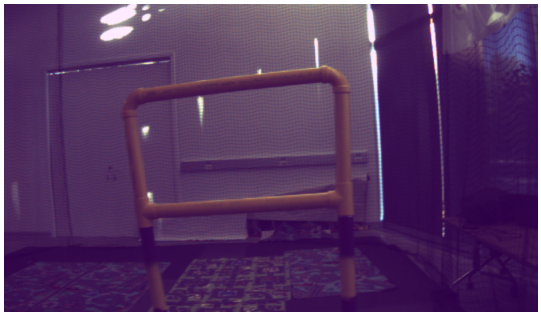

Fig. 3. Camera image with low lighting

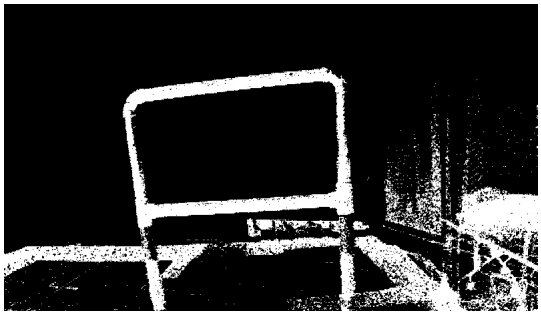

Fig. 4. GMM model with k = 2 for low-lighting image

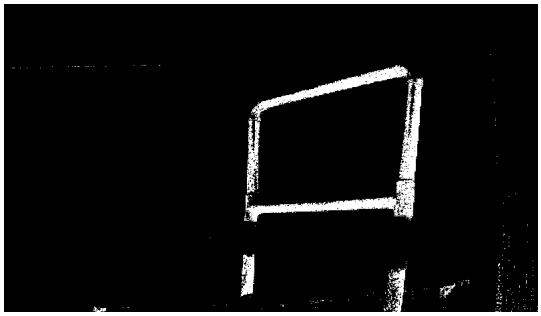

Fig. 5. GMM model with k = 2 for high-lighting image

1) Number of clusters, k = 2:

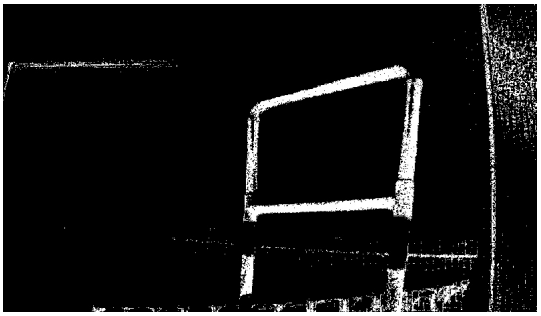

Fig. 6. GMM model with k = 3 for low-lighting image

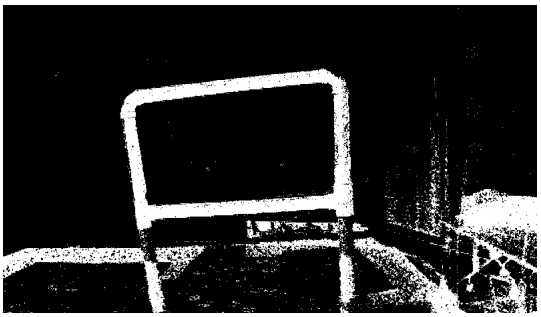

Fig. 7. GMM model with k = 3 for high-lighting image

2) Number of clusters, k = 3:

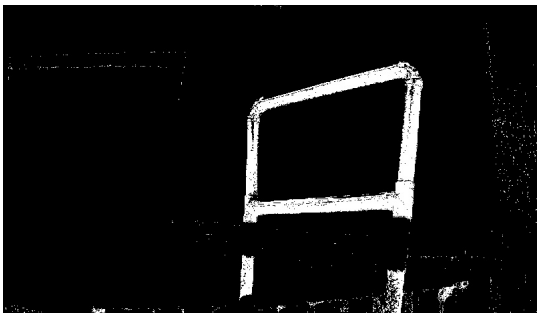

Fig. 8. GMM model with k = 6 for low-lighting image

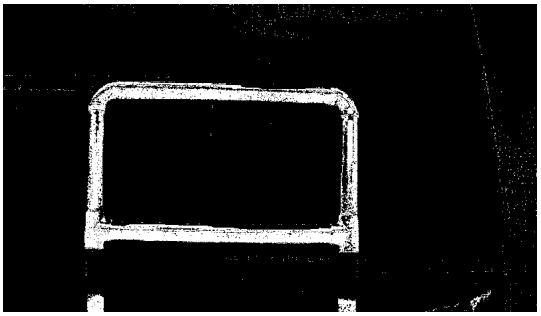

Fig. 9. GMM model with k = 6 for high-lighting image

3) Number of clusters, k = 6: In conclusion, increasing the number of clusters did not show much help in the window detection.

D. Line/Shape Fitting

To take the results from color identification and translate it into useful knowledge about the window's position relative to the quadcopter, vertices corresponding to the corners of the window had to be detected. To accomplish this, first Canny edge detection was applied to the camera's image after it had been filtered by the GMM to quantify where the edges of the window were in the camera's view. The Open CV function Hough Lines was then called with the output of the edge detection algorithm to get position and angle parameters for lines fit to edges that had been detected by the camera. The lines were then clustered using the select k means algorithm with a k of two to be able to separate detected lines into categories with large relative angles between them. Window vertices were then detected by identifying where Hough Lines in one cluster intersected with Hough Lines in the other cluster. The intersections were then clustered and each cluster was averaged to assign a single point for each vertex. In the event that there were more than 4 clusters of Hough Line intersections, the algorithm took the 4 clusters with the highest number of intersections in them and identified those as the window vertices. The positions of the vertices in the image and the coordinates of the window (assuming the centroid of the window was at (0, 0, 0)) were then fed into the Open CV Solve PNP function along with the intrinsic calibration parameters of the camera to solve for the camera's position and orientation relative to the center of the window.

In practice, this worked with sufficient accuracy in the event that the windows was entirely in the camera's field of view. However, when only a part of the window was in view of the camera, there were problems that occurred. When only one edge of the window was detected, and no edges were detected perpendicular to that edge, the select k means clustering would still create two clusters of lines, and as a result, line intersections would be detected along the one edge that was detected, and the algorithm would detect 4 vertices along the edge. When these vertices were then fed into the Solve PNP function, it tried to fit a rectangular shape to a line, which gave particularly bad position and orientation results. A filter was written to detect instances where 4 vertices were detected, and the 4 vertices were close to a line shape, which helped to prevent this problem. In the event that 2 vertices were seen, the code estimated which edge of the window was detected based on its orientation and which quadrant of the field of view it was in, and then ran Solve PNP on those two points and the corresponding 2 points of the physical window. This also caused some problems as only detecting 2 points yielded highly varying and imprecise positions and orientations. In the event one vertex was detected, the assumption that if only one vertex was in the field of view, which vertex it was could be assumed from which quadrant of the image it was showing up in. (i.e. if in the top right part of the picture, it would be assumed to be the bottom left corner of the window) In the event one or two vertices were seen, rather than rely on PNP data, instructions were sent to the controller to move the quadcopter in small increments to try and get more of the window in the field of view of the camera so that better pose data could be collected.

IV. Controller

From a trajectory standpoint, this project presents a step up in difficulty from the last project as now the trajectories are unknown until runtime. For the last project, an open loop control system was be used, as the exact trajectory was known beforehand, and the control inputs could be tuned to obtain that exact trajectory. When testing the open loop controller, it was noticed that if command inputs were nonlinear, as if the velocity was doubled, the quad wouldn't go exactly twice as far. Therefore, for an arbitrary trajectory, such as the one found in this project, a closed loop control was desired for robustness.

A. Controller Logic

During flight, the bebop publishes its position and velocity (bebop odometry). Based on this, it was decided that a PD controller would be used.

Very quickly, it was discovered that coordinate transformations would be a problem, and a potential source for error (see Lessons Learned). The position given by the bebop is published in a global reference frame, with the origin and orientation of this frame set when the bebop takes off for the first time after being turned on (and will only reset when turned off/on again). However, the velocity given by this channel was in body frame of the bebop. The control inputs for the bebop are also given in the body frame.

To deal with this, the following steps were taken:

1) The desired waypoint was converted from the local frame to the global frame set by the first odometry message.
2) The raw odometry data was converted into the global frame. This only affected the velocities in x and y (the quad can only sustain steady pitch and roll angles of 0) which were transformed using the yaw angle.
3) The error in position and orientation (just yaw) was found in the global frame. The since the desired velocity was 0, The velocity itself was the change in error vs. time.
4) The new control input was calcualted using the gains Kp and Kd and the position and velocity errors.
5) Steps 2-4 were repeated until the quad arrived at the waypoint.

Convergence is when the quad position, orientation, and velocity errors are all under some threshold. Originally, just position and orientation errors were used and the quad would fly through the waypoint (on the way to an overshoot) and then try to fly through the next waypoint. This would lead

to increased drift and instability over time, and adding the velocity tolerance ensured the quad had reached a stable position at the waypoint before moving to the next waypoint. Tuning the controller also helped reduce the overshoot.

B. Tuning

The PD controller was tuned by flying two trajectories. The first was a simple straight line just to isolate one direction at a time. This straight line trajectory wa repeated in different orientations (global frame) and used to verify that all of the axis transformations made in the controller were right (this is how it was discovered that the velocity is given in the body frame). Next, a 3D diamond (similar to the one from Project 2) was used to test what happened when the quad was moving in all 3 directions at once.

The goal of the project was not speed, but accuracy and the choice of gains reflected that. The emphasis on Kd was higher than Kp, increasing the initial rise time but decreasing the overshoot.

V. Integration

Next, the controller was integrated with the camera. The camera gives the location of the gate in the camera frame (seen in Figure XX), which is different from both the bebop body frame and the global odometry frame. To fix this, the first waypoint was defined as the yaw angle between the gate and the current bebop position. With this yaw, the local bebop frame and the camera frame align, making it much easier. The next waypoint was defined using the camera x,y, and z inputs, and was set to be 1.5m behind the gate in x and aligned in y and z. An offset of .1m was used in the z direction to account of the fact that the camera was mounted below the center of the quad. This entire process was done using only one camera input (to obtain the global waypoints), the rest was done using the closed loop controller described above.

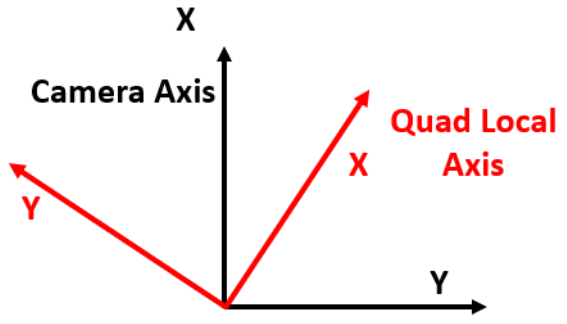

Fig. 10. Camera and Local Axis System. The X, Y, and Z values given by the PNP (whats shown in the video TestSetOutput.mp4) are in this axis.

To summarize, the overall control loop can be summarized using the following steps:

1) The location of the quad with respect to the camera was given to the controller and used to set the waypoints (one message, not a subscription).
2) Next, the quad yaws such that the camera axis lines up with the local bebop axis.
3) The quad then moves to line up with the window in y and z with a 1.5m offset. The initial x, y, and z positions from the camera are used to set the waypoints.
4) The position relative to the camera is found again, and if the drone is not lined up, new waypoints are set and steps 2 and 3 are repeated.
5) If the drone meets a threshold for being "lined up", then it moves forward through the gate (past the gate by 1m (just moved in x).

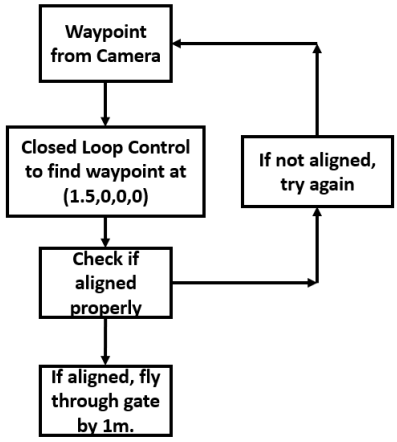

Fig. 11. Controller Logic

The threshold for being line up was set to be less than $\pm$ 6 degrees in yaw and $\pm$ .1m in x and y respectively. The x distance from the gate had to be between 1 and 2 meters (as long as it was line up straight, x really didn't matter too much, as the final approach was only given in x).

VI. Results

During the test, we only had time to test the first two trajectories (both straight line trajectories). We were able to successfully complete the short trajectory, but not the long run. During the long run, there were several attempts that looked good, in that they reached the final waypoint 1.5m normal to the gate and looked lined up, but the code never recognized this. There were also several attempts where the quadrotor would yaw by a very large angle (not close to the proper angle), causing the camera to lose sight of the gate. After these attempts, we would immediately land and try again. Some reasons for why we believe both of these issues occured, and how we might improve on this in the future, are presented in the next section.

VII. Conclusion and Lessons Learned

One issue that was accounted was the errant messages that would come in from the camera (from the PNP function).

This problem was evident especially in yaw, where the angles would vary by as much as $\pm$ 30 degrees. We believe this was a large reason for why we would occasionally have yaw angles that would take our camera out of sight of the gate.

We learned that the PNP data is noisy, even if the GMM is tuned, because the waypoints are in a 2D plane (hard to get yaw) and because the lighting through the window can cause streaks that lead to errant data. One solution to this is using multiple data points from the PNP instead of just relying on one data point, as that one point might be wrong. Some options for how to implement this could be as fancy as an EKF filter or as simple as a moving average. For future projects, we will use some averaging for data from our cameras to help reduce noise.

Another issue that we encountered was that the closed loop controller sometimes would seem like it converged but wouldn't. This happened when we would be lined up in front of the gate, with the proper orientation, but would eventually hover away. When first developing and testing the closed loop controller, we discovered this problem, but were able to fix it by adjusting the tolerances (especially in yaw and z). However, this testing was done in a different location and perhaps the odometry data on this surface is a little more noisy (although the IRB surface should be better). In summary, we haven't identified exactly what this issue is, but we have a test plan for how to combat it. We are planning to tune our closed loop controller at IRB to try and recreate this problem. Then we can look at the position and velocity error, hopefully allowing us to identify and fix the issue.

REFERENCES

[1] ENAE788 Class 5 Slides
[2] Some Code taken from learnopencv.com/rotation-matrix-to-euler-angles/