

# Image Processing Lab Project Report

## Introduction

The purpose of this project is to mimic the game Lego's Life of George, in which a player must recall and reassemble Lego blocks from an image that is displayed on the screen for a brief period of time. MATLAB was used to create a program that would correct a set of images with various flaws, such as noise, rotation, and projections, and then return an array that accurately represented the 16 square coloured blocks as strings.

## Strategy

To retrieve an array representing the colour pattern of a colour pattern image from a hard disc. Initially, the image will be loaded into the program. Then, the image will be checked for any transformations needed. If the image is projected or rotated, it will be corrected to ensure proper alignment. Next, the colour matrix will be found by locating the position of each square using a segmentation technique, such as thresholding, and then extracting the colour value from each square. This process will be robust to noise and changes in lighting conditions by using appropriate filtering and normalization techniques. Finally, the colour values will be returned as an array of strings representing each square in the image.

## Functions

To obtain the colour matrix for all PNG files in the 'images' folder, simply run the 'main.mlx' MATLAB live script, which loads and processes each file. During this process, it invokes the 'getColourMatrix' function to process the current file and get the colour matrix.

- **getColourMatrix()**

The function '**getColourMatrix**' takes a filename as input and returns the colour matrix of the image file. It first reads the image data from the file and displays the original image. Then it checks if the image can be transformed using the '**canBeTransformed**' function. If it can be transformed, the image is corrected using the '**correctImage**' function, and the corrected image is displayed. Finally, the '**findColours**' function is called to obtain the colour matrix of either the corrected or original image depending on the transformation possibility.

- **canBeTransformed()**

The function '**canBeTransformed**' determines whether an input image can be transformed. The function first converts the input image to grayscale, then applies the Canny edge detection algorithm to obtain a binary edge image. The edge image is morphologically thickened to make the edges more prominent. The Radon transform is then applied to the thickened edge image, and the angle associated with the maximum value in the Radon transform is calculated. If the absolute value of the angle is greater than 50 degrees, the image is considered transformable, and the function returns **true**. Otherwise, the function returns **false**.

- **correctImage()**

The function '**correctImage**' takes an image as an input and performs the following tasks:

1. It calls the '**findCircles**' function to detect the centroids of four circles in the image.

2. It defines a set of fixed points for the transformation using the reference image **'org\_1.png'**.
3. It reorders the detected circle centroids to match the fixed points using the helper function **'orderPoints'**. The function **'orderPoints'** initializes an output array with one point at a fixed position and then loops over each point in "movingPoints". For each point, it finds the closest point in "fixedPoints" using Euclidean distance, removes that point from "fixedPoints", and adds the corresponding point from "movingPoints" to the output array.
4. It performs an affine transformation using the matched points and the **'fitgeotrans'** function.
5. It crops the transformed image to a specific size, to get rid of the uninterested region. As depicted in figure 1.

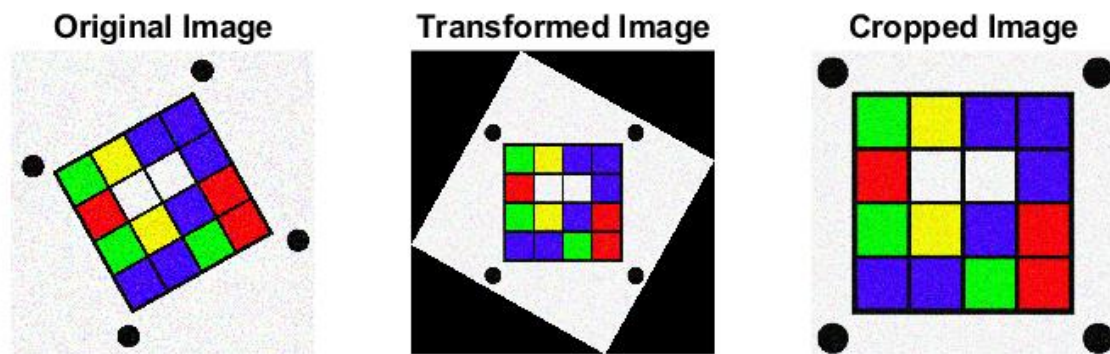


Figure 1

6. It returns the corrected image.

NOTE: **'orderPoints'** function uses **pdist()** which is available in Statics and Machine Learning toolkit of MATLAB.

- **findCircles()**

The function **'findCircles'** locates four black circles in an input image and returns their centroids. It starts by converting the image to grayscale and applying Canny edge detection to extract edges. It then uses **'bwconncomp'** to find the connected components in the edge image and **'regionprops'** to get their convex areas. The four smallest convex areas are then selected using **'ismember'** and **'mink'**. Connected components with convex areas other than the four smallest are discarded. The centroids of the remaining four connected components are then computed using **'mean'** and **'ind2sub'**. Finally, the function returns an array of centroid coordinates.

- **findColours()**

The function **'findColours'** converts the input image from RGB to LAB colour space, as LAB Colour is a more accurate colour space, and then crops the image to remove the circles. Then, the function initializes an output array of size 4x4 and defines coordinates for regions of interest. It iterates over each coordinate pair, extracts a single patch of the cropped image, calculates the mean LAB values of the region, classifies the region based on its LAB values, and stores the result in the output array. Two helper functions, **'meanLAB'** and **'classifyColour'** are used to compute the mean  $L^*$ ,  $a^*$ , and  $b^*$  values separately for each colour

channel (R,G,B) of the patch using the ‘mean’ function and to classify colour label by checking the values of L, A, and B against certain thresholds, respectively.

## Comments on Real Photographs



Figure 2

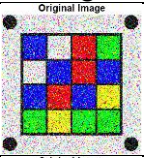
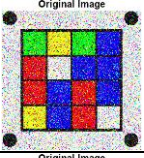
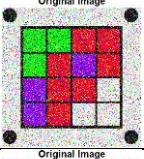
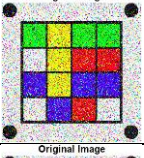
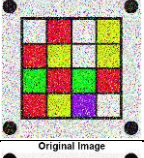
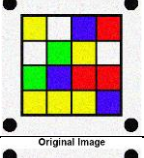
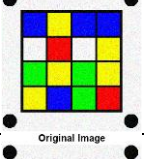
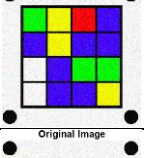
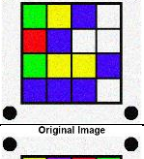
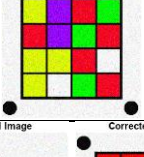
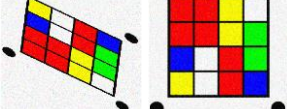
Figure 2 shows four images with uneven lighting and shadow patches, IMAG0032, IMAG0033, IMAG0034, and IMAG0035. These images can have their lighting evened out using morphological operators. While having a dark and light contrast, respectively, IMAG0036 and IMAG0037 have the most consistent lighting. Due to the fact that IMAG0038 contains several circles of the same size, it would be difficult to distinguish the coloured square block using the four circles at its corners. The image IMAG0042 has a noticeable blue cast that can be removed by switching the image's colour space from RGB to LAB (Luminance, A, B). Despite IMAG0041's slight crumbling and IMAG0044's blur, the canny edge detection algorithm may still be able to handle these photos.

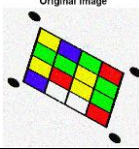
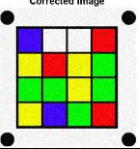
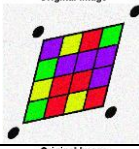
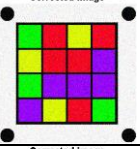
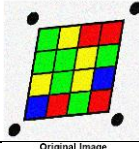
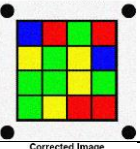
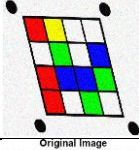
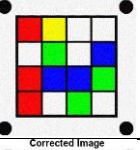
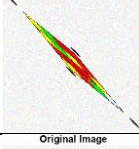
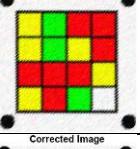
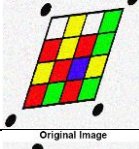
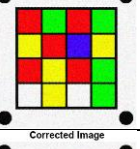
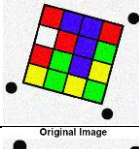
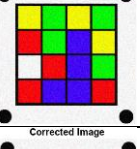
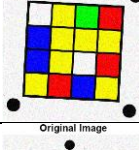
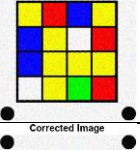
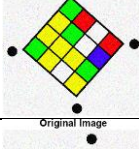
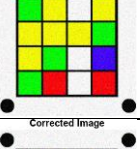
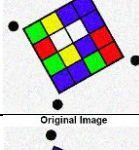
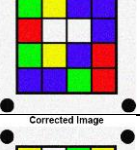
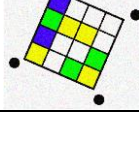
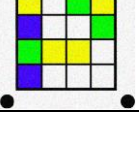
## Conclusion

The project aimed to automatically read a coloured pattern image and return an array representing the colour pattern. To achieve this, a strategy was devised that involved loading the image, checking if it needed any transformation, correcting it if required, and finally finding and returning the colour matrix. The results show that the task of identifying the colours of the squares in each image was successfully fulfilled for all the images with 100% accuracy. Moreover, the function that dealt with circle detection and fixing the orientation of the rotated/projected images had an overall 100% success rate, making it a robust and reliable solution for this image processing problem.

# Appendix

Results Table

Filename	Image	Output	Success	Notes
noise_1.png		<pre>result = 4x4 string "B"      "W"      "R"      "G" "W"      "B"      "R"      "B" "B"      "R"      "B"      "Y" "G"      "Y"      "G"      "G"</pre>	Yes	None
noise_2.png		<pre>result = 4x4 string "G"      "Y"      "G"      "B" "R"      "W"      "B"      "B" "R"      "B"      "R"      "B" "Y"      "B"      "R"      "W"</pre>	Yes	None
noise_3.png		<pre>result = 4x4 string "G"      "G"      "R"      "R" "G"      "R"      "B"      "R" "B"      "R"      "R"      "W" "B"      "R"      "W"      "W"</pre>	Yes	None
noise_4.png		<pre>result = 4x4 string "G"      "Y"      "G"      "G" "W"      "Y"      "R"      "R" "B"      "Y"      "B"      "B" "W"      "B"      "R"      "W"</pre>	Yes	None
noise_5.png		<pre>result = 4x4 string "W"      "R"      "W"      "Y" "R"      "Y"      "W"      "Y" "G"      "R"      "G"      "R" "R"      "Y"      "B"      "W"</pre>	Yes	None
org_1.png		<pre>result = 4x4 string "Y"      "W"      "B"      "R" "W"      "G"      "Y"      "W" "G"      "B"      "R"      "R" "Y"      "Y"      "Y"      "B"</pre>	Yes	None
org_2.png		<pre>result = 4x4 string "B"      "Y"      "B"      "B" "W"      "R"      "W"      "Y" "G"      "Y"      "G"      "Y" "Y"      "B"      "G"      "R"</pre>	Yes	None
org_3.png		<pre>result = 4x4 string "G"      "Y"      "R"      "B" "B"      "Y"      "B"      "B" "W"      "B"      "G"      "G" "W"      "B"      "B"      "Y"</pre>	Yes	None
org_4.png		<pre>result = 4x4 string "G"      "Y"      "B"      "W" "R"      "B"      "W"      "W" "G"      "Y"      "Y"      "B" "B"      "B"      "B"      "W"</pre>	Yes	None
org_5.png		<pre>result = 4x4 string "Y"      "B"      "R"      "G" "R"      "B"      "G"      "R" "Y"      "Y"      "R"      "W" "Y"      "W"      "G"      "R"</pre>	Yes	None
proj_1.png		<pre>result = 4x4 string "R"      "R"      "Y"      "W" "R"      "R"      "Y"      "G" "B"      "W"      "R"      "G" "Y"      "W"      "R"      "B"</pre>	Yes	None

proj_2.png			result = 4x4 string "B" "W" "W" "R" "Y" "R" "Y" "G" "G" "G" "Y" "G" "Y" "B" "G" "R"	Yes	None
proj_3.png			result = 4x4 string "G" "R" "Y" "R" "Y" "R" "R" "B" "G" "B" "B" "B" "B" "Y" "R" "G"	Yes	None
proj_4.png			result = 4x4 string "B" "R" "G" "R" "Y" "G" "Y" "B" "G" "G" "Y" "G" "G" "Y" "R" "R"	Yes	None
proj_5.png			result = 4x4 string "R" "Y" "W" "W" "W" "G" "W" "B" "R" "B" "B" "G" "R" "W" "G" "W"	Yes	None
proj_6.png			result = 4x4 string "Y" "G" "R" "R" "Y" "G" "Y" "R" "R" "R" "R" "Y" "Y" "R" "G" "W"	Yes	None
proj_7.png			result = 4x4 string "R" "G" "R" "G" "Y" "R" "B" "Y" "R" "Y" "R" "G" "W" "Y" "W" "G"	Yes	None
rot_1.png			result = 4x4 string "Y" "G" "Y" "G" "R" "G" "B" "Y" "W" "R" "B" "G" "R" "B" "B" "R"	Yes	None
rot_2.png			result = 4x4 string "Y" "R" "B" "Y" "B" "Y" "W" "R" "B" "Y" "Y" "Y" "W" "Y" "G" "R"	Yes	None
rot_3.png			result = 4x4 string "G" "Y" "W" "Y" "Y" "Y" "Y" "G" "Y" "G" "W" "B" "G" "R" "W" "R"	Yes	None
rot_4.png			result = 4x4 string "G" "Y" "B" "B" "R" "W" "W" "B" "G" "Y" "B" "R" "B" "B" "G" "R"	Yes	None
rot_5.png			result = 4x4 string "Y" "W" "G" "Y" "B" "W" "W" "G" "G" "Y" "Y" "W" "B" "W" "W" "W"	Yes	None

## Code

- **main.mlx**

```
% Find all png files in images folder
D=dir('images/*.png');

% Load and process each file
for ind=1:length(D)
    % Name of png file
    filename = fullfile(D(ind).folder,D(ind).name);
    % Printing the filename
    fprintf('Filename: <strong> %s </strong>\n',D(ind).name)
    % Getting colour matrix for the file
    result = getColourMatrix(filename)
end
```

- **getColourMatrix.m**

```
function colors = getColourMatrix(filename)
    % Reads an image from a file, applies correction if possible, and
    % returns the color matrix

    % Read image data from file
    img = imread(filename);

    % Display the original image
    figure;
    subplot(1, 2, 1)
    imshow(img)
    title("Original Image")

    % Check if image can be transformed
    if (canBeTransformed(img))
        % Correct the image
        correctedImg = correctImage(img);

        % Display the corrected image
        subplot(1, 2, 2)
        imshow(correctedImg)
        title("Corrected Image")

        % Get colors from corrected image
        colors = findColours(correctedImg);
    else
        % Get colors from original image
        colors = findColours(img);
    end
end
```

- **canBeTransformed.m**

```
function transformable = canBeTransformed(img)
    % Determine whether the image can be transformed

    % Convert the image to grayscale
    gray_img = rgb2gray(img);

    % Apply the Canny edge detection algorithm to the grayscale image
```



```

edge_img = edge(gray_img, 'canny');

% Morphologically thicken the edges to make them more prominent
thickened_img = bwmorph(edge_img, 'thicken');

% Define the range of angles to use for the Radon transform
theta_range = -90:89;

% Perform the Radon transform on the thicken image using the specified
% theta range
[R,~] = radon(thickened_img, theta_range);

% Find the maximum value in each column of the Radon transform
max_values = max(R);

% Determine the angle associated with the maximum value in the
% Radon transform
angle = 90;
while(angle > 50 || angle < -50)
    [~, angle] = max(max_values);
    max_values(angle) = 0;
    angle = angle - 91;
end

% Determine whether the image is transformable based on the angle
if (angle ~= 0)
    transformable = true;
else
    transformable = false;
end
end

```

- **correctImage.m**

```

function correctedImage = correctImage(image)
    % Finds centroids of four circles in the image
    circlesCentroids = findCircles(image);

    % Define fixed points for transformation taking 'org_1.png' as reference
    fixedPoints = [26.1005  26.0050; 26.3870  445.6450;
                   445.7411  26.5075; 445.9658  445.0861];

    % Reorder the points to match in the corresponding images
    circlesCentroids = cell2mat(orderPoints(circlesCentroids, fixedPoints));

    % Perform affine transformation with the given points
    myTransform = fitgeotrans(circlesCentroids, fixedPoints, 'affine');
    transformedImg = imwarp(image, myTransform);

    % Crop image and convert to LAB color space for output
    cropSize = [480 480];
    cropWindow = centerCropWindow2d(size(transformedImg), cropSize);
    correctedImage = imcrop(transformedImg, cropWindow);
end

function orderedPoints = orderPoints(movingPoints, fixedPoints)
    % Order points based on their euclidean proximity

    % Initialize output array with one point at a fixed position

```

```

orderedPoints = {[4, 2]};

% Create a copy of movingPoints
tempPoints = movingPoints;

% Loop over each point in movingPoints
for i = 1:length(movingPoints)
    minDistance = Inf;
    minDistanceIndex = 0;

    % Select the current point from fixedPoints
    currentFixedPoint = fixedPoints(i, :);

    % Loop over each point in fixedPoints to find the closest one
    for j = 1:length(fixedPoints)
        % Get the current point from fixedPoints
        p2 = fixedPoints(j, :);

        % Calculate the Euclidean distance between the two points
        X = [currentFixedPoint; p2];
        currentDistance = pdist(X, 'euclidean');

        % Update the minimum distance and index if the current
        % distance is smaller
        if currentDistance < minDistance
            minDistance = currentDistance;
            minDistanceIndex = j;
        end
    end

    % Remove the fixed point from fixedPoints by replacing it with
    % a point at infinity to avoid selecting it again
    fixedPoints(minDistanceIndex, :) = [Inf -Inf];

    % Add the corresponding point from movingPointsCopy to orderedPoints
    orderedPoints{i, 1} = tempPoints(minDistanceIndex, 1);
    orderedPoints{i, 2} = tempPoints(minDistanceIndex, 2);
end
end

```

- **findCircles.m**

```

function centroids = findCircles(img)
    % Locates four black circles in the image and returns their centroids

    % Convert the image to grayscale and apply Canny edge detection
    gray_img = rgb2gray(img);
    edge_img = edge(gray_img, 'Canny', [0.01, 0.9]);

    % Find connected components and their convex areas
    objects = bwconncomp(edge_img);
    CC = regionprops("table", edge_img, 'ConvexArea');

    % Find the four smallest convex areas
    min_four = mink(CC.ConvexArea, 4);

    % Select connected components with the four smallest convex areas
    idx = ismember(CC.ConvexArea, min_four);
    pxllist = objects.PixelIdxList;

```



```

% Deletes all false elements
pxlList(~idx) = [];

% Calculate the centroid of each connected component
centroids = zeros(numel(pxlList), 2);
for i = 1:numel(pxlList)
    [r, c] = ind2sub(size(img), pxlList{i});
    centroids(i, :) = mean([c r]);
end
end

```

- **findColours.m**

```

function result = findColours(img)
% This function is used to return an array of strings of
% categorical colors

% Convert image to LAB color space
labImage = rgb2lab(img);

% Crop the image to the center region without the circles
croppedImage = labImage(75:405, 75:405, :);

% Initialize the result array
result = string(zeros(4, 4));

% Define the coordinates of the regions of interest
coordinates = [10, 100, 200, 260];

% Iterate over the coordinates
for i = 1:length(coordinates)
    y = coordinates(i);
    for j = 1:length(coordinates)
        x = coordinates(j);
        % Extract a single patch of the cropped image
        patch = croppedImage(20+y:25+y, 20+x:25+x, :);
        % Calculate the mean LAB values of the region
        [L, A, B] = meanLAB(patch);
        % Classify the region based on its LAB values
        label = classifyColour(L, A, B);
        % Store the result in the output array
        result(i, j) = label;
    end
end
end

function [avgL, avgA, avgB] = meanLAB(patch)
% Calculates and returns the average L*, a*, and b* values from an
% input image

avgL = mean(patch(:, :, 1), 'all');
avgA = mean(patch(:, :, 2), 'all');
avgB = mean(patch(:, :, 3), 'all');
end

function colorLabel = classifyColour(L, A, B)
% Accepts mean LAB values and returns color label
if L > 74
    if A < -36

```

```
        colorLabel = "G"; % Green
elseif B > 35
    colorLabel = "Y"; % Yellow
else
    colorLabel = "W"; % White
end
elseif L < 74 && B < -15
    colorLabel = "B"; % Blue
elseif abs(A) < 3 && abs(B) < 3
    colorLabel = "W"; % White
else
    colorLabel = "R"; % Red
end
end
```