ENAE788M Assignment 4 - Flying quadrotor through a colored window

Estefany Carrillo, Mohamed Khalid M, and Sharan Nayak

I. INTRODUCTION

In this project, we provide the ability for PRG Husky quadrotor to fly through a colored window. We use leopard imaging camera to get images of the colored window and perform segmentation of the colored window using Gaussian Mixture Model (GMM). The segmented image is passed through a image processing pipline to generate centroid of the gate. The calculated centroid (desired location) and the current location is fed as input to a bang-bang controller to generate the necessary control inputs to control the quadrotor.

II. COLOR SEGMENTATION

A. Color Segmentation Using Thresholding

We implement color thresholding by first converting the image to HSV color. Using the function *inrange*, we obtain a mask by identifying the pixel HSV values that are between 26 and 33 for the hue value, and between 0 and 255 for the saturation and value. Then, we perform the function *bitwiseand* from OpenCV on the image and the mask to filter out the pixel values that are not within the color range desired. We observe that this method is very simple and fast, however, its performance degrades under darker lighting conditions. This can be noted in Figs. 1 - 4 below for different outputs obtained under bright and dark lighting conditions.

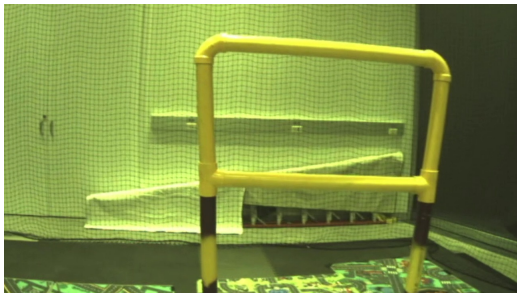

Fig. 1: Test image for color thresholding under bright light conditions.

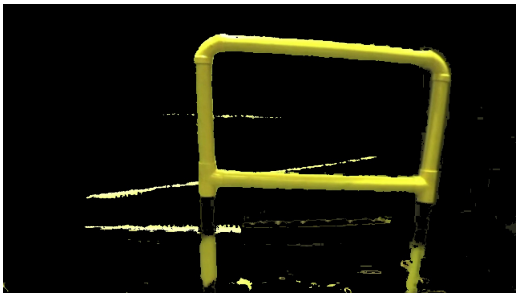

Fig. 2: Segmented image by using color thresholding under bright lighting conditions.

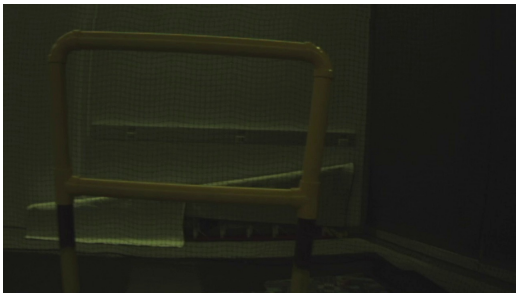

Fig. 3: Test image for color thresholding under dark lighting conditions.

B. Color Segmentation Using Gaussian mixture model

The color segmentation is performed using GMM using the Expectation Maximization (EM) iterative algorithm. The EM algorithm consists of two steps - E and M step. The E-step is used to calculate the cluster weights α_{ij} where i is the data point and j is the cluster index. The M-step is used to estimate the parameters maximizing the expected posterior log-likelihood $p(C_l/\boldsymbol{x})$ where C_l is the given color (purple or yellow) and $\boldsymbol{x}$ is the data vector. The iterative algorithm ends when the number of iterations exceeds the max number of iterations (M_{iter}) or when the norm $\|\bar{\boldsymbol{x}_c} - \bar{\boldsymbol{x}_{c-1}}\| < e_{thres}$ where $\bar{\boldsymbol{x}_c}$ is the mean at the current iteration and $\bar{\boldsymbol{x}_{c-1}}$ is the mean at previous iteration and e_{thres} is the error threshold. The number of Gaussians K in the mixture model is chosen to be 6. The initial weights for each of the Gaussians is chosen to be $\frac{1}{6}$. The means of the Gaussians are randomly chosen from the training set X_{train} of x. The co-variance matrices are taken to be diagonal matrices with random numbers between 0 and 1 on the diagonal.

The training set X_{train} is obtained by taking images of the colored window and cropping regions of interest corresponding to the sides of the colored window. See Fig. 10 for an example. The cropped images which are in RGB color space are converted to HSV color space and then formed into a long vector with each row consisting of three values representing a pixel. The training set is run through the EM algorithm to generate the model (means, co-variances and weights). An image and the output of the model for this image are shown in Fig. 11.

Fig. 4: Segmented image by using color thresholding under dark lighting conditions.

Fig. 5: Original and cropped images

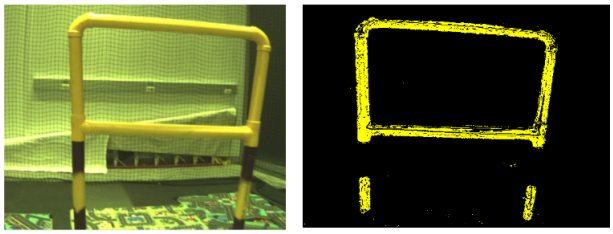

Fig. 6: Original and segmented image using GMM

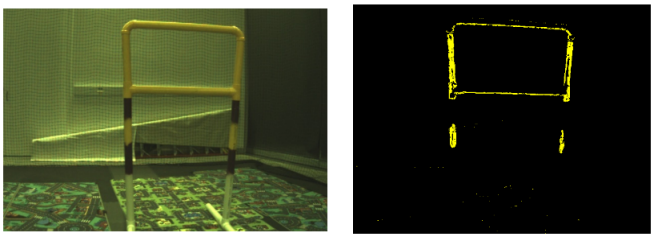

Fig. 7: Original and segmented image using Single Gaussian

C. Color Segmentation Using Single Gaussian

We use the same method as above but set $K = 1$. The segmentation is not found to be as great as that obtained using mixture of gaussians.

III. LINE FITTING

Once we apply color segmentation to an image, we proceed to detect the shape of the window and its centroid using the following methods.

A. Finding closed contours

The first method consists of detecting the largest connected component in the image and checking whether this component is convex. Before detecting the connected components, we apply morphological operations to erode the image by a small amount using a matrix of ones of size 3×3 and thus remove noise. Then, we dilate the image by a larger amount using a matrix of ones of size 20×55 to fill in gaps in the gate portion of the image and erode once more by a small amount using a matrix of ones of size 5×5 to decrease the size of the connected parts.

After applying morphological operations, we apply the function *connectedComponentsWithStats* to identify connected components and check the area percentage of each connected component with respect to the area of the entire image in order to keep the largest component. This is the component whose area percentage is above 5 percent of the total area. We set the pixel values corresponding to the largest connected component identified to 1 and the rest to 0. We then convert the resulting matrix of 1's and 0's to an image of color by multiplying each pixel value by 255.

In order to detect the edges of the largest connected component, we apply the Canny edge detection algorithm with a low threshold value of 100 and high threshold value of 200. Both of these values are determined empirically. From the resulting image, we only keep edges that are above a threshold by using the function *threshold* with the option THRESHBINARY.

Then, we use *findcontours* and *isConvex* from OpenCV to obtain convex and closed contours. Since we can get more than 1 contour from the image, we obtain the centroid for each convex contour and select the centroid with the smaller y-coordinate since this corresponds to the gate as opposed to its bottom part. The centroid of each contour can be obtained from the function *moments*. In Fig. 8, the contours of the largest connected component obtained from the test image in Fig. 1 are shown, and in Fig. 9, the centroid computed is shown.

Fig. 8: Contours of largest connected component obtained from test image in Fig. 1.

B. Finding Hough Lines

In this method, we first apply dilation to the image by a small amount using a matrix of ones of size 10×10 to fill in the gaps in the image from the color segmentation. Then, we apply Canny edges detection algorithm to find the edges

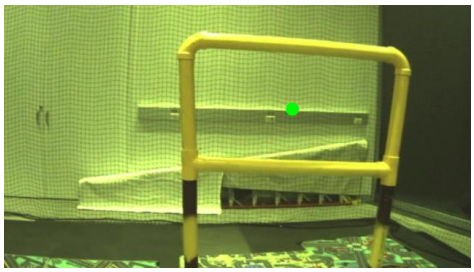

Fig. 9: Centroid (shown as a green circle) computed using contours from test image in Fig. 1.

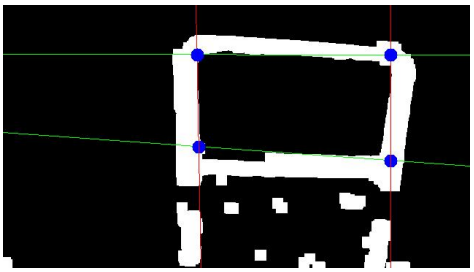

Fig. 10: Hough Lines obtained after Canny edges detection is applied on test image in Fig. 1.

using a lower threshold of 0 and a highes threshold of 200, determined empirically.

The next step consists of applying the function *Hough* from OpenCV to identify lines from the edges detected in the case edges are obtained. We notice that if the segmentation is of poor quality, Canny edges detection does not work and therefore this method cannot return an output.

The k-means algorithm is then used in order to classify the obtained Hough lines into horizontal and vertical lines. We use the available function *kmeans* from OpenCV to cluster lines into two groups by their angles. Once we obtain our two groups, we then apply k-means again to separate horizontal lines by their ρ value in order to distinguish top horizontal lines from bottom horizontal lines and choose among the top lines, the line with the largest ρ value and among the bottom lines, the line with the smallest ρ value. The same procedure is applied to identify the vertical lines corresponding to the inner part of the gate. Once, the set of two horizontal lines and two vertical lines is obtained, we compute the intersections of each pair of horizontal and vertical line as a solution to a linear system of equations using *linalg.solve(A,b)*, with A being the matrix with the cosine and sine of the orientations of each line, i.e. and b the vector with the ρ values of each line:

$$A = \begin{bmatrix} \cos\theta_1 & \sin\theta_1 \\ \cos\theta_2 & \sin\theta_2 \end{bmatrix} \tag{1}$$

$$b = \begin{bmatrix} \rho_1 \\ \rho_2 \end{bmatrix} \tag{2}$$

From the intersections found, we identify the corner points and can easily compute the centroid by averaging out the *min* and *max* of the x-coordinates and doing the same for the y-coordinates. Figs. 10 - 11 demonstrate the output of this method.

C. Filtering

From the centroids obtained using both methods mentioned above, we compute an average centroid estimate. We also observe that the centroid given in the first method is more consistent, while the second method requires a better image to return an output. We also implement a weighted moving average combining the centroid estimate at current time step and the average centroid estimate from all previous time steps to improve on the estimate of the centroid as we obtain more images during execution. We set a lower weight coefficient for the average centroid and a higher weight coefficient for the current centroid estimate. This helped reduce large deviations from the centroid estimate as the quadrotor moves.

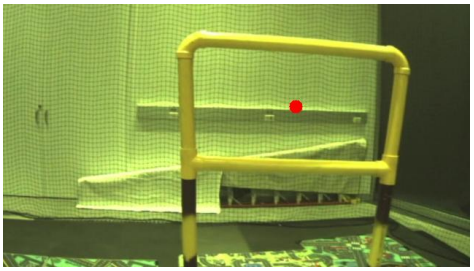

Fig. 11: Centroid (shown as a red circle) computed using Hough lines from test image in Fig. 1.

IV. CAMERA CALIBRATION

A. Estimating camera intrinsics

Camera intrinsic calibration comprises of estimating the camera calibration matrix K which includes the focal length and the principal point and the distortion parameters. A *pinhole model* was assumed as the projection model and *radtan* as the distortion model. `Kalibr` [1][2][3] package was utilized for obtaining the system parameters. An Apil grid was used as the basis for the callibration. The camera intrinsic parameters identified are as follows.

$$\begin{aligned} K &= \begin{bmatrix} f_u & 0 & p_u \\ 0 & f_v & p_v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 694.14 & 0 & 435.89 \\ 0 & 692.51 & 210.68 \\ 0 & 0 & 1 \end{bmatrix} \\ d_{coeff} &= \begin{bmatrix} k_1 & k_2 & r_1 & r_2 \end{bmatrix} \\ &= \begin{bmatrix} -0.3873 & 0.3011 & 0.0017 & -0.0030 \end{bmatrix} \end{aligned} \tag{3}$$

Multiple sample videos were recorded so as to obtain the least re-projection error. Fig. 12 illustrates that re-projection error is less than 1 pixel.

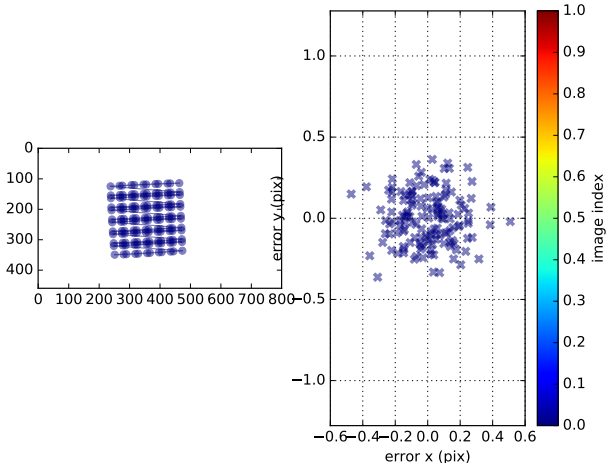

Fig. 12: Re-projection error obtained from `Kalibr` package

B. Color calibration

Any camera needs to be color-corrected so that the image sensor data resembles what a human perceives. Although it is not essential for GMM implementation, it is a good practice to color-correct the images for easier implementation and debugging. Further, it enables us to utilise the complete dynamic range of all channels when converting from bayer to RGB image.

The raw data from the camera was analysed when it was pointed at the pure Red, Green and Blue colors. The requisite biases for correction were estimated and used for calibration through `bcorrect`, `gcorrect` and `rcorrect` in `__init__.py`

V. POSE ESTIMATION

In a pinhole projection model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation. The model is given mathematically by,

$$s\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_C = K[R|t]\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_E \tag{4}$$

where, s is the image scaling, K the camera intrinsics and $[R|t]$ is the composite rotation-translation matrix. 'C' and 'E' represent the camera and earth frames respectively.

From 3D-2D point correspondences and camera intrinsics, one can determine object pose with respect to camera by using (4). Note, that a minimum of 3 corresponding points are required to estimate the pose. More the number of points, better the estimate would be, however at the cost of computation.

Given the information of dimensions of the the window and the corners from the GMM-CV pipeline, `solvePnP` routine from `openCV` library was used to estimate the pose of window. `cv2.rodrigues` had to be used to evaluate the rotation matrix, R_{CE}. Since, the camera is rigidly mounted onto the quadrotor, the rotation matrix from camera to body frame, R_{BC} is constant and expressed as,

$$R_{BC} = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \tag{5}$$

`cv2.solvePnP` returns the position vector of origin of earth frame expressed in the camera frame. The origin of earth frame was arbitrarily chosen to be one of the corners of the window. The position vector of center of window expressed in body frame, $\bar{d}$ is given by,

$$\begin{aligned} \bar{d}_B &= R_{BE}(\bar{r}_E + \bar{x}_E) \\ R_{BE} &= R_{BC}R_{CE} \end{aligned} \tag{6}$$

where, $\bar{r}$ is the position vector of origin of earth frame from quadrotor and $\bar{x}$ is the relative location of the centre of the window from the arbitrarily chosen corner (origin of earth frame).

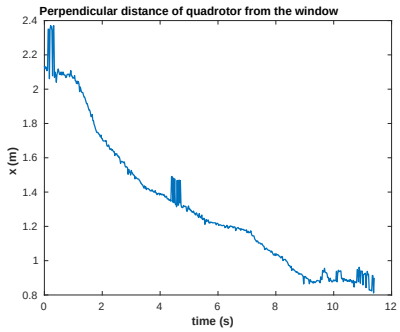

Fig. 13: Distance of the window from the quadrotor

As observed in Fig. 13 and 14, the `solvePnP` outputs position data with relatively less noise. A low-pass filter can be implemented to futher remove the sudden spikes in the distance data. Further from Fig. 13, it can be seen that after $x = 0.8$, we no longer get the data, implies that the camera's field of view is not that large to observe the 4 corners of the window for distances less than $0.8m$ and hence, this information should be utilized when computing the trajectory.

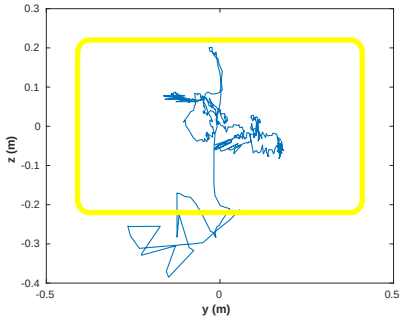

Fig. 14: Alignment of quadrotor with respect to centre of the window

The pose of window w.r.t. quadrotor expressed in Euler angles (yaw, pitch and roll) is computed through the following set of equations.

$$\psi = \tan^{-1}(r_{21}/r_{11})$$
$$\theta = \tan^{-1}(-r_{31}/\sqrt{{r_{32}}^2 + {r_{33}}^2}) \quad (7)$$
$$\phi = \tan^{-1}(r_{32}/r_{33})$$

where r_{ij} are the entries of the transformation matrix, R_{BE}.

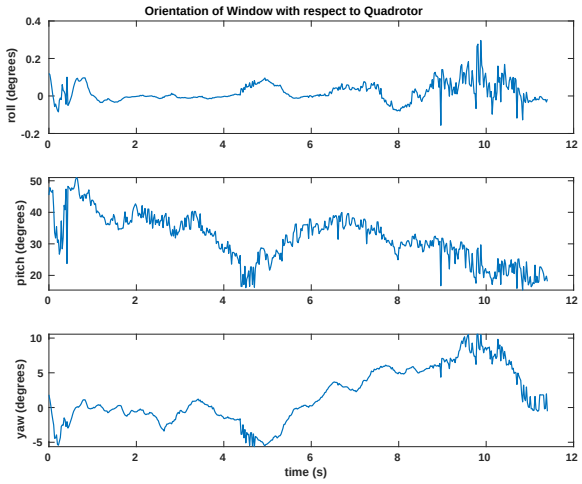

Fig. 15: Orientation of window w.r.t quadrotor

Fig. 15 illustrates how noisy the orientation estimated from `solvePnP` is. We have to implement some sort of filters - low pass, moving average or fuse IMU data using EKF to get a better orientation estimate. Owing to the noisy characteristics of the estimate, we did not utilise the orientation data in our control strategy.

VI. TRAJECTORY CONTROLLER

The trajectory controller is implemented using a simple bang-bang control. The control law is implemented in image space using the deviation from the current location as the control input Fig. 16. We use bang-bang control because the motion of the quadrotor is stable and the control law causes little drift. The center location of the image is taken as the current location of the quadrotor. The centroid of the colored gate calculated from the image processing pipeline is taken as the desired location. The deviation in the locations is given as input to the bang bang controller which generates a fixed control signal. The actual values for the bang-bang control in the X, Y, Z axis were determined empirically. The control input in the X direction (towards the gate) is always provided after the quadrotor reaches the hover position so that the quadrotor makes progress in moving towards the gate which in turn causes the image input to the image processing pipeline to get better. We use the image processing pipline to determine when we cross the gate. This is determined by the pipline seeing more background noise than the gate itself. Once we have crossed the gate, we provide a control input in the X-direction to move away from the gate and then land.

The current limitation of our controller is that there is no control input for performing a yaw. This is because it is not possible to estimate the orientation of the gate directly by processing a single image. We tried using the orientation estimate using the PNP method but the orientation values were too noisy. Our future work for our controller will include filtering the orientation values from PNP and then using the estimated yaw to orient in the right direction and then move forward.

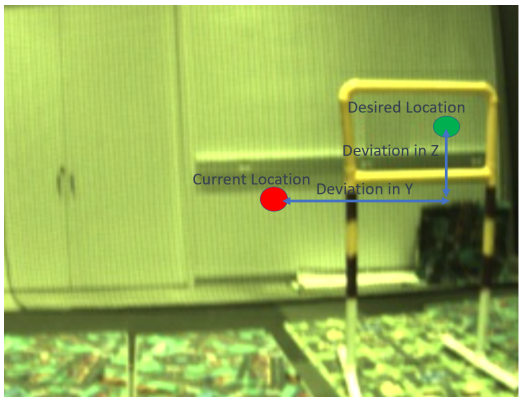

Fig. 16: Current (red) and desired (green) locations given as input to controller.

VII. PLOTTING IN RVIZ

We use Rviz to plot the colored window and real-time waypoint positions of the quadrotor. The colored window is plotted using the visualization_marker array message which helps plot the individual sides of the colored window. For plotting the position of the quadrotor, the ROS TF transform containing the position coordinates were broadcasted from our program to Rviz every 0.1 sec. The position coordinates of the quadrotor were generated using PNP. Since PNP does not perform well when the quadrotor is close to the gate, we were not able to plot the quadrotor going through the window. Fig. 17 shows the yellow colored window, world and quadrotor coordinate systems being displayed in Rviz.

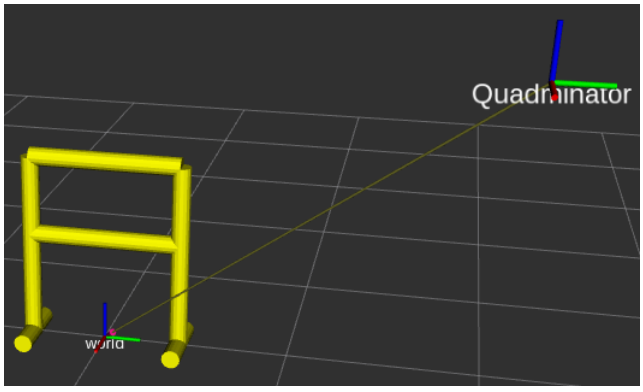

Fig. 17: World and quad frame displayed in Rviz

VIII. CONCLUSION)

We provided the ability to PRG Husky quadrotor to fly through a colored window. We used GMM to perform color segmentation and Hough lines and closed contours method to

generate the centroid. The difference in the current location and desired location (centroid) is input to the bang-bang controller to control the quadrotor. We used PNP to generate the position coordinates of the quadrotor in the world frame for display in Rviz. Due to very noisy orientation estimates from PNP, our yaw controller did not work as expected and hence we did not use yaw controller in this project. Our future work will include filtering the noisy orientation estimates and integrating the yaw controller into our system.

REFERENCES

[1] Paul Furgale, Joern Rehder, and Roland Siegwart. Unified temporal and spatial calibration for multi-sensor systems. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1280–1286. IEEE, 2013.

[2] Paul Furgale, Timothy D Barfoot, and Gabe Sibley. Continuous-time batch estimation using temporal basis functions. In *2012 IEEE International Conference on Robotics and Automation*, pages 2088–2095. IEEE, 2012.

[3] Jérôme Maye, Paul Furgale, and Roland Siegwart. Self-supervised calibration for robotic systems. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 473–480. IEEE, 2013.