Crousework Report of Image Processing

Bo Hu
Candidate No. 285585

May 30, 2024

1 Methodology

I selected the Images1 dataset for processing. I developed a function, $result = colourMatrix(filename)$, which reads images from Images1 and returns a matrix representing the color pattern of the image. The functionality of this function has been verified to achieve 100% accuracy in color pattern recognition, with the entire process being fully automated, eliminating the need for manual parameter adjustments per image. The comprehensive workflow of the $colourMatrix$ function is depicted in Figure 1.

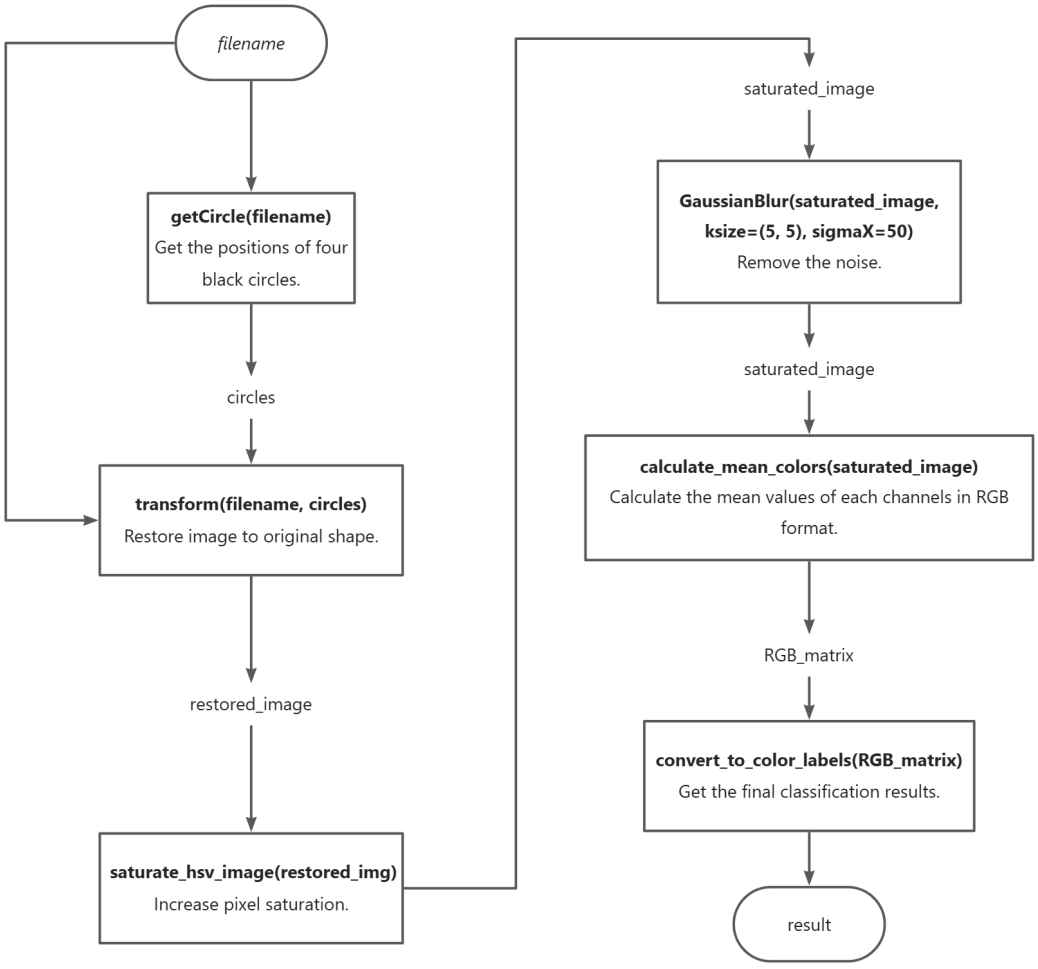


Figure 1: The breakdown process of the function $result = colourMatrix(filename)$

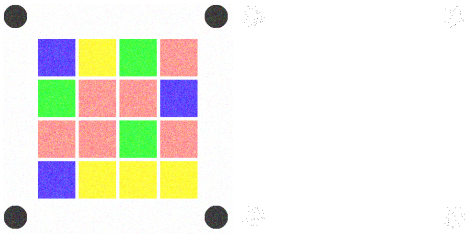
1.1 Get Positions of Four Black Circles

The initial step in the *colourMatrix* function entails acquiring coordinates of four black circles in the image, which facilitate subsequent shape restoration. Given that conversion of an RGB image to grayscale results in lower pixel intensity values for originally black regions compared to their colored counterparts, the first operation involves converting the original image into grayscale. Subsequently, thresholding is applied to the grayscale image to isolate regions of black pixels, serving as a preliminary selection of coordinates for the black circles present in the original image.

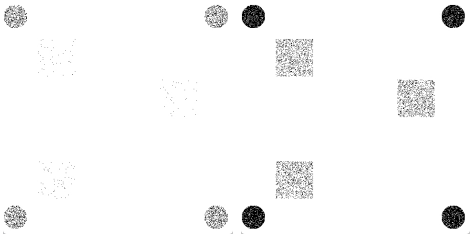
Filter Black Pixels

Through experimentation, I encountered challenges in applying a uniform threshold to effectively segment black circle regions across all images in the Images1 dataset. This difficulty arises due to variations in image content; specifically, some images contain grid squares with notably dark colors that, upon conversion to grayscale, also exhibit low pixel intensity values, thereby complicating distinction from black circle areas at higher threshold settings. Conversely, if the threshold is set too low, a substantial portion of pixels within the black circle regions of some images are inadvertently excluded. Figure 2 shows the effect of different thresholds on black pixel filtering.

Ultimately, I elected to set the threshold to 20. Prioritizing the exclusion of grid square pixels over achieving a more pronounced differentiation of black circle regions led to this decision. Consequently, a more stringent, lower threshold was employed for the selective filtering of black pixels, yielding the outcome depicted in Figure 3.

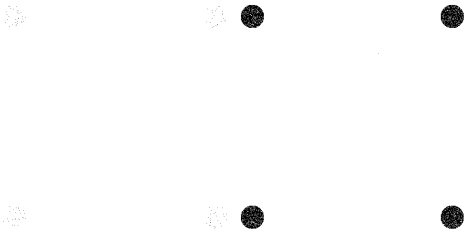


(a) noise_1.png (b) threshold = 20

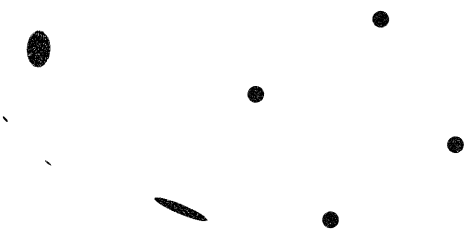


(c) threshold = 50 (d) threshold = 80

Figure 2: The results of thresholding on the image noise_1.png at values of 20, 50, and 80 reveal distinct outcomes. A lower threshold effectively preserves pixels within the black circles but inadvertently discards a significant portion of the desired pixels within these circles. Conversely, higher thresholds retain numerous pixels from the grid squares, which ideally should be filtered out, indicating a compromise in selective segmentation of the targeted black circle regions.



(a) noise_1.png (b) org_1.png



(c) proj_2.png (d) rot_1.png

Figure 3: Applying thresholding with a value of 20 to the image results in a segmentation where predominantly pixels corresponding to the black circles are retained. However, this comes at the expense of also excluding a considerable number of desirable pixels within those circles.

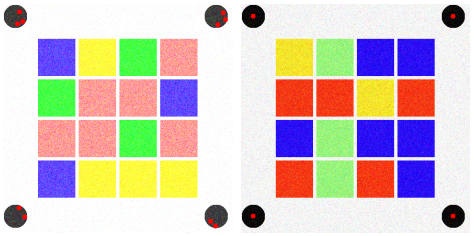
Get Four Center Positions Via Clustering Algorithms

Within the $find\_dense\_black\_clusters$ function depicted in Figure 1, I employ the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering algorithm to identify and localize dense clusters of black pixels in the image. DBSCAN, a density-based spatial clustering method, excels at detecting clusters of arbitrary shape and efficiently discarding noise points. Parameters $eps = 5$ and $min\_samples = 3$ are used, implying that two points are considered part of the same cluster if they are within 5 pixels of each other

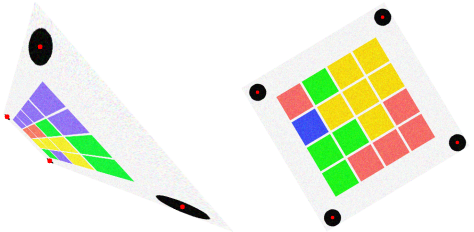
and each has at least 3 such neighboring points.

Given that the uniform threshold applied in the preceding step yields varying pixel segmentation outcomes across different images, application of the DBSCAN algorithm for clustering black pixel clusters also produces divergent results. Some images undergo perfect clustering around the centers of the four black circles, while others yield more than four cluster centers, indicating the presence of multiple cluster centroids within a single black circle due to the initial segmentation variability. The results are depicted in Figure 4.

To further categorize the central coordinates of previously identified dense black pixel regions (clusters) into four distinct classes and ascertain the centroid of each category, I subsequently applied the k-means clustering algorithm to the DBSCAN clustering outputs. By setting $n\_clusters = 4$, the intention is to subdivide the cluster centroids into precisely four separate categories. The results are depicted in Figure 5.

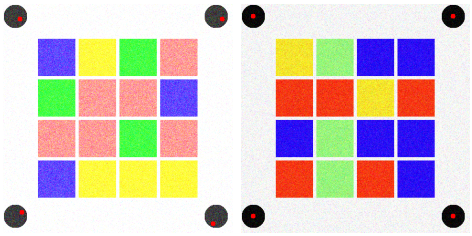


(a) noise_1.png (b) org_1.png

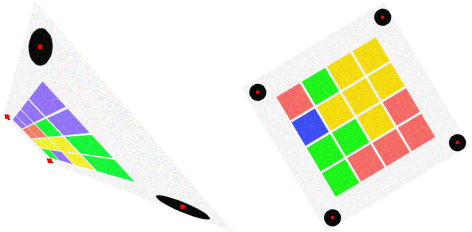


(c) proj_2.png (d) rot_1.png

Figure 4: Red dots denote the cluster centroids obtained after employing the DBSCAN algorithm. As illustrated in (a), instances arise where a unique cluster center within a black circle cannot be singularly identified.



(a) noise_1.png (b) org_1.png



(c) proj_2.png (d) rot_1.png

Figure 5: Following the refinement process using k-means clustering, each black circle ultimately contains a single red dot, successfully representing the centroid of the black circle as determined by the clustering.

1.2 Restore Images to Original Shape

After obtaining the central coordinates of the four black circles through clustering, we proceed to restore the perspective-transformed image to its original shape, using these circles as reference points. First the coordinates of these points are ordered clockwise.

In the *calculate_center* function depicted in Figure 1, the centroid of the four points is computed. The *polar_angle* function depicted in Figure 1 determines the polar angle of each point relative to the centroid.

The polar angle θ_i for each of four points $P_i(x_i, y_i)$ with respect to the centroid (x_{center}, y_{center}) is calculated by the atan2 function:

$$\theta_i = \text{atan2}(y_i - y_{center}, x_i - x_{center})$$

Here, $\text{atan2}(y_i - y_{center}, x_i - x_{center})$ yields the angle in radians from the positive x-axis to the point P_i, spanning from 0 to 2π. These computed polar angles serve as the pivotal criteria for subsequently ordering the points in a clockwise manner.

The *sort_points_clockwise* depicted in Figure 1 function then arranges these four points in clockwise order, serving as the reference points post-perspective transformation. Similarly, the reference points in the original image are also arranged clockwise: $[(0, 0), (0, 480), (480, 480), (480, 0)]$.

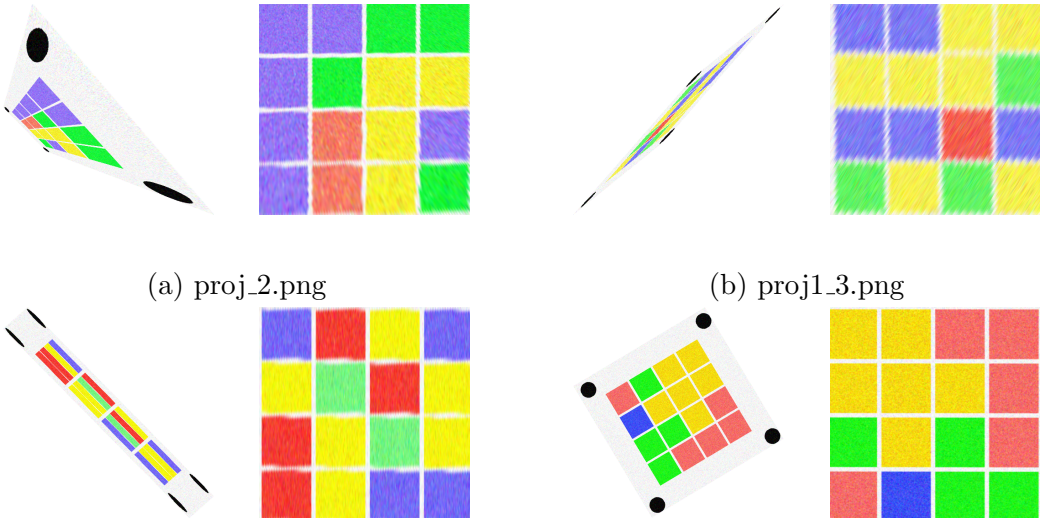
Utilizing OpenCV's $getPerspectiveTransform$ function, the inverse perspective transformation matrix, m, is calculated based on the coordinates before and after transformation. This matrix facilitates reversing the perspective transformation, reverting the image back to its initial state. Thereafter, the inverse perspective transformation is executed on the image using the derived matrix m and $warpPerspective$ function, restoring its original form.

Finally, the restored image is cropped to retain only the region spanning pixel positions $(50, 50)$ to $(430, 430)$, encompassing the color grid area, thereby facilitating subsequent color identification steps. The final results of restoration are depicted in Figure 6.

1.3 Identify Color of Blocks

Following the restoration of the image to its original form and cropping to isolate the color grid, the grid is further divided into 4×4 smaller squares for individual color identification within each.

Initiating the process, the input RGB image is transformed into the HSV (Hue, Saturation, Value) color space using OpenCV's $cvtColor$ function, as HSV is more conducive to adjusting color saturation compared to RGB. The



(a) proj_2.png (b) proj1_3.png

(c) proj1_1.png (d) rot_1.png

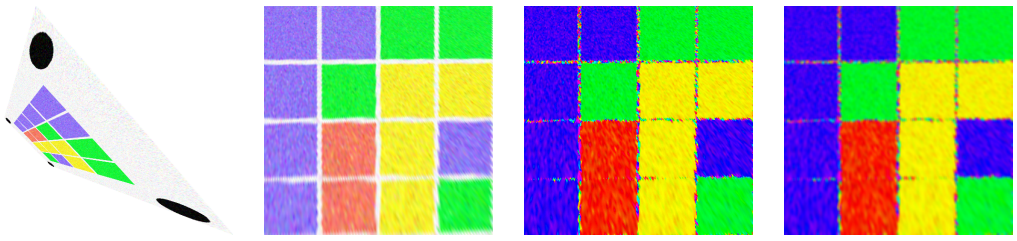
Figure 6: The results after image restoration and cropping. Notably, even severely distorted original images can attain significantly improved recovery outcomes.

HSV image is then decomposed into three separate channels – Hue (H), Saturation (S), and Value (V) – using OpenCV's *split* function, with the saturation of every pixel maximized to enhance color purity and vividness. Subsequently, the altered HSV image is converted back to the BGR format.

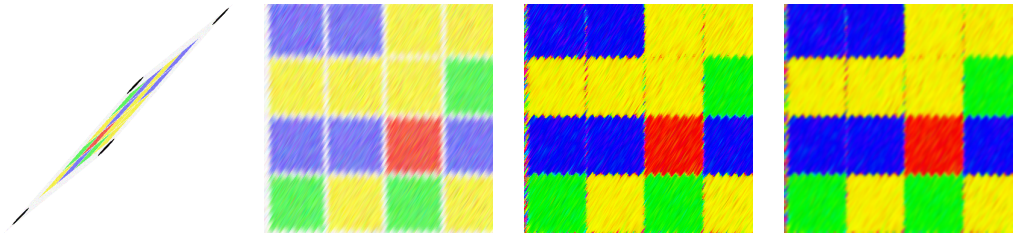
A Gaussian blur is applied to the image post-saturation adjustment to mitigate noise present in the image. The resultant processed image, as depicted in Figure 7, evidences these enhancements.

The function *calculate_mean_colors* depicted in Figure 1 segments the input RGB image into 4×4 grids and computes the mean value of each color channel (Red, Green, Blue) across all pixels within each grid, subsequently returning these averages in a matrix format.

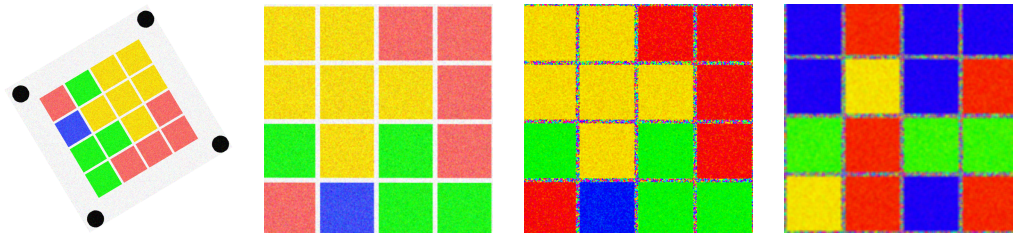
The *convert_to_color_labels* function depicted in Figure 1, which takes as input the matrix output from *calculate_mean_colors*, categorizes each grid based on the mean RGB values into one of the following classes: red (r), blue (b), green (g), yellow (y), or white (w), and outputs this classification in a matrix form. The algorithm for color classification is shown in Algorithm 1.



(a) proj_2.png



(b) proj1_3.png



(c) rot_1.png

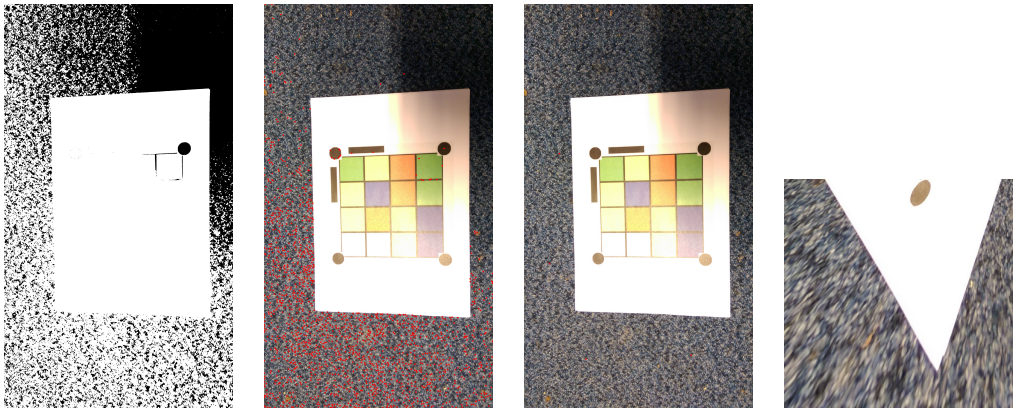
Figure 7: From left to right, the images are as follows: the original image, the restored image, the image with increased saturation, and the image subjected to Gaussian smoothing.

Algorithm 1 Color Identification Algorithm

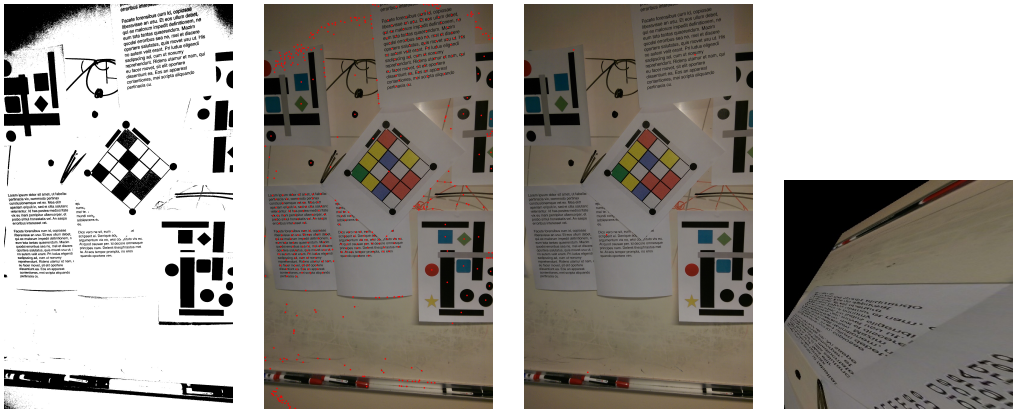
```
if b > 150 & r < 150 & g < 150 then
    return 'r'
else if r > 150 & g < 150 & b < 150 then
    return 'b'
else if g > 150 & r < 150 & b < 150 then
    return 'g'
else if g > 150 & b > 150 then
    return 'y'
else
    return 'w'
end if
```

2 Comment on Real Photos

My *colourMatrix* function fails to process real photographs from the Photos dataset. It encounters difficulties at the initial step of identifying coordinates of four fiducial points around color regions, as illustrated in Figure 8. These images, due to high-intensity illumination and cluttered backgrounds, prove resistant to thresholding techniques aimed at isolating pixels within the black circular regions, thereby hindering successful segmentation.



(a) IMAG0034.jpg



(b) IMAG0038.jpg

Figure 8: Each row, proceeding left to right, depicts: the original photograph after thresholding; preliminary localization points identified via DBSCAN clustering; refined localization points through k-means algorithm; and the restored image to its original form.

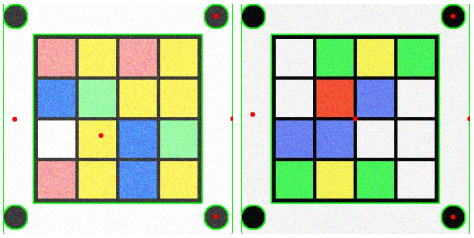
3 Discussion

My function has achieved 100% accuracy in color recognition on the Images1 dataset, with the entire process automated, eliminating the need for manual adjustments to individual images. In Appendix Table 1, I document the processing time required for each image, utilizing an i5-12600KF CPU.

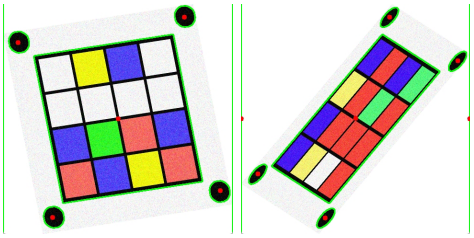
Attempts to process the Images2 dataset encountered initial hurdles during the stage of localizing the four black circles. Thresholding, initially attempted to isolate black pixels, proved ineffective due to inadequate differentiation between black borders and circles present in the Images2 set, rendering this method suboptimal.

Subsequently, contour detection was employed using $findContours$ function in OpenCV to extract both the black circles and the borders perimeter surrounding color blocks, yielding a set of coordinate points outlining these contours. By statistically analyzing point density—assuming higher density for black circle contours compared to black border perimeters—I endeavored to differentiate them. However, this approach did not yield satisfactory results, as depicted in Figure 9.

Further, I explored Hough line transform to detect all coordinate points along the black gridlines in the Images2 dataset, aiming to isolate the four corner points of black borders by comparing x and y coordinates, followed by applying a mask to remove the border area. Theoretically, this should have left only the black circles visible; however, the practical outcome, also shown in Figure 10, was less than ideal.

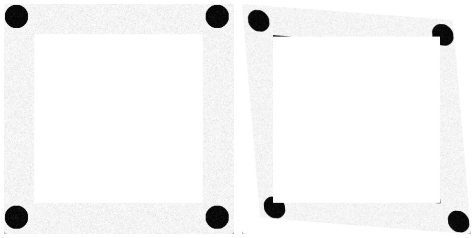


(a) noise_5.png (b) org_2.png

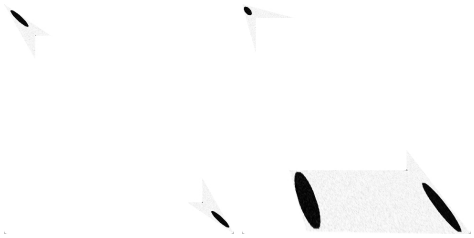


(c) proj1_1.png (d) proj1_4.png

Figure 9: The green lines represent contours derived from contour detection, while red dots indicate processed contour centroids. Instances arise where not all black circle centers can be located, as evidenced in (a), and distinguishing black circle centroids from those of other contours poses a challenge.



(a) org_1.png (b) proj1_2.png



(c) proj1_3.png (d) proj_4.png

Figure 10: Utilizing Hough line transform to identify grid areas, followed by masking to eliminate these regions, is impaired in severely distorted images where the removal process inadvertently affects the black circle areas, as observed in cases (c) and (d).

Appendix

1.Results

Table 1: Results of Images1 dataset

Filename	Time(s)	Result	Filename	Time(s)	Result
noise_1.png	0.0481	$\begin{bmatrix} r & b & r & y \\ g & r & g & y \\ y & r & r & y \\ b & g & r & b \end{bmatrix}$	noise_2.png	0.0465	$\begin{bmatrix} g & y & b & r \\ r & g & y & r \\ r & g & g & y \\ y & y & g & y \end{bmatrix}$
noise_3.png	0.0374	$\begin{bmatrix} b & y & g & b \\ r & b & r & g \\ r & y & y & b \\ b & y & r & y \end{bmatrix}$	noise_4.png	0.0331	$\begin{bmatrix} r & g & g & b \\ b & y & r & r \\ y & b & g & y \\ r & y & g & g \end{bmatrix}$
noise_5.png	0.0335	$\begin{bmatrix} b & r & g & g \\ y & g & g & y \\ g & b & b & y \\ y & b & r & y \end{bmatrix}$	org_1.png	0.0949	$\begin{bmatrix} b & r & b & b \\ b & y & b & r \\ g & r & g & g \\ y & r & b & r \end{bmatrix}$
org_2.png	0.0916	$\begin{bmatrix} b & y & g & r \\ y & y & b & y \\ b & b & b & g \\ r & b & g & y \end{bmatrix}$	org_3.png	0.0888	$\begin{bmatrix} g & b & b & b \\ y & y & y & b \\ b & r & g & g \\ b & g & r & r \end{bmatrix}$
org_4.png	0.092	$\begin{bmatrix} b & y & g & r \\ r & r & g & y \\ r & b & g & r \\ b & r & b & r \end{bmatrix}$	org_5.png	0.0748	$\begin{bmatrix} r & b & y & b \\ r & g & b & b \\ g & g & g & r \\ r & g & g & r \end{bmatrix}$
proj1_1.png	0.0414	$\begin{bmatrix} b & r & y & b \\ y & g & r & y \\ r & y & g & y \\ r & y & b & b \end{bmatrix}$	proj1_2.png	0.0337	$\begin{bmatrix} b & g & y & r \\ g & r & b & g \\ r & r & r & r \\ g & g & b & r \end{bmatrix}$

proj1_3.png	0.033	$\begin{bmatrix} b & b & y & y \\ y & y & y & g \\ b & b & r & b \\ g & y & g & y \end{bmatrix}$	proj1_4.png	0.0591	$\begin{bmatrix} y & r & b & y \\ y & y & y & g \\ g & r & b & r \\ b & r & b & y \end{bmatrix}$
proj1_5.png	0.0598	$\begin{bmatrix} y & r & g & r \\ r & g & g & b \\ g & r & b & g \\ r & y & r & g \end{bmatrix}$	proj2_1.png	0.0672	$\begin{bmatrix} b & g & r & g \\ y & b & y & b \\ b & g & g & y \\ g & y & g & y \end{bmatrix}$
proj2_2.png	0.06	$\begin{bmatrix} r & g & g & g \\ g & b & y & b \\ y & y & g & g \\ b & y & g & b \end{bmatrix}$	proj2_3.png	0.0738	$\begin{bmatrix} r & g & y & y \\ b & g & g & b \\ y & r & g & g \\ r & g & r & r \end{bmatrix}$
proj2_4.png	0.0641	$\begin{bmatrix} g & y & b & r \\ g & r & y & b \\ y & y & r & g \\ r & y & g & b \end{bmatrix}$	proj2_5.png	0.0442	$\begin{bmatrix} y & g & r & b \\ r & b & y & y \\ r & b & b & r \\ r & b & g & r \end{bmatrix}$
proj_1.png	0.0475	$\begin{bmatrix} y & b & g & y \\ g & g & b & g \\ g & y & b & g \\ y & y & r & b \end{bmatrix}$	proj_2.png	0.0751	$\begin{bmatrix} b & b & g & g \\ b & g & y & y \\ b & r & y & b \\ b & r & y & g \end{bmatrix}$
proj_3.png	0.0748	$\begin{bmatrix} r & g & g & y \\ g & g & b & y \\ r & y & b & b \\ r & r & b & r \end{bmatrix}$	proj_4.png	0.0823	$\begin{bmatrix} b & y & y & b \\ b & r & r & y \\ r & g & g & g \\ g & y & r & r \end{bmatrix}$
proj_5.png	0.0712	$\begin{bmatrix} r & g & y & y \\ b & r & y & b \\ g & y & b & y \\ y & y & b & y \end{bmatrix}$	rot_1.png	0.0968	$\begin{bmatrix} y & y & r & r \\ y & y & y & r \\ g & y & g & r \\ r & b & g & g \end{bmatrix}$
rot_2.png	0.0994	$\begin{bmatrix} r & b & y & y \\ r & b & b & b \\ y & r & b & y \\ g & r & g & b \end{bmatrix}$	rot_3.png	0.1	$\begin{bmatrix} r & b & b & b \\ b & r & y & b \\ b & b & b & b \\ r & r & y & r \end{bmatrix}$

rot_4.png	0.0918	$\begin{bmatrix} b & y & b & r \\ y & g & b & y \\ y & g & y & g \\ g & b & y & g \end{bmatrix}$	rot_5.png	0.0932	$\begin{bmatrix} g & r & r & y \\ y & y & b & g \\ b & r & r & r \\ y & y & y & b \end{bmatrix}$

2.Code

Listing 1: ColourMatrix.py

```
import cv2
import numpy as np
import math
from sklearn.cluster import DBSCAN
from sklearn.cluster import KMeans


def find_dense_black_clusters(image):
    """
    Find the clusters of dense black pixels.
    :param image: Grayscale map with white pixels as
        the background and black pixels as the
       foreground.
    :return: A list of tuples. Each tuple is a
       coordinate representing the center of the
       cluster.
    """

    # Get the coordinate of black pixels.
    black_pixels = np.argwhere(image == 0)

    # Convert the coordinates to a format suitable
       for DBSCAN. The coordinates of OpenCV images
       are (y, x),
    # while the usual data processing is (x, y).
    X = black_pixels[:, [1, 0]]

    # Apply DBSCAN clustering.
    db = DBSCAN(eps=5, min_samples=3).fit(X)

    # Get clustering results.
    labels = db.labels_
```

```
27      unique_labels = set(labels)
28
29      clusters = []
30      for k in unique_labels:
31          # -1 indicates the noise point.
32          if k == -1:
33              continue
34          class_member_mask = (labels == k)
35          xy = X[class_member_mask & (labels != -1)]
36          # Get the center of mass of the cluster as
               the representative coordinates.
37          c = xy.mean(axis=0)
38          clusters.append(tuple(map(int, c)))
39
40      return clusters
41
42
43  def calculate_center(points):
44      """
45      Calculate all the centers of points.
46      :param points: A list of points.
47      :return: The center of these list of points.
48      """
49      x_sum, y_sum = sum(x for x, y in points), sum(y
           for x, y in points)
50      center_x, center_y = x_sum / len(points), y_sum
           / len(points)
51      return center_x, center_y
52
53
54  def polar_angle(point, center):
55      """
56      Calculate the polar Angle of the point with
           respect to the center.
57      :param point: The coordinate of a point.
58      :param center: The center of a list of points.
59      :return: The polar Angle of the point with
           respect to the center.
60      """
61      dx, dy = point[0] - center[0], point[1] - center
           [1]
```

```
62      angle = math.atan2(dy, dx)
63      # Convert the Angle to a range of 0 to 2  , ensuring that the clockwise ordering is correct.
64      if angle < 0:
65          angle += 2 * math.pi
66      return angle
67
68
69  def sort_points_clockwise(points):
70      """
71      Sort the coordinate points in clockwise order.
72      :param points: A list of points.
73      :return: Points listed in clockwise order.
74      """
75      center = calculate_center(points)
76      sorted_points = sorted(points, key=lambda p: polar_angle(p, center), reverse=True)
77      return sorted_points
78
79
80  def getCircle(image_path):
81      """
82      Get the coordinates to locate the black circle around the image.
83      :param image_path: The path of image.
84      :return: The coordinates of the four black circles around the image.
85      """
86      image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
87      # Use a threshold to process the image.
88      thresh = 20
89      _, threshold = cv2.threshold(image, thresh, 255, cv2.THRESH_BINARY)
90
91      # Find the clusters of black pixels.
92      clusters = find_dense_black_clusters(threshold)
93
94      # Divide clusters into four categories via k-means algorithm.
```

```
95      coordinates_np = np.array(clusters)
96      kmeans = KMeans(n_clusters=4, random_state=0)
97      kmeans.fit(coordinates_np)
98
99      # Gets the center coordinates for each category.
100     cluster_centers = kmeans.cluster_centers_
101     corners = []
102     for i, center in enumerate(cluster_centers):
103         x, y = center
104         x, y = int(x), int(y)
105         corners.append((x, y))
106
107     # Order the four coordinates clockwise.
108     sorted_corners = sort_points_clockwise(corners)
109     return sorted_corners
110
111
112 def transform(image_path, trans, org=None):
113     """
114     Restore perspective transformed images to their
          original shape.
115     :param image_path: The path of image.
116     :param trans: The coordinates of the four anchor
          points of the image after perspective
          transformation.
117     :param org: The coordinates of the four anchor
          points of the original image.
118     :return: The image after recovery.
119     """
120     # Define the four corner coordinates of the
          transformed rectangle.
121     if org is None:
122         org = [(0, 0), (0, 480), (480, 480), (480,
              0)]
123
124     original_corners = np.float32(org)
125     transformed_corners = np.float32(trans)
126
127     # Calculate perspective transformation matrix.
128     m = cv2.getPerspectiveTransform(
          transformed_corners, original_corners)
```

```
129
130     # Read the transformed image.
131     image = cv2.imread(image_path)
132
133     # Apply the reverse perspective transform.
134     restored_image = cv2.warpPerspective(image, m, (
            image.shape[1], image.shape[0]))
135
136     # Crop image.
137     restored_image = restored_image[50:430, 50:430]
138
139     return restored_image
140
141
142 def saturate_hsv_image(image):
143     """
144     Saturate the pixels to the maximum.
145     :param image: Image in RGB format.
146     :return: Image in RGB format.
147     """
148
149     # Transform image from RGB to HSV.
150     hsv_img = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
151
152     # Split H, S and V channels.
153     h, s, v = cv2.split(hsv_img)
154
155     # Set all values of saturation channel to
            maximum (255).
156     s_max = np.ones_like(s) * 255
157
158     # Merge channels with saturation set to maximum.
159     hsv_img_max_saturation = cv2.merge((h, s_max, v)
            )
160
161     # Converts the modified HSV image back into the
            BGR color space.
162     bgr_img = cv2.cvtColor(hsv_img_max_saturation,
            cv2.COLOR_HSV2BGR)
163
164     return bgr_img
```

```
165
166
167 def calculate_mean_colors(image):
168     """
169     Calculate the mean values of each channels in
        RGB format.
170     :param image: Image in RGB format.
171     :return: A matrix of tuple consists of the mean
        value of each channels.
172     """
173     # Divide image into 4x4 blocks.
174     block_size = image.shape[0] // 4
175
176     # Initialise result matrix.
177     result_matrix = np.zeros((4, 4, 3), dtype=np.
        float32)
178
179     for i in range(4):
180         for j in range(4):
181             # Focus on one block.
182             block = image[i * block_size:(i + 1) *
                block_size, j * block_size:(j + 1) *
                block_size]
183
184             # Calculate the mean value of each
                channels.
185             mean_bgr = np.mean(block, axis=(0, 1))
186
187             # Converts to a tuple and stores it in
                the result list
188             result_matrix[i, j] = tuple(map(int,
                mean_bgr))
189
190     return result_matrix.tolist()
191
192
193 def convert_to_color_labels(matrix):
194     """
195     Get the final classification results.
196     :param matrix: A matrix of tuples. Tuples
        consist of means of each RGB channels.
```

```
197     :return: The matrix of classification result.
198     """
199
200     def map_color(row):
201         r, g, b = row
202         if b > 150 and r < 150 and g < 150:
203             return 'r'
204         elif r > 150 and g < 150 and b < 150:
205             return 'b'
206         elif g > 150 and r < 150 and b < 150:
207             return 'g'
208         elif g > 150 and b > 150:
209             return 'y'
210         else:
211             return 'w'
212
213     # Apply conversion rules to each element.
214     color_labels = [[map_color(pixel) for pixel in
            row] for row in matrix]
215     return color_labels
216
217
218 def colorMatrix(image_path):
219     """
220     Function that automatically reads a colour
            pattern image from hard-disc and returns an
            array representing the colour pattern.
221     :param image_path: The file name of image.
222     :return: The matrix representing the color
            pattern.
223     """
224     # Get the coordinate of four black circles
            around the image.
225     circles = getCircle(image_path)
226
227     # Restored image to original shape.
228     restored_img = transform(image_path, circles)
229
230     # Increase pixel saturation.
231     restored_img = saturate_hsv_image(restored_img)
232
```

```
233     # Apply Gaussian Blur
234     restored_img = cv2.GaussianBlur(restored_img,
            ksize=(5, 5), sigmaX=50)
235
236     # Get the color pattern.
237     res = calculate_mean_colors(restored_img)
238     res = convert_to_color_labels(res)
239
240     return res
241
242
243 if __name__ == "__main__":
244     image_path = 'images1/noise_1.png'
245
246     # Get the color matrix.
247     result = colorMatrix(image_path)
248     print(result)
```