Image Processing Report
Candidate No: 229538
Word Count: 2872

Contents

1. Importing Images from Disk

I first implemented a function to import and process any input image, designated by the path on disk. **processImage**, takes two parameters: the input path and a sigmoid value - this corresponds to the level of Gaussian smoothing applied to the image using MATLAB's **imgaussfilt**. I decided naively applying a relatively low level of smoothing to every image was the most straightforward approach to dealing with the inherent noise in the images. I also decided to have my function output the smoothed image in both RGB and LAB colour spaces; this was to allow for colour classification (LAB) and verification of correct classification via console output (RGB).

1.1. LAB Colour Space Rationale

Although RGB is typically what we are used to seeing on our screens, the RGB colour space has limitations that make this task more challenging when it comes to colour thresholding. Colours in RGB are a additive mixture of the three channels and are often badly effected by changes in brightness. One solution to the brightness problem would be to use the RG chromaticity colour space which normalises the colours by removing any information of intensity. More commonly, however, the solution to this is to transform the images from RGB to either HSV or LAB colour spaces. Whilst LAB and HSV are similar, and are both considered more true to life than RGB for colour perception, LAB is often preferred as it is perceptually linear. LAB also separates contrast from colour meaning that changes in illumination do not alter the colour channel values. This was ideal for the problem of colour classification because the saturation's often differed in each image and would have lead to a more complicated colour thresholding algorithm.

2. Creating a Colour Classifier

To begin with, I decided to focus on the "noise_x.png" images as these were the most straightforward and allowed me to refine the LAB thresholds for colour classification. Before I could classify the colours, I implemented an iterative approach to slice small squares from within each colour block. Because I only applied a low level of Gaussian smoothing, I decided that a small slice would be more reliable for

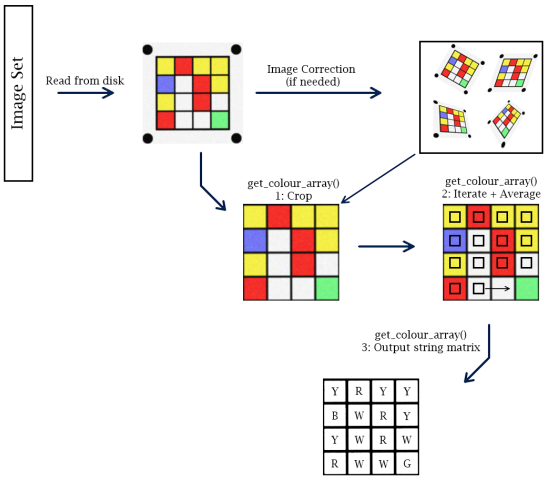


Figure 1. A simplified visual overview of my implemented image classification algorithm.

colour classification than a single pixel and easier to implement than finding the entire colour square to average across. The mean LAB values were then calculated within this localised areas using my **meanLab** function. My **classifyColour** function accepts the mean LAB values and using if-else logic blocks, is able to provide a classifier for the colours. I encapsulated the logic for image classification in my **getColourMatrix** function; this crops the image so it contains only the colour blocks, it then iterates through each block, classifies the colour and saves the output in a string array (Figure 1). The decision to crop the image was made when I was originally experimenting with ideas in Python, a language I am more comfortable with. I found it easier to find a reliable iterative loop when the image only contained 'valuable' colour blocks rather than including the excess clutter surrounding the colour matrix. Furthermore, I decided to keep the colour slice small to make the algorithm more robust in the case where the automatic correction did not perfectly square the image.

3. Rotated Images

3.1. First solution: Radon Transformation

I next focused on the rotated images "rot_x". My first problem was to identify when an image needed to be rotated; my first naive solution was exploiting that

the size of all the rotated images were greater than 480 x 480 - a filter on image size worked well for identifying rotated images. Following this, I needed an automated means to correct the rotation. I found an elegant solution to this [2] which utilises MATLAB's Radon transform. A Radon transform is "Hough-like" and is a common technique used in medical imaging [3]. It can be thought of as a mapping between Cartesian coordinates (x, y) to Polar coordinates (p, θ) where p is the distance from a point source and θ is the angle. By iterating through all angles (θ) and radii (p) found using MATLAB's **radon** function, this algorithm finds the maximum angle in which a straight line is detected between 50 and -50 degrees. Figure 2. illustrates how lines in the image are detected, appearing as bright spots. Using this maximum angle θ^*, we can correct the image using the MATLAB function **imrotate**.

3.2. Second solution: Geometric Transformation

The solution I settled on came organically through me solving the more difficult "proj1_x" and "proj2_x" images. When I realised my solution also worked for rotated images, I decided to use my own code - I will outline my implementation and difficulties in the following section.

4. Transformed Images

The main difficulty for this project was the automatic correction of the transformed "proj_x" images. Whilst using MATLAB's **cpselect** was the obvious option, I wanted to see how we might automate this.

4.1. Finding Four Control Points

In order to utilise MATLAB's **fitgeotrans** function, we need a minimum of four control points - here, the most intuitive implementation exploited the four black circles in each corner of the images. I experimented with thresholding to extract the dark circles, however I was not able to develop a robust system that could be applied to all images. Following this, I found that a Canny edge detector worked well here to find the circles as well as few other unwanted edges due to the distinct black border surrounding the colour matrix. By using **regionprops**, and by filtering out the four smallest objects according to their 'ConvexArea', I was able to isolate the circles. Whilst I experimented with KMeans to find centroids, I found it was simpler

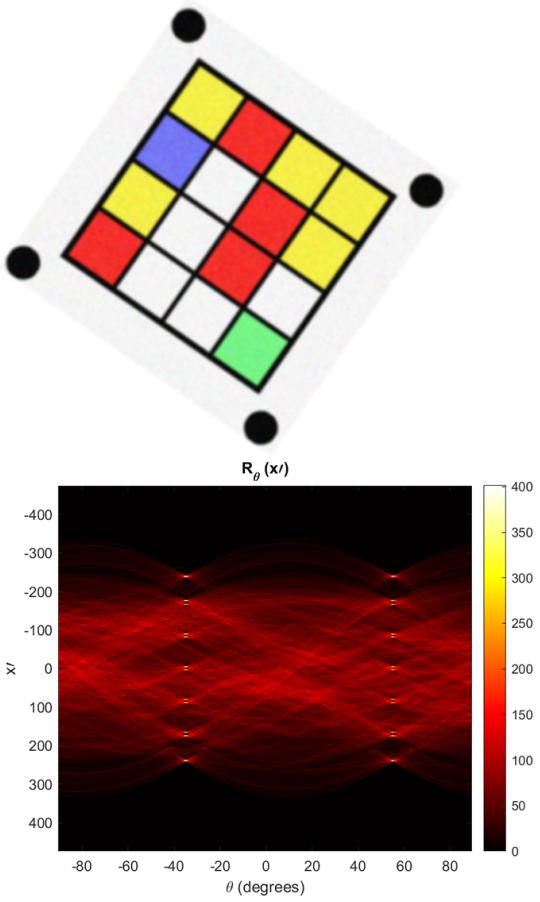


Figure 2. A visual example of MATLAB's Radon Transform on image "rot_1.png" as seen above. The brighter spots of colour indicate points of intersection. We can see 2 collections of points in this $p\theta$ space suggesting 2 collections of lines - 1 rotated ≈ -35 degrees counter-clockwise and the other collection of lines rotated ≈ 60 degrees clockwise.

to just find the average coordinates across all points for each circle. When provided a target image for transformation, my function will generate the centroids for this image; this acts as the 'moving points' within the **fitgeotrans** function. The 'fixed points' are hard coded but were derived from running my function on the "noise_1.png" image.

4.2. Ordering the Control Points

Whilst my method of finding control points was robust, the major difficulty was that if these points did not correctly match - that is top-left with top-left, bottom-right with bottom-right etc - then the geomet-

ric transformation often incorrectly warped the image. My first attempt at solving this was to order the coordinates with **sortrows**; whilst this worked for some images it largely failed. After considerable experimentation, I developed my **orderPoints** function. Here I exploit the fact that despite different transformations being applied, I could order the points by euclidean distance, that is, find which corner they were nearest too. This was made possible by the fact that our task was *'orientation invariant'* meaning that there is more than one viable solution. Whilst I experimented with different distance measures, I found Euclidean to be sufficiently fast and capable at ordering the points correctly. One extra consideration was to ensure that each corner could only have a single match to handle the possibility where a point was equidistant between two different corners. To achieve this, once a point had been matched to the nearest corner, I then maximise the distance between that point and every other corner. I achieved this by setting the selected point equal $(\infty, -\infty)$. In MATLAB, the keyword **Inf** can be used for this and represents numbers that are larger than MATLAB's *realmax*: $1.7977e + 308$. With correctly ordered points, the geometric transformation is a lot more reliable.

4.3. Cropping Images

After image correction had been applied, the output images were considerably larger than the standard 480 x 480. As a result, a simple solution was to apply a centred cropping window to extract the desired image. Whilst some of the cropped images appear extremely blurred, my colour classification function was robust enough to handle this.

4.4. Automatic Transformation Detection

Whilst I had developed a reliable system for the correction of the majority of the "proj_x" images, I wanted to develop logic to determine automatically whether an image required transformation - this would prevent unnecessary computation as I highlight later. Utilising parts of the previously mentioned rotational correction code which exploited Radon transforms, I was able to simplify my main function logic. Rather than having separate blocks for each type of transformed image, I utilised the fact that images that had not been transformed - the noise and org image sets - returned a maximum $\theta = 0$ from a Radon transform (Figure 3). As a result, I created the function **isTransformed** which applies a Canny edge detector followed by the Radon transform. The function returns a bool as to whether the image requires transformation based on the images maximum θ value.

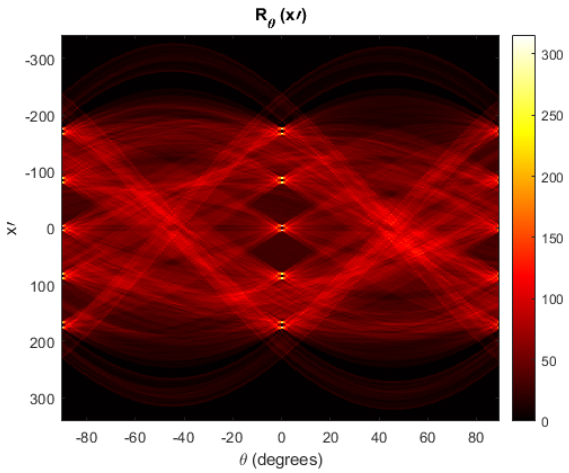


Figure 3. The visualised Radon transform for the "noise_1.png" image. As you can see, there are a single collection of points centred at 0 degrees - this indicates that the lines are perfectly square. I exploited this property to filter which images required geometric correction.

4.5. Noise_x, Org_x, Rot, Proj1_x and Proj2_x Image Sets

Using my **autoCorrection** function which encapsulates the above mentioned logic, it returns a corrected image in the LAB colour space to supply to my **getColourMatrix** function. These combined are able to correctly solve all the noise_x, org_x, rot_x, proj1_x and proj2_x images. It is worth noting that I tried both 'affine' and 'projective' transformations applied to these images and both types successfully handled all the required images. When applied to the rotated images, I noticed that using **fitgeotrans** versus the Radon rotation function produced output that was horizontally mirrored as I have illustrated in Figure 4.

5. Proj_x Image Set: Not Solved

These images were considerably more challenging that the previous images; they appear to combine several transformations, rotation, affine and projective.

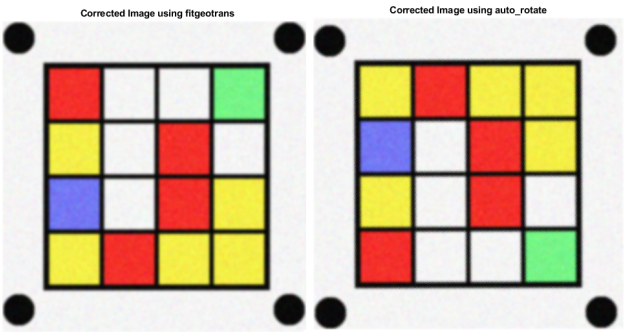


Figure 4. Two different solutions to solving rotational correction. The left-hand image is transformed using my algorithm which uses MATLAB's fitgeotrans function. On the right, the image has been corrected using code which incorporates the rotation correcting algorithm from [2]. We can see that they are both valid solutions but they are also horizontally mirrored.

Even with the use of the manual **cpselect** for selecting control points, I was unable to solve the proj_x images. Some transformations, when combined, resulted in enormous images - MATLAB reported one to be 34GB - whilst others caused MATLAB to simply crash. Below I have outlined the progress I made and the issues I encountered.

5.1. Tilt correction

All of the images in this set appeared to have a tilt transformation applied to them which has the following transformation matrix:

$$\begin{bmatrix} 1 & 0 & E \\ 0 & 1 & F \\ 0 & 0 & 1 \end{bmatrix}$$

Where E and F influence the vanishing point [1]. As a result, I tried to recover the vanishing point using Hough line detection with limited success - I will use the image "Proj_1" from image set 2 to illustrate this. Figure 5 shows the original Proj_1 image alongside the image after Hough line detection. We can see that the image needed to be zoomed out sufficiently to see the points of intersection. Because MATLAB's **houghlines** function returns lines as two endpoint coordinates, I needed to derive the line equation using MATLAB's **polyfit** function. By finding the gradient and intercept, I was able to firstly plot the lines, as seen in Figure 5, and then find the points in which these lines

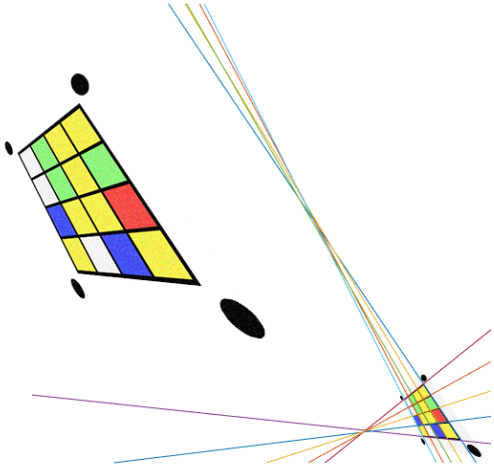


Figure 5. Proj_1 is shown on the left with an example of my Hough line finder.

intersect. The theory behind this algorithm is that the vanishing point will be the point with the highest number of intersections.

In order to find the line intersections, I used the following equations:

$$\text{X intercept} = \frac{c_1 - c_2}{m_1 - m_2}$$
$$\text{Y intercept} = m_1 x_0 + c_1$$

Where c_1 and c_2 are the line intercepts and m_1 and m_2 are the respective line gradients. Whilst this was an effective method for finding all the points of intersection, as seen in Figure 6, it still needed further refinement. Firstly, it found all intersecting points rather than the points which had the most intersections. Secondly, we can see in Figure 6 that there is a clear convergence point to the left-hand side where as the convergence point above the colour matrix is less well defined. As a result, it will be harder to determine the vanishing point for these 'scattered' intersecting points.

Unfortunately, I was unable to find a robust method to solve the Proj_x image set, even using manual control point selection using **cpselect**. I think the major obstacle is recovery of the tilt transformation information in order to accurately reverse the transformation.

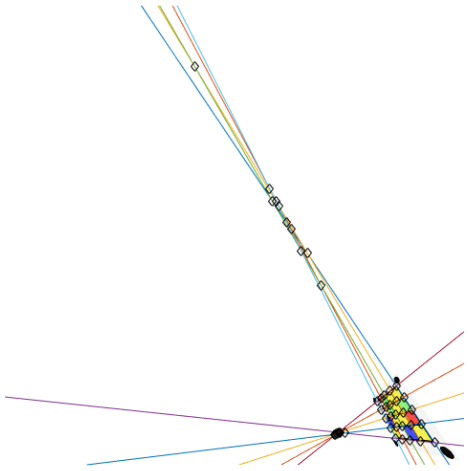


Figure 6. An illustration of a function that finds all the points of intersection for the lines provided. A point of intersection is plotted as a hollow black square.

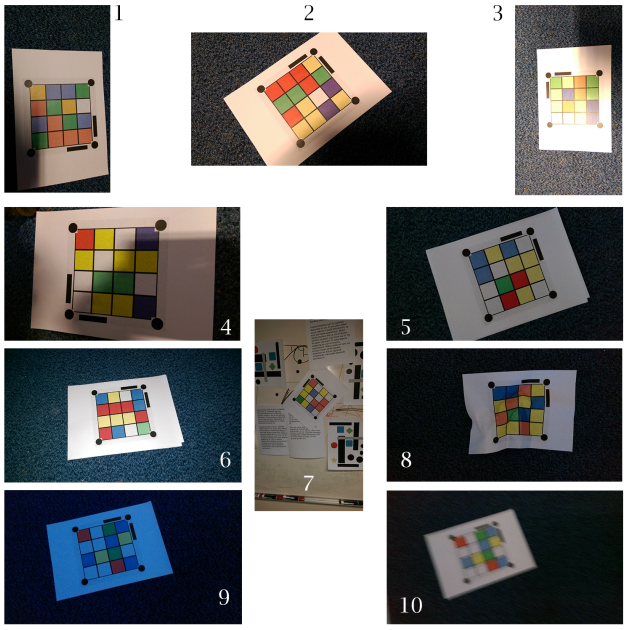


Figure 7. A collage of the 'real' images that we were provided with. Whilst the .zip file had numbered them 0032-0044, there are only 10 images and I have labeled them accordingly.

6. Real Images: Discussion

Whilst our report was based on classifying the computer generated colour matrices in image set 1 or 2, it is worth discussing how we would have to adapt my algorithm to cope with 'real' images. Despite the fact the answer can often be simplified to 'take a superior image', some of these issues may be unavoidable and as such, a discussion of their difficulties and possible solutions is worthwhile. Within the provided .zip file the images are numbered as 0032 - 0044 despite there being only 10 images; from now on, I will refer to the images as image 1 - 10 as seen in Figure 7.

6.1. Variations in Lighting

A significant difference can be seen throughout the ten images in terms of their overall lighting. For example, in images 1, 2 and 4, we see a significantly darker shadow, most likely caused by the camera occluding the light source. Conversely, image 3 has a much brighter spot caused by a non-diffuse light source as well as a shadow. In terms of lighting balance, image 6 has the most uniform lighting with no distinct bright or dark spots - it appears as if the camera's flash was used here. Another lighting anomaly can be seen in image 9 - here there is a considerable blue tint perhaps caused by significantly lower light or a hardware issue with the cameras sensor. Whilst the colours in image 9 are relatively uniform, the blue hue would impact colour classification accuracy. More generally, with a range of lighting conditions the naive 'if-else' logic blocks would need considerable refinement and the images would require suitable pre-processing. Whilst saturation stretching is an option, using the LAB colour space would be essential as this is a illumination invariant colourway.

6.2. Variations in Orientation

Whilst the colour matrices are mostly central within each image, they all appear to vary in their orientation. Image 3 appears to be the most 'square-on', but the majority of the images have been taken at an angle. This causes images to be rotated to varying degrees as well as what appear to be projective and affine transformations. My algorithm is comfortable with simple rotation correction as well as images that are not heavily skewed - like the proj_x images - so in theory, adapting it to a real image may not be too complex. Again, any tilt transformations would be the most difficult - my attempt at solving this would perform even

poorer here as there are more distinct lines introduced by the edges of the white paper against a dark background.

6.3. Distortion in the Images

Whilst the images in image set 1 and 2 had a low level of noise, this could be largely circumvented through a small amount of Gaussian smoothing. The majority of the real images have the inherent noise associated with the camera being used - this noise is exacerbated by low light levels. Two images, however, have a significantly higher level of image distortion; images 8 and 10. Image 10 is heavily blurred, most likely caused by an unstable camera as the picture was taken. Whilst the blurring is not an inherent problem for classifying the colours within the matrix (this is in fact what my Gaussian smoothing did), my method of finding the black circles within my transformation correction algorithm would be considerably harder. Image 8 on the other hand is a unique problem to the real images with the paper being physically deformed, causing each colour square to be a slightly different shape and size. In theory, because my algorithm takes the mean colour value from a small centred square, it could deal with a small level of deformation. If the colour classification was more sensitive, that is, classifying differences between light and dark green for example, the creases in the paper may significantly alter the true colour values by introducing shadows. Whilst we were unable to use techniques such as phase correlation in our algorithm due to the lack of texture in our images, it would be interesting to see if this 'crumpled' image would respond better to these techniques.

6.4. Automatic Transformation Correction

In my algorithm, I relied heavily on the ability to locate the dark circles, find their centre, and use these as control points for geometric transformations. Whilst this is still theoretically possible on some of the images, most images would produce unwanted behaviour. For example, images 1, 3 and 4 have some of their circles in significantly different light levels making thresholding difficult. The deformed image 8 may lead to a 'less circular' circle - as I filtered the circles by their convex area, this may also be impacted. As mention previously, high levels of blurring make edge detection considerably more challenging when trying to locate the circles. The most obvious example of issues with my algorithm is highlighted by image 7 which is surrounded by unwanted 'clutter'. This is made worse by the presence of other shapes, including circles, that would compete with the circles on the desired image. The best hope for an image like this would be perhaps to hope that your desired image is centred and crop the image accordingly - this obviously makes automation increasingly difficult.

7. Algorithm Performance

Whilst 'performance' in this task was primarily judged on the accuracy of colour classification, I wanted to briefly cover the computational performance of my algorithm. Unsurprisingly, the most computationally expensive part of the algorithm is the correction of warped images, specifically my **autoCorrection** function and the calls to **findCircles** - I used MATLAB's profiler to quantify these differences. In my final code, my **colourMatrix** function will include plotting of the original and corrected image for comparison, however, when examining run time I have excluded the plotting time for a more accurate run time. Figure 8 is a comparison of performance between an image that required transformation and one that did not. We can see that images that require correction take almost double the amount of time to execute. Even with the plotting of images, however, my algorithm still takes less than a second to return a color matrix.

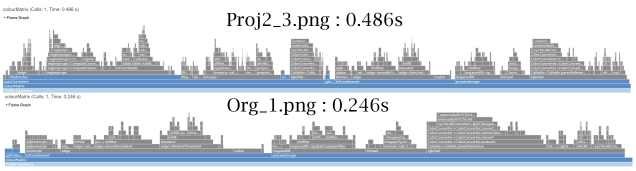


Figure 8. A comparison of MATLAB's profiler run on two different images. Proj2_3 requires transformation correction whilst Org_1 does not. We can see the significant increase in run time (almost double) when the image requires correction.

8. Results

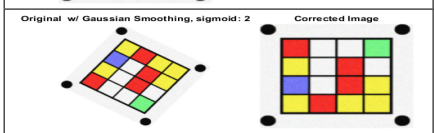
Image	String Array	Comments
	resultMatrix = 4×4 string "B" "Y" "Y" "B" "W" "R" "Y" "G" "R" "Y" "Y" "B" "G" "Y" "W" "R"	• noise_1.png
	resultMatrix = 4×4 string "Y" "B" "R" "G" "G" "G" "W" "Y" "G" "B" "B" "W" "R" "Y" "B" "Y"	• noise_2.png
	resultMatrix = 4×4 string "R" "R" "R" "Y" "B" "B" "Y" "W" "G" "B" "Y" "B" "R" "B" "W" "W"	• noise_3.png
	resultMatrix = 4×4 string "Y" "G" "Y" "W" "R" "W" "W" "W" "R" "W" "G" "G" "R" "G" "B" "G"	• noise_4.png
	resultMatrix = 4×4 string "R" "Y" "R" "Y" "B" "G" "Y" "Y" "W" "Y" "B" "G" "R" "Y" "B" "Y"	• noise_5.png
	resultMatrix = 4×4 string "B" "Y" "W" "Y" "Y" "W" "W" "R" "W" "Y" "R" "R" "G" "W" "W" "R"	• org_1.png
	resultMatrix = 4×4 string "W" "G" "Y" "G" "W" "R" "B" "W" "B" "B" "W" "W" "G" "Y" "G" "W"	• org_2.png
	resultMatrix = 4×4 string "G" "R" "Y" "G" "Y" "B" "G" "G" "B" "R" "R" "W" "G" "Y" "W" "Y"	• org_3.png
	resultMatrix = 4×4 string "W" "W" "B" "Y" "G" "G" "B" "R" "W" "W" "G" "Y" "G" "W" "Y" "R"	• org_4.png
	resultMatrix = 4×4 string "R" "Y" "G" "Y" "W" "G" "G" "G" "Y" "W" "G" "R" "G" "G" "Y" "W"	• org_5.png
	resultMatrix = 4×4 string "R" "Y" "G" "Y" "W" "G" "G" "G" "Y" "W" "G" "R" "G" "G" "Y" "W"	• org_5.png
	resultMatrix = 4×4 string "R" "W" "W" "G" "Y" "W" "R" "W" "B" "W" "R" "Y" "Y" "R" "Y" "Y"	• rot_1.png

Table 1. My Algorithm Results

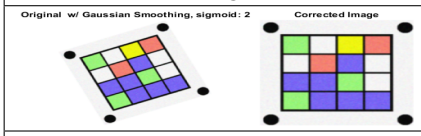
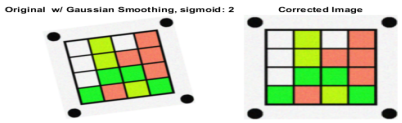
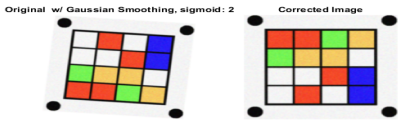
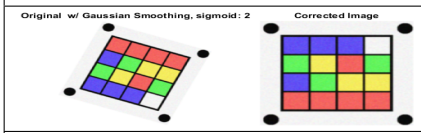
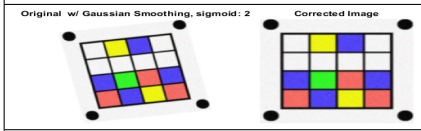
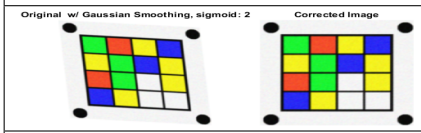
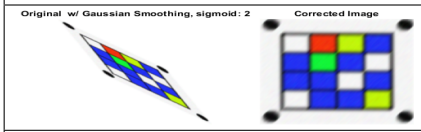
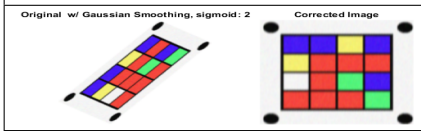
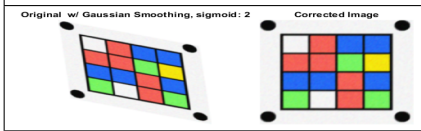
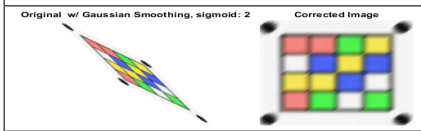
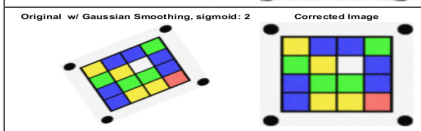
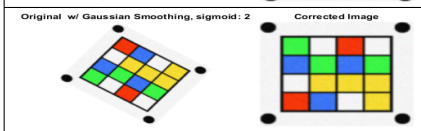
Image	String Array	Comments
	resultMatrix = 4×4 string "G" "W" "Y" "R" "W" "R" "B" "W" "B" "B" "G" "W" "G" "B" "B" "B"	• rot_2.png
	resultMatrix = 4×4 string "W" "G" "W" "R" "W" "G" "R" "R" "W" "G" "G" "R" "G" "R" "G" "G"	• rot_3.png • Appear to be 2 different 'greens' in this matrix. My algorithm does not discriminate between them
	resultMatrix = 4×4 string "R" "R" "G" "Y" "G" "Y" "Y" "W" "W" "W" "R" "B" "W" "R" "W" "B"	• rot_4.png • Yellows here arguably look more orange
	resultMatrix = 4×4 string "B" "B" "B" "W" "G" "Y" "R" "G" "B" "G" "Y" "Y" "R" "R" "R" "R"	• rot_5.png
	resultMatrix = 4×4 string "W" "Y" "B" "W" "W" "W" "W" "W" "B" "G" "R" "B" "R" "B" "Y" "R"	• proj1_1.png
	resultMatrix = 4×4 string "G" "R" "Y" "B" "Y" "G" "B" "Y" "R" "G" "W" "Y" "B" "Y" "W" "W"	• proj1_2.png
	resultMatrix = 4×4 string "W" "R" "G" "B" "B" "G" "B" "W" "B" "B" "W" "B" "W" "B" "B" "G"	• proj1_3.png
	resultMatrix = 4×4 string "B" "B" "Y" "B" "Y" "R" "R" "R" "W" "R" "G" "B" "R" "R" "R" "G"	• proj1_4.png
	resultMatrix = 4×4 string "W" "R" "B" "B" "R" "R" "G" "Y" "B" "B" "R" "B" "G" "W" "R" "G"	• proj1_5.png
	resultMatrix = 4×4 string "R" "R" "G" "Y" "W" "B" "Y" "B" "Y" "Y" "B" "W" "R" "G" "W" "G"	• proj2_1.png
	resultMatrix = 4×4 string "Y" "B" "B" "G" "G" "Y" "W" "B" "B" "G" "G" "B" "B" "Y" "Y" "R"	• proj2_2.png
	resultMatrix = 4×4 string "G" "W" "R" "W" "B" "G" "B" "G" "W" "Y" "Y" "Y" "R" "B" "W" "Y"	• proj2_3.png

Table 2. My Algorithm Results

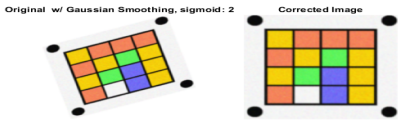
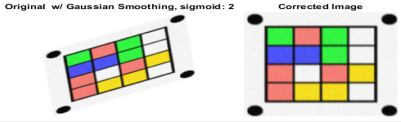
Image	String Array	Comments
	resultMatrix = 4×4 string "Y" "R" "R" "R" "R" "Y" "G" "Y" "Y" "G" "B" "Y" "R" "W" "B" "Y"	• proj2_4.png • Reds appear more orange. My classifier still correctly classified it as red. Compare this orange to rot_4 which was classified correctly as yellow
	resultMatrix = 4×4 string "G" "R" "G" "W" "B" "B" "G" "W" "R" "W" "R" "Y" "R" "Y" "Y" "W"	• proj2_5.png

Table 3. My Algorithm Results

9. Bibliography

References

[1] MATLAB. Matrix representation of geometric transformations. *https://uk.mathworks.com/help/images/matrix-representation-of-geometric-transformations.html*, 2021. 5

[2] A. Omidvarnia. Image rotation correction. *https://www.mathworks.com/matlabcentral/fileexchange/16968-image-rotation-correction), MATLAB Central File Exchange*, 2021. 3, 5

[3] A. I. Zayed. A new perspective on the role of mathematics in medicine. *Journal of advanced research*, 17:49–54, 2019. 3

Appendices

A. Main Function

A.1. colourMatrix

This function is considerably faster without the plotting functionality

```
function result = colourMatrix(targetImage)
    % colourMatrix: Returns a string array of colour classifications
    % PARAMS: targetImage: path to image on disk
    % RETURNS: string array

    % Import image: sigmoid = level of smoothing
    sigmoid = 2;
    [rgbImg, labImg] = processImage(targetImage, sigmoid);

    % Plotting the original image for comparison
    figure;
    subplot(1, 2, 1)
    imshow(rgbImg)
    title("Original " + " w/ Gaussian Smoothing, sigmoid: " + sigmoid)

    if (isTransformed(rgbImg))
        [rgbImgCorrected , labImgCorrected] = autoCorrection(rgbImg);

        subplot(1, 2, 2)
        imshow(rgbImgCorrected)
        title("Corrected Image")

        result = getColourMatrix(labImgCorrected);
    else
        % No plot here because image is not transformed
        result = getColourMatrix(labImg);
    end
end
```

B. Auxiliary Functions

B.1. processInput

```
function [inputImage, outputImage] = processImage(path, sigmoid)
   % processImage: Returns smoothed image located at path
   originalImage = imread(path);
   % Applies gaussian smoothing. High sigmoid = High smoothing
   inputImage = imgaussfilt(originalImage, sigmoid);
   outputImage = rgb2lab(inputImage);
end
```

B.2. isTransformed

N.B In MATLAB not equal is written as $\sim=$ (tilde equals). Latex has interpreted this as $\neq$

```
function answer = isTransformed(img)
   % isTransformed: Returns TRUE if the image requires transformation
   %  Accepts an image in the RGB colour space, applies a radon transform
   %  and determines whether the image needs correction based on max theta
   gray_img = rgb2gray(img);
   bin_img = edge(gray_img,'canny');
   bin_img = bwmorph(bin_img,'thicken');

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

   theta_max = 90;
   while(theta_max > 50 || theta_max < -50)
      % R2: Maximum Radon transform value over all angles.
      [R2,theta_max] = max(R1);
      % Remove element 'R2' from vector 'R1', so that other maximum
      % values can be found.
      R1(theta_max) = 0;
      theta_max = theta_max - 91;
   end

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

B.3. autoCorrection

```
function [cropped,labImg] = autoCorrection(targetImage)
   % autoCorrection: Returns corrected image in LAB colour space: labImg
   %                 Returns cropped RGB image for comparison if required

   % Get centroids of 4 circles in each image
   movingPoints = findCircles(targetImage);
   % fixedPoints are derived from findCircle(noise_1.png).
   fixedPoints = [27.0282 26.5028; 26.7486 445.7151;
      445.3812 26.6354; 445.5667 445.7056];

   % Re-order coordinates so that the corners in each respective image are
   % matched
   movingPoints = cell2mat(orderPoints(movingPoints, fixedPoints));
   % Transform here can be affine or projective
   mytform = fitgeotrans(movingPoints, fixedPoints, 'affine');
   out = imwarp(targetImage, mytform);

   % Resize / crop image and convert to LAB for output
   cropSize = [480 480];
   r = centerCropWindow2d(size(out), cropSize);
   cropped = imcrop(out,r);

   labImg = rgb2lab(cropped);
end
```

B.4. getColourMatrix

N.B Here in latex, my variable deltaX and deltaY has been shown as ΔX and ΔY

```
function results = getColourMatrix(img)
   % getColourMatrix: Returns string array of classified colours

   % Crop centre colour matrix
   coloursCropped = img(75:405, 75:405, :);
   % Works with a char array. Difference is in output
   % "BWYR" rather than "B" "W" "Y" "R"
   results = string(zeros(4, 4));
   % Fixed coordinates to iterate through colour matrix
   coords = [10, 100, 200, 260];
   for i = 1:length(coords)
   ΔY = coords(i);
      for j = 1:length(coords)
         ΔX = coords(j);
         squareSlice = coloursCropped(20+ΔY:25+ΔY, 20+ΔX:25+ΔX, :);
         [l, a, b] = meanLab(squareSlice);
         myLabel = classifyColour(l, a, b);
         results(i, j) = myLabel;
      end
   end
end
```

B.5. classifyLab

If else logic could be streamlined but this was sufficient for the task and did not bottleneck algorithm performance

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

   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
```

B.6. findCircles

N.B The line starting with: pxlList(¬idx) should be pxlList(∼idx). Tilde is a special character in Latex but is used in MATLAB as the negation operator

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

   % Binarise Image + find connected components
   BW = edge(rgb2gray(img), "Canny", [0.01, 0.9]);
   objects = bwconncomp(BW);

   % Returns the convex area of every connected component
   CC = regionprops("table", BW, 'ConvexArea');
   minFour = mink(CC.ConvexArea, 4);
   % Bool array to locate the smallest 4 areas
   % In theory the smallest areas should be all the circles
   idx = ismember(CC.ConvexArea, minFour);

   pxlList = objects.PixelIdxList;
   pxlList(¬idx) = []; % Deletes all false (0) elements

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

B.7. meanLab

Could have added a logic check to assert that input image was in LAB colour space

```
function [meanL, meanA, meanB] = meanLab(labImg)
    % meanLab: Returns the average LAB values for provided LAB image

    % 'all' = computes mean over all elements in channel
    meanL = mean(labImg(:,:,1), 'all');
    meanA = mean(labImg(:,:,2), 'all');
    meanB = mean(labImg(:,:,3), 'all');
end
```

B.8. orderPoints

```
function outputPoints = orderPoints(movingPoints, fixedPoints)
    % orderPoints: Matches pairs of points based on their euclidean proximity
    % This means top-left with top-left, bottom-right with bottom-right etc.

    outputPoints = {[4, 2]};
    % Create a copy of movingPoints. We can then retrieve the desired
    % original values from this copy even when I maximise points to
    % [Inf -Inf]
    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 point that has been selected
        % This prevents re-selection
        fixedPoints(minDistanceIndex, :) = [Inf -Inf];
        outputPoints{i, 1} = valueArray(minDistanceIndex, 1);
        outputPoints{i, 2} = valueArray(minDistanceIndex, 2);
    end
end
```

C. Extra Python Code

To extract the images for my results table, I took monitor screenshots and then developed two small Python scripts to automatically rename, crop and split images into the respective colour matrix and string matrix for each solved image.

C.1. Renaming Script

```
from PIL import Image
import glob

"""
This script opens, renames and saves the images in a different directory before ...
    they are cropped by my "Results Cropping Script"
Takes image called "MATLAB R2020b - academic use 15_04_2021 17_01_36"
Returns image called "noise_1.png"
"""

img_types = ['noise', 'org', 'rot', 'proj1', 'proj2']
filenames = []

for img in img_types:
    for i in range(1, 6):
        filenames.append(f"{img}_{i}")

for i, filepath in enumerate(glob.iglob(r"ORIGINAL SCREENSHOT LOCATION\*.png")):
    temp_img = Image.open(filepath)
    temp_img.save(rf"NEW SCREENSHOT LOCATION\{filenames[i]}.png")
```

C.2. Cropping Script

```
from PIL import Image
import glob


def get_results(file):
    """
    Given a screenshot of my MATLAB window (2560 x 1377 - 1440p), this crops the ...
        image to only contain the MATLAB script output, then individually crops ...
        each 'useful' part: the colour matrix and the results matrix
    """
    img = Image.open(file)
    # Slices path to only contain file name
    # Will need changing depending on your selected path
    filename = img.filename[84:-4]

    # Crop majority of screenshot
    left = 1430
    top = 395
    right = 2000
    bottom = 835
    first_crop = img.crop((left, top, right, bottom))

    # 570 x 440
    # Colour Matrix Cropping
    cm_left = 25
    cm_top = 20
    cm_right = 510
    cm_bottom = 225
    cm_crop = first_crop.crop((cm_left, cm_top, cm_right, cm_bottom))

    # Result Matrix Cropping
    rm_left = 5
    rm_top = 355
    rm_right = 295
    rm_bottom = 430
    rm_crop = first_crop.crop((rm_left, rm_top, rm_right, rm_bottom))

    # Save images to respective folders
    # Working on windows machine so may have to change path formatting
    cm_crop.save(rf"YOUR PATH\Colour Matrices\{filename}_CM.png")
    rm_crop.save(rf"YOUR PATH\Result Matrices\{filename}_RM.png")


PATH = r"SCREENSHOT PATH\*.png"

for filepath in glob.iglob(PATH):
    get_results(filepath)
```