Circular Bullseye

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 problem of target identification and pose estimation using a Bebop Parrot. First, a method for identifying the target in multiple light conditions is introduced, along with methods to reduce all noise and fals detections. Then a simple pose estimation is presented that only needs one point on the target to find the relative pose. Finally, these two methods are combined and used to search, identify, and land on the target.**

I. Introduction/Problem Statement

The goal of this project to find and land on a circular bullseye target. The target will be black and white (Figure 1a) and the approximate coordinates of the target will be given relative to the quadrotors starting position.

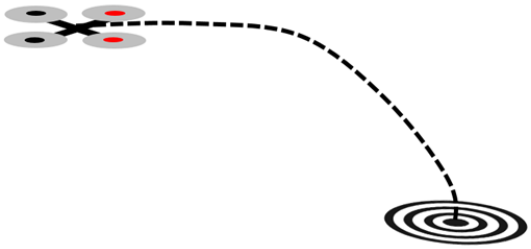

Fig. 1. The goal of this project is to find and land on a target.

For this project, the first task was to identify the relative position of the target relative to the current quadrotor position. Two different approaches were used for problem. First, a simplified PNP was implemented to find the position of the target to the quadrotor in the xy plane (only x and y coordinates were obtained). In this approach, the z distance to the target was considered to be the altitude (it was assumed that the target was on the ground). Once the position of the target was obtained, a closed loop 3D Controller was used to move to the target position quickly.

II. Target Identification

In this section, the challenge of finding, identifying, and calculating the position of the target are discussed. The main challenge of this project was to find the position of the quad relative to the target. A down-facing stereo camera was used to search for the target. A simple color thresholding algorithm was used to reduce the noise in the image and allow the circles on the bullseye to be identified.

To identify the target, the python function *HoughCircles* was used. This circle took in a grey scale image (the bottom-facing stereo camera was a greyscale camera) and fits circles to features in the image by using the gradient of the pixel value. The user can also specify other inputs, including the minimum and maximum radius of the circles (in pixels), the minimum distance between the centers of two circles, and a threshold for how ”circular” the circles need to be (how many possible false detections will there be). The output of the function is the center and radius (in pixels) of all circles found.

A. Color Thresholding

Originally, *HoughCircles* was tried on the raw image coming from stereo camera. However, this resulted in many false detections, as seen in Figures 2 and 3 below.

Fig. 2. Raw camera image of the ground.

Fig. 3. Without any thresholding, *HoughCircles* will pick up extraneous circles.

First, the input parameters for the *HoughCircles* function was adjusted to attempt to reduce the noise. Adjusting the minimum and maximum radii for the circles helped, but there were still many false detections, especially because there were circular features on the carpet.

Despite how the input parameters were adjusted, these circles would show up and give false detections. Therefore, a more robust method was desired, which involved thresholding the image to only show the bullseye. This was done by taking all pixel values that were below a certain value (between 0 and 255), and making them all zero (black), as seen in Figures 4 and 5.

Fig. 4. With a proper threshold, only the target will appear, making it easier to use *Hough Circles.*

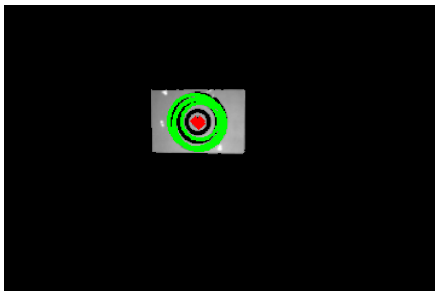

Fig. 5. With a proper threshold, only the target will appear, making it easier to use *Hough Circles.*

This removed all of the noise from the image, and allowed *HoughCircles* to only capture the circles given. In fact, the input parameters for *HoughCircles* could be relaxed, as there is no chance of false detection as long as the thresholding is tuned properly.

The thresholding values were obtained roughly 24 hours before the live demo to ensure that the lighting conditions were as close to testing conditions as possible. The thresholding values were found for 25, 50, 75, and 100 percent lighting. Zero percent lighting was attempted, and the thresholding would work if the quad was close to the target, but if it was very high or at an angle it was able to consistently identify the target. If the thresholding value was lowered to where it did, the background of the mask wouldn't remain black and there would be many false detections. Therefore, only 4 lighting conditions were truly found.

B. Calculating Pose - Simple Method

After obtaining a clean target image, the pose of the quad relative to the target could be found only using the center (with certain assumptions). This simple method was implemented first to check how robust and accurate it would be.

To find the pose of the quad relative to the target using only the center, similar triangles were used. First, the field of view of the camera was desired, as shown in Figure XX.

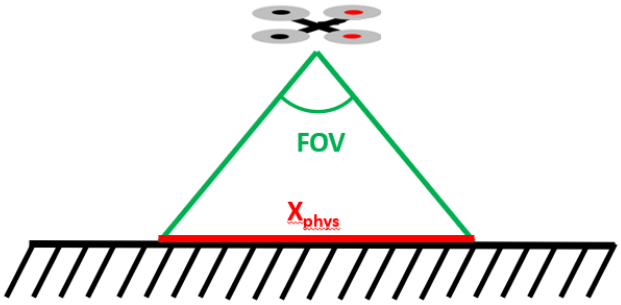

Fig. 6. The field of view of the camera is used to calculate the physical distance of the image.

The field of view was found in both X and Y direction by holding the quad at 1m and measuring what was visible on the camera. With the field of view, the physical distance could be related to what the camera frame saw using the following equation (there is a similar equation for Y).

$$X_{phys} = Altitude * tan(FOV_X/2) \quad (1)$$

If the physical distance from the edge of the image to the quad was known, then the x and y distances from any point on the image could be calculated using the pixel value of the center of the circle in the camera frame(given by *HoughCircles*) and the resolution of the camer.

The physical distance from the camera to the target is calculated using the following equation, where x_{center} and y_{center} are the pixel values given by *HoughCircles*.

$$X = Altitude * tan(FOV_X/2) * ((x_{center} - 160)/160) \quad (2)$$

$$Y = Altitude * tan(FOV_Y/2) * ((x_{center} - 120)/120) \quad (3)$$

For this method, x_{center} and y_{center} were calculated by averaging the center locations for the left and right camera. The altitude was determined by using the odeometry data published during flight. After calculating the positions in the camera frame, they need to be moved into the quad body frame so proper velocity commands can be given. For us, this involved switching X and Y and then multiplying Y by -1.

However, this will depend on how the camera is mounted.

This method is simple, easy to implement, and seems to be robust at several different lighting conditions. The two big assumptions made in this method are that pitch and roll are negligible, and that the altitude value given by the odometry is reasonable.

The first assumption we felt good making, as we can ensure that this is close when we collect data to send to the controller. A quadcopter cannot maintain hover with a nonzero pitch and roll, so the only times we would see large pitch and roll values would be if we were flying very fast. Our controller is tuned to such a way that don't move too fast (more on this will be given later), and once we arrive near the target we wait a half second for the quad to stabilize before taking the inputs.

For the second assumption, we found the controller is robust even if the altitude is not that accurate. The altitude is used to determine the scale factor of how far it needs to move - it doesn't influence the direction (unless the altitude is negative). Therefore, the quad will either overshoot or undershoot the target if the altitude is not exact. This just means that it will take longer to lock onto the target, but it will still eventually get to the same place, as long as the altitude error is reasonable (not negative, and within .5-.8m).

III. Controller

The controller used in this project is the same one created for Project 2. The controller is a closed loop controller that uses feedback from bebop autonomy during flight. Since the controller was introduced in detail in the previous report, this paper will provide a brief summary of how it works.

The controller is given a waypoint, which includes a Δx, Δy, Δz, and $\Delta\psi$ (although for this project, yaw is not used). From this information, it follows these steps:

1) The desired waypoint is converted from the local frame to the global frame set by the first odometry message.
2) The raw odometry data is converted into the global frame. This only affects 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) is found in the global frame. Since the desired velocity was 0, the velocity itself was the change in error vs. time.
4) The new control input was calculated 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.

From last project to this project, several updates were made to the controller, specifically dealing with tuning. Last project the controller was only used to fly small distances (¡1m at a time) and since this project could involve a large initial flight distance, we wanted to tune the quad for a larger distance. This was smart, as our original gains had a high reliance on Kp compared to Kd, which caused a very large overshoot when the waypoint is very far from the initial point. Our new gains had a higher Kd compared to Kp, drastically reducing the overshoot. These gains still work well for smaller distances.

Another change made to the controller was that we added in a speed cap. As mentioned above, for very large initial errors (large waypoints), we would have very large initial velocities that would cause an overshoot. Increasing Kd helped, but didn't completely fix the problem. We found that if we artificially capped the velocity input to the quad, this helped almost eliminate overshoot. This also helped the quad maintain a relatively level orientation (pitch and roll close to 0), which helps us with our methods for calculating the quad position relative to the target.

The second change made to the controller was a new set of gains was introduced for the z direction. We noticed that our movement in the z direction would take a very long time to converge, and over that time the quad would drift in x and y. Therefore, we increased both Kp and Kd, while making Kp higher than Kd, while verifying that there was a very small overshoot. During our mission, we change altitudes while searching for the target, so moving faster in z can significantly speed up our run.

IV. Integration

There were two python codes, the target identification code and the controller, that needed to be integrated. It was decided that a method similar to the approach taken in Project 3a would be used. Although the performance on Project 3a was lacking, it was determined that the main cause was the inability to recognize the gate, and if that was fixed, the method itself would work.

This method involved having the color thresholding code take in the images from the duo camera (right and left camera images), and constantly publish the x and y positions of the target relative to the quad (we want the target relative to the quad so that we can use these as waypoints for the controller). The controller would take this waypoint, move to it (while ignoring the continuous update of waypoints), and then get the next waypoint only after the first trajectory was completed. The target identification code would also send a flag along with the waypoint. If this flag was set to 0, the controller should fly to the waypoint, but if the flag is set to 1, the quad is on top of the waypoint and the controller should land the quad.

V. Mission Profile

This section will contain a brief overview of the mission profile, or how the quad will attempt to find and land on the target. First, the quad will takeoff from the given position and fly to the given "position" of the target (the mean of the gaussian distribution for where the target will be). Next, the quad will ascend from the takeoff height to a search

height which is an input parameter (usually around 1.5-2m). Next, the controller wait for a waypoint from the target identification code. At this altitude, with our cameras large field of view, we are normally able to identify the target that is roughly 1.5m in each direction. However, if it is unable to identify the target it will move up another half meter, expanding the area that it can visualize the target to close to 2m. After identifying the target, the controller will fly to the given set of waypoints and then lower down to 1m.

Next, the controller will get the waypoint from the target identification code, move there, and receive another waypoint. This process is repeated until the target identification code sends a flag telling the controller to land.

VI. Results

During the test, the location of the target was given as 2.7m in x and 1.6m in y away from the original drone location with $\pm$ .6m in both x and y. Our original search used the mean values and flew there first, before moving to 1.8m for our search. We were able to correctly identify and land on the target in roughly 50 seconds. Next, we tried searching at 1m instead of 1.8m. Because of the wide field of view, we were still able to find the target. Because we didn't take time to move up and down, we were able to reduce our time to 25 seconds.

For the next 10 minutes of testing time, we attempted to improve our speed by lowering the tolerances for both the target identification (when should we land) and the closed loop controller (when have we hit the waypoint). With these methods, we were able to reduce our time to 17 seconds.

VII. Conclusion and Lessons Learned

One major conclusion that we learned from this project was that lighting matters. We learned this a little in Project 3a, but we had several other issues with that project. In this project we were able to see that even waiting 1 hr from 5 pm to 6 pm can change the lighting to where the thresholding values need to change. One way to get around this would be using an adaptive thresholding technique, which would be especially helpful when reducing glare.

References

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