ENAE788M Project 4b
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*—This project has two parts. The first to fly over or under a wall of unknown vertical position. The second to recognize a 'bridge' and fly the Bebop directly over the center.**

I. Wall Detection and Avoidance

A. Detection

We initially tried using the optical flow equations with the Bebop Odometry's linear velocity measurement to compute the depth to each detected pixel using,

$$\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} = \frac{1}{Z}\begin{bmatrix}-f & 0 & x\\ 0 & -f & y\end{bmatrix}\begin{bmatrix}V_x\\ V_y\\ V_z\end{bmatrix}. \quad (1)$$

Where V_x, V_y, and V_z are the velocity's in the Camera frame, f is the focal length, and $\dot{x}$, $\dot{y}$ are the pixel velocities. Given the velocities, we can simply solve for the depth, Z. Note that we assume $\mathbf{\Omega}$ to be 0. Unfortunately, the conversion from the Bebop's odometry frame to the camera frame was too difficult to work out, and we got reliable measurements very infrequently (maybe 1 in 10 Odometry updates).

We ended up going with a simpler method of simply 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.
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.

This was the basic starting point for us, everything else we did was simply add-ons to try to improve the performance of this basic algorithm.

The first step was to reduce the number of extraneous features detected by OpenCV's [1] $goodFeaturesToTrack$ function. The first portion of this was restricting the field of view to a narrower image, which we do using the fact that we are starting with the wall directly in front of us, as well as the Bebop's altitude estimate, Z_B. We start by defining a rectangle of interest in front of the camera frame. Say we want to focus on a region that is a meters in width, at a distance d_x, in front of the Bebop. Then we can simply eliminate all pixels greater than a distance $x_{pixel} = (af)/(2d_x)$ from the center C_x.

Because of the carpet, the ground is also a big problem. Hence we set a distance, d_z, in front of the camera frame that we also want to ignore. The logic being that if we only care about the ground if we can see it behind the wall. If we are at altitude, Z_B, and we only care about ground data at a distance, d_z, directly in front of the drone, then from similar triangles we can mask and pixel with y value greater than,

$$y_p = C_y + \frac{Z_B}{d_z} f.$$

Given the mask, we are able to only consider features within a certain area of the image. An example masked image is given in figure 1. Sample feature detection is provided in figure 1.

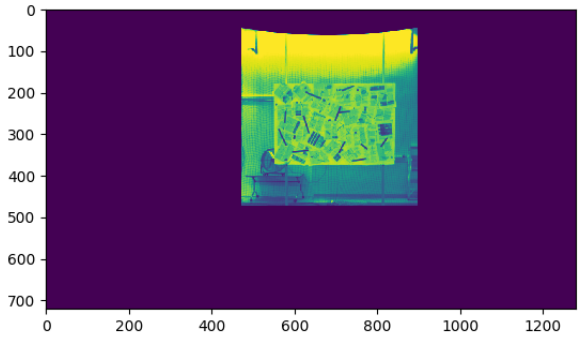

Fig. 1. Example of a simple rectangular mask we apply for feature recognition. The lower limit is dependent on the altitude of the Bebop. Note the mask is applied, and then the image is undistorted.

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 3 has a sample of the optical flow for the features given in figure 2.

Finally, we come to the problem of determining which pixels are close and which are far. First off, when searching

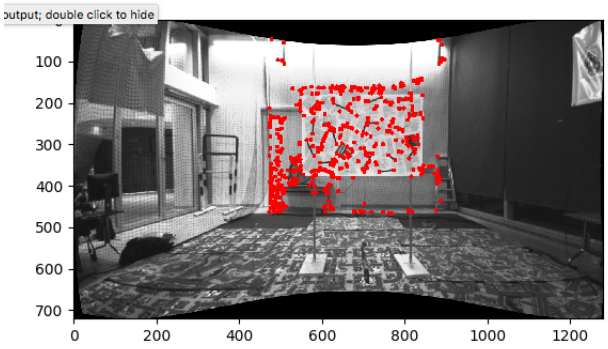

Fig. 2. Feature detection on masked image using the Shi-Tomasi Corner Detector in OpenCVs $goodFeaturesToTrack$.

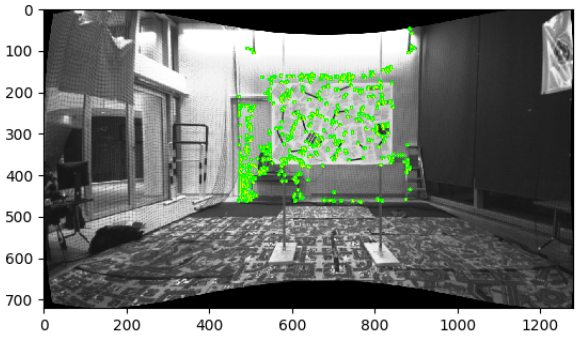

Fig. 3. Feature flow tracked over 4 images, at a rate of 20 Hz.

for the map, we commanded the Bebop to simply move up and down in the Z direction, allowing us to immediately ignore pixels with large Δx greater than some threshold. Next, we used K-means to cluster the flow coordinates by Δy into two groups, F_{near} and F_{far}, under the presumption that the features with large Δy are closer, and small Δy are more distant.

We noticed in testing that the clustering was prone to the occasional large of outlier getting into F_{near}, but for the most part, the clustering was successful when the optical flow was accurate. However, since we used the topmost and bottom most pixels in F_{near} to estimate the location of the top and bottom of the wall, these outliers could be fatal. To 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 4.

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,

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

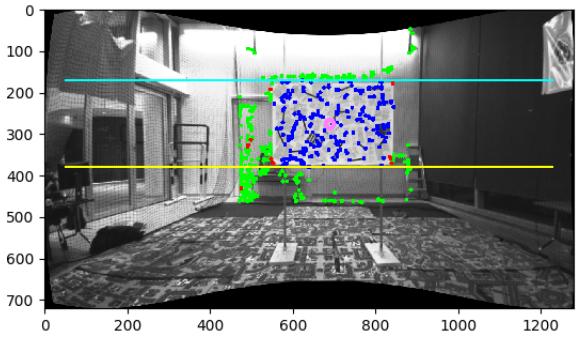

Fig. 4. Pixel depth map from K-means clustering on y-deltas. Blue dots are near, green are far, red are rejected by the standard deviation from the mean near point (pink). The topmost and bottom most pixels are marked with horizontal lines.

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 θ_{min} is the smaller absolute value between the angle to the top and to the bottom of the wall cluster.

$$Z_{wall} = Z_{bebop} + X_{wall} * sin(\theta_{min}) + sign(\theta_{min}) * 0.5 \quad (2)$$

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.

II. Bridge Detection

A. Detection

Bridge detection utilized similar functionality as the bullseye detection performed for Project 3b. Specifically, this was done by using the down-facing Duo camera used to capture raw grayscale imagery, creating a masked image from these raw images by applying an adaptive thresholding algorithm, detecting contours within the masked image, applying bounding rectangles to each of the contours, then extracting the center of the bounding rectangle that best fit the expected bridge shape.

To accomplish the above process, several built-in functions from the OpenCV Python library were used. Namely, the $adaptiveThreshold$ was used with a specified Gaussian threshold model to apply an adaptive Gaussian threshold to the raw grayscale image produced by the Duo camera, the $findContours$ function was used to detect and extract all contours in an image, and $boundingRect$ function was used to find the bounding rectangle around a given contour. We experimented with multiple methods to pre-process the masked image prior to detecting contours (including erosion, dilation, and multiple types of blur), however, experimenting in different lighting conditions revealed that the most effective and reliable bridge detection occurred with no pre-processing. Instead, the application of constraints to determine the bridge location was done on the detected contours and the bounding rectangles. In detecting the bridge, we aimed to detect the smaller rectangles that comprised the bridge and use those to align the drone laterally with the bridge and move forward until we were over it, then we could do a brief period of open-loop control continuing in that direction until we were across the bridge.

We were allowed to orient the drone directly for this challenge, and therefore we could assume that we would always be perpendicular to the bridge. As such, we could apply rather stringent constraints on the size of the detected contours and the size and aspect ratio of the bounding rectangles. Specifically, we could ignore very small and very large contours and bounding rectangles; indeed, for the bounding rectangles, we knew the height we would be flying at for this challenge, and as such we could be very specific in the size of the bounding rectangles sch that they would align with the smaller rectangles composing the bridge. Additionally, the aspect ratio for the bounding rectangles (defined as the length of the rectangle divided by the width) could also be constrained to limit the results. This generally resulted in a single rectangle corresponding to the center of the bridge. The pixel location of the center of the rectangle was used to compute the roll and pitch angle of the bridge center relative to the drone, which was published as a ROS topic and subscribed to by the controller. In the event that there were multiple detected rectangles, the average of the rectangle centers was used.

Figures 5 - 9 show the iterative application of these constraints on a sample set of data.

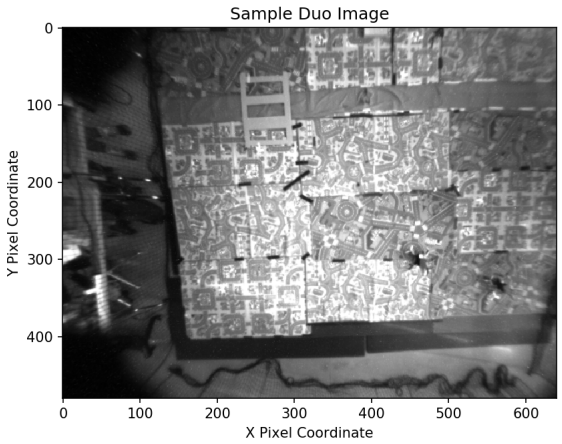

Fig. 5. Sample image taken from Duo camera. Note IR LED value is set ot zero here.

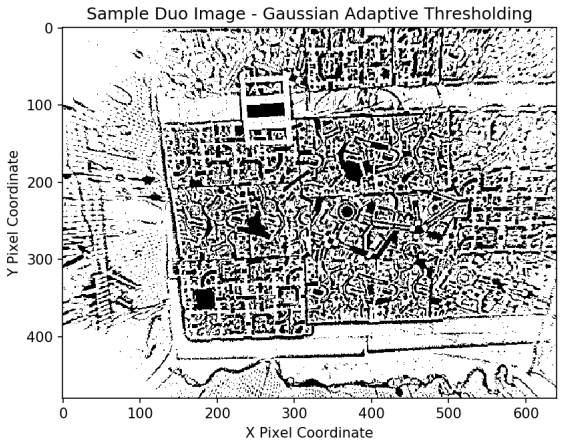

Fig. 6. Sample image from Duo camera with OpenCV's adaptive Gaussian thresholding applied.

B. Controls

The controller for the bridge traversal was derived from the bullseye target control strategy from project 3b. Using the same nonlinear controller described in previous projects the drone was initially taken up to a preset altitude where it could search for the bridge. The drone was commanded to wait here until it had a confirmed position of the bridge. In the same method as with the bullseye controller in project 3b the camera measurement rotations θ_x, and θ_y were added to the drones attitude rotation to get a vector towards the target from the drones position. The position of the bridge was solved through intersecting this vector with the ground plane. Once again these measurements were input into a moving average filter until the standard deviation reached below a threshold.

With the position of the bridge the algorithm used our nonlinear controller to position the drone 1 meter behind the bridge. When the position error dropped below a threshold and the drone's velocity was low enough the drone was instructed

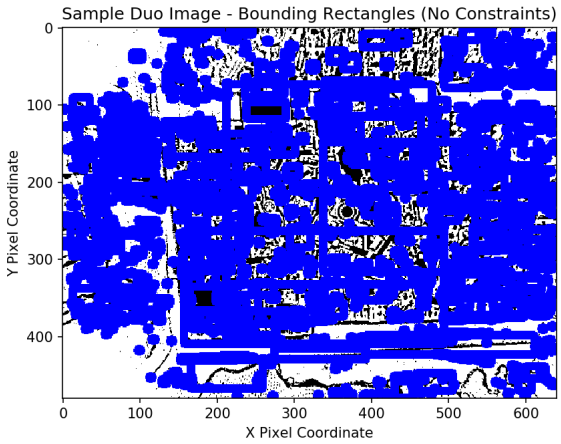

Fig. 7. Sample masked image from Duo camera with bounding rectangles applied to every detected contour.

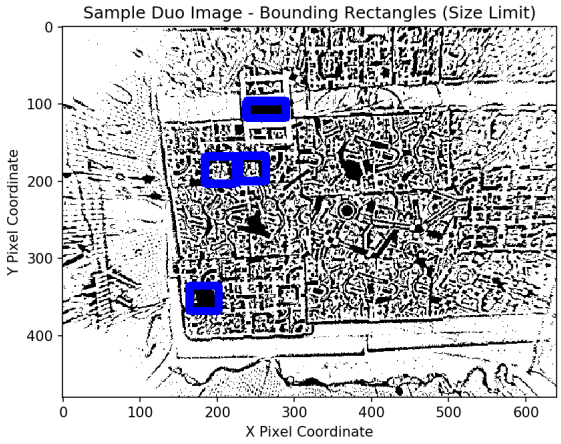

Fig. 8. Sample masked image from Duo camera with bounding rectangles applied but with minimum and maximum size limitations.

to move forward a set distance.

III. Videos

Wall Flight:

1) Rviz Positioning: https://youtu.be/E9io6ojWuag

Bridge Flight:

1) Rviz Positioning: https://youtu.be/Fm8jLlAzX0Q

IV. Important Lessons Learned

A. Keep it Simple

We wasted a lot of time trying to get the optical flow equations working to reliable give pixel depth in the physical world. However, once we gave up on that and went with the simple clustering method, we found something that worked even more robustly, with much less complexity. In the future we should always test the simpler idea to see if it works, then add complexity only if we need it. As an additional application

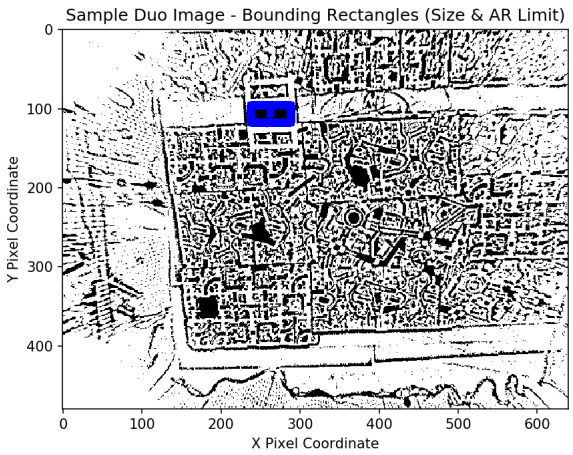

Fig. 9. Sample masked image from Duo camera with bounding rectangles applied but with minimum and maximum size and aspect ratio limitations.

to the keep it simple rule, we reused a significant portion of our code from the bullseye landing functionality for the bridge detection. Although the less reflective color of the bridge made it much more difficult to detect, once detected, the logic to travel to the bridge was very similar to that used to travel to the bullseye (minus the landing part).

B. Camera Calibrations are Tough

Making the Bebop Odometry or the Duo Stereo camera velocities line up enough with the pixel velocities from the forward facing camera proved to be more difficult than we though. We wasted a good deal of time tweaking parameters and trying to understand why our data was so poor, and really should have just gone with the simple option to begin with. That being said, in the real world if we absolutely needed depth from the forward facing camera, we would simple put the stereo camera/imu on the front, which would be much, much simpler.

C. Understand Parameters

Using built in methods from cv2 is great for feature detection and optical flow, but we discovered that we really have to play with the parameters for each to get them working reliably. The default values for most of these things are unreasonable. Hence, once we found reasonable lighting conditions, we manually tuned thresholds, window sizes, etc for feature detection and flow that worked ONLY in those conditions. If the lighting or camera image size change, we should not expect those parameters to continue working!

Acknowledgment

The authors would like to thank the professors for this course, Nitin J. Sanket and Chahat Deep Singh, as well as Dr. Inderjit Chopra.

References

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