**Robotics Lab 7 Report**

**Nimra Sohail Areeb Adnan**

Task 7.3

Our object detection strategy:

For detecting each colored cube we used the same strategy. Hence if a strategy holds true for red cube detection, it holds true for green,blue and yellow cube detection too. Lets understand first how thr=e red cube gets detected:

In the code, the red cube is detected through a series of image processing steps.

1)The image is loaded using the imread() function and converted to double precision using im2double().The im2double function scales the pixel values of the image to the range [0,1] by dividing all pixel values by the maximum possible value for the image format. For example, if the input image is of type uint8 (8-bit unsigned integer), the maximum pixel value is 255, so the im2double function divides all pixel values by 255 to scale them to the range [0,1].

2)The red color range in the RGB color space is defined using the min and max variables. The red color range is defined as [0.2 0 0] to [0.4 0.16 0.18], which represents a range of dark to medium red colors. The range of acceptable RGB values for red is defined by the min and max variables, which specify the lower and upper bounds, respectively. We do this by trial and error method for example: 

This is an enrlaged view of the red block only and we see that the point highlighted has some [R G B] colors, we find the maximum and minimum of eacg R,G,B color within the red box to identify the red color. The 4 by 6 matrix in code specifies each row for each color and 3 columns first for min value and the next 3 columns for max vaules.

Note:We are runnig a for loop of size 4 as we have 4 rows each corresponsing to a colored cube. Then we extract the min and max value of each colored cube and then create a color mask. The color mask concept is explained below.

3)The red\_mask is created by thresholding the image to extract the red regions. This is done by checking if each pixel in the image falls within the red color range using logical operations on the red, green, and blue channels of the image. The threshold is applied to each color channel of the image separately. Specifically, the red\_mask variable is created by combining three separate logical conditions, corresponding to the red channel, green channel, and blue channel of the image, respectively. The logical conditions are defined using the >= and <= operators to check if each pixel's RGB values fall within the acceptable range for red.In other words, each pixel's red value must be greater than or equal to the min value for the red channel, and less than or equal to the max value for the red channel. Similarly, each pixel's green value must be greater than or equal to the min value for the green channel, and less than or equal to the max value for the green channel. Finally, each pixel's blue value must be less than or equal to the max value for the blue channel, and greater than or equal to the min value for the blue channel.

4)The red\_mask is cleaned up using morphological operations. This is done to remove small regions of noise in the mask and ensure that the red regions are connected. In this code, the imopen() and imclose() functions are used to remove small objects and fill in any gaps between larger objects in the mask. First, a square structuring element is created using the strel function. The structuring element is a small binary image that defines the neighborhood around each pixel that will be used for the morphological operation. In this case, we are using a square structuring element with a side length of 10 pixels, which means that each pixel will be evaluated based on its 10x10 square neighborhood. Next, two morphological operations, imopen and imclose, are applied to the thresholded image red\_mask using the square structuring element se. imopen is an erosion followed by dilation operation, which helps to remove small objects and smooth out the boundaries of larger objects in the image. imclose is a dilation followed by erosion operation, which helps to fill in small gaps and holes in objects. Morphological operations are mathematical operations that are used to manipulate binary images. In the context of image processing, they are often used for tasks such as removing noise or filling in gaps in binary images. Two morphological operations are applied to the red\_mask. The first operation is an opening operation, which is performed using the imopen function with a square structuring element of size 10. An opening operation is typically used to remove small objects from a binary image while preserving larger ones. This is achieved by eroding the image first, which shrinks the objects, and then dilating it, which expands them back out. The idea is that small objects will be eroded away completely and will not be able to expand back out again, while larger objects will remain mostly intact. The second operation is a closing operation, which is performed using the imclose function with the same square structuring element. A closing operation is essentially the opposite of an opening operation: it is used to fill in small gaps in binary images while preserving larger structures. This is achieved by dilating the image first, which expands the objects, and then eroding it, which shrinks them back down. The idea is that small gaps between objects will be filled in by the dilation, but larger structures will remain mostly intact. By performing both an opening and a closing operation on the red\_mask, the code is able to remove small noise and fill in small gaps, which should help to produce a more accurate detection of the red cubes.

5)Connected components in the mask are identified using the bwconncomp() function. Connected components are sets of adjacent pixels that have the same value. In this case, the connected components correspond to the red regions in the mask. Step 5 in the given code involves finding the connected components in the binary mask generated in step 3. Connected components are defined as sets of pixels in an image that are connected to each other, either directly or indirectly. In the binary mask, the connected components correspond to the regions that meet the criteria set in the red color range.

The MATLAB function bwconncomp is used to find the connected components in the binary mask. The function takes the binary mask as an input and returns a structure “cc” that contains information about the connected components. Specifically, cc.NumObjects provides the total number of connected components, and cc.PixelIdxList is a cell array containing the indices of the pixels that belong to each connected component.

6)The location of the red cubes is extracted from the connected components using the regionprops() function. The 'Centroid' property is used to extract the center coordinates of each connected component. Once the connected components are found, the 'regionprops' function is used to extract the properties of the connected components. In this case, we are interested in the centroids of the connected components, which are the pixel coordinates at the center of each red region. The 'regionprops' function takes two arguments: the binary mask and the type of properties to extract. In this case, the type of properties is 'Centroid', which returns the centroid of each connected component. The output of 'regionprops' is a structure containing information about the properties of each connected component. This information is stored in the 'stats' variable. Finally, the centroid locations of the red regions are extracted from the 'stats' variable and stored in the 'red\_cubes' variable. The 'cat' function is used to concatenate the centroid locations into a single matrix. The resulting matrix contains the x and y coordinates of the centroids, where each row represents a single red region.

Note: “all\_cubes” is an empty array that is defined to store the centroids of all detected cubes. At each iteration of the for loop, stats.Centroid contains the centroid coordinates of all the connected components found in the binary mask for the current color range. cat(1, stats.Centroid) concatenates the centroid coordinates vertically into a single array. The semicolon ; is used to concatenate this array with the existing all\_cubes array along the first dimension. This means that the new centroids will be added as new rows to the bottom of the array. This way, at the end of the for loop, all\_cubes will contain the centroid coordinates of all the detected cubes in the image. For example all\_cubes had the following information:

>> all\_cubes

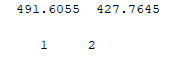
all\_cubes =

491.6055 427.7645

264.3077 376.2885

538.5000 245.5000

290.6541 163.6541

7)The location of the red cubes is displayed using the disp() function. The size of the red cubes array is also displayed.For example for the red cube, the variable “red\_cubes” contained the following information of x and y coordinates in the image  and 1 by 2 is the size of the matrix being formed.

Note: We use a loop that iterates over the four color ranges defined earlier in the color\_ranges matrix. For each color range, it selects the corresponding centroids from the all\_cubes array and stores them in color\_cubes. The first line inside the loop defines a string array color\_name with the names of the four colors: 'Red', 'Blue', 'Green', and 'Yellow'. The second line uses indexing to extract the centroids corresponding to the current color range. The expression (i-1)\*cc.NumObjects+1:i\*cc.NumObjects is used to select the correct range of rows from the all\_cubes array. cc.NumObjects is the number of connected components (i.e., cubes) detected for the current color range, so the expression (i-1)\*cc.NumObjects+1 gives the index of the first row to select, and i\*cc.NumObjects gives the index of the last row to select. The : in the second dimension selects all columns.

8)The original image is displayed using the imshow() function. The location of the red cubes is displayed as red circles using the plot() function. The 'MarkerSize' and 'LineWidth' options are used to control the size and thickness of the circle markers.

Note: we are using a for loop here also. hold on;: Tells MATLAB to keep the current plot active, so that additional plots can be added to it.

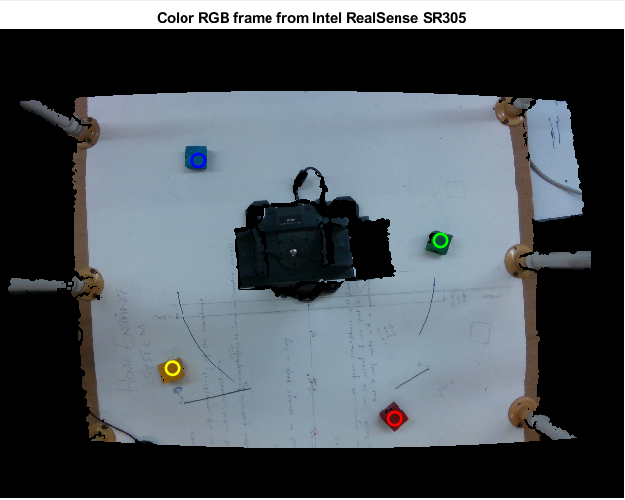
colors = ['r', 'b', 'g', 'y'];: Defines an array of colors to use for each set of cubes.

The loop iterates through each color range in the color\_ranges matrix.

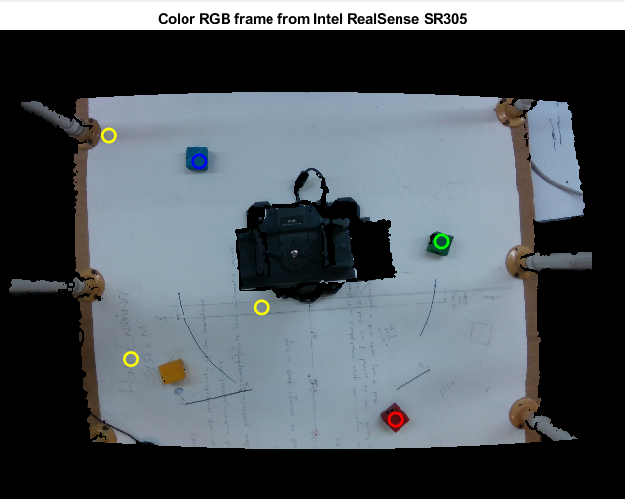
color\_cubes = all\_cubes((i-1)\*cc.NumObjects+1:i\*cc.NumObjects, :); Selects the centroids of the cubes that fall within the current color range.

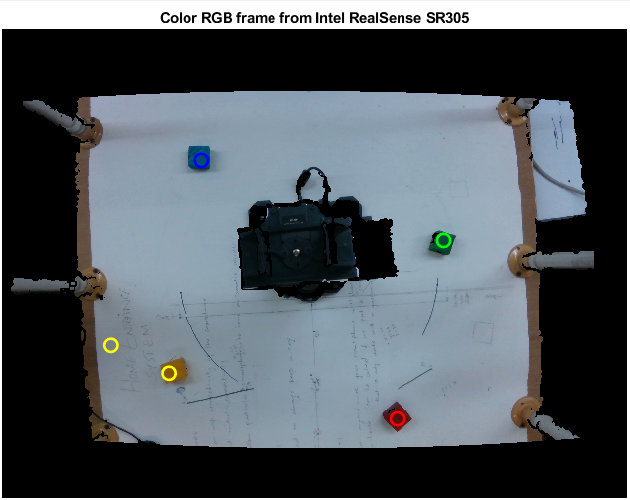
Now moving on, as part of the pre-processing stage we adjusted the lightning of the frame such the cubes get identified more prominently with the base becoming more white and making cubes more visible. The cubes were intentionally placed at smooth surfaces and not at bumpy places on the cardboard because that hinders the right information being captured for the block. We didn’t updated any parameter from intelrealsense camera but ran the depthexample() code to get the rgb image and used that for processing.

Images after test run:



Before when we were doing trial and error, some of the following runs:

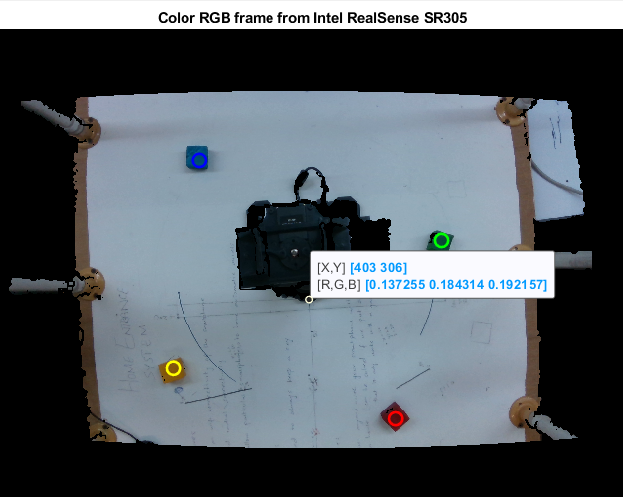




By trial and error and making the min and max ranges more accurate we ended up getting a single yellow block being identified. Same was done for other blocks.

For this setup, the red block was identified with a higher accuracy then other blocks. When we placed these blocks at some other places, because the light falling on them changed hence their RGB values also varied somehow. Amongst the four colored cubes, red cube was still being identified the most easily within the frame. For the given setup, all colors were identified separately and when we placed two red blocks, they both were getting identified as well.

Now for determining the position of each block we assumed that our centre lies at following point in workspace:



Assuming this point is our origin we subtracted this point from all the other cube’s coordinates and got new coordinates in the specified workspace. The achieved coordinates were:

R\_CUBE\_COORD =

88.6055 121.7645

B\_CUBE\_COORD =

-112.3459 -142.3459

G\_CUBE\_COORD =

135.5000 -60.5000

Y\_CUBE\_COORD =

-138.6923 70.2885

Code:

% Clear the command window and any existing variables

clc;

clear;

% Read in the image and convert it to a double precision format

im = imread('untitled.png');

img = im2double(im);

% Define the color ranges for the cubes in the RGB color space

color\_ranges = [0.2 0 0 0.4 0.16 0.18

0.43 0.3 0 1 0.35 1

0.02 0.24 0.2 0.2 0.31 0.3

0 0.2 0.4 0 0.5 0.5];

% Create an empty array to store the centroids of the detected cubes

all\_cubes = [];

% Loop through each color range and detect the cubes in that color

for i = 1:4

% Define the minimum and maximum values for the color range

min = color\_ranges(i, 1:3);

max = color\_ranges(i, 4:6);

% Create a binary mask that is true for all pixels that fall within the specified color range

color\_mask = (img(:,:,1) >= min(1)) & (img(:,:,1) <= max(1)) & ...

(img(:,:,2) >= min(2)) & (img(:,:,2) <= max(2)) & ...

(img(:,:,3) >= min(3)) & (img(:,:,3) <= max(3));

% Clean up the mask using morphological operations to remove any small holes or gaps in the color regions

se = strel('square', 10);

color\_mask = imclose(imopen(color\_mask, se), se);

% Find connected components in the mask (i.e. clusters of adjacent pixels that are true)

cc = bwconncomp(color\_mask);

% Extract the centroid locations of each connected component as a list of points

stats = regionprops(cc, 'Centroid');

% Append the centroids to the list of all cubes

all\_cubes = [all\_cubes; cat(1, stats.Centroid)]

end

% Display the number and coordinates of the detected cubes for each color

for i = 1:4

color\_name = ['Red', 'Blue', 'Green', 'Yellow']

color\_cubes = all\_cubes((i-1)\*cc.NumObjects+1:i\*cc.NumObjects, :)

disp([color\_name(i) ' cubes:'])

disp(color\_cubes)

disp(size(color\_cubes))

%fprintf("error check\n")

end

% Display the original image with the detected cubes marked as colored circles

imshow(img);

hold on;

colors = ['r', 'b', 'g', 'y'];

for i = 1:size(color\_ranges, 1)

color\_cubes = all\_cubes((i-1)\*cc.NumObjects+1:i\*cc.NumObjects, :);

plot(color\_cubes(:,1), color\_cubes(:,2), [colors(i) 'o'], 'MarkerSize', 10, 'LineWidth', 2);

end