function result = colourMatrix(filename)
% This function is designed to process images read from a file

```
% Initializes the Sigmoid value as 2
sigmoid = 1.5;
% Enhance the image by gaussian filter
[rgb_Img, lab_Img] = processImage(filename, sigmoid);

% Plot the original image and contrasts the corrected image
figure;
subplot(1, 2, 1)
imshow(filename)
title("Original Image")

if (isTransformed(rgb_Img))
    % if this image can be transformed
    [rgbImg_Corrected , labImg_Corrected] = autoCorrection(rgb_Img);

    % Plot the corrected Image
    subplot(1, 2, 2)
    imshow(rgbImg_Corrected)
    title("Corrected Image")
    % Show the result
    result = getColourMatrix(labImg_Corrected);
else
    % if it can not be transformed
    result = getColourMatrix(lab_Img);
end
```

end

function results = getColourMatrix(img)
% This function is used to return an array of strings of categorical colors
coloursCropped = img(75:405, 75:405, :);
results = string(zeros(4, 4));
% Fixed the coords
coords = [10, 100, 200, 260];

```
% interate the coords
for i = 1:length(coords)
deltaY = coords(i);
    for j = 1:length(coords)
        deltaX = coords(j);
        squareSlice = coloursCropped(20+deltaY:25+deltaY, 20+deltaX:25+deltaX, :);
        [l, a, b] = meanLab(squareSlice);
        myLabel = classifyColour(l, a, b);
        results(i, j) = myLabel;
    end
end
```

end

function label = classifyColour(l, a, b)
% classifyColour: Accepts mean LAB values, returns char colour labe

```
if (l > 82)
    if (a < -35)
        label = "G";  % Green
    elseif (b > 35)
        label = "Y";  % Yellow
    else
        label = "W";  % White
    end
elseif (l < 82) && (b < -15)
    label = "B";  % Blue
elseif (abs(a) < 3) && (abs(b) < 3)
    label = "W";  % White again
else
    label = "R";  % Red
end
```

end

function[inputImage, outputImage] = processImage(path, sigmoid)
% processImage: enhance the image by gauss smooth
originalImage = imread(path);
% Appling a gaussian smoothing
inputImage = imgaussfilt(originalImage, sigmoid);
outputImage = rgb2lab(inputImage);
end

function [cropped,labImg] = autoCorrection(filename)
% autoCorrection: to corrected the image in LAB colour space

```
% Gets centroids of four circles for each image
movingPoints = findCircles(filename);
fixedPoints = [27.0282   26.5028; 26.7486  445.7151;
    445.3812   26.6354; 445.5667  445.7056];

% Rearrange coordinates so that the corners in each corresponding image match
movingPoints = cell2mat(orderPoints(movingPoints, fixedPoints));
%Conversion can be affine or projection
mytform = fitgeotrans(movingPoints, fixedPoints, 'affine');
out = imwarp(filename, mytform);

% Crop image and convert to LAB for output
cropSize = [480 480];
r = centerCropWindow2d(size(out), cropSize);
cropped = imcrop(out,r);
labImg = rgb2lab(cropped);
```

end

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

```
% Binarise the Image
BW = edge(rgb2gray(img), "Canny", [0.01, 0.9]);
objects = bwconncomp(BW);
CC = regionprops("table", BW, 'ConvexArea');
minFour = mink(CC.ConvexArea, 4);
% Locates the smallest 4 areas
idx = ismember(CC.ConvexArea, minFour);

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

centroids = zeros(numel(pxlList), 2);
for i=1:numel(pxlList)
    [r, c] = ind2sub(size(img), pxlList{i});
    centroids(i,:) = mean([c r]);
end
```

end

function [meanL, meanA, meanB] = meanLab(lab_Img)
  % meanLab: Returns the average LAB values

```
meanL = mean(lab_Img(:,:,1), 'all');
meanA = mean(lab_Img(:,:,2), 'all');
meanB = mean(lab_Img(:,:,3), 'all');
```

end

function answer = isTransformed(img)
% isTransformed: whether the Img can be transformed

```
gray_img = rgb2gray(img);
bin_img = edge(gray_img,'canny');
bin_img = bwmorph(bin_img,'thicken');

theta = -90:89;
[R,~] = radon(bin_img,theta);
[R1,~] = max(R);

theta_max = 90;
while(theta_max > 50 || theta_max < -50)
    [~,theta_max] = max(R1);
    R1(theta_max) = 0;
    theta_max = theta_max - 91;
end

if (theta_max ~= 0)
    answer = true;
else
    answer = false;
end
```

end

function outputPoints = orderPoints(movingPoints, fixedPoints)
% orderPoints: Patterns points based on their euclidean proximity

```
outputPoints = {[4, 2]};
% Make a duplicate of movingPoints.
valueArray = movingPoints;

for i = 1:length(movingPoints)
    minDistance = Inf;
    minDistanceIndex = 0;
    % Select entire row at index i
    p1 = fixedPoints(i, :);
    for j = 1:length(fixedPoints)
        p2 = fixedPoints(j, :);
        X = [p1;p2];
```

```
        currentDistance = pdist(X, 'euclidean');
        if currentDistance < minDistance
            minDistance = currentDistance;
            minDistanceIndex = j;
        end
    end
    % Maximise the selected point
    fixedPoints(minDistanceIndex, :) = [Inf -Inf];
    outputPoints{i, 1} = valueArray(minDistanceIndex, 1);
    outputPoints{i, 2} = valueArray(minDistanceIndex, 2);
end
```

end