# ENAE788M Final Project
# Team Bouncing Rainbow Zebras


Erik Holum
Graduate Student
University of Maryland
Email: eholum@gmail.com

Edward Carney
Graduate Student
University of Maryland
Email: carneyedwardj@gmail.com

Derek Thompson
Graduate Student
University of Maryland
Email: derekbt@yahoo.com



***Abstract*—We are given the task of developing a control strategy to autonomously navigate a Bebop quadcopter through a series of obstacles. The obstacle types and order are known, but their relative positions and orientations are uncertain. In this paper, we discuss the visual processing and control methods used to detect, sense, and ultimately bypass each obstacle in the course.**


## I. Introduction

This paper describes the approach taken to navigate through the predetermined course of obstacles. The course layout is shown in 1 with the following obstacles:

1) Avoid low wall.
2) Traverse through gate.
3) Traverse over bridge (avoid river).
4) Land on circular bullseye target.
5) Avoid high wall.
6) Land on square bullseye target.

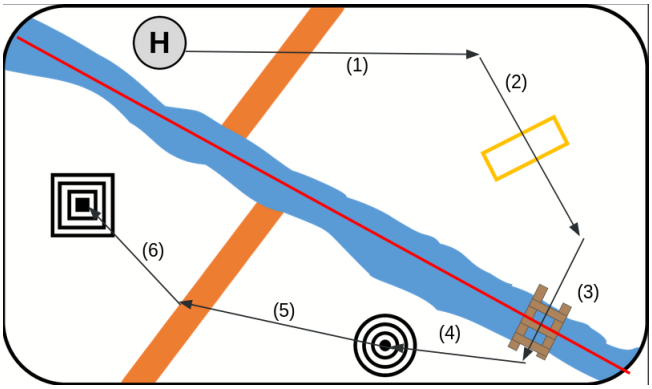


Fig. 1. Final course layout with numbered obstacles.

The implemented solutions for each of these obstacles relied heavily on the work done in prior projects and utilized the ROS [1] node architecture established and built upon for those projects. It should be noted that the bridge and bullseye detection in prior projects utilized the downward-facing Duo camera. However, due to a hardware failure with the Duo camera, this implementation was updated to use the front-facing camera. As such, the entire course was traversed using the front-facing camera only.

In this paper, we will first provide a brief summary of the methods used to navigate each obstacle. We then discuss putting everything together in a single state machine to navigate the entire course in sequence.

## II. Wall Detection and Avoidance

### A. Wall Detection

We a simple using pixel velocities to determine which features were 'close' and which were 'far'. In the simplest description, the algorithm for wall detection is,

1) Detect features in an initial image using OpenCV's $goodFeaturesToTrack$ [2].
2) Track the optical flow of the features for a set amount of time.
3) Compute the raw pixel deltas in the optical flow.
4) Cluster into two groups with K-means to determine which pixels moved a lot and which did not. The pixels that moved more are the nearer features, and most likely the wall.
5) The top most pixel (based on Y position) is the bottom of the wall, the bottom most is the top of wall.

In order to only focus on interested areas, we construct a mask that changes depending on the altitude of the bebop. An example masked image is given in figure 2. Sample feature detection within this mask is provided in figure 2.

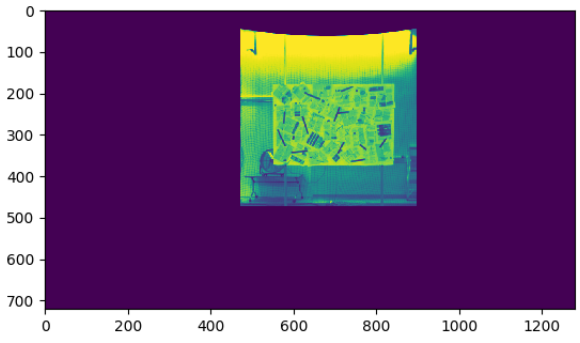


Fig. 2. Example of a simple rectangular mask we apply for feature recognition. The lower limit is dependent on the altitude of the Bebop. Note the mask is applied, and then the image is undistorted.

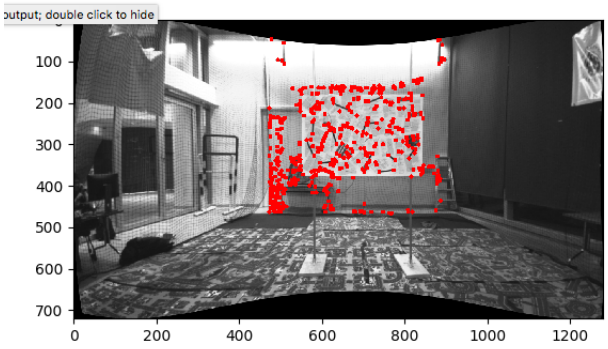


Fig. 3. Feature detection on masked image using the Shi-Tomasi Corner Detector in OpenCVs $goodFeaturesToTrack$.

Next, we use Lucas-Kanade optical flow in OpenCV's $calcOpticalFlowPyrLK$ to compute $[\Delta x_i, \Delta y_i]$ for each feature. We run the image capture and flow at 20 Hz, and update the features to track every 4 images, in other words, we track flow for 4 images and reset the features to track every Bebop odometry reading. Figure 4 has a sample of the optical flow for the features given in figure 3.

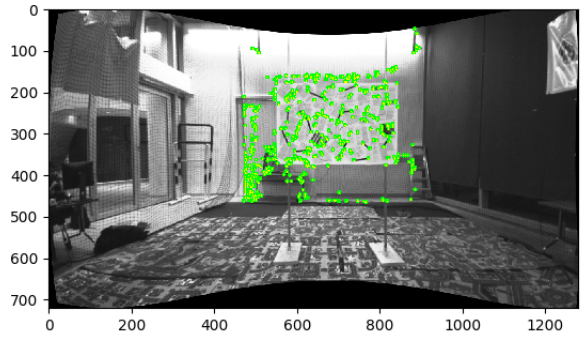


Fig. 4. Feature flow tracked over 4 images, at a rate of 20 Hz.

Given the tracked features, the problem simply becomes clustering using K-means based on total pixel distance per feature. We grouped into two clusters, $F_{near}$ and $F_{far}$ (greater pixel distance implies closer features). One minor improvement was made, since clustering was prone to the occasional large of outlier getting into $F_{near}$. To help remove them, we simply used a threshold based on standard deviations from the mean coordinate of $F_{near}$. An image demonstrating the clustering, distance filtering, then computing the upper, lower, and center of wall is provided in figure 5.

Given the focal length of the camera, $f$, and the location of the center, top, and bottom of the wall in pixel coordinates, it was trivial to compute the estimated angle from the camera frame to these relative poses. E.g., to compute the angle between the camera and the top of the wall,

$$\theta = \arctan \frac{y_{max} - C_y}{f}.$$

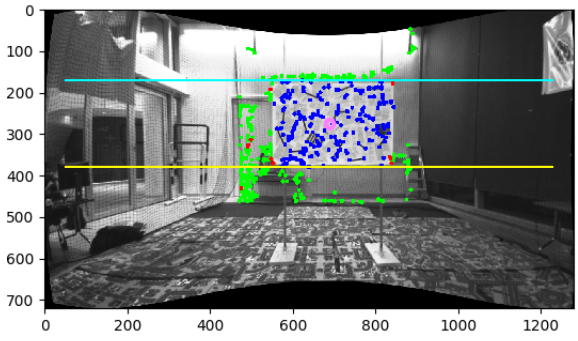


Fig. 5. Pixel depth map from K-means clustering on y-deltas. Blue dots are near, green are far, red are rejected by the standard deviation from the mean near point (pink). The topmost and bottom most pixels are marked with horizontal lines.

The controller leveraged a variety of these angles in its strategy, as discussed in the following section.

### B. Controls

The implementation for the wall avoidance and traversal was very rudimentary due to the difficulty detecting the center of the wall. The algorithm started with the assumption that the wall was relatively centered in the drone field of view. Assuming it would be able to see the wall the drone would move between 0.5 and 2.5 meters until it determined that it was safe to move forward through the gate. This required a change to the overall state machine allowing for multiple state exit conditions and subsequent state options.

To aid wall detection from optical flow the drone was set to climb at a relatively slow constant rate between the maximum allowed altitude (2.5m), and the minimum flyable altitude (0.5) meters. While the drone was moving between the altitudes, the controller was receiving measurement data from the wall detection script. From this data the controller was able to yaw to the center of the estimated wall position and estimate the top and bottom of the wall from a prior estimated distance to wall. The altitude of the wall center was calculated with the following equation, where $\theta_{min}$ is the smaller absolute value between the angle to the top and to the bottom of the wall cluster.

$$Z_{wall} = Z_{bebop} + X_{wall} * sin(\theta_{min}) + sign(\theta_{min}) * 0.5 \quad (1)$$

This method gave us a more reliable way to predict the true height of the wall based in scenarios where the wall was only partially seen. In cases where the entire wall was not seen, taking the average of the computed cluster would return a height closer to the drone, not giving the drone enough altitude clearance to fly through the gate.

With the measurements, the altitude of the gate was computed from a moving average. When the standard deviation of the measurements went below a threshold and the estimated height was calculated above a clearance distance, the state was progressed and the drone was instructed to move forward a set distance through the gate.

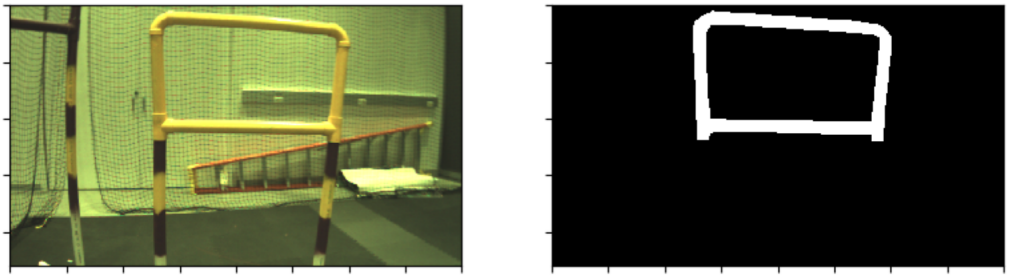

Fig. 6. Left: Raw image of yellow at 800x460 resolution. Right: Mask generated in Matlab using $roipoly$.

## III. Gate Detection

In this section we discuss our approach for detecting the yellow gate using Gaussian Mixture Models (GMMs), isolating and computing the gate's position in physical coordinates, then a control scheme to fly through it.

### A. Training GMMs

The first step in the process was to gather a significant amount of test data for training and testing our implementations. We captured images using the Leopard color camera and manually constructed masked regions of interest using Matlab's ROI tool. See Figure 6.

Given the training data and masks, we implemented a color segmentation tool using a trained Gaussian Mixture Model (GMM). This included implementation of an EM algorithm to converge to optimal weights $\pi_i$, means $\sigma_i$, and covariances $A_i$ to fit our data. Likelihood is computed with,

$$P(x|c_i) = \sum_{i=1}^{k} \pi_k \sqrt{\frac{\det A}{(2\pi)^3}} \exp\left[-\frac{1}{2}(x-\mu)^T A^{-1}(x-\mu)\right].$$

Testing empirically, we found that the HSV color scheme generally gave the best segmentation for isolating yellow colors. A sample masked image is provided in Figure 7.

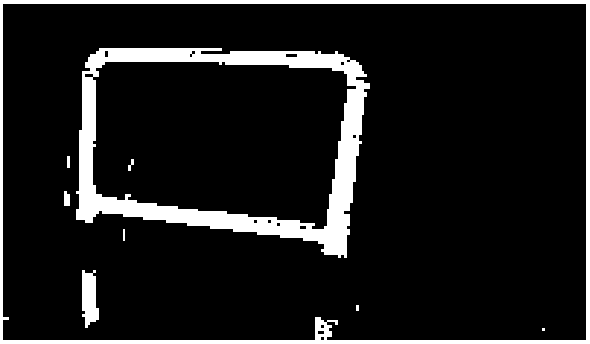

Fig. 7. Masked Gate Image

### B. Determine Gate Position and Orientation

The masked images produced from the color thresholding were post-processed to extract the information necessary to determine the position and orientation of the gate. This process included extracting edges via Canny edge determination, finding line segments from the given edges via a probabilistic Hough transformation, calculating the intersection points of the Hough lines, and determining the four primary intersection point clusters (i.e. the gate edges) via K-Means clustering.

Figure 8 shows the result of our process visually on a sample image from testing, including the isoated corners determined from clustering on intersections of perpendicular lines.

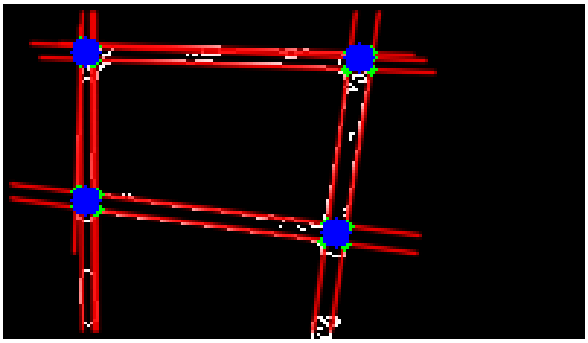

Fig. 8. Gate Pose and Orientation Determination

Given that we know the physical characteristics of the gate, we can use OpenCVs $solvePNP$ function to map the corners to real world coordinates. We presume the average of all four corners to be the center of the gate, and compute the relative position and rotation of the center of the gate. The sample output of this process is demonstrated in Figure 9. Note the blue line extending from the center marker visualizes the computed angle between the camera and center line.

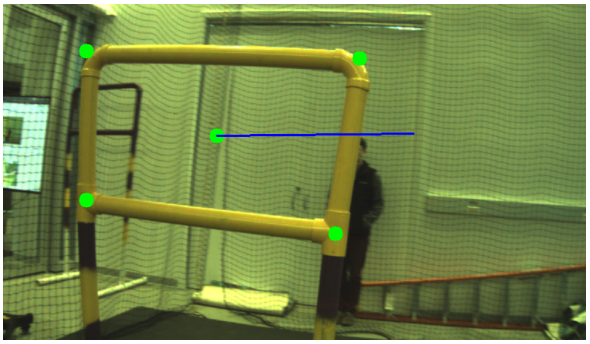

Fig. 9. Gate Pose and Orientation Determination

### C. State Logic

In order to control what the drone was doing at various steps in going through a gate, logic to control the process was created. The states follow a linear progression from the initial state to a defined final state. Each state contains information on what the drone needs to be doing at during this state and what conditions need to be met in order to move to the next state. The states are initialized as follows.

$$\begin{aligned} states[n] = States(&own\_State\_Number,\\ &next\_States\_Number,\\ &exit\_Condition\_Type,\\ &exit\_Condition\_Threshold,\\ &gate\_Detection\_Active,\\ &controller\_Type,\\ &fly\_WP\_Location,\\ &look\_WP\_Location) \end{aligned} \tag{2}$$

During the control loop the exit condition is checked and the proper controller called. The exit condition can be time based, distance based, or a check of if it has confirmed a gate. Once the exit condition is met the states is progressed to the next state number. The controller type defines the how the vehicle navigates towards its desired location. This is further explained in the next section. The $fly\_WP\_Location$ defines where the drone will try to fly if it is flying to a point in space, while the $look\_WP\_Location$ defines where the vehicle will point towards while it is flying to the point. This is set to null when the vehicle is flying towards a gate. These point are also relative so they are defined relative to the drones position when it exits the previous state. This is to counter drift in the odometry over numerous states.

When flying through a gate 3 states are used. The first states guides the vehicle to a point in space where it will have the best chance at seeing the gate. During this state the detection is active, trying to lock onto a gate location. This states is exited once a gate location has been found. The next state works off of the gate visuals to align itself with the gate and move towards it. At a certain point the gate is no longer visible in the camera frame, at which point the states is switched to the next again. This is trigger by distance to the gate. The final state traverse the gate for a certain amount of time before the state is exited and the system is shutdown.

### D. Controller

The controllers used for the system were defined in three separate functions, forward navigation, gate navigation, and point navigation. The point navigation is the same method used for project 2, and is used to fly to a position in space. This controller that uses a receding horizons approach is outlined in the paper for project 2. The forward navigation is the simplest controller as it just commands the vehicle to pitch forward at a desired angle. This controller is fully open loop and is just used to navigate through the gate. When this controller is used it is assumed the vehicle is already perfectly lined up with the gate.

The gate navigation was the designed to align the vehicle with the gate and put it on a trajectory to safely pass through. The controller executes the altitude and rotational controller independent of the lateral positional controller. If the position of the gate is found before the heading is confirmed the vehicle will rotate towards the gate and climb to the correct altitude to get a better view of gate. Once the gate heading is confirmed the lateral controller moves towards the normal line and through the gate. The basis of the lateral controller uses the same receding horizons approach as the point navigation, except the axis in which the positional error is calculated. Instead of the X and Y position error, the controller uses the distance from the gate and the angular distance from the normal line of the gate. The velocity desired towards the gate is calculated as a function of the lateral error from the normal line. The vehicle will stay 2 meters from the gate until the lateral distance from the normal line is below a threshold. The desired velocity graph of the gate navigation controller is displayed in 10.

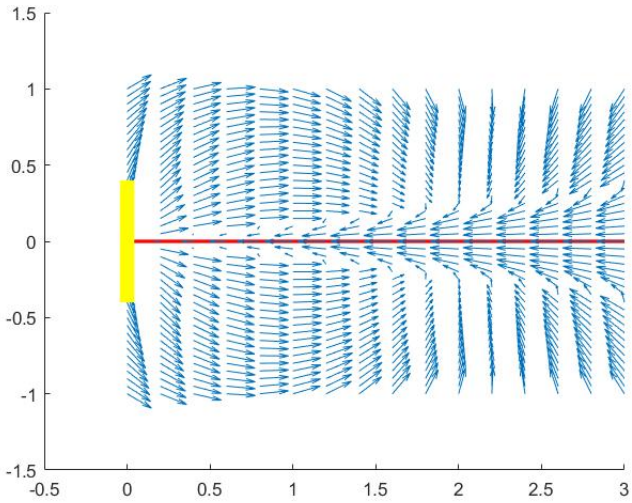


Fig. 10. Gate navigation velocity

## IV. Bridge Detection

In this section, we are given the task of traversing a bridge laid over a river with a Bebop quadrotor. Using image data from the front-facing camera, we use the OpenCV [2] to locate the the bridge in the image frame and find the center of the bridge in pixel coordinates. We discuss the process of deducing the location of the bridge in real world coordinates using the processed image data. Finally we implement a controller and provide results from implementation for this portion of the final project.

### *A. Image Processing and Shape Finding*

Due to the high contrast of the bridge compared to the river background, we were able to perform a conversion to HSV color space and apply simple color thresholding on the images returned from the front-facing Leopard camera to create an accurate binary mask. The HSV color space performed well for this task due to the color and intensity (i.e. saturation) of the bridge in our test lighting conditions. The exact threshold values were determined empirically using data captured from the camera around the time of the final flight. Figures 11-13 show the intensity values for a sample raw image of the target from the Leopard camera in HSV color space, note the small number of pixels with higher intensity value for the lightness (V) and saturation (S) parameters. By simply thresholding the image to isolate higher values of V and S parameters we were able to isolate the majority of the target whitespace.

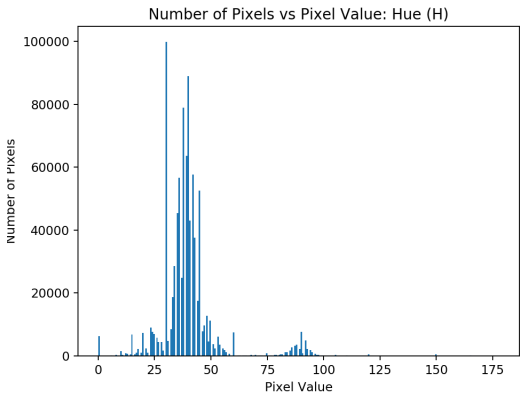


Fig. 11. Hue (H) intensity histogram for a sample bridge image.

A combined approach of erosion, dilation, and inversion is then used to remove regions of noise in the image; this was accomplished using OpenCV's built-in erosion and dilation functions. All the contours in the binary image were then extracted and bounding rectangles fit to each contour (again using built-in OpenCV functions). Small contours (i.e. those with small bounding rectangles) were removed, and the arameters for the remaining bounding rectangles were retained. The largest rectangle was then isolated, and the center of the whitespace in that rectangle was found. This centerpoint was then exported from the image processing function as the location of the bridge.

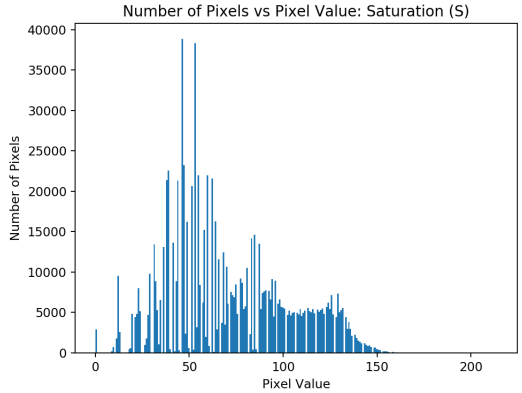


Fig. 12. Saturation (S) intensity histogram for a sample bridge image.

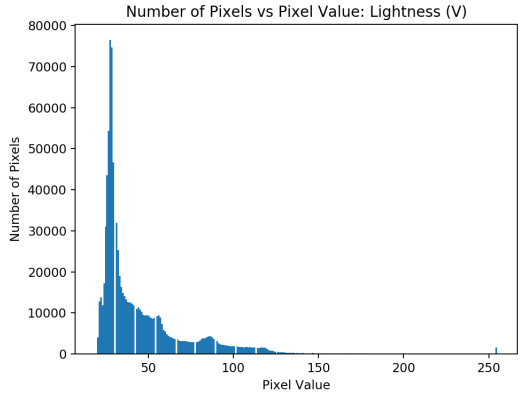


Fig. 13. Lightness (V) intensity histogram for a sample bridge image.

This method of image processing generally yielded reliable positional information for the bridge, provided that the color thresholding values were determined using data with lighting values reasonably similar to those in the actual test environment. The image processing method is shown in Figures 14, 15, 16, and 17.

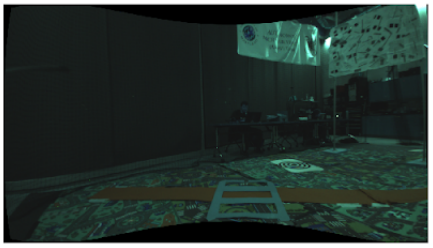

Fig. 14. Raw Leopard camera image.

Fig. 15. Conversion to HSV color space.

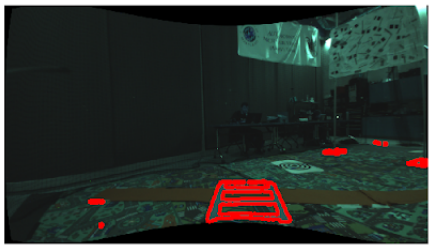

Fig. 16. Resulting mask after inversion, erosion, dilation, application of Canny edge detection, and removal of small contours.

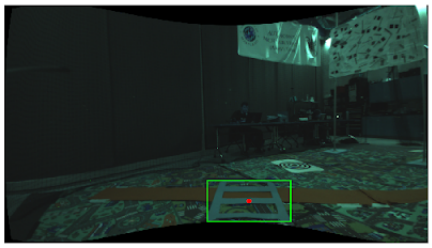

Fig. 17. Bounding rectangle for largest contour (green) with the center pixel of the bounding rectangle (red).

### B. Pose Estimation

Ultimately, we found that the simplest approach was the best approach for providing target position relative the quadrotor. Using just the computed center of the bounding rectangle, we are able to get reliable guesses for the *direction* of the target. Ultimately, we are able to rely on the Bebop's own altitude estimation for $Z$, then use basic trigonometry and similar triangles to produce a heading.

Let $f$ be the focal length of the camera, $(c_x, c_y)$ be the center of the projected ellipse in pixel coordinates. And let $\theta_x$ be the angle formed by $[0,0,0]$, $[0,0,f]$, and $[c_x,0,0]$. Likewise, let $\theta_y$ be the angle formed by $[0,0,0]$, $[0,0,f]$, and $[0,c_y,0]$. See figure 18.

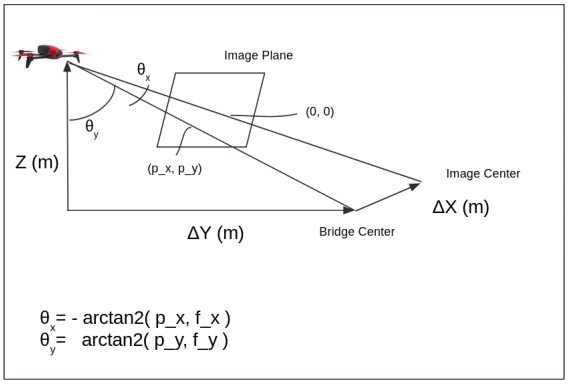


Fig. 18. Projection of the bridge in the image plane, together with the center of the image frame and the focal length.

Then, rather than computing the full estimated position, we only compute the angles $\theta_x$ and $\theta_y$ as

$$\begin{bmatrix}\theta_x \\ \theta_y\end{bmatrix} = \begin{bmatrix}\arctan 2\,(c_x, f) \\ \arctan 2\,(c_y, f)\end{bmatrix}$$

However, in this case we must deal with the reality of the distortion of the camera. Using the focal lengths $f_x$, $f_y$, and the image center $u_x, u_y$ returned by Kalibr, we can compute the physical world values of $\theta_x$ and $\theta_y$ as

$$\begin{bmatrix}\theta_x \\ \theta_y\end{bmatrix} = \begin{bmatrix}\arctan 2\,(c_x - u_x, f_x) \\ \arctan 2\,(c_y - u_y, f_y)\end{bmatrix}.$$

Where $(c_x, c_y)$ is the center of the rectangle in image frame coordinates. Ultimately, we are able to accomplish the task of traversing the bridge using only these two values, which is discussed further in the Controller section.

After computing the angles $\theta_x$ and $\theta_y$ the detection node sends these values in an array to the navigation node. When the navigation node receives these values it uses the attitude from the last odometry update to project the position of the target on the ground plane using the equations below.

$$\begin{aligned} R_{att} &= Rot(\phi, \theta, \psi) \\ R_{cam} &= Rot(\theta_x, \theta_y, 0) \\ V_{proj} &= R_{att} * R_{cam} * [0,0,1]^T \\ t &= altitude / V_{proj}[2] \\ X_{bullseye} &= [bebop_x + V_{proj}[0] * t, bebop_y + V_{proj}[1] * t, 0] \end{aligned} \tag{3}$$

These prediction were input to the filter designed for the gate traversal in project 3a to get a final filtered result. The filter used a moving average of the last ten measurement

while discarding new measurements whose error was outside a defined threshold.

### C. Controller

The control of the drone used the same state machine implementation as project 3a to maintain the flexibility and ability to extend the drones capabilities to longer and more complex tasks. The navigation process was composed of a total of 5 states.

Between these processes the states composed of a point navigation process, a hold position process, and a bridge traversal navigation process. The point navigation would guide the vehicle to the best estimate of the target with an altitude of 0.75 meters. The altitude was set low in order to get a wide coverage with the front-facing camera. At that altitude the camera was able to see the majority of the test area so we were confident that we would be able to see the bridge. The point navigation ends when the bebop has reached the desired view point, at which point it moves to the next state where it is commanded to hold its position. This state waits until a bridge position is confirmed before it enters the final navigation state and moves to traverse the bridge. The bridge navigation is based on the point navigation function with no change to the altitude control.

With the position of the bridge the algorithm used our nonlinear controller to position the drone one meter behind the bridge. When the position error dropped below a threshold and the drone's velocity was low enough the drone was instructed to move forward a set distance.

## V. Square and Round Target Detection

In this section, we are given the task of landing a Bebop quadrotor on the center of target made of black and white concentric shapes (squares or circles). We are provided with a stereo Duo3d camera, which we mount on the nadir-facing side of the drone so that it is as in line with the body fixed $z$-axis as much as possible, and the front-facing Leopard Imaging M021 Camera.

Using image data from the front-facing camera, we use the OpenCV [2] to locate the the target in the image frame and find the center of the target in pixel coordinates. We discuss the process of deducing the location of the target in real world coordinates using the processed image data. Finally we implement a controller and provide results from implementation for this portion of the final project.

### A. Image Processing and Shape Finding

Due to the high contrast of the black-and-white bullseye target, we were able to perform a conversion to LAB color space and apply simple color thresholding on the images returned from the front-facing Leopard camera to create an accurate binary mask. The LAB color space performed well for this task due to the color and brightness (i.e. intensity) of the bullseye targets. The exact threshold values were determined empirically using data captured from the camera around the time of the final flight. Figures 19-20 show the intensity values for a sample raw image of the target from the Leopard camera in LAB color space, note the small number of pixels with higher intensity value for the lightness (L) and red/green (A) parameters. By simply thresholding the image to isolate higher values of L and B parameters we were able to isolate the majority of the target whitespace.

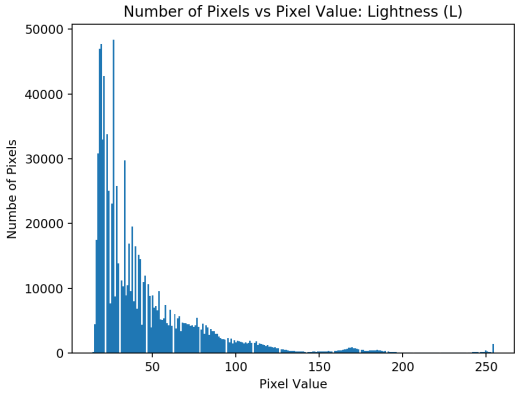


Fig. 19. Lightness (L) intensity histogram for a sample bullseye target image.

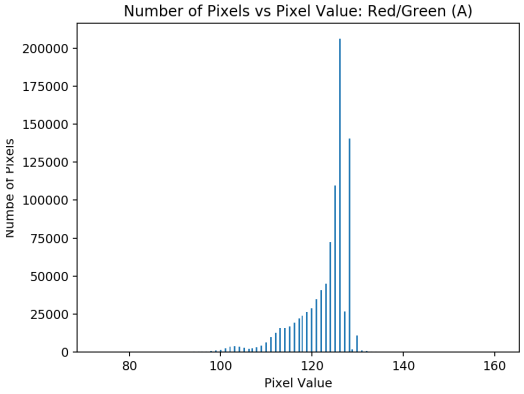


Fig. 20. Red/Green (A) intensity histogram for a sample bullseye target image.

The exact same image processing approach that was used for bridge detection and traversal was then performed on the resultant image. Similar to the bridge, this method of image processing generally yielded reliable positional information for the bullseye targets, provided that the color thresholding values were determined using data with lighting values reasonably similar to those in the actual test environment. The image processing method is shown in Figures 22, 23, 24, and 25.

### B. Controller

The control of the drone used the same state machine implementation as project 3a to maintain the flexibility and ability to extend the drones capabilities to longer and more complex tasks. The navigation process was composed of a total of 5 states.

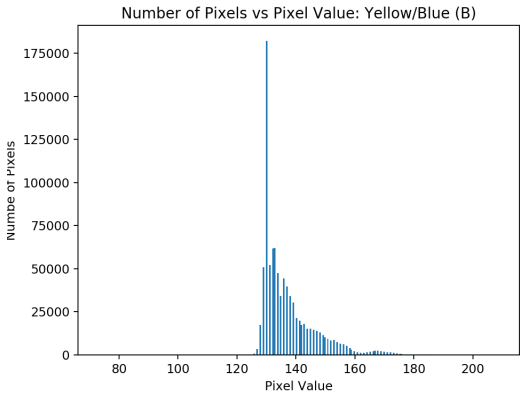


Fig. 21. Yellow/Blue (B) intensity histogram for a sample bullseye target image.

Fig. 22. Raw Leopard camera image.

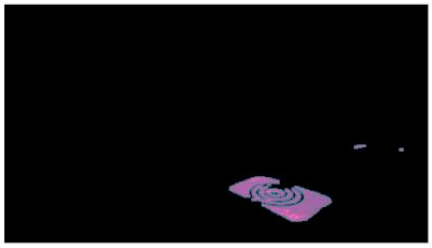

Fig. 23. Conversion to LAB color space.

Between these processes the states composed of a point navigation process, a hold position process, and a target landing navigation process. The point navigation would guide the vehicle to the best estimate of the target with an altitude of 0.75 meters. The altitude was set low in order to get a wide coverage with the front-facing camera. At that altitude the camera was able to see the majority of the test area so

Fig. 24. Resulting mask after inversion, erosion, dilation, application of Canny edge detection, and removal of small contours.

Fig. 25. Bounding rectangle for largest contour (green) with the center pixel of the bounding rectangle (red).

we were confident that we would be able to see the target. The point navigation ends when the bebop has reached the desired view point, at which point it moves to the next state where it is commanded to hold its position. This state waits until a target position is confirmed before it enters the final navigation state and moves to to the determined position above the target. The target navigation uses point navigation based on the determined position of the target from the angles computed earlier. Once the bebop has reached the determined point, the landing command is issued.

The same process was used for both bullseye targets. It was kicked off twice: once after the bebop had travered the bridge, and again after the bebop had avoided the second wall.

## VI. Navigation

Given the previous sections, we see we have clear methods for detecting and bypassing each obstacle in the course. However, as made clear in the introduction, the objective of this project is the successfully navigate the entire obstacle course, where the positions and orientations of each obstacle are only approximately known. To accomplish this task, we

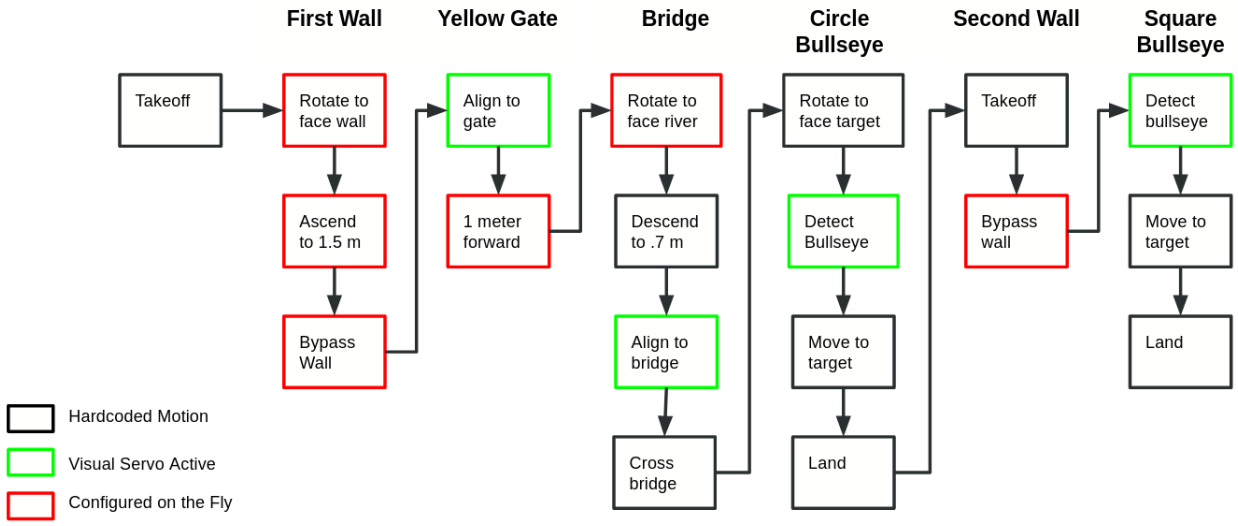


Fig. 26. Logical flow diagram for course obstacles.

have implemented a linear state machine for traversing each obstacle.

Figure 26 presents the high-level logical flow implemented to manage flying through the course. We will briefly discuss the sequence of events and the conditions for moving from state to state.

First, we note that as depicted in the diagram, many of our states simply use open loop control - rather than visual feedback. In testing, we found many of our tools as implemented above to be slow. In the interest of navigating the course as quickly as possible, we have chosen to measure certain angles and distances immediately prior to taking off, and have left easy spaces in the code to change these on the fly during the demonstration.

Of particular importance to this approach is initializing the Bebop's on-board rotational measurement to be perfectly inline with the river (see the red line in Figure 1. This sets the starting angle of the Bebop to $0°$, and allows us to reconfigure expected angles relative that diagonal. For example, since the bridge is always crossing the river, we can tell the Bebop to align to an orientation of $270°$ after coming out of the gate.

### A. Takeoff and the First Wall

Given that the position of the wall is known, and we can set the rotor into any particular desired start orientation, the first phase of the state machine is simple. We simply open loop control over the wall, then turn to put the gate in view of the front camera using a simple point-to-point controller. Once the odometry readings have reached the desired coordinates, we move out of these states.

### B. The Yellow Gate

We presumably will have the gate in view of the front camera immediately upon entering the 'Align to Gate' state. With that assumption, we line up with the perpendicular center line through the gate according to the control strategy discussed in Section III. Once the visual feedback has the gate lined up at an approximate distance of $0.75$ m, we deactivate the gate detection algorithm and simply send a command to move 1 m forward (this distance is configurable).

### C. The Bridge

Coming out of the gate requires reasonably accurate point-to-point commands to ensure the front-facing camera has the bridge in view. Hence careful measurements and rotational commands are needed. Once the bridge is in view, however, simply lining up to have the bridge in the center is straightforward. Hence, moving out of the 'Align to Bridge' phase depends heavily on the 'Rotate to face River' point-to-point command. Once the center has been detected to be directly in front of the Bebop for enough image cycles, we use the estimated center pose to command a forward motion that is sufficient to cross the river.

### D. The Circular Bullseye

Again, we use a point-to-point command to ensure the Bebop can see the Bullseye. Once the center location of the Bullseye has stabilized, we take the average translational reading to command an $X$-$Y$ motion that will put the drone over the target, then immediately land.

### E. The Second Wall

This is the most dangerous part of the course in terms of collisions with obstacles. Basically the wall detection algorithm presented in Section II relies on moving the drone up and down until the wall pose has been reliably detected. This process is can take many seconds. Rather than rely on the slow method, we have again opted to hard code a point-to-point motion that will take the drone from the center of the circular bullseye to directly underneath the wall. Here is where the initial starting

angular pose comes in very handy, since we can essentially guarantee that the bebop will takeoff facing the wall. Once it is facing the wall, it is simple to eye-ball the distance to it.

### *F. The Square Bullseye*

Assuming we have successfully navigated underneath the second wall, the bullseye target should be in view of the front camera. Once there, we use the exact same process as the first circular bullseye.

## VII. Presentation Video

We have uploaded a powerpoint presentation to YouTube, available at:

- https://youtu.be/X0qug6wOVEo

If required the .mov video file is available for download through Google Drive:

- https://tinyurl.com/sv7zn56

## References


[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[2] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.