

## Problem Statement:

For our project, we are writing software in MATLAB that automatically reads a colour pattern image and returns an array of the colour pattern.

## Introduction:

In the video game Lego's Life of George, the screen briefly displays a picture before going black. The game's player must then build the shape from memory using Lego blocks. As soon as the player takes a snapshot of the finished blocks, a computer compares the brick design to the original image. In the code below, I've built a function that takes the supplied image and, with the help of a few additional functions, returns an array containing the colour pattern of the given image.

Firstly, we are defining the *'findcolour'* function which calls the other required functions.

### Function findcolours(filename)

This method *'findColours'* accepts a filename and gives out an array of colour names in the colour data matrix image.

1. This function initially loads a picture supplied by the filename and saves it to the image variable and then the *'loadImage'* function converts the image to type double, which is a typical image processing format.
2. Next, the function calls *'findCircles'* on the image to find the coordinates of any circles present.
3. Then the coordinates is passed as a parameter for the function *'correctImage'* which undistorts the image.
4. The code then checks to see if the filename contains the strings 'noise\_' or 'org\_'. If this doesn't occur, it shows a warning informing that the method *getColor*s can only be used on undistorted images and sets the variable result to 0. Otherwise, it calls the *getColours* function on the image and assigns the array of colour names found in the image to the variable result.

And then finally gives the 'result' and returns it as *'colorarray'*.

### 1.Image=loadImage(filename)

The loadImage MATLAB function loads an image from a file specified by 'filename' and converts it to a double-precision array. Here is a description of what the function does:

- The function *loadImage* has one input argument, filename.

- *Imread* is utilised within the function to read picture data from the file supplied by *filename*.
- The image data is saved in the variable *img*.
- The *image* data is then converted to type double using the '*im2double*' function, and the output is saved in the variable *image*.

The function returns the *image* variable as its output.

## 2.circleCoordinates = findCircles(image)

This function takes an image as input and returns the coordinates of the four largest black circles in the image. Here is a description of what the function does:

- The function '*findCircles*' has one input argument, *image*.
- The image input is saved in the variable *img*.
- Using the *rgb2gray* function, the image is transformed to grayscale and saved in the variable *gray\_img*.
- The '*graythresh*' function is used to determine an image threshold value, and the resulting binary image is saved in the variable *binary\_img*.
- The '*imcomplement*' function is used to invert the binary image, which is then stored in the variable *inverted\_binary\_img*.
- The *bwconncomp* function is used to name the linked components in the inverted binary image, and the results are saved in the variable *cc*.
- The *cellfun* function is used to compute the area of each connected component in *cc*'s *PixelIdxList*, and the resulting areas are saved in the variable *areas*.
- The *sort* function is used to sort the areas descendingly, and the sorted areas and indices are kept in the variables *sorted\_areas* and *sorted\_indices*, respectively.
- A for loop is used to obtain the coordinates of the image's first four greatest black blobs (circles). The loop begins at 2 to exclude the largest white blob which is the background, and the coordinates of each blob are kept in the *blob\_coords* matrix.
- The first coordinate in the *blob\_coords* matrix is eliminated since it relates to the backdrop.
- The *sortrows* function is used to sort the coordinates clockwise, beginning with the bottom-left blob, and the sorted coordinates are saved in the variable *sortedCoordinates*.
- The second and fourth coordinates are compared to the first and third coordinates to see if they are in the proper positions using two if statements. The coordinates are switched if they are not.
- The output of the function is stored in the variable *circleCoordinates*, which contains the sorted coordinates of the four biggest black circles in the image.

The overall purpose of this code is to locate and sort the coordinates of the biggest black circles in an image through image processing procedures. It recognises the circles by combining picture thresholding, connected component labelling, and area computations, and then arranges their coordinates in a particular manner.

### 3.correctImage(circleCoordinates, image)

This MATLAB function uses the supplied coordinates to adjust an image. Here's the breakdown of how the functions works:

- The function '*correctImage*' requires two input arguments: *image*, the image that has to be rectified, and *coordinates*, a set of coordinates that specify the region of interest.
- The corners of a rectangle into which the image will be transformed are specified by the 4x2 matrix *boxf*.
- The region is defined by four points: [0,0], [0,480], [480,480], and [480,0].
- The transformation that transfers the Coordinates to the rectangle area is made using this matrix.
- TF is the transformation matrix produced by *fitgeotrans*, which applies a projective transformation to map the Coordinates to *boxf*.
- An *imref2d* object called *outview* establishes the spatial reference for the final image. It is made using the input image's size.
- *B* is the rectified image that was produced by using *imwarp* to apply the projective transformation. The *fillvalues* option specifies the values to be applied to pixels outside the input image's boundaries. It is set to 255 (*white*) in this instance to fill in any gaps left by the transformation.
- The output image has the same spatial reference as *outview* because the *outputview* parameter is set to *outview*.
- *Imcrop* is used to crop *B* to a size of 480x480 pixels and to remove any glare from the image *imflatfield* is applied to *B* with a value of 40 and the image contrast is adjusted using *imadjust* with the settings [0.4,0.65].

The function returns the undistorted corrected version of the given image.

### 4.colours=getColours(image)

The '*getColours*' function, which takes an image as input and outputs a 4x4 matrix of colour names, is defined in this code. The following actions are carried out by the code:

- Use *im2uint8* to convert the input picture to 8-bit unsigned integer format.
- Employ *medfilt3* and *imadjust* to apply a 3D median filter to reduce noise and boost contrast.
- Using *rgb2gray* and a threshold value of 20, construct a binary mask using the filtered image.

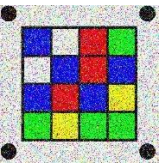
- Make use of *bwareaopen* and its complement *bwareaopen* to eliminate minute positive and negative specks from the mask.
- Using an *imerode* tool and a structural element of size 10x10, erode the mask to remove the outer white area.
- Segment the picture with *bwlabel*.
- Determine the average colour in each section of the mask and store the results in *Concolors*.
- Attempt to align the region's centres with a 4x4 grid.
- Rearrange the colour samples to correspond to the centres' snapped-in order.
- Determine the RGB colour separations between each sample and a collection of reference colours.
- From the list of reference colours, choose the sample that is the closest match.
- Check the list of reference colours for the colour names that correspond to the matched colours.
- Convert the list of colour names into a 4x4 matrix and output it as the function's result.

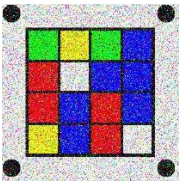
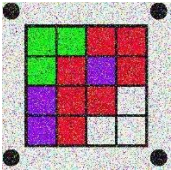
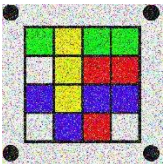
The pre-trained neural network can also be used in this function to reduce noise and improve colour recognition.

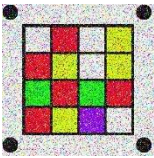
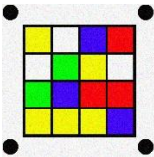
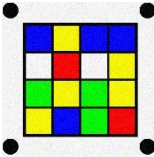
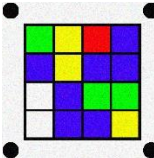
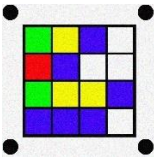
## Conclusion:

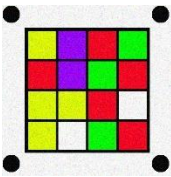
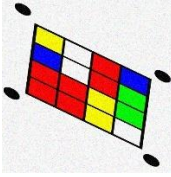
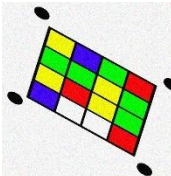
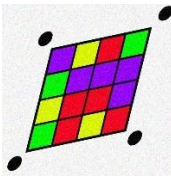
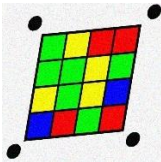
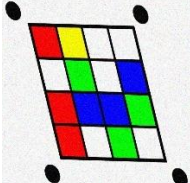
In order to perform this, several functions are utilised, including `loadImage`, `findCircles`, `correctImage`, and `getColours`. The image is loaded, modified, its circles are located, any distortion is fixed, and the image's colours are extracted. Although the results could occasionally be off, they give the desired output for most of the images.

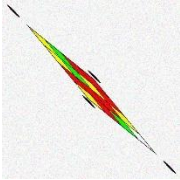
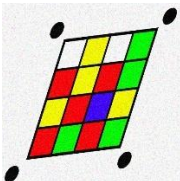
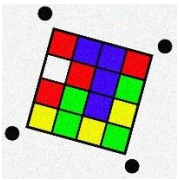
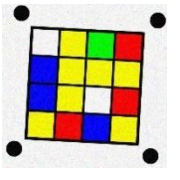
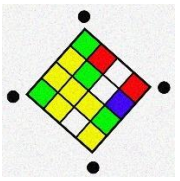
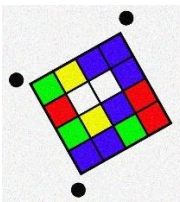
## Results Table:

Filename	Image	Output	Success	Notes
noise_1.png		<div> <div>'blue'</div> <div>'white'</div> <div>'red'</div> <div>'green'</div> <div>'white'</div> <div>'blue'</div> <div>'red'</div> <div>'blue'</div> <div>'blue'</div> <div>'red'</div> <div>'blue'</div> <div>'yellow'</div> <div>'green'</div> <div>'yellow'</div> <div>'green'</div> <div>'green'</div> </div>	Yes	None

noise_2.png		<div> <div>'green'</div> <div>'red'</div> <div>'red'</div> <div>'yellow'</div> </div> <div> <div>'yellow'</div> <div>'white'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'green'</div> <div>'blue'</div> <div>'red'</div> <div>'red'</div> </div> <div> <div>'blue'</div> <div>'blue'</div> <div>'blue'</div> <div>'white'</div> </div>	Yes	None
noise_3.png		<div> <div>'green'</div> <div>'green'</div> <div>'blue'</div> <div>'blue'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'red'</div> <div>'red'</div> </div> <div> <div>'red'</div> <div>'blue'</div> <div>'red'</div> <div>'white'</div> </div> <div> <div>'red'</div> <div>'red'</div> <div>'white'</div> <div>'white'</div> </div>	No	We got 75% accuracy as the violet colours are shown as blue colour in the output this is because we haven't given the reference for that colour, so it gave the nearest colour blue as output.
noise_4.png		<div> <div>'green'</div> <div>'white'</div> <div>'blue'</div> <div>'white'</div> </div> <div> <div>'yellow'</div> <div>'yellow'</div> <div>'yellow'</div> <div>'blue'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'blue'</div> <div>'red'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'blue'</div> <div>'white'</div> </div>	Yes	None

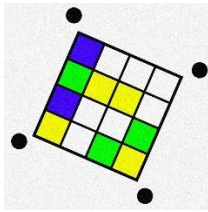
noise_5.png		<div> <div>'white'</div> <div>'red'</div> <div>'white'</div> <div>'yellow'</div> </div> <div> <div>'red'</div> <div>'yellow'</div> <div>'white'</div> <div>'yellow'</div> </div> <div> <div>'green'</div> <div>'red'</div> <div>'green'</div> <div>'red'</div> </div> <div> <div>'red'</div> <div>'yellow'</div> <div>'blue'</div> <div>'white'</div> </div>	No	We got 91% accuracy because of the violet-coloured block in the image.
org_1.png		<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	None
org_2.png		<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	None
org_3.png		<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	None
org_4.png		<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	None

org_5.png		<div> 'yellow'   'blue'   'red'   'green'  'red'   'blue'   'green'   'red'  'yellow'   'yellow'   'red'   'white'  'yellow'   'white'   'green'   'red' </div>	no	We got 83% accuracy because of the violet blocks.
proj_1.png		Not applicable for distorted images	None	None
proj_2.png		Not applicable for distorted images	None	None
proj_3.png		Not applicable for distorted images	None	None
proj_4.png		Not applicable for distorted images	None	None
proj_5.png		Not applicable for distorted images	None	None

proj_6.png		Not applicable for distorted images	None	None
proj_7.png		Not applicable for distorted images	None	None
rot_1.png		Not applicable for distorted images	None	None
rot_2.png		Not applicable for distorted images	None	None
rot_3.png		Not applicable for distorted images	None	None
rot_4.png		Not applicable for distorted images	None	None



rot\_5.png



Not applicable for distorted images

None

None

### Thoughts and comments on the given Real Images

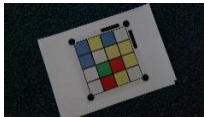
Filename	Image	Comments
IMAG0032.jpg	A 4x4 grid of colored squares (yellow, blue, green, orange) on a white background, with four black dots at the corners. The image is slightly tilted and has a shadow in the bottom left.	This image rectangles in addition to the four circles and there is also a shadow in the bottom left
IMAG0033.jpg	A 4x4 grid of colored squares (red, green, blue, yellow) on a white background, with four black dots at the corners. The image is oriented and has some noise.	This image is oriented and has some noise but looks better to process.
IMAG0034.jpg	A 4x4 grid of colored squares (green, yellow, orange, white) on a white background, with four black dots at the corners. The image has significant light glare.	Too much of light glare in the image makes it difficult to recognise the exact colours.

IMAG0035.jpg



There is a huge shadow on the image but with the use of use certain filters, we might still be able to perform the desired action on it.

IMAG0036.jpg



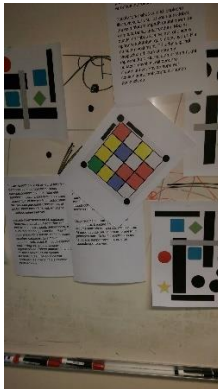
This image looks better to process

IMAG0037.jpg



Here the image is exposed to bright light.

IMAG0038.jpg



Too many objects are present in the image to detect the corners of the colour matrix.

IMAG0041.jpg



The image seems to be shrunk but using the '*imwarp*' technique we can undistort this image.

IMAG0042.jpg



This image has a blue filter over it.  
We have to remove it before processing.

IMAG0044.jpg



This image is too blurry to detect the edges precisely and extract the colours from it.

## Code Appendix:

### Function: FindColours(filename)

```
function colorArray = findColours(filename)

% Loads an image from the file specified by filename, and returns it as type double.
image = loadImage(filename);

% this function finds the circles in the image and returns its co-ordinates
circleCoordinates = findCircles(image);

%this function undistorts the image by using the circle coordinates from
%the below function
images=correctImage(circleCoordinates,image);
imshow(images)

if contains(filename, 'noise_') || contains(filename, 'org_')
    %this function getColours returns an array of colours in the
    % given image
    result = getColours(image);
else
    disp('The getColors function can be used only on un distorted images');
    result=0;
end
disp(result)
colorArray = result;
end
```

### Function: loadImage(filename)

```
function image = loadImage(filename)
% Loads an image from the file specified by filename, and returns it as type double.

% Read in the image using imread
img = imread(filename);
```

```
% Convert the image to double precision
image = im2double(img);

end
```

## Function: findCircles(image)

```
function circleCoordinates = findCircles(image)

% Load the image
img = image;

% Changing the given image to grey image
gray_img = rgb2gray(img);

% Thresholding to get a binary image
threshold = graythresh(gray_img);
binary_img = imbinarize(gray_img, threshold);

% inverting the binary image
inverted_binary_img = imcomplement(binary_img);

% Labelling the components that are connected in the inverted_binary_img
cc = bwconncomp(inverted_binary_img);

% Calculating the area of each component
areas = cellfun(@numel, cc.PixelIdxList);

% Sorting in descending order
[sorted_areas, sorted_indices] = sort(areas, 'descend');

% Getting the coordinates of the first four largest black blobs
num_blobs = 5;
blob_coords = zeros(num_blobs, 2);
for i = 2:num_blobs
    blob_indices = cc.PixelIdxList{sorted_indices(i)};
    [rows, cols] = ind2sub(size(inverted_binary_img), blob_indices);
    blob_coords(i, :) = [ mean(cols),mean(rows)];
end
% Removing the first coordinate from the blob_coords matrix
blob_coords(1, :) = [];

% Sort the coordinates in clockwise order starting from bottom-left
sortedCoordinates = sortrows(blob_coords);

if sortedCoordinates(2,2) < sortedCoordinates(1,2)
    % If the second coordinate is below the first, swap them
    sortedCoordinates([1 2],:) = sortedCoordinates([2 1],:);
```

```

end

if sortedCoordinates(4,2) > sortedCoordinates(3,2)
    % If the fourth coordinate is above the third, swap them
    sortedCoordinates([3 4],:) = sortedCoordinates([4 3],:);
end

circleCoordinates=sortedCoordinates;

```

end

### Function: correctImage(Coordinates, image)

```

function outputImage = correctImage(Coordinates, image)

% Define a fixed box with coordinates
boxf = [[0 ,0]; [0 ,480];[480 ,480]; [480 ,0]];

% Calculating the transformation matrix from the given Coordinates to
% transform the matrix to the fixed box using projective transformation
TF = fitgeotrans(Coordinates,boxf,'projective');

% Create an image reference object with the size of the input image
outview = imref2d(size(image));

% Apply the calculated transformation matrix to the input image
% and create a new image with fill value 255 (white) outside the boundaries of the
input image
B = imwarp(image,TF, 'fillvalues',255,outputview=outview);

% Crop the image to a size of 480x480
B = imcrop(B,[0 0 480 480]);

% Try to suppress the glare in the image using flat-field correction
B = imflatfield(B,40);

% Adjust the levels of the image to improve contrast
B = imadjust(B,[0.4 0.65]);

% Assign the corrected image to the outputImage variable
outputImage = B;
end

```

### Function: getColours(image)

```

function colours=getColours(image)

% Convert the image to uint8 format
W=im2uint8(image);

% Median filter to suppress noise

```

```

W = medfilt3(W,[7 7 1]);

% Increase contrast
W = imadjust(W,stretchlim(W,0.025));

% Convert the RGB image to grayscale and threshold
Conimage = rgb2gray(W)>20;

% Remove positive specks from binary image
Conimage = bwareaopen(Conimage,100);

% Remove negative specks from binary image
Conimage = ~bwareaopen(~Conimage,100);

% Remove outer white region
Conimage = imclearborder(Conimage);

% Erode image
Conimage = imerode(Conimage,ones(10));

% Segmenting the image
[K 0] = bwlabel(Conimage);

% Storing the average color of each region
Concolors = zeros(0,3);

% Getting the average color in each labeled region
for p = 1:0 % step through patches
    each_pch = K==p;
    all_pch_areas = W(each_pch(:,:),[1 1 1]);
    Concolors(p,:) = mean(reshape(all_pch_areas,[],3),1);
end

% Normalizing the color values to the required range [0, 1]
Concolors = Concolors./255;

% Snapping centers to grid
Y = regionprops(Conimage,'centroid');
X = vertcat(Y.Centroid);
lim_X = [min(X,[],1); max(X,[],1)];
X = round((X-lim_X(1,:))./range(lim_X,1)*3 + 1);

% Reordering the color samples
idx = sub2ind([4 4],X(:,2),X(:,1));
Concolors(idx,:) = Concolors;

% Specifying color names
clrnames = {'white','red','green','blue','yellow'};

% declaring a reference colors list in RGB
clrrefs = [1 1 1; 1 0 0; 0 1 0; 0 0 1; 1 1 0];

% measuring distance of colours in RGB
I = Concolors - permute(clrrefs,[3 2 1]);
I = squeeze(sum(I.^2,2));

```

```

% finding the nearest match
[~,idx] = min(I,[],2);

% Looking for the colour names in each patch
Colornames = reshape(clnames(idx),4,4);

% Returns the array color names
colours= Colornames;

end

```

## References:

[Clay Swackhamer](#) [2019], Mathworks.com, How to make an image straightener like that of cam scanner app? .Available at: <https://uk.mathworks.com/matlabcentral/answers/456973-how-to-make-an-image-straightener-like-that-of-cam-scanner-app>

[DGM](#) [2023], Mathworks.com, Car Plate Perspective Transform, Available at: [https://uk.mathworks.com/matlabcentral/answers/1954429-car-plate-perspective-transform/?s\\_tid=ans\\_lp\\_feed\\_leaf](https://uk.mathworks.com/matlabcentral/answers/1954429-car-plate-perspective-transform/?s_tid=ans_lp_feed_leaf)

[Image Analyst](#) [2022], Mathworks.com. Detecting colors on a RGB image. Available at: [https://uk.mathworks.com/matlabcentral/answers/547560-detecting-colors-on-a-rgb-image?s\\_tid=srchtitle](https://uk.mathworks.com/matlabcentral/answers/547560-detecting-colors-on-a-rgb-image?s_tid=srchtitle)

Noise Removal, Matlab Documentation, Available at: <https://uk.mathworks.com/help/images/noise-removal.html>

Imwarp, Matlab Documentation. Available at: <https://uk.mathworks.com/help/images/ref/imwarp.html>

[DGM](#), 2023, Mathworks.com, Returning an array of colors from a double. Available at: <https://uk.mathworks.com/matlabcentral/answers/1664934-returning-an-array-of-colors-from-a-double-image>

Mathworks.com, How to recognize 6 colors of a face rubik's cube at the same time?. Available at: [https://uk.mathworks.com/matlabcentral/answers/1827523-how-to-recognize-6-colors-of-a-face-rubik-s-cube-at-the-same-time?s\\_tid=prof\\_contriblnk](https://uk.mathworks.com/matlabcentral/answers/1827523-how-to-recognize-6-colors-of-a-face-rubik-s-cube-at-the-same-time?s_tid=prof_contriblnk)