Image processing lab project

Andres Sebastián Salazar Alturo

Candidate number: 276209

Lecturer:
Dr Phil Birch

Module:

Image processing

University of Sussex
Artificial Intelligence and adaptive systems MSc
Brighton, UK
2024

Introduction

Image processing has multiple applications, using machine learning models and different strategies to process information effectively to be applied in autonomous vehicles, social media, and security. This project objective is to process images that are based on the Lego's life of George. The images are 4x4 boards with colors red, blue, white, and yellow. To achieve this an algorithm in MATLAB is proposed. The algorithm is made of the functions *findColors, loadImage, findCircles, correctImage* and *getColors.* These functions will be explained, the logic of the functions, design decisions and suggestions to improve algorithm's performance.

Methods

The functions used and explanations were:

1. **FindColors**

The function has as input image from the hard disk. This function calls the functions *loadImage, findCircles, correctImage* and *getColors*. Finally, it returns the result that is a 4x4 matrix with the colors of the input image in the same orientations as the image viewed from standard view.

This function works as the main for the MATLAB file findColors.m. Hence, do not have a complex structure or logic implementation.

2. **LoadImage**

The function has as input the file name. Then, the image is imported and denoised. The strategy used to denoise the images was to apply a median filter to each RGB channel and to the median filtered image is applied a Gaussian filter. As this decision increased the algorithm effectiveness to detect the colors. Using one filter leads to confusion of red and blue.

It was decided to filter the image before detecting the reference black circles, to guarantee that the coordinates found of the circle are as precise as possible.

It could be possible to improve the filter by tuning the neighborhood size in the median filter and Gaussian filter.

3. **FindCircles**

The function’s input is the filtered image. It processes the image to extract the coordinates of the black circles. It converts the image to grayscale, applies the threshold to get the binary image and then inverts to get the black circles as white. The minimum and maximum values of the circles are set to the simplicity of the algorithm and as the images are known is possible to do it.

Then, the function imfindCircles that is a MATLAB function to detect circles based on the Hough transform, a technique to detect shapes. After that, detects the circles and their positions based on how clear the circles are, using sensitivity setting to tune the detection accuracy. Just the four most ‘clear’ circles based on the *metric* value and select them using *maxk.* Finally, the coordinates are sorted into the top positions first, then the bottom positions.

The function could be improved by changing the strategy to get the circles. As in the projected images, is not possible to effectively detect the black circles. Also, this strategy was done to rotate the image, but not to modify the angle of the processed image.

4. **correctImage**

The function is designed to rotate the image given the reference coordinates that are 2 black circles. The two at the top. It extracts the circles coordinates, sets the position of the circles, estimates the rotation angle by the difference in x and y of the circles. It calculates the angles using the function *atan2* and converts the angle to degrees using the function *rad2deg*. Finally, the image is rotated by the calculated angle. The rotation uses bilinear interpolation for better results and the image is cropped to maintain the original size of the image. To rotate the image, the function *imrotate* was used. It rotates the image given the input angle.

To improve the function as in the previous one, is necessary to implement an algorithm that can handle the projected images. Therefore, a more robust technique to identify the black circles is necessary. Moreover, the logic of the general algorithm just processes original, rotated and noise images. It raises an error when a projected image is used as input.

5. **getColors**

This is the largest function. It has as input the rotated image. Even if the image does not require to be rotated, as the orientation is not a restriction, it was decided to rotate all the image and have a general process. The input image is converted to HSV color space, to make it easier to detect the squares in the grid. The masks in HSV for the colors were created, these masks detect a range of the colors, so the filtered image spectrum is covered.

The functions *imopen* and *bwlabel* are used. These functions perform a morphological operation that removes small objects from the foreground based on the structuring image and labels connected components in a binary image where uses the cleanedCombineMask to label a set as white (True) or black (False) background.

Then, the squares that have areas greater than 5800 and smaller than 6200 are selected as the color squares needed to classify next. Then, the structure variable that has the information of the squares is sorted by the position of the squares in the image, as the coordinates are x, y and ensure that when detecting the colors, the output is going to be in the same order as the image. Then, after filtering the squares and sorting them, the squares are extracted from the filtered image and stored in a cell to be processed.

Finally, the color detection is done by creating a structure variable with the color ranges in LAB color space because is more uniform and it's easier to distinguish the red and blue colors. For this, every square is converted to LAB and the mean LAB color is calculated. The mean is compared to the preselected ranges and given the smaller distance a color is selected as the color of the square. The color is appended to the final matrix and returns the solution for the original image.

Is possible to improve the function by splitting the process in at least another function, so the function would be easier to understand and avoid that a lot of the process is done in one single function.

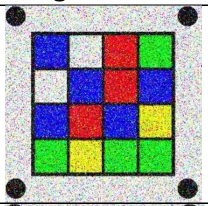
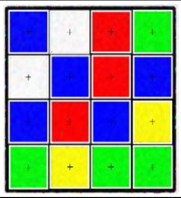
Discussion

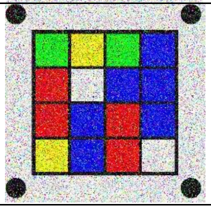
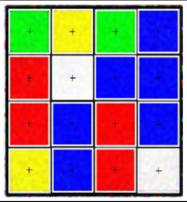
The solution in general works well. However, is not robust enough to process projected images. A new approach for these images must be implemented. I could be a process where the image is enlarged and the black circles are easier to detect, then calculate not just the angle between two circles but provide a reference image to project to standard view.

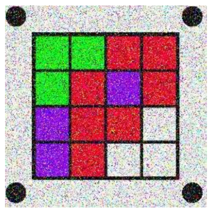
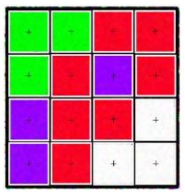
In my results, the projected images are not working and were giving errors. I took those images out of my test. That is why my mean score is 100. Nevertheless, I am not processing the seven projected images.

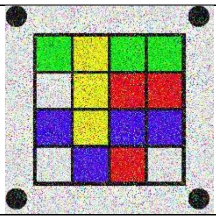
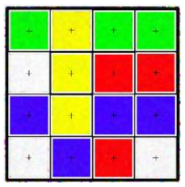
Results

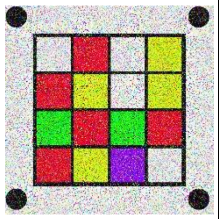
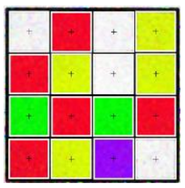
Filename	Image	Output image	Output	Success	Notes

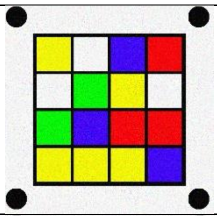
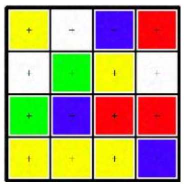
noise_1.pg			{'blue' } {'white' } {'red' } {'green' }
			{'white'} {'blue' } {'red' } {'blue' }
			{'blue' } {'red' } {'blue' } {'yellow'}
			{'green'} {'yellow'} {'green'} {'green' }
				Yes	No notes.

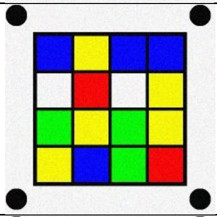
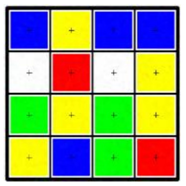
noise_2.png			{'green' } {'yellow'} {'green'} {'blue' }
			{'red' } {'white' } {'blue' } {'blue' }
			{'red' } {'blue' } {'red' } {'blue' }
			{'yellow'} {'blue' } {'red' } {'white'}
				Yes	No notes.

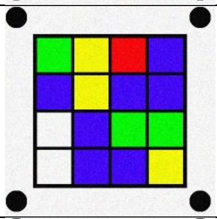
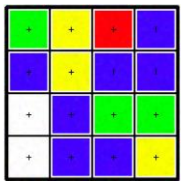
noise_3.png			{'green'} {'green'} {'red' } {'red' }
			{'green'} {'red' } {'blue' } {'red' }
			{'blue' } {'red' } {'red' } {'white'}
			{'blue' } {'red' } {'white'} {'white'}
				Yes	Purple has similar values to blue in LAB color space, that is why is detected as blue.

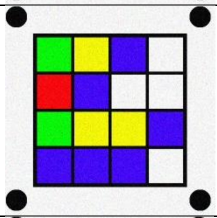
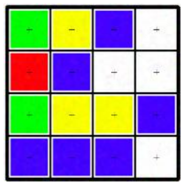
noise_4.png			{'green'} {'yellow'} {'green'} {'green'}
			{'white'} {'yellow'} {'red' } {'red' }
			{'blue' } {'yellow'} {'blue' } {'blue' }
			{'white'} {'blue' } {'red' } {'white'}
				Yes	No notes.

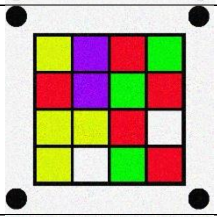
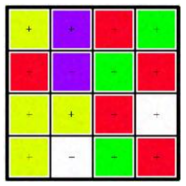
noise_5.png			{'white'} {'red' } {'white'} {'yellow'}
			{'red' } {'yellow'} {'white'} {'yellow'}
			{'green'} {'red' } {'green'} {'red' }
			{'red' } {'yellow'} {'blue' } {'white' }
				Yes	Purple has similar values to blue in LAB color space, that is why is detected as blue.

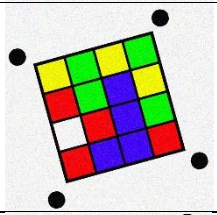
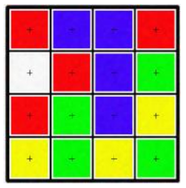
org_1.png			{'yellow'} {'white' } {'blue' } {'red' }
			{'white' } {'green' } {'yellow'} {'white'}
			{'green' } {'blue' } {'red' } {'red' }
			{'yellow'} {'yellow'} {'yellow'} {'blue' }
				Yes	No notes.

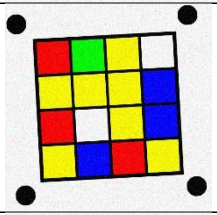
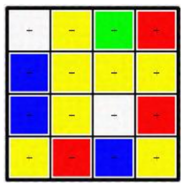
org_2.png			{'blue' } {'yellow'} {'blue' } {'blue' }
			{'white' } {'red' } {'white'} {'yellow'}
			{'green' } {'yellow'} {'green'} {'yellow'}
			{'yellow'} {'blue' } {'green'} {'red' }
				Yes	No notes.

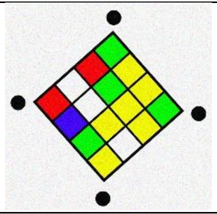
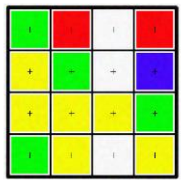
org_3.png			{'green'} {'yellow'} {'red' } {'blue' }
			{'blue' } {'yellow'} {'blue' } {'blue' }
			{'white'} {'blue' } {'green'} {'green' }
			{'white'} {'blue' } {'blue' } {'yellow'}
				Yes	No notes.

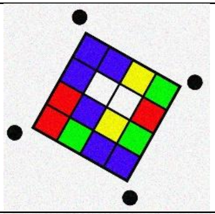
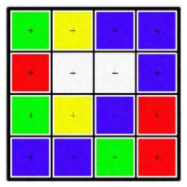
org_4.png			{'green'} {'yellow'} {'blue' } {'white'}
			{'red' } {'blue' } {'white' } {'white'}
			{'green'} {'yellow'} {'yellow'} {'blue' }
			{'blue' } {'blue' } {'blue' } {'white'}
				Yes	No notes.

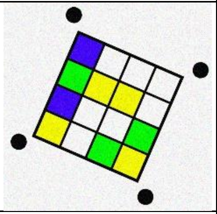
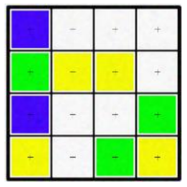
org_5.png			{'yellow'} {'blue' } {'red' } {'green'}
			{'red' } {'blue' } {'green'} {'red' }
			{'yellow'} {'yellow'} {'red' } {'white'}
			{'yellow'} {'white' } {'green'} {'red' }
				Yes	No notes.

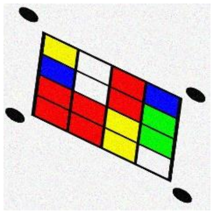
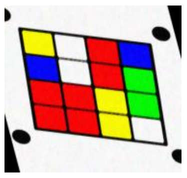
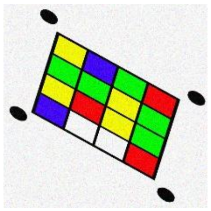
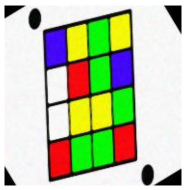
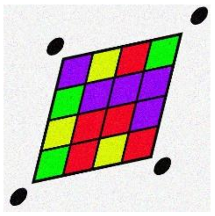
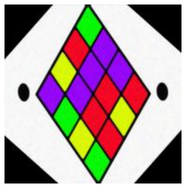
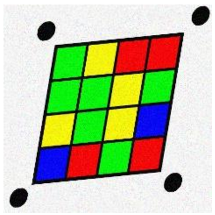
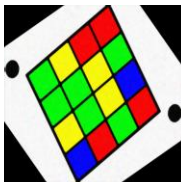
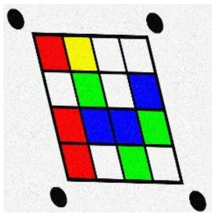
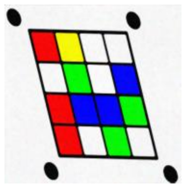
rot_1.png			{'red' } {'blue' } {'blue' } {'red' }
			{'white' } {'red' } {'blue' } {'green' }
			{'red' } {'green'} {'blue' } {'yellow'}
			{'yellow'} {'green'} {'yellow'} {'green' }
				Yes	No notes.

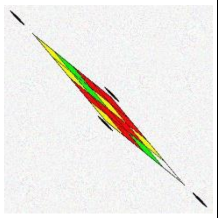
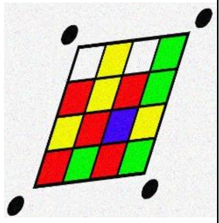
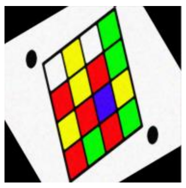
rot_2.png			{'white' } {'yellow'} {'green' } {'red' }
			{'blue' } {'yellow'} {'yellow'} {'yellow'}
			{'blue' } {'yellow'} {'white' } {'red' }
			{'yellow'} {'red' } {'blue' } {'yellow'}
				Yes	No notes.

rot_3.png			{'green' } {'red' } {'white' } {'red' }
			{'yellow'} {'green' } {'white' } {'blue' }
			{'yellow'} {'yellow'} {'yellow'} {'green' }
			{'green' } {'yellow'} {'white' } {'yellow'}
				Yes	No notes.

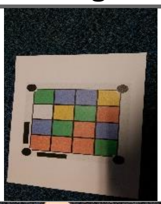
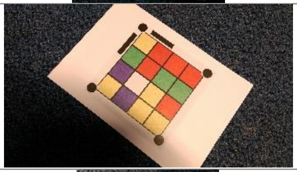
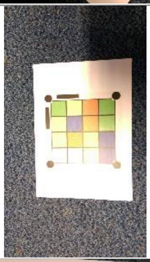
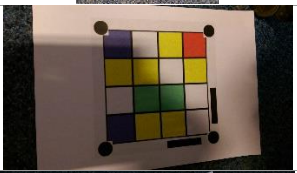
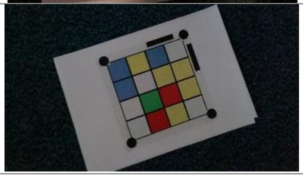
rot_4.png			{'green'} {'yellow'} {'blue' } {'blue'}
			{'red' } {'white' } {'white'} {'blue'}
			{'green'} {'yellow'} {'blue' } {'red' }
			{'blue' } {'blue' } {'green'} {'red' }
				Yes	No notes.

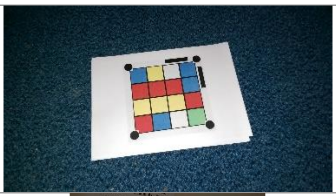
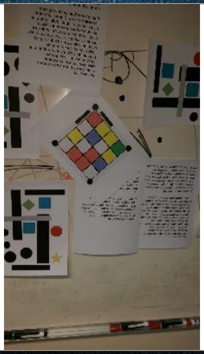
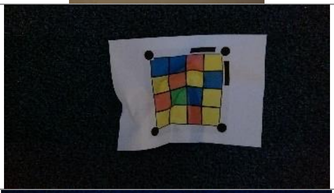
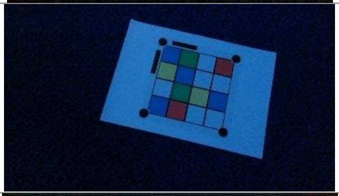
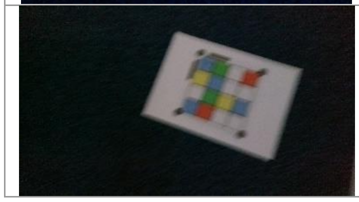
rot_5.png			{'blue' } {'white' } {'white' } {'white' }
			{'green' } {'yellow'} {'yellow'} {'white' }
			{'blue' } {'white' } {'white' } {'green' }
			{'yellow'} {'white' } {'green' } {'yellow'}
				Yes	No notes.

proj_1.png			No matrix.	No	Projections are not processed efficiently.
proj_2.png			No matrix.	No	Projections are not processed efficiently.
proj_3.png			No matrix.	No	Projections are not processed efficiently.
proj_4.png			No matrix.	No	Projections are not processed efficiently.
proj_5.png			No matrix.	No	Projections are not processed efficiently.

proj_6.png		No image.	No matrix.	No	Projections are not processed efficiently.
proj_7.png			No matrix.	No	Projections are not processed efficiently.

Real images

Image	Comments
	The image has extra black lines. Maybe to use it as reference to rotate and project the image. It has a shade. That could interfere to detect the black circle in that corner
	The quality of the image is better. It is a bit tilted but can be processed. If the purple is not initialized as a possible color, blue may be selected as the color of those squares.
	The image is bright, and its bottom section is distorted by the light. This may lead to misclassify white and yellow.
	It has a shade that covers half of the image. Nonetheless, the colors are clear and use LAB as color space could provide the necessary information to detect the right colors.
	The image could be processed in the developed algorithm. However, the additional black lines would not be used.

	The image is lit and projected. However, the colors are clear and could be processed.
	In this case the image is clear and rotated. Nevertheless, there are multiple black shapes in the photo. Therefore, is necessary to implement a strategy that sets a relation between the black circles and the black lines; thus, we work just with the section of the image we need.
	The image surface seems wrinkled. However, is not extreme enough to make it impossible to process.
	The blue light may be a problem. It is necessary to remove the blue light and keep the relevant image information.
	The image is highly blurred. That increases the difficulty of detecting the black circles. A strategy that detects the darker area in the photo could be applied. However, the grid is not clear and could misclassify the color. Moreover, the colors look light so the HSV mask should be extended.

Code

```matlab
% my_colors = findColours1('images\org_1.png');

function res=findColours(filename)

% Load and filter the image
importedImage = loadImage(filename);
% Find the position of the reference black circles
circleCenters = findCircles(importedImage);
% Correct the image to standard view
filteredImageDouble = correctImage(circleCenters, importedImage);

% Get the final matrix
res = getColors(filteredImageDouble);

disp(res)

% % Display the filtered image
% figure;
% imshow(importedImage);
% title('Filtered Image');

end

function filteredImageMedGau = loadImage(filename)
% Read the image
importedImage = imread(filename);

% Remove noise

% Define a larger neighborhood size for more smoothing
neighborhoodSize = [7 7];

% Apply median filter to each color channel separately
redChannelFiltered = medfilt2(importedImage(:, :, 1), neighborhoodSize);
greenChannelFiltered = medfilt2(importedImage(:, :, 2), neighborhoodSize);
blueChannelFiltered = medfilt2(importedImage(:, :, 3), neighborhoodSize);

% Recombine the filtered channels into one RGB image
filteredImage = cat(3, redChannelFiltered, greenChannelFiltered,
blueChannelFiltered);

% Apply Gaussian smoothing
% '1' is the deviation of the Gaussian distribution
filteredImageMedGau = imgaussfilt(filteredImage, 1);

end

function circleCenters = findCircles(image)

% Convert to grayscale
grayImage = rgb2gray(image);

% Apply threshold to get binary image for circle detection
```

```
binaryImage = imbinarize(grayImage);
% Invert to get black circles as white
binaryImage = ~binaryImage;

% Define the radius range, as the images are known is possible to set the
% values
minRadius = 20;
maxRadius = 40;

% Find circles using imfindcircles that uses the Hough Transform to detect
% the circle shapes
[centers, radii, metric] = imfindcircles(binaryImage, [minRadius maxRadius],
'ObjectPolarity','bright', 'Sensitivity',0.92);

% Keep only the four strongest detections based on metric value
numCircles = 4; % number of circles I want to keep
if length(metric) > numCircles
    [~, idx] = maxk(metric, numCircles);
    centers = centers(idx, :);
end

% Sort the black circles coordinates

% 'centers' contains circle centers

% First, sort by y-coordinate to separate top from bottom
[sortedY, orderY] = sort(centers(:, 2), 'ascend');
sortedCentersY = centers(orderY, :);

% Now top circles as the first two entries and the bottom circles as the last two
% Next, sort each of these pairs by their x-coordinate to separate left from right

% For the top two circles
[topSortedX, topOrderX] = sort(sortedCentersY(1:2, 1), 'ascend');
topCircles = sortedCentersY(topOrderX, :);

% For the bottom two circles
[bottomSortedX, bottomOrderX] = sort(sortedCentersY(3:4, 1), 'ascend');
bottomCircles = sortedCentersY(2 + bottomOrderX, :);

% Now concatenate the top and bottom circles to get the final sorted list
% The first row is top-left, second is top-right, third is bottom-left, and fourth is
bottom-right
circleCenters = [topCircles; bottomCircles];

end

function imageRotated = correctImage(circleCoordinates, originalImage)

% Rotate the image

% Circles coordinates
x1 = circleCoordinates(1);
y1 = circleCoordinates(5);
```

```
x2 = circleCoordinates(2);
y2 = circleCoordinates(6);

% Coordinates of the two top circles in the original image that should be
horizontally aligned
circle1 = [x1, y1];
circle2 = [x2, y2];

% Calculate the angle with respect to the horizontal axis
dx = circle2(1) - circle1(1);
dy = circle2(2) - circle1(2);
% The negative sign is to correct the y-axis direction
angle_to_horizontal = atan2(-dy, dx);

% Convert the angle to degrees
angle_degrees = rad2deg(angle_to_horizontal);

% Rotate the image to align it horizontally
imageRotated = imrotate(originalImage, -angle_degrees, 'bilinear', 'crop');

end

function colors = getColors(imageFiltered)

% Detect the grid

% Convert the filtered image from RGB to HSV color space
hsvImage = rgb2hsv(imageFiltered);

% Define the color thresholds for red, blue, green, yellow, and white in HSV space

% for red
redMask1 = (hsvImage(:,:,1) >= 0.0 & hsvImage(:,:,1) <= 0.1) & (hsvImage(:,:,2) >
0.6) & (hsvImage(:,:,3) > 0.5);
redMask2 = (hsvImage(:,:,1) >= 0.9 & hsvImage(:,:,1) <= 1.0) & (hsvImage(:,:,2) >
0.6) & (hsvImage(:,:,3) > 0.5);
redMask = redMask1 | redMask2;

% For blue
blueMask = (hsvImage(:,:,1) >= 0.55 & hsvImage(:,:,1) <= 0.75) & (hsvImage(:,:,2) >
0.6) & (hsvImage(:,:,3) > 0.5);

% For green
greenMask = (hsvImage(:,:,1) > 0.25 & hsvImage(:,:,1) < 0.4) & (hsvImage(:,:,2) >
0.7) & (hsvImage(:,:,3) > 0.5);

%For yellow
yellowMask = (hsvImage(:,:,1) > 0.1 & hsvImage(:,:,1) < 0.22) & (hsvImage(:,:,2) >
0.6) & (hsvImage(:,:,3) > 0.5);

% For white
whiteMask = (hsvImage(:,:,2) <= 0.5) & (hsvImage(:,:,3) >= 0.5);

% For purple
```

```
purpleMask = (hsvImage(:,:,1) >= 0.65 & hsvImage(:,:,1) <= 0.8) & (hsvImage(:,:,2) >
0.4) & (hsvImage(:,:,3) > 0.4);

% Combine masks for multi-color detection
combinedMask = redMask | blueMask | greenMask | yellowMask | whiteMask | purpleMask;

% Perform morphological operations to clean up the mask
% Remove small objects from the foreground based on the structuring element
cleanedCombinedMask = imopen(combinedMask, strel('disk', 5));

% Label connected components on the combined color mask
[labeledImage, numSquares] = bwlabel(cleanedCombinedMask);

% % Display original image
% figure;
% imshow(imageFiltered)
% title("Filtered image")
% hold on;
%
% % Measure properties of image regions
% squareMeasurements = regionprops(labeledImage, 'Area', 'BoundingBox', 'Centroid');
%
% % Loop through all squares and only process those with areas between 5900 and 6100
% for k = 1 : numSquares
%     % Check if the area of the square is within the specified range
%     if squareMeasurements(k).Area > 5800 && squareMeasurements(k).Area < 6200
%         thisBlobsBoundingBox = squareMeasurements(k).BoundingBox;
%         thisBlobCentroid = squareMeasurements(k).Centroid;
%
%         % Draw a rectangle around the detected color
%         rectangle('Position', thisBlobsBoundingBox, 'EdgeColor', 'w', 'LineWidth',
2);
%
%         % Mark the centroids
%         plot(thisBlobCentroid(1), thisBlobCentroid(2), 'k+');
%
%         % % Print the position of the centroid
%         % fprintf('Square #%d (color detected) is at position: x=%.1f, y=%.1f\n',
...
%         %     k, thisBlobCentroid(1), thisBlobCentroid(2));
%     end
% end
%
% hold off;

%% Get the squares

% Measure properties of image regions
squareMeasurementsExtraction = regionprops(labeledImage, 'Area', 'BoundingBox',
'Centroid');

% Initialize a new struct to store filtered square measurements
squareMeasurementsFiltered = struct('Area', {}, 'BoundingBox', {}, 'Centroid', {});

% Loop through all squares and filter by area
```

```
for k = 1 : numSquares
    % Check if the area of the square is within the range of the colored
    % squares
    if squareMeasurementsExtraction(k).Area > 5800 &&
squareMeasurementsExtraction(k).Area < 6200
        % Extract the bounding box of each square
        squareBoundingBox = floor(squareMeasurementsExtraction(k).BoundingBox);

        % Append the current square's measurements to the new struct if they meet the
area criteria
        currentMeasurement = struct(...
            'Area', squareMeasurementsExtraction(k).Area, ...
            'BoundingBox', squareBoundingBox, ...
            'Centroid', squareMeasurementsExtraction(k).Centroid ...
        );
        squareMeasurementsFiltered(end + 1) = currentMeasurement;
    end
end

% Display the filtered measurements
% disp(squareMeasurementsFiltered);

%% Process the squareMeasurementsExtraction

% Extract centroids
centroids = vertcat(squareMeasurementsFiltered.Centroid);

% Truncate the decimal part of each centroid coordinate
truncatedCentroids = floor(centroids);

% Sort the truncated centroids by x and then y
[~, sortedIndices] = sortrows(truncatedCentroids, [1, 2]);

% Reorder the struct array based on sorted indices
sortedSquareMeasurementsFiltered = squareMeasurementsFiltered(sortedIndices);

% % Display the sorted centroids
% for i = 1:length(sortedSquareMeasurementsFiltered)
%     disp(sortedSquareMeasurementsFiltered(i).Centroid);
% end

% Initialize a cell array to store the extracted squares
extractedSquares = {};

% Loop through all squares and extract them from the original image
for k = 1 : length(sortedSquareMeasurementsFiltered)
    % Check if the area of the square is within the specified range

    % Extract the bounding box of each square
    squareBoundingBox = floor(sortedSquareMeasurementsFiltered(k).BoundingBox);

    % Extract the coordinates and dimensions
    x = squareBoundingBox(1);
    y = squareBoundingBox(2);
    width = squareBoundingBox(3);
    height = squareBoundingBox(4);

    % Extract the region from the original image using the bounding box
    extractedSquare = imcrop(imageFiltered, [x, y, width, height]);

    % Store the extracted square in the cell array
    extractedSquares{end + 1} = extractedSquare;

end

%% Detect the colors of the squares

% Initialize a 4x4 cell array to store the colors
colorsMatrix = cell(4, 4);

% Define LAB color centers for matching
colors = struct();

colors.red = [50, 70, 50];         % LAB for Red
colors.green = [50, -50, 50];      % LAB for Green
colors.blue = [50, 20, -50];       % LAB for Blue
colors.yellow = [80, 0, 80];       % LAB for Yellow
colors.white = [100, 0, 0];        % LAB for White
% colors.purple = [55, 50, -40];  % Added range for purple

% Loop through the squares and compare the mean LAB value
for k = 1:length(extractedSquares)

    % Get the current square image
    currentSquare = extractedSquares{k};
    % Convert the current square from RGB to LAB
    labSquare = rgb2lab(currentSquare);
    % Calculate the mean LAB color of the square
    meanLAB = squeeze(mean(mean(labSquare, 1), 2));

    % Find the closest color
    detectedColorName = '';
    smallestDistance = inf;
    colorNames = fieldnames(colors);
    for c = 1:length(colorNames)
        colorName = colorNames{c};
        colorValue = colors.(colorName);
        distance = sqrt((meanLAB(1) - colorValue(1))^2 + (meanLAB(2) -
colorValue(2))^2 + (meanLAB(3) - colorValue(3))^2);

        if distance < smallestDistance
            smallestDistance = distance;
            detectedColorName = colorName;
        end
    end

    % Calculate the correct row and column in the 4x4 matrix for column-major order
    % Adjusting index to start from 0 for proper calculation
    index = k - 1;
    % Calculate column by dividing the adjusted index by 4
    col = ceil((index + 1) / 4);
    % Calculate row using modulus to cycle through 1 to 4
    row = mod(index, 4) + 1;
    % Store the detected color in the result matrix
    colorsMatrix{row, col} = detectedColorName;
end

colors = colorsMatrix;

end
```