

521H3: Image Processing

Candidate Number: 277260

Name and Surname: Mert OLCAMAN

May, 2024

Contents

Approaches.....	2
1. Filtering:.....	2
2. To Find All Circle Centers with Regions	2
3. Specifying Fixed Centre Points.....	3
4. How to Understand the Type of Image and How to Correct.....	3
a. Projection-required Images (proj)	3
b. Rotation-required Images (rot).....	3
c. Other Types of Images (noise, org)	4
5. How to Find Each Colour	4
How to Work	5
Real Images and Discussion.....	6
Appendix	9
Result Table	9
Accuracy Table for Each Image.....	11
Functions with Explanations.....	12
Functions with Codes	13
getColours.m	13
filtering_image.m.....	15
find_circle_centers.m	15
find_circle_or_square.m	17
find_slope.m.....	18
findBestOrientation.m	18
findColours.m	19
loadImage.m.....	21
margins_calculator.m.....	22
rotatedPredictionAndAccuracy.m	22
refImgCircleCenters.m	23
find_each_accuracy.mlx.....	23

Approaches

1. Filtering:

order of filters: median filter → average filter → median filter → Gaussian filter → sharpening filter

The median filter takes the value at the middle and the median value is less affected by outliers. It's useful when the picture is noisy.

Gaussian filter blurs the image. So, the noise gets decreased.

The sharpening filter makes the image sharpened. It was required after the Gaussian filter.

2. To Find All Circle Centers with Regions

If I had taken 2 random circle centres, the orientation degree of the image would have been found wrong because of the location of the circles. I needed to take 2-neighbour centres into account while calculating the degree.

On the other hand, it was a must to know in projection process to match the correct circular objects.

After providing the stats of the labelled image for each object (for example, adding the value of 1 to all pixels for the object labelled as 1), the area values are found. I checked the values for areas and saw that more than half of the image is covered by a square. So, I took a threshold of 0.5 of the whole image area. If that area value is smaller than this area, it means that each object is circular (circle or ellipse). Then, I took all of the indexes of circles and excluded the square from the image.

Again, I labelled the image with a new number to be sure that it is ordered from 1 to 4 to iterate in the loop accurately. If any of the circles had labelled as 5 in the first section, I would have got an error. After that, I divided the image into 4-imaginary-quarter pieces as a 2D-coordinat space, which can be seen in Figure 1.

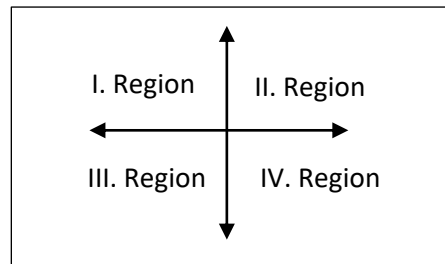


Figure 1: Regions in 2D-coordinate space

Half of the image from both vertical and horizontal was the origin according to the coordinate system. That's why I divided the size of the image by 2. Then, I found the row and column indexes of each circle and took the average value to find the centre. After this point, I added conditions to find where each region is located in.

I could get the circle centres in an order. The first row was located in the first region, the second one in the second region and so on.

3. Specifying Fixed Centre Points

I needed to find the locations of each circle in a well-oriented picture. I used the image named “org_2.png” to find the reference points.

4. How to Understand the Type of Image and How to Correct

In the beginning, I converted the image into binary form. In that way, I could detect the edges and shapes in the picture.

a. Projection-required Images (proj)

In such pictures, there was no circular object. All were elliptical, so the eccentricity had a different value than other types. I specified a threshold for eccentricity as 0.4, after checking the values in each image. The average eccentricity value was higher than 0.6. However, I wanted to make sure.

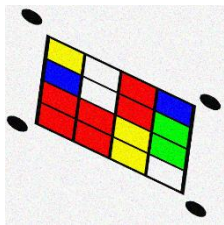


Figure 2: Distorted Image

As it can be seen in Figure 2, the circles look like ellipse, while they look like nearly exact circle in Figure 3.

After that point, I found the all ellipse centers with regions as **described**. They were set as moving point.

The **fixed points** were taken from the reference image. Then, the image was projected by matching each moving point with equivalent fixed point.

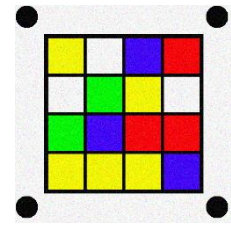


Figure 3: Corrected Image

b. Rotation-required Images (rot)

These types of pictures had different orientations like in Figure 4, which had slope.

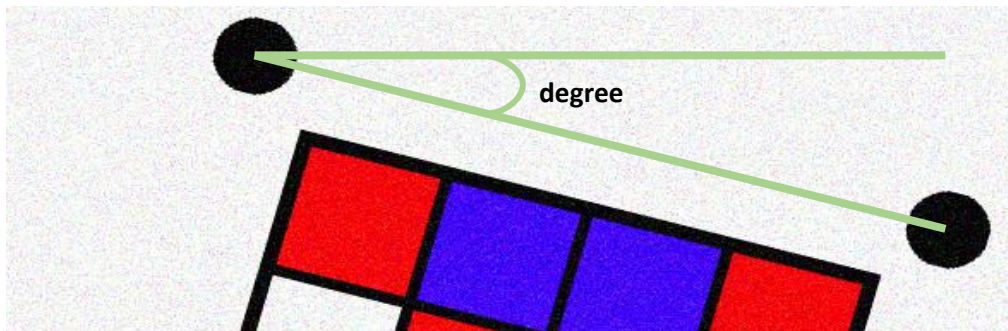


Figure 4: Degree Representation over the Image

To find the degree, the coordinates of each circle center was found at the beginning, with region.

Then, I took the first 2 neighbours into account in Equation 1.

$$\text{degree} = \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

Equation 1: Degree Calculation

c. Other Types of Images (noise, org)

If the degree found was 90 degrees, the image was well-oriented.

5. How to Find Each Colour

The row and column indexes of the square object were found. Then, a side length was calculated and divided by 4 to find approximate length of each sub-square.

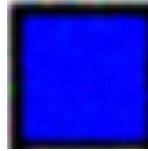


Figure 5: A representation of Sub-square

After taking the relevant indexes, I got Figure 5. It didn't give me the colour of blue when I got the average of median values of the image due to the black border. Then, I decided to find exactly blue part of the image.

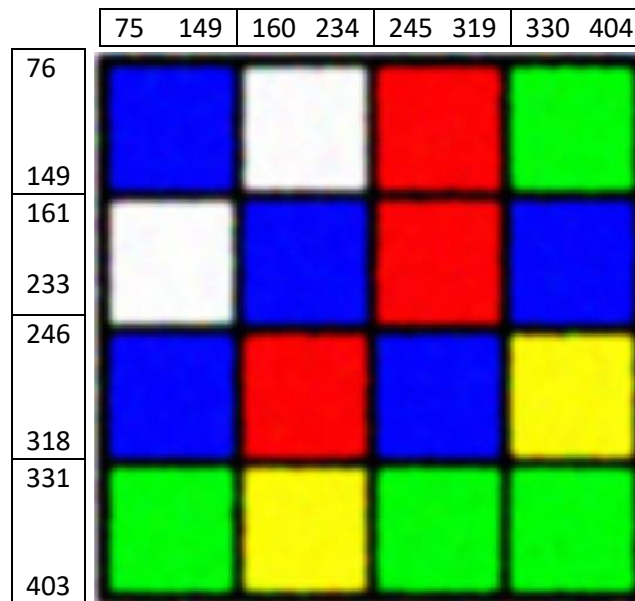


Figure 6: Row and Column Indexes of Square Element

If the first blue square is taken as an example in Figure 6, the column index for the right border ends at 149, while the starting index of white square located in the right-hand side of the blue square is 160. The missing part belongs to the border line (black grids).

I gave offset from each side to sub-square.

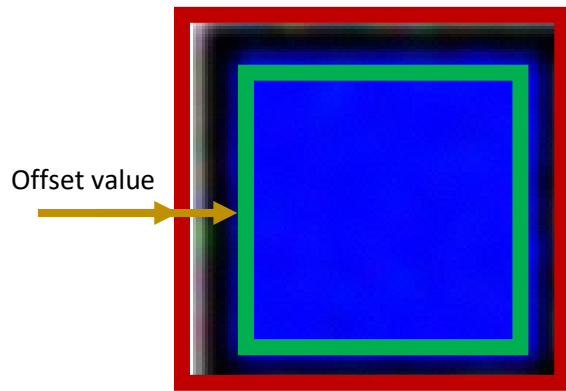


Figure 7: Before Offset

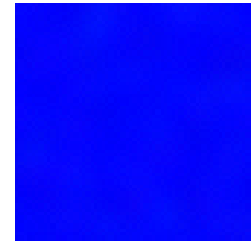


Figure 8: After Offset

Hence, I could find the correct colour value by taking the median of the colour values. Even though it seemed there was no noise, I wanted to take the median (less affected value by outliers) to make it accurately.

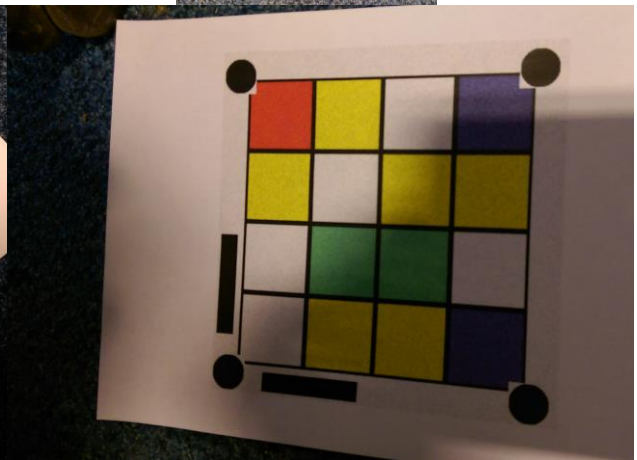
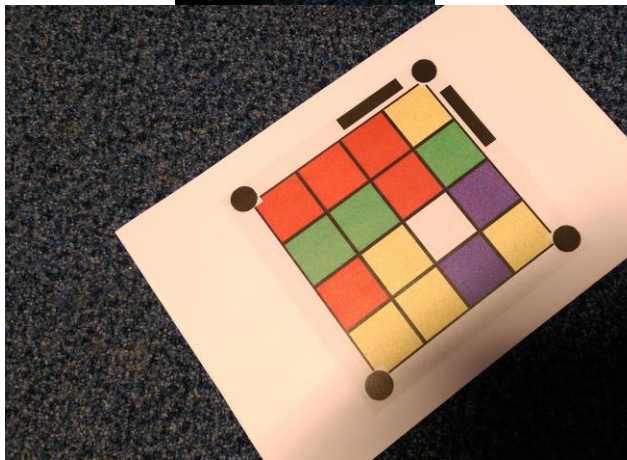
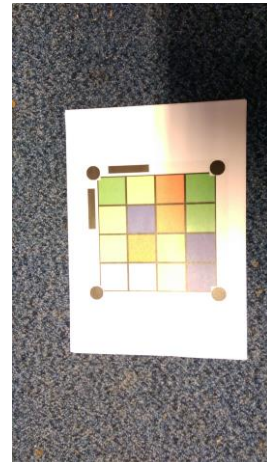
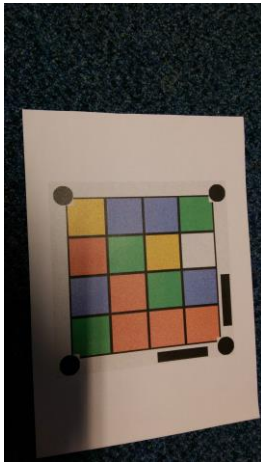
How to Work

function: findColours (image_location)	
Inputs: image_location	Outputs: the cell of predicted colours a figure where distorted and corrected images are

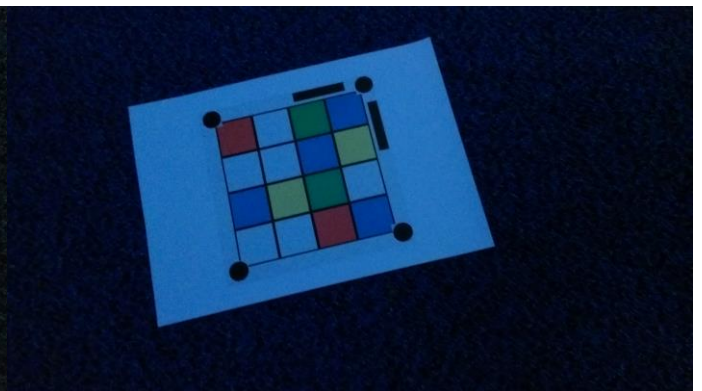
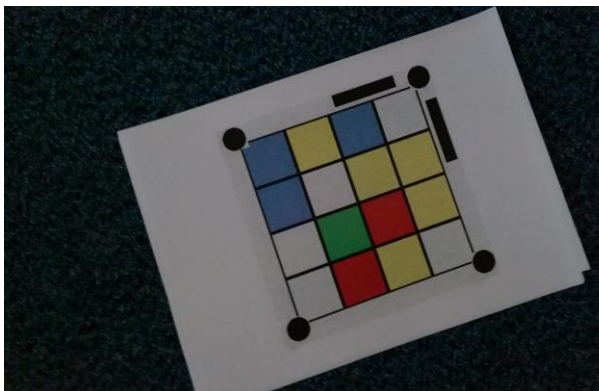
(Note: Addition to that, if the file named “**find_each_accuracy.mlx**” is run, the accuracy table for all, and the last image with corrected version can be seen.)

Real Images and Discussion

- **Shadow or Light Reflection:** it changes the colour value



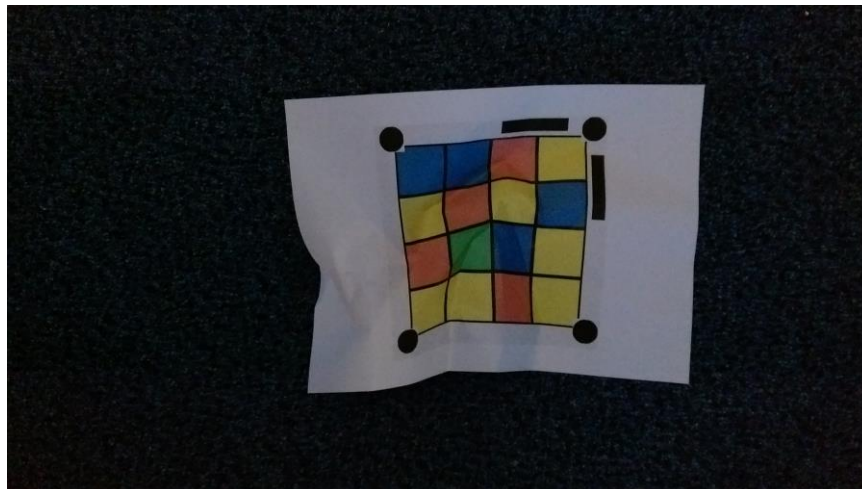
- **Insufficient or Wrong Brightness:** the colours are predicted incorrect



- **Containing Other Objects:** since there are other objects around the image, the model can detect other things rather than the actual image



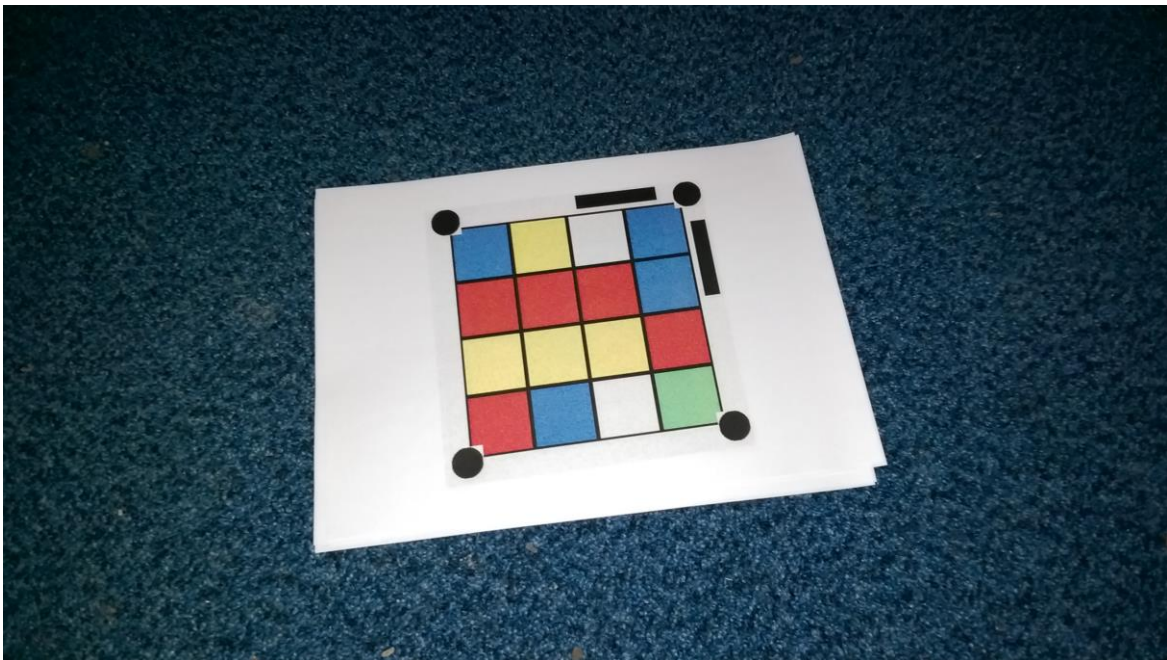
- **Crumpled Paper:** the colour values are distorted, and the element which checks colours over sub-squares detects other colours because of not being able to follow parallel to the lines




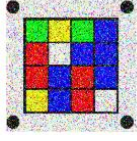
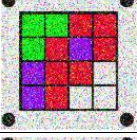
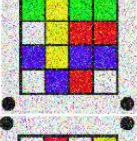
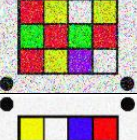
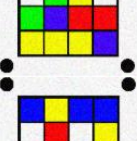
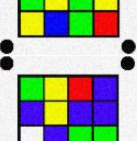
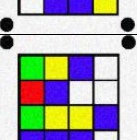
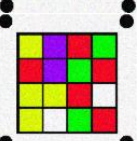

- **Blurred Image:** Normally, there is no correction for blurred images in the filter function. So, any objects cannot be detected.



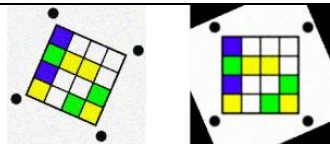
- **Printing Quality:** The density of some colours (like green) can cause misprediction.



Appendix Result Table

Filename	Image	Output	Success	Notes
noise_1.png		prediction = 4x4 cell <div> <div>'blue'</div> <div>'white'</div> <div>'red'</div> <div>'green'</div> </div> <div> <div>'white'</div> <div>'blue'</div> <div>'red'</div> <div>'blue'</div> </div> <div> <div>'blue'</div> <div>'red'</div> <div>'blue'</div> <div>'yellow'</div> </div> <div> <div>'green'</div> <div>'yellow'</div> <div>'green'</div> <div>'green'</div> </div>	Yes	
noise_2.png		prediction = 4x4 cell <div> <div>'green'</div> <div>'yellow'</div> <div>'green'</div> <div>'blue'</div> </div> <div> <div>'red'</div> <div>'white'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'red'</div> <div>'blue'</div> <div>'red'</div> <div>'blue'</div> </div> <div> <div>'yellow'</div> <div>'blue'</div> <div>'red'</div> <div>'white'</div> </div>	Yes	
noise_3.png		prediction = 4x4 cell <div> <div>'green'</div> <div>'green'</div> <div>'red'</div> <div>'red'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'blue'</div> <div>'red'</div> </div> <div> <div>'blue'</div> <div>'red'</div> <div>'red'</div> <div>'white'</div> </div> <div> <div>'blue'</div> <div>'red'</div> <div>'white'</div> <div>'white'</div> </div>	Yes	
noise_4.png		prediction = 4x4 cell <div> <div>'green'</div> <div>'yellow'</div> <div>'green'</div> <div>'green'</div> </div> <div> <div>'white'</div> <div>'yellow'</div> <div>'red'</div> <div>'red'</div> </div> <div> <div>'blue'</div> <div>'yellow'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'white'</div> <div>'blue'</div> <div>'red'</div> <div>'white'</div> </div>	Yes	
noise_5.png		prediction = 4x4 cell <div> <div>'white'</div> <div>'red'</div> <div>'white'</div> <div>'yellow'</div> </div> <div> <div>'red'</div> <div>'yellow'</div> <div>'white'</div> <div>'yellow'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'green'</div> <div>'red'</div> </div> <div> <div>'red'</div> <div>'yellow'</div> <div>'blue'</div> <div>'white'</div> </div>	Yes	
org_1.png		prediction = 4x4 cell <div> <div>'yellow'</div> <div>'white'</div> <div>'blue'</div> <div>'red'</div> </div> <div> <div>'white'</div> <div>'green'</div> <div>'yellow'</div> <div>'white'</div> </div> <div> <div>'green'</div> <div>'blue'</div> <div>'red'</div> <div>'red'</div> </div> <div> <div>'yellow'</div> <div>'yellow'</div> <div>'yellow'</div> <div>'blue'</div> </div>	Yes	
org_2.png		prediction = 4x4 cell <div> <div>'blue'</div> <div>'yellow'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'white'</div> <div>'red'</div> <div>'white'</div> <div>'yellow'</div> </div> <div> <div>'green'</div> <div>'yellow'</div> <div>'green'</div> <div>'yellow'</div> </div> <div> <div>'yellow'</div> <div>'blue'</div> <div>'green'</div> <div>'red'</div> </div>	Yes	
org_3.png		prediction = 4x4 cell <div> <div>'green'</div> <div>'yellow'</div> <div>'red'</div> <div>'blue'</div> </div> <div> <div>'blue'</div> <div>'yellow'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'white'</div> <div>'blue'</div> <div>'green'</div> <div>'green'</div> </div> <div> <div>'white'</div> <div>'blue'</div> <div>'blue'</div> <div>'yellow'</div> </div>	Yes	
org_4.png		prediction = 4x4 cell <div> <div>'green'</div> <div>'yellow'</div> <div>'blue'</div> <div>'white'</div> </div> <div> <div>'red'</div> <div>'blue'</div> <div>'white'</div> <div>'white'</div> </div> <div> <div>'green'</div> <div>'yellow'</div> <div>'yellow'</div> <div>'blue'</div> </div> <div> <div>'blue'</div> <div>'blue'</div> <div>'blue'</div> <div>'white'</div> </div>	Yes	
org_5.png		prediction = 4x4 cell <div> <div>'yellow'</div> <div>'blue'</div> <div>'red'</div> <div>'green'</div> </div> <div> <div>'red'</div> <div>'blue'</div> <div>'green'</div> <div>'red'</div> </div> <div> <div>'yellow'</div> <div>'yellow'</div> <div>'red'</div> <div>'white'</div> </div> <div> <div>'yellow'</div> <div>'white'</div> <div>'green'</div> <div>'red'</div> </div>	Yes	

proj_1.png			prediction = 4x4 cell 'yellow' 'white' 'red' 'blue' 'blue' 'white' 'red' 'green' 'red' 'red' 'yellow' 'green' 'red' 'red' 'yellow' 'white'	Yes	
proj_2.png			prediction = 4x4 cell 'yellow' 'blue' 'green' 'red' 'green' 'green' 'UNDEFINED' 'UNDEFINED' 'yellow' 'UNDEFINED' 'green' 'white' 'blue' 'white' 'UNDEFINED' 'white'	No	It doesn't work since the circle centre on the bottom right-hand corner is not found properly
proj_3.png			prediction = 4x4 cell 'blue' 'yellow' 'red' 'green' 'green' 'blue' 'blue' 'blue' 'yellow' 'red' 'red' 'blue' 'green' 'red' 'yellow' 'red'	Yes	
proj_4.png			prediction = 4x4 cell 'green' 'yellow' 'red' 'red' 'green' 'green' 'yellow' 'green' 'yellow' 'green' 'yellow' 'blue' 'blue' 'red' 'green' 'red'	Yes(?)	My function gives accurate orientation, but it's reversed according to the answer matrix, so it seems that it's wrong.
proj_5.png			prediction = 4x4 cell 'red' 'yellow' 'white' 'white' 'white' 'green' 'white' 'blue' 'red' 'blue' 'blue' 'green' 'red' 'white' 'green' 'white'	Yes	
proj_6.png			prediction = 4x4 cell 'green' 'green' 'yellow' 'yellow' 'green' 'green' 'yellow' 'red' 'red' 'red' 'red' 'red' 'red' 'red' 'red' 'red'	No	It is not correctly oriented, because of being laid
proj_7.png			prediction = 4x4 cell 'white' 'yellow' 'white' 'green' 'red' 'yellow' 'red' 'green' 'yellow' 'red' 'blue' 'yellow' 'red' 'green' 'red' 'green'	Yes	
rot_1.png			prediction = 4x4 cell 'red' 'blue' 'blue' 'red' 'white' 'red' 'blue' 'green' 'red' 'green' 'blue' 'yellow' 'yellow' 'green' 'yellow' 'green'	Yes	
rot_2.png			prediction = 4x4 cell 'white' 'yellow' 'green' 'red' 'blue' 'yellow' 'yellow' 'yellow' 'blue' 'yellow' 'white' 'red' 'yellow' 'red' 'blue' 'yellow'	Yes	
rot_3.png			prediction = 4x4 cell 'green' 'red' 'white' 'red' 'yellow' 'green' 'white' 'blue' 'yellow' 'yellow' 'yellow' 'green' 'green' 'yellow' 'white' 'yellow'	Yes	
rot_4.png			prediction = 4x4 cell 'red' 'green' 'blue' 'blue' 'red' 'blue' 'yellow' 'green' 'blue' 'white' 'white' 'red' 'blue' 'blue' 'yellow' 'green'	Yes	

rot_5.png		<pre> prediction = 4x4 cell 'blue' 'white' 'white' 'white' 'green' 'yellow' 'yellow' 'white' 'blue' 'white' 'white' 'green' 'yellow' 'white' 'green' 'yellow' </pre>	Yes	
-----------	---	--	-----	--

Accuracy Table for Each Image

As it can be seen from the table, accuracy for each image is in the second column, while the first column represents the image name. Overall accuracy is found as **0.93** for my functions. However, the answer matrix is reversed for “**proj_4.png**”, so the actual accuracy is **0.957%**.

	1	2
1	'noise_1.png'	1
2	'noise_2.png'	1
3	'noise_3.png'	1
4	'noise_4.png'	1
5	'noise_5.png'	1
6	'org_1.png'	1
7	'org_2.png'	1
8	'org_3.png'	1
9	'org_4.png'	1
10	'org_5.png'	1
11	'proj_1.png'	1
12	'proj_2.png'	0.5625
13	'proj_3.png'	1
14	'proj_4.png'	0.3750
15	'proj_5.png'	1
16	'proj_6.png'	0.5000
17	'proj_7.png'	1
18	'rot_1.png'	1
19	'rot_2.png'	1
20	'rot_3.png'	1
21	'rot_4.png'	1
22	'rot_5.png'	1

Functions with Explanations

getColours.m	<p>Takes the inputs and finds the borders of the element which checks colours in each sub-square.</p> <p>While finding borders of the element adds margin values from each border, it can be thought as an offset</p> <p>The reason why of it is that the border colour is black and it affects the accuracy of the value of the median colour</p> <p>After borders are found, the check element goes over each sub-square and finds the median value and gives it as a number</p> <p>The value is divided by 255 to make it binary</p> <p>Then the value is sent to if conditions to find the colour, it is given as output in a cell</p>
filtering_image.m	<p>Takes the rgb image and apply some filters;</p> <p>3d-median filter [5 5 1], average filter (6), 3d-median filter [5 5 1], gaussian filter (sigma=2), sharpening filter [0 -1 0; -1 5 -1; 0 -1 0]</p>
find_circle_centers.m	<p>Takes the inputs and gives each coordinate with the order of the region:</p> <p>I.region: Top-Left, II.region: Top-Right, III.region: Bottom-Left, IV.region: Bottom-Right</p>
find_circle_or_square.m	<p>Takes the input as binary image and finds the desired shape</p> <p>Then, gives only the desired shapes in the binary image</p>
find_slope.m	<p>Takes the circle centers, Gives the slope in the image in degree for rotation</p>
findBestOrientation.m	<p>Finds the best orientation, which gives the highest accuracy</p> <p>Even if the image corrected with a different position than the actual one, it gives the accuracy for the algorithm and how many time it is required to be rotated in 90 degree</p>
findColours.m	<p>Takes image location, gives the cell for predicted colours</p>
loadImage.m	<p>Takes the location of the image</p> <p>Reads, apply filters, and converts it into binary</p>
margins_calculator.m	<p>While checking each sub-square element in the main square, the borders of each is seen as black, so it affects the rgb value for each sub-square. To eliminate it, offset is taken into account, this function calculates these values</p>
rotatedPredictionAndAccuracy.m	<p>takes the predicted cell and correct oriantation, gives the correct rotated prediction cell</p>
refImgCircleCenters.m	<p>(*Note: it can be used only in case of using a reference image from a folder directly)</p> <p>Takes the reference image location, gives each circle's center in the image</p> <p>[When it is desired to be added, the steps below must be done:</p> <p>1- uncomment this in "find_each_accuracy"</p> <p>%%referenceImage='images/org_2.png'; %reference image for projection</p> <p>2- uncomment this in "findColours"</p> <p>%%[circle_centers_ref,img_ref]=refImgCircleCenters(refere nce_image_location);</p> <p>3-add parameter named "referenceImage" as an input to the function of "findColours"</p>
find_each_accuracy.mlx	<p>The live script to see each accuracy one by one in a table and see the last image</p>

Functions with Codes

getColours.m

```
function [prediction] =  
getColours(rotated_img,outer_square_size,side_colour_check_element,row_start,col_start,  
margin_left,margin_right,margin_bottom)  
  
%                               INPUTS  
%  
%rotated_img: correct image  
%outer_square_size: bi"gger square's size (4x4 square)  
%side_colour_check_element: the side size of the sub-squares  
%row_start: the row index where the bigger square starts  
%col_start: the column index where the bigger square starts  
%margin_left: the offset value from the left border  
%margin_right: the offset value from the right border  
%margin_bottom: the offset value from the bottom border  
%  
%  
%                               OUTPUTS  
%>>  
%prediction =  
%  
%   4x4 cell array  
%  
%   {'predicted_colour'}   {'predicted_colour'}   {'predicted_colour'}  
%   {'predicted_colour'}  
%   {'predicted_colour'}   {'predicted_colour'}   {'predicted_colour'}  
%   {'predicted_colour'}  
%   {'predicted_colour'}   {'predicted_colour'}   {'predicted_colour'}  
%   {'predicted_colour'}  
%   {'predicted_colour'}   {'predicted_colour'}   {'predicted_colour'}  
%   {'predicted_colour'}  
%  
%                               EXPLANATION  
%  
% takes the inputs and finds the borders of the element which checks colours in each  
% sub-square  
% while finding borders of the element adds margin values from each border, it can be  
% thought as an offset  
% The reason why of it is that the border colour is black and it affects the accuracy  
% of the value of the median colour  
% After borders are found, the check element goes over each sub-square and finds the  
% median value and gives it as a number  
% The value is divided by 255 to make it binary  
% Then the value is sent to if conditions to find the colour  
% It is given as output in a cell  
  
repeat=3; %there are 4 sub-squares in each row and column, so it is used in loop  
index=0;  
  
prediction=cell(repeat+1); %4x4 prediction cell is created  
for row_k=0:repeat  
  
    row_plus=row_k*side_colour_check_element; %check element's side size is  
    provided to the function, it's the element which checks each sub-square  
    row_idx=row_start+row_plus; %new starting row index is found  
  
    for col_k=0:repeat
```

```

        col_plus=col_k*side_colour_check_element; %like in row, the value for the
column incrementation is found
        col_idx=col_start+col_plus; %new starting column index is found

        index= index + 1;

        %margin in each can be thought as an offset from border
        %since the border colour is black, it affects the value of
        %the colour, so it can be found more accurately like this
        row_bottom=row_idx+side_colour_check_element-margin_bottom; %it's the
bottom border of the element which checks the sub-squares
        col_left=col_idx+margin_left; %left border of the element, the margin is
added because of being on the left-hand side
        col_right=col_idx+side_colour_check_element-margin_right; %left border of
the element, the margin is subtracted because of being on the right-hand side

        red_value=rotated_img(row_idx:row_bottom,col_left:col_right,1); %1st
dimension is associated with red
        green_value=rotated_img(row_idx:row_bottom,col_left:col_right,2); %2nd
dimension is associated with red
        blue_value=rotated_img(row_idx:row_bottom,col_left:col_right,3); %3rd
dimension is associated with red

        %to get the most frequent value in the square element which
        %checks each sub-square
        red_mean=median(median(red_value));
        green_mean=median(median(green_value));
        blue_mean=median(median(blue_value));

        %to get binary value for each colour
        binary_red=red_mean/255;
        binary_green=green_mean/255;
        binary_blue=blue_mean/255;

        %conditions for specifying colours in binary values
        if binary_red==0 && binary_green==0 && binary_blue==1
            colour='blue';
        elseif binary_red==1 && binary_green==1 && binary_blue==1
            colour='white';
        elseif binary_red==1 && binary_green==0 && binary_blue==0
            colour='red';
        elseif binary_red==0 && binary_green==1 && binary_blue==0
            colour='green';
        elseif binary_red==1 && binary_green==1 && binary_blue==0
            colour='yellow';
        elseif binary_red==1 && binary_green==0 && binary_blue==1
            colour='blue'; %even though they seem like purple in the image, they
are blue normally because of being affected by red light
        else
            colour='UNDEFINED'; %for other values, I added UNDEFINED colour name so
that It can be understandable that the binary value for the colour is not defined
        end

        prediction{row_k+1,col_k+1}=colour;

    end
end
end

```


filtering_image.m

```
function [B] = filtering_image(img)

%           INPUTS
%img= original rgb image
%
%           OUTPUTS
%B= filtered rgb image
%
%           EXPLANATION
%takes the rgb image and apply some filters;
%
%3d-median filter [5 5 1]
%average filter (6)
%3d-median filter [5 5 1]
%gaussian filter (sigma=2)
%sharpening filter [0 -1 0; -1 5 -1; 0 -1 0]

%-----FILTERING-----%
%rgb median filter
filtered=medfilt3(img,[5 5 1]);

%6 by 6 averaging filter
avg=fspecial('average',6);
filteredIMG=imfilter(filtered, avg);

%rgb median filter
filtered=medfilt3(filteredIMG,[5 5 1]);

%gaussian filter sigma=2
B = imgaussfilt(filtered,2);

%%Explanation: sharpening is necessary, otherwise some circles may not be detected
%%because of being close to the circle at the center and being blurred
%%after filtering

%sharpening filter matrix
sharpening_filter=[0 -1 0; -1 5 -1; 0 -1 0];
%applying sharpening filter
B=imfilter(B,sharpening_filter);

end
```

find_circle_centers.m

```
function [circle_centers] = find_circle_centers(img,cc,stats)
%FIND CENTER OF EACH CIRCLE BY DIVIDING THE PICTURE INTO REGIONS
%
%           INPUTS
%img: image
%cc: connected components in binary image
%stats: measure properties of image regions
%
%           OUTPUTS
%circle_centers: the matrix which consists of the center of each circle in
%the image
% circle_centers= [ (c1x, c1y) => area1
%                  (c2x, c2y) => area2
%                  (c3x, c3y) => area3
%                  (c4x, c4y)] => area4
%
%           EXPLANATION
```

```

%Takes the inputs and gives each coordinate with the order of the region
%I.region: Top-Left
%II.region: Top-Right
%III.region: Bottom-Left
%IV.region: Bottom-Right
%
%
%      ^
%      |
%  I.region  |  II.region
%      |
%      |
% <----->
%      |
%  III.region |  IV.region
%      |
%      |
%      v
%
%-----latest added start-----%
%-----finding circles in the image-----%
%finding total area of the shapes in the image
total_area=sum([stats.Area]);
%finding the half of the area in the image to use as a threshold
%%exp: the square at the center covers more pixels than the circles around
%%I took threshold 50% of them, it could be taken even less
half_area=total_area*0.5;

%taking the index of the shapes which covers less than threshold
%in other words, these indexes belong to circles
idx = find([stats.Area] < half_area);
%taking only the shapes suitable for the condition related to threshold
BW2 = ismember(labelmatrix(cc),idx);
%displaying circles
%labelling and numbering each circle
[L,num]=bwlablel(BW2);

%-----latest added end-----%

%-----finding area of each circle-----%
%finding the size of the picture in terms of row and column
size_image=size(img);
row_size=size_image(1);
column_size=size_image(2);
%thresholds can be thought as the x and y axis of a coordinate system
row_threshold=row_size/2;
col_threshold=column_size/2;

%At this point, I divided the picture into 4 imaginary-quarter area
%In other words, the center of the image is the origin of the
%coordinate system, but the row and column numbers is not changed

%I.region: Top-Left
%II.region: Top-Right
%III.region: Bottom-Left
%IV.region: Bottom-Right
%
%
%      ^
%      |
%  I.region  |  II.region
%      |
%      |
%

```

```

% <----->
%
%   III.region   |   IV.region
%   |            |
%   |            |
%   |            |
%   v
%

%an empty matrix to store center points of each circle
circle_centers=zeros(4,2);
%loop to iterate over circles
for i=1:num
    %finding ith elements' row and column values
    [row_idx, col_idx]=find(L==i);
    %taking the mean of each area gives approximately center of circle
    %also rounding is required to get exact pixel
    row_idx=round(mean(row_idx));
    col_idx=round(mean(col_idx));

    if row_idx<row_threshold && col_idx<col_threshold
        region=1;
        circle_centers(1,1)=row_idx;
        circle_centers(1,2)=col_idx;
    elseif row_idx<row_threshold && col_idx>col_threshold
        region=2;
        circle_centers(2,1)=row_idx;
        circle_centers(2,2)=col_idx;
    elseif row_idx>row_threshold && col_idx<col_threshold
        region=3;
        circle_centers(3,1)=row_idx;
        circle_centers(3,2)=col_idx;
    elseif row_idx>row_threshold && col_idx>col_threshold
        region=4;
        circle_centers(4,1)=row_idx;
        circle_centers(4,2)=col_idx;
    end
end
end

```

find_circle_or_square.m

```

function [L,num,BW2] = find_circle_or_square(BW,par)

%           INPUTS
%BW: binary image
%par: "circle" or "square"
%
%           OUTPUTS
%L: Labelled-connected components in 2-D binary image
%num: The number of connected objects found in BW2
%BW2: The binary image of circle or square
%
%           EXPLANATION
%Takes the input as binary image and finds the desired shape
%Then, gives only the desired shapes in the binary image

%finding connected components
cc = bwconncomp(BW);
%taking area and eccentricity values of each component
stats = regionprops(cc,"Area","Eccentricity");

```

```

%-----finding circles in the image-----%
%finding total area of the shapes in the image
total_area=sum([stats.Area]);
%finding the half of the area in the image to use as a threshold
%%exp: the square at the center covers more pixels than the circles around
%%I took threshold 50% of them, it could be taken even less
half_area=total_area*0.5;

if strcmp(par, 'circle')
    %taking the index of the shapes which covers less than threshold
    %in other words, these indexes belong to circles
    idx = find([stats.Area] < half_area);
elseif strcmp(par, 'square')
    idx = find([stats.Area] >= half_area);
else
    error('Invalid parameter. Please use ''circle'' or ''square''.');
end

%taking only the shapes suitable for the condition related to threshold
BW2 = ismember(labelmatrix(cc),idx);
%labelling and numbering each circle
[L,num]=bwlabel(BW2);
end

```

find_slope.m

```

function [degree] = find_slope(circle_centers)
%FIND SLOPE THROUGH CIRCLE CENTERS
%
%           INPUTS
%circle_centers: the matrix of circle centers ordered by region
%
%           OUTPUTS
%degree: the orientation degree
%
%           EXPLANATION
%Takes the circle centers
%Gives the slope in the image in degree for rotation

%-----finding slope-----%
y2=circle_centers(2,2);
y1=circle_centers(1,2);

x2=circle_centers(2,1);
x1=circle_centers(1,1);
slope=(y2-y1)/(x2-x1);
degree=atand(slope);
end

```

findBestOrientation.m

```

function [value,degree] = findBestOrientation(prediction,check_matrix)

%           INPUTS
%prediction: predicted colours' cell
%check_matrix: the actual colours' cell
%
%           OUTPUTS
%value: the best accuracy value
%degree: how many times it is required to be rotated in 90 degree
%
%           EXPLANATION

```

```

%Finds the best orientation, which gives the highest accuracy
%Even if the image corrected with a different position than the actual one,
%it gives the accuracy for the algorithm and how many time it is required
%to be rotated in 90 degree

%an empty vector to storage the accuracies based on 90-degree rotation each
degreeList=zeros(1,4);

for i=[0,1,2,3] %degrees to rotate, each indicates how many times of 90
    rotated_prediction=rot90(prediction,i); %rotate the prediction matrix
    correct=0; %reset the value in each loop to start from 0 for each rotation
    total=0; %reset the value in each loop to start from 0 for each rotation
    for row= 1:size(prediction, 1)
        for column= 1:size(prediction, 2)
            if isequal(rotated_prediction(row,column),check_matrix.res{row,column})
                %check the values in relevant row and index for prediction and other matrix are the
                same
                correct=correct+1; %if they are the same add 1
            end
            total=total+1; %adds 1 to find the total element number
        end
    end
    accuracy=correct/total; %finds accuracy
    degreeList(1,i+1)=accuracy; %adds the accuracy to the list created at the beginning

end

[value,position]=max(degreeList); %finds the maximum accuracy value and position of it
to get how many degrees it is required to be rotated for the best match with the answer
degree=position-1; %for example, position=3 means 2 times 90 degrees rotation is
required
end

```

findColours.m

```

function [prediction] = findColours(image_location)
%
%      INPUTS
%image_location: image's location
%
%      OUTPUTS
%prediction: the cell which consist of predicted colours
%
%
%      EXPLANATION
%Takes image location and actual colours
%Gives corrected image, actual image, and accuracy

%-----IMAGE READING-----%
[img,B,BW]=loadImage(image_location);

%-----EDGE DETECTION-----%
% Threshold or apply edge detection to detect the shape boundaries
binaryImage = edge(BW, 'sobel');

%-----FILLING HOLES-----%

```

```

% Fill inside the shapes
filledImage = imfill(binaryImage, 'holes');

%-----CHECK THE TYPE OF IMAGE-----%
%-----BINARY COMPONENT ANALYSIS-----%
%finding connected components
cc = bwconncomp(filledImage);
%taking area and eccentricity values of each component
stats = regionprops(cc,"Area","Eccentricity");
%specifying the circular shapes in the image
threshold_eccentricity=0.4; %after checking the circles in all images, the eccentricity
values of the circles in the images in which the projection is required is higher than
0.6 . While it was lower than 0.4 for projection-not-required images.
max_eccentricity=max([stats.Eccentricity]);

BW2=filledImage;

if max_eccentricity<threshold_eccentricity
    %%disp('It''s not a projection image');

    circle_centers=find_circle_centers(img,cc,stats);

    degree=find_slope(circle_centers);

    if degree==90
        %%disp('The picture is well-oriented');

    else
        %-----rotate image-----%
        rotation_degree=90-degree;
        %%disp('The rotation is required');
        BW2=imrotate(filledImage,rotation_degree,"crop");
        B=imrotate(B,rotation_degree,"crop");

    end

else
    %%disp('It''s a projection image');
    circle_centers=find_circle_centers(img,cc,stats);

    %the correct positions of the center of each circle
    circle_centers_ref=[26 26; 26 445; 445 26; 445 445];

    %the reference image can be provided with location either
    %%[circle_centers_ref,img_ref]=refImgCircleCenters(reference_image_location);
    %changing column row values for transformation function
    fixedPoints=circle_centers_ref(:, [2, 1]);
    movingPoints=circle_centers(:, [2, 1]);
    %find transformation matrix for projection
    mytform = fitgeotrans(movingPoints,fixedPoints, 'projective');
    %finds the size of image
    Rfixed = imref2d(size(img));
    %crop according to size of the reference image
    cropped = imwarp(B,mytform,'OutputView',Rfixed);

```

```

    %apply transformation for projection
    newIMG=imwarp(B,mytform);

    B=cropped;

end

%-----COMMON PART FOR ALL-----%
%After this point the image is oriented as expected
[L,num,BW3]=find_circle_or_square(BW2,'square');
%displaying the square

%-----Finding Square and Sub-square Size-----%

%finding the row and column indexes of square
[row_idx, col_idx]=find(L==num);
%finding the side of the square
row_start=min(row_idx);
row_end=max(row_idx);

col_start=min(col_idx);
col_end=max(col_idx);
%I wanted to take the average from each side, normally they must be
%equal, but because of the image quality, it may not work properly
side_square=round(((row_end-row_start)+(col_end-col_start))/2);

%finding the size of side of each sub-square
side_colour_check_element=floor(side_square/4); %there are 4 squares at each side

[margin_left,margin_right,margin_bottom]=margins_calculator(side_colour_check_element);

%prediction=getColours(rotated_img,side_square,side_colour_check_element,col_start,col_
end)

%creates subplot;
%left-hand side: distorted image
%left-hand side: corrected image
subplot(1, 2, 1);
imshow(img)
subplot(1, 2, 2);
imshow(B)

[prediction] =
getColours(B,side_square,side_colour_check_element,row_start,col_start,margin_left,marg
in_right,margin_bottom);
end

```

loadImage.m

```

function [img,B,BW] = loadImage(location)

%           INPUTS
%location= the location of the file in string format
%
%           OUTPUTS
%img=the original rgb image
%B=filtered image
%BW= black&white (binary) image

```



```
%
%
%           EXPLANATION
%Takes the location of the image
%Reads, apply filters, and converts it into binary
```

```
%read image
img=imread(location);
```

```
%applying filter function
B=filtering_image(img);
```

```
%giving the binary of the filtered image
BW=im2bw(B);
```

```
end
```

margins_calculator.m

```
function [margin_left,margin_right,margin_bottom] = margins_calculator(side)
```

```
%
%           INPUTS
%side: the size of side of the sub-square
%
%           OUTPUTS
%margin_left: left border offset value
%margin_right: right border offset value
%margin_bottom: bottom border offset value
%
```

```
%
%           EXPLANATION
%while checking each sub-square element in the main square, the borders of
%each is seen as black, so it effects the rgb value for each sub-square, to
%eliminate it offset is taken into account, this function calculates these
%values
```

```
    %there is a black frame which changes the rgb values, so offset is
    %required
```

```
    %margin proportions from each side to give offset from each border
    margin_proportion_bottom=0.2;
    margin_proportion_left=0.1;
    margin_proportion_right=0.1;
```

```
    %margin values for each border, found by taking the ceil
    margin_bottom=ceil(side*margin_proportion_bottom);
    margin_left=ceil(side*margin_proportion_left);
    margin_right=ceil(side*margin_proportion_right);
```

```
end
```

rotatedPredictionAndAccuracy.m

```
function [accuracy,rotated_prediction] =
rotatedPredictionAndAccuracy(prediction,matrix_location)
```

```
%
%           INPUTS
%prediction: prediction of cell of colours
%matrix_location: the location of the correct cell of colours
%
```

```
%           OUTPUTS
%accuracy: accuracy score between 0-1
%rotated_prediction: rotated prediction cell in case of necessity
%
```

```
%           EXPLANATION
%takes the predicted cell and correct oriantation, gives the correct rotated prediction
cell
```

```
%colour values matrix
check_matrix=load(matrix_location);
```

```
[accuracy,degree]=findBestOrientation(prediction,check_matrix);
%rotates the prediction matrix as many times as required
rotated_prediction=rot90(prediction,degree);
end
```

refImgCircleCenters.m

```
function [circle_centers_ref,img_ref] = refImgCircleCenters(location)

%           INPUTS
%location: location of the reference image
%
%           OUTPUTS
%circle_centers_ref: circle centers' matrix
%img_ref: reference image
%
%           EXPLANATION
%Takes the reference image location, gives each circle's center in the
%image
%
%-----REFERENCE IMAGE PART-----%

%taking the reference picture for projection correct
[img_ref,B_ref,BW_ref] = loadImage(location); %org_2 was selected for reference

%-----EDGE DETECTION-----%
% Threshold or apply edge detection to detect the shape boundaries
binaryImage_ref = edge(BW_ref, 'sobel');

%-----FILLING HOLES-----%
% Fill inside the shapes
filledImage_ref = imfill(binaryImage_ref, 'holes');

%-----CHECK THE TYPE OF IMAGE-----%
%-----BINARY COMPONENT ANALYSIS-----%
%finding connected components
cc_ref = bwconncomp(filledImage_ref);
%taking area and eccentricity values of each component
stats_ref = regionprops(cc_ref,"Area","Eccentricity");
%fill the holes, we can get whole shapes to find the exact area values
BW2_ref=filledImage_ref;

%finding circle centers for reference image
circle_centers_ref=find_circle_centers(img_ref,cc_ref,stats_ref);
%-----REFERENCE IMAGE END-----%
end
```

find_each_accuracy.mlx

```
directory_mat='images/*.mat';
directory_png='images/*.png';
%in case reference image is desired to be used, uncommend below
%%referenceImage='images/org_2.png'; %reference image for projection
%Also, uncommend refImgCircleCenters in the function of findColours
%Additionally, add a variable to the input as referenceImage
```

```

files_mat=dir(directory_mat); %matrix files
files_png=dir(directory_png); %images

Accuracy_Matrix = cell(min(length(files_mat), length(files_png)), 2); %accuracy matrix for all
images

for idx = 1:min(length(files_mat), length(files_png)) %loop to get each image and matrix from
file
    if files_mat(idx).isdir == 0 || files_png(idx).isdir == 0 % Check if it's not a directory
        %disp(files_png(idx).name)
        image_loc = fullfile('images/', files_png(idx).name); %image location
        matrix_loc = fullfile('images',files_mat(idx).name); %matrix location

        prediction=findColours(image_loc);
        [accuracy,rotated_prediction]=rotatedPredictionAndAccuracy(prediction,matrix_loc);

        Accuracy_Matrix{idx, 1} = files_png(idx).name; %adding file name
        Accuracy_Matrix{idx, 2} = accuracy; %adding accuracy
    end
end
Accuracy_Matrix

```