Avoid the wall and find the bridge - 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 examines the challenge of target detection and tracking using the bebop parrot quadcopter. The first model problem used to examine this problem was finding a bridge and crossing a river. This involved detecting a blue river and not crossing it, while simultaneously searching for a river. Only when the bridge was successfully identified could the river be crossed (over the bridge). The second model problem involved avoiding a wall. After takeoff, the quadcopter would see a wall that would be high (¿ 1m off the ground) or low (¡ .5m below the ground). The quad must identify which configuration the wall was in and then fly between two poles supporting the wall (above or below the wall depending on the walls position). For both of these problems, the vision identification method is introduced. Finally the overall controller logic combining a closed loop controller and the vision software is presented.**

I. Introduction/Problem Statement

The goal of this object is to complete two seperate tasks with the bebop quadrotor. This first task involves using a bridge to cross a river. The enviornment will be set up as seen in Figure 1, and the quadrotor is no allowed to cross the river unless over the bridge. The quadrotor must move to the river (without crossing it), and then search along the river until it finds the bridge, only crossing the river at this point.

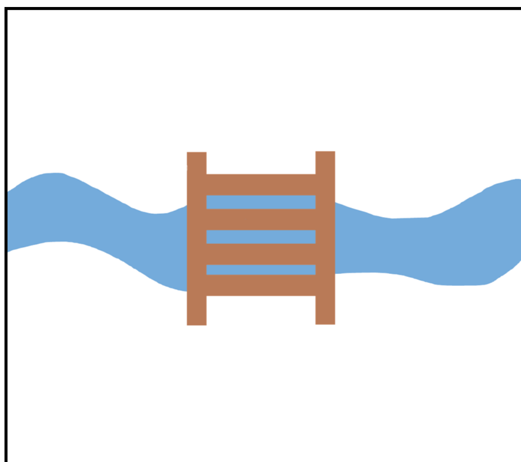

Fig. 1. The quad must find the bridge and use it to cross the river.

In the second task, the quadrotor must detect and avoid a wall. The wall has dimensions (1mx1.25m), and can be placed in either a high or low configuration, as seen in Figure 2. The quadrotor will be initially placed about 2-3 meters away, roughly facing the wall with up to 15 degrees of yaw. The quad must detect the wall, determine if it is a high or low wall, center itself, and then fly through the poles used to hold the wall. The quad may not exceed a height of 2.5m for this project. The wall has been marked with extra features to help with detection.

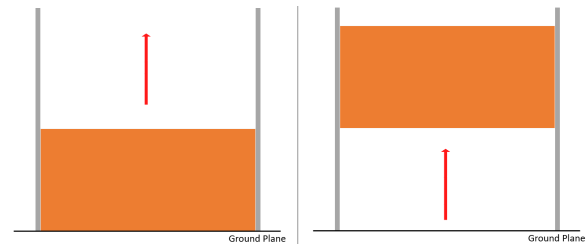

Fig. 2. The quad must detect if the window is low or high and adjust its flight path correspondingly.

II. Find the Bridge

The bridge task was broken into several different components. The main piece was the visual portion, or actually identifying the bridge. This was done using the bottom stereo camera. The second component was the controller, or how to make the quad fly to a desired location. Finally, we needed to merge the two components together to complete the mission efficiently. This is covered into the integration section.

A. Identifying the Bridge

To identify the bridge and the river, the bottom facing stereo camera was used. For a single image, a window is swept through the entire image without overlap. The size of this window is a user input, and was taken to be 17 pixels (square window) for this testing. For each window, the variance of the greyscale values is computed. If this variance is lower than a threshold (taken to be 200 for this analysis), then this square is marked as a possible river. This results in an image as seen in Figure3.

Fig. 3. All of the areas with uniform grayscale values are river candidates.

Next, a filter is applied to find the exact river. This involves taking the median and standard deviation of the y values (in image frame) for the center of each river candidate. Then, only the candidates that are within a certain percentage of the standard deviation from the median are kept. This technique relies on the fact that the river will lead to many more river candidates than false detections, and that the quad has relatively small yaw relative to the river. After filtering, the image from Figure 3 will turn into Figure 4.

Fig. 4. Filtering the River candidates by y position (in camera frame) allows for the river to be found identified correctly.

After identifying the river, the code will look for gaps in the river, as this will indicate the bridge. The window size was tuned so that neither the bridge or the river in between the bridge panels will register as river candidates. If a gap is found, the code first checks if it is greater than a certain pixel value that corresponds to the approximate size of the bridge. If all of these conditions are met, the position of the bridge in the image frame is equal to the position of the center of the gap in the river, shown in Figure 5.

Fig. 5. The largest gap in the river is assumed to be the bridge - as long as this gap is larger than a certain threshold.

The physical position of the bridge relative to the quad is found using a similar method to the scheme implemented in Project 3b - Bullseye. Based on the height from the odometry and the position of the bridge in the image frame, the physical X and Y positions can be calculated using the following equations.

*1) **Averaging Scheme**:* The procedure above works very well for a single measurement, however there can be some outliers. To ensure that these outliers are not used, multiple measurements were used for each command. The procedure outlined above was run for 5 different images, and the position of the bridge is identified in all. If 3 or more images do not see the bridge, this means that the bridge is assumed not to be in view, and the search algorithm takes over (explained in integration section). If 3 or more images do see the bridge, then the physical coordinates of where the bridge is relative to the quad are compared. If the standard deviations of these points are below a certain tolerance, only then does the code believe it is able to see the bridge. Lastly, these coordinates are sent to the controller.

B. Controller

A closed loop controller was used to move the quad to any given waypoint. This controller uses feedback from the odometry given by the bebop quad itself to accurately navigate to any point. For more information, refer to the report from Project 3a - Mini Drone Race.

C. Integration

There were two codes created for the previous two sections that run well on their own - the last task was integrating them so they run well together.

First, we addressed to problem of the quad trying to establish the position of the bridge during flight. We were able to find the bridge in flight, but the position of the bridge would vary, as it was the position of the bridge relative to the

quad at the instant that the image was obtained. This meant that the spread of bridge locations would be high and would be indistinguishable from false detections. Two methods were conceived to address this. The first involved adding the bebop odometry to the bridge finding code, so that the position of the bridge could be relative from some arbitrary point to account for the movement of the quad. The second method involved waiting until the quad had reached a waypoint until it looked for the bridge. The second method was simpler and didn't add much more time compared to the first method so it was implemented.

The code was setup such that the quad would hover and look for the bridge. After 5 images, the bridge finder would send one of three messages to the controller. These messages included a flag and coordinates, packaged in pose message. We assume the orientation is fixed to where the quad is perpendicular to the river so the orientation is not needed.

1) For a flag equal to 0, the bridge finder code was not able to find the bridge. This could mean that there were no gaps in the river large enough or that the bridge measurements were too scattered, indicating false detections. In this case, the controller is instructed to move the left .75m.
2) When the flag is equal to 1, the bridge finder is able to find a good estimate for the bridge. The coordinates included in the message indicate the x and y position of the bridge relative to the quad. The controller moves to a certain distance behind the bridge, set to be .7m in preparation to cross.
3) When the flag is equal to 2, the bridge finder indicates that the quad is converged to its desired location pre-crossing the bridge, set to be centered in y and offset back from the bridge by .7m. At this point, it moves forward 1.5m to cross the bridge.

After receiving a message, the controller will move to the desired waypoint, and then send a message to the bridge finder code to again look for the bridge.

It was decided to only move over .75m if the river is not detected, as this would give us multiple attempts to see the river. For example, at 1m height, we can see roughly 2m in width with out camera. However, even, with the filter, we have still gotten false detections (due to the carpet itself fluttering when the quad is nearby) or we have not been able to pick up the bridge with one sample set. However, moving only .75m will allow us at least 2 attempts to find the bridge, which has worked consistently so far.

III. Avoid the Wall

The front camera was used to during this portion of the project. The approach was developed with the assumption that the quad would be placed between 5 and 10 ft with the quad being able to see at least half the wall at takeoff. For cases outside the bounds listed the quad still might be able to complete the mission, however those cases must be tested and performance would be much less robust.

A. Wall Detection

To detect the location of the wall, feature matching from sequential images of the front facing monocular camera was used. In each image that was captured by the camera, features were detected using the OpenCV Orb detect and compute function. The OpenCV matcher BFmatcher was then used to detect feature matches with the previously captured image. The locations of the feature matches along with the disparities were then stored. To make an initial estimation of which feature detections were in the foreground of the image and which ones were in the background, the feature matches were run through a select K means clustering algorithm by disparity with K = 2. This gave a pretty reasonable determination in which features were detected on the wall and which ones were in the background.

To further filter out bad feature matches, the "good" feature matches were saved for the previous two images received, and then a median/standard deviation filter was applied to remove feature matches that were spatial outliers compared to the rest of the features. This further reduced the number of false features estimated to be on the wall. To estimate where the center of the wall was in the image plane, the 5 maximum and 5 minimum x and y values corresponding to the filtered feature matches were taken and the corners of the wall were estimated by taking the medians of these samples. From there, the center was estimated to be the centroid of these corners. We found this method for estimating the centroid of the wall to be more accurate than simply taking the mean of all features thought to be on the wall.

B. Wall Avoidance

To avoid the wall, it was advantageous to be able to estimate the position of the quad relative to the centroid of the wall. To accomplish this, we needed to be able to estimate depth, or how far away the quad was from the wall, so that the conversion between image frame and real world coordinates could be computed. It was discovered that estimating depth using the disparities between two successive camera frames was very imprecise due to the very small distance between the two images. To solve this problem, upon takeoff, the quad would take an image, would then travel up 0.35 meters, take a second image, and then use the wider distance between the two images to compute the wall depth at the initial position. Odometry was then used to update the depth estimate in each successive frame. With depth, the centroid of the wall relative to the quad could be computed in physical units, granting the ability to generate waypoints to give to the quadcopter.

To avoid the wall, the depth was first computed, and then the quad then determined whether the wall needed to be traversed over the top or underneath. Given the altitude of the quad and the position of the centroid and edges of the wall relative to

the quad, the distance from the floor to the bottom of the wall could be computed. If there was 1 meter of space, the quad would determine it should go under the wall, if there was less space, the quad would decide to go over the wall. After this decision was made, the quad was programmed to align itself up with the center of the wall side to side, and either the top or bottom edge of the wall (depending if it would go over or under) while being about 1.5 meters away. Once the quad converged on this point using feedback from the wall centroid estimation, a command was then given to either raise or lower elevation to provide sufficient clearance to avoid the wall, and then the quad was instructed to proceed forward until it passed the plane of the wall, at which point it was instructed to land.

C. Controller

A closed loop controller was used to move the quad to any given waypoint. This controller uses feedback from the odometry given by the bebop quad itself to accurately navigate to any point.

D. Integration

Similar to the bridge problem, there were two codes that were used for this task - one that handled the vision and one that handled the movement. This section addresses how these codes were integrated together.

After takeoff, the controller would send a message to the vision code to take a picture. Next, the controller would move the quad up .35m, before again sending a message to the vision code to take a picture. At this point, the vison code would calculate the depth from the disparity between the two pictures.

Next, the vision code would send the depth and the relative position of the wall relative to the quad back to the controller. The controller would move roughly 1/3 of the depth toward the wall and attempt to center itself in y and z (for the quad). After this, coupling iterations would begin. After the quad had reached the desired waypoint, the controller would send a message to the vision code and it would calculate the lateral distance needed to center the quad on the gate (as mentioned earlier, the coupling iterations don't use z, as if the full gate is not found, the z estimate for the gate center will be off. Only the initial estimate for Z before moving closer is used). This process was repeated until the quad was relatively centered on the gate center.

This process does not involve any yaw, so it only works if the initial yaw position is relatively small (¡ 35 degrees). This just means that the quad will fly through the gate at an angle.

IV. Results

A. Find the Bridge

During the testing, the quad was able to find the bridge relatively quickly on the first attempt. However, it did not see the bridge the first possible opportunity, instead moving past the bridge and then coming back to it. We believe that this was because the lights were lower than we had tested out (the other team was testing with lower lights and we forgot to change). When the lights are lower, there is less difference in pixel values on the surrounding carpet, leading to false river detections (we were tuned for 100 percent light). However, despite this the quad was still able to find and cross the river on the first attempt.

B. Avoid the Wall

During the live demo, the wall was placed in a ”medium” position - the bottom of the wall was roughly .5m above the ground. This first time, the quad was not able to detect the proper z location of the wall. However, the second attempt, the quad was able to sucessfully detect and fly over the wall.

V. Conclusion and Lessons Learned

One of the problems that we initially had was if we were too close to the wall, the field of view on the camera in the y direction (camera frame) was too small so that the camera couldn't see very much of the wall. This would lead to an estimation that the center of the wall was either higher or lower than it actually was, and would occasionally cause the quad to fly too high or low so that the wall left the camera's field of view completely. The way that we got around this was by having the controller only make adjustments to its height when the quad was far away so that it could keep most of the wall in the field of view at all times. Another way to address this problem would be to use a different resolution on the camera so that the field of view isn't cropped.

The second major lesson learned was about computational time. During initial testing of the bride finding algorithm, cv.circle was called many times for each pixel that was a river candidate (just for tuning using Rviz). On a laptop, this code could process one image in roughly .05-.1 seconds. However, it took over 1.5 seconds on the upboard. When the visualization was changed to cv.rectangle for the window instead of a nested for loop with cv.circle (1 cv.rectangle vs. 17*17 cv.rectangles), the time reduced back to around 10 Hz.

References

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