Lab 3 Report

Group 7: Yu GU (yg2466), Boyuan Chen (bc2699), Yueting Lu(yl3607)

Question 1:

Implemented color tracker. Code Implementation can be found in cart_color_tracker.py

Comment: There were certain problems present during the course of experiment – mostly due to hardware limitations. As a result, we had to make some adjustments during implementation to compensate for these limitations.

a. The perceived fps of the cart, while connect through the 150MB WI-FI port, is roughly 2 – 3 or less viewing on the monitor from desktop. However, when printing out the center of target for each frame, we get around 10 fps judging from the output in the console.

   Since there is significant lag between cart and monitor, we limited the fps to 4 in the code in order to reduce the difference between what we perceive and what the cart receives.

   We also reduced the resolution to 320 by 240 in order to reduce the lag between cart and monitor, which helped to some extend but did not provide substantial improvement.

b. The tracker performance is highly correlated with the light condition of the environment. As the surface of the target might reflect lights and cause changes in both luminance and perceived color.

   In order to compensate for the changes in environment, we calculated the Hue based on the statistics and fixed the saturation and value with a range of 50 – 255. While certain level of accuracy lose is inevitable, our methods guaranteed that we can track the target in most cases provided that noise in the background is negligible.

Implementation

1. Initiating Tracker and Create Binary Image

In our implementation, we used the getXY function provided to obtain a list of clicked points. Then used the obtained points coupled with the values of these pixel in the converted hsv image to generate the Binary Image required.

As mentioned earlier, we fixed the saturation and value of the hsv spectrum while deducing hue from the given points. In terms of smoothing, meanwhile, we adopted a Gaussian with a

kernel size of 15 by 15. Then we eroded the image with a rectangular area of 3 by 5 and dilated the resulting mask.

After these procedures, the resulting mask is less sharp than original mask and most of the noises were wiped out from the final mask.

Steps are as follows:

Step 1. Set picamera configuration (resolution 320 240, fps 4)
Step 2. Read Picamera stream using capture_continous.
Loop:
    Step 3. Convert the Image to HSV
    Step 4. Check for record_flag (if set to true: start initializing color tracker)
    If True:
        Step 5. Get list of clicked points
        Step 6. Obtain the lower and upper bound of hue based on points and hsv image
        Step 7. Set threshold
        Step 8. Calculating Mask (will be explained later)
        Step 9. Detect blob (will be explained later)
        Step 10. Show image and set record_flag to false
    Else:
        Step 5. Check if thresh is set.
        If set:
            Step 6. Calculate Mask
            Step 7. Detect Blob
            Step 8. Show image

        If not:
            Step 6. Display unfiltered image
    Step 5. Set waitkey. If ‘s’ is pressed, set record_flag to true.

For mask calculation, we used the inRange function provided by opencv to obtain the mask based on threshold obtained through clicked points.
Steps are as follows:

get_mask
Step 1. Filter hsv image with lower and upper bound using inRange
Step 2. Gaussian Blur the mask with 15*15 kernel
Step 3. Erode the mask with 3*5 kernel
Step 4. Dilate the mask

2. Find Blob and Centroid

detect_blob
The movement detection and blob finding functions are integrated as we need to update the area and center accordingly, which will be explained later.

When finding the blobs, we used the function provided by opencv findContours, which returns a list of points of all detected contours in the image. We than get the area of each perceived contour and find the largest one, denoting it as the target.

Steps are as follows:

Step 1. Find contours in the graph
Step 2. For each contour found, loop and find the contour with the largest area
Step 3. Check update parameter
If 0:
    Step 4. Update center, area
    Step 5. Parse cart action
Step 4. Draw Contours and Center.
Step 5. Return resulting image to caller

3. Determine Movement and Orientation

For the movement determination part, we adopted two algorithms for implementation.
The most important component of this part if the determine the distance for which the cart moves.

Tag: Naïve_Movement
For starters, we adopted the methods provided in class: determining movement based on the initial image and resulting image.

Steps are as follows:

Step 1. Check if new area is 105% larger than initial area
If True:
    Move forward for a set amount
Step 2. Check if new area is 95% less than initial area
If True:
    Move backward for a set amount

In our implementation, we simply instructed the cart to move forward/backward if area difference is triggered and compare the resulting area size with the initial area size.
If the new area size is within 5% difference of the old area size, then we consider the cart has moved to an acceptable position.

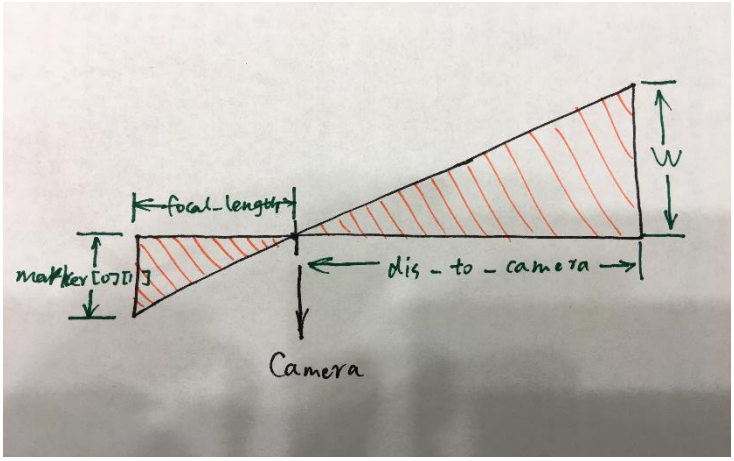
Method 2.
Parse_cart_command
Our second implementation of the movement determination used simple geometry to determine the distance of the target from the cart.

Show below is the pinhole model. We assumed that the encompassing target area has proportional changes with respect to changes in area size. The implementations can be found in *move_cart_forward* and *move_cart_backward.*

The trigger condition is similar to the one we used in naïve movement method: if the area difference is greater than 5%, we initiate the movement to keep the area difference within 5%.

We deduct the change in distance based on the input and convert the resulting distance difference to encoder count using tire perimeter and encoder count for a full rotation.

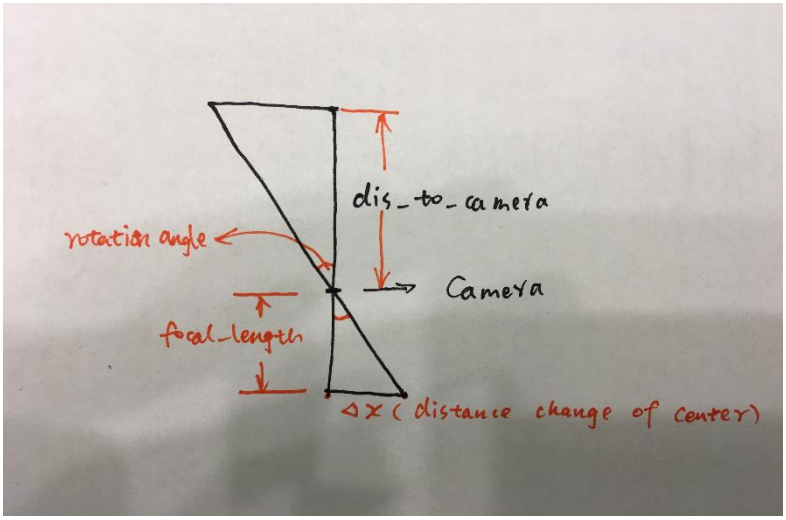
It should be noted that, however, cart movement (forward and backward) is performed after the cart has turned in order to reduce the additional complexity in turning caused by movement.



Angle of Movement

Since the GoPiGo has limited movement and camera is fixed on the servo(90) on the cart, we can only turn the camera left and right in terms of movement. Furthurmore, we assumed that the depth difference will be compensated by the movement method we dopted.
Therefore, we calculated the angle of movement based on the x-coordinate difference of the center.

We want to know how the object move left or right, so that the robot can rotate with the object. The key to this question is to find out how the center of the object moves. By using the triangle relationship again, we can get the rotation angle very easily. The details can be shown in the following picture:



After obatining the angle of movement, we simply calculated the corresponding encoder count and instructed the cart to turn left or right accordingly. The conversion functions are the same as we used in bug2.py

Question 2: Done. Implementation can found in hand_detector.py

The basic idea of the hand_detector algorithm is to find the defects of every hand gesture.

Detecting different gestures based on the number of defects recognized by robot's camera. Different number of defects correspond to different meaning, so that the robot can move according to different hand gestures.

Users can define specific moves by setting it up according to the number of defects.