Computer Vision Histogram Matching

Ian Sinclair
ENCE 3620-1

March 30, 2022
Dr. Mohammad H. Mahoor

Abstract

A technique for image contrast enhancement is explored. Specifically, analyzing a histogram matching method that maps the contrast of an input image to that of a pre-determine target image. For example, if an image has low contrast and is dark, the majority of its pixel intensity values will be on the lower range of the image scale $[0, 255]$. And so it may be possible to improve the image quality by pairing the input image with a high contrast brighter 'target' image and adjusting the contrast of the input to match the target. This process involves generating a normalized histogram of the both images. Equalizing both images by created a cumulative distribution function (CDF) from the histograms, then mapping the input CDF to the target CDF, and finally performing the inverse equalization to create the new image. As a result, image equalization techniques are also explored. Within the scope of the analysis, two gray scale image sets are considered, each with two input images and a single target image. In regards to histogram matching, it is not possible to numerically confirm the effectiveness of the technique and so results of each set are verified visually.

1 Introduction

Histogram matching is a technique to adjust the contrast of an input image to match that of a target image. This is generally a three step process, to generate normalized image histograms, equalize those histograms, then creates maps between equalized histograms and their inversions.

1.1 Histogram of an image

The histogram of an image can be defined as a map of the number of pixels in the image that belong to a particular intensity value. And so for any intensity value, the histogram evaluates to the number of pixels in the image that have that intensity.

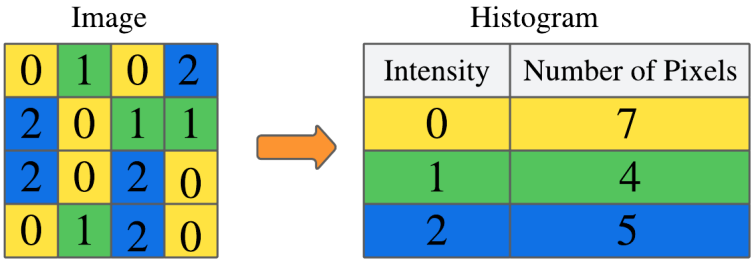


Figure 1: Image displaying concept art an image and corresponding histogram

For convention let the maximum intensity of an image be $L = 256$, then the image intensity is on the range, $[0, 255]$. And so deriving a more formal definition for the histogram of an image, let $f(x, y)$ by any gray scale image, and let $r_k \in [0, 255]$ be the intensity value for any pixel $k = (x, y)$

in the image. Then the histogram at intensity i can be defined by,

$$n_i = h(i) = \sum_{(x,y)\in I} 1, \qquad \text{Where } r_k = i, \quad i \in [0, L-1]$$

Where n_i is the magnitude of the histogram at intensity i.
Which when applied to a full image results in a curve displaying the pixel intensity distribution.

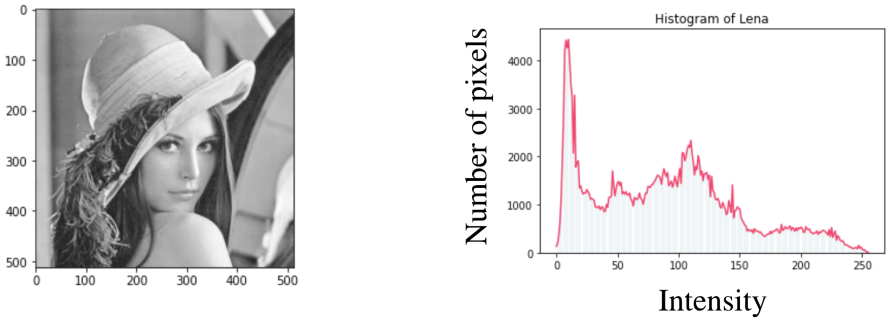


Figure 2: Image displaying an image of famous computer vision example Lena, and its corresponding histogram.

1.2 Histogram Matching

Correspondingly, histogram matching is the process of mapping an input histogram to a target output. This is commonly done to enhance the amount of contrast in the input image. It is important to note that the effectiveness of the algorithms is depended on the choice of target histogram and so that respect there is no fully correct choice. And so this technique is largely trial and error to find an image that is visually superior for whatever application it is being applied to.

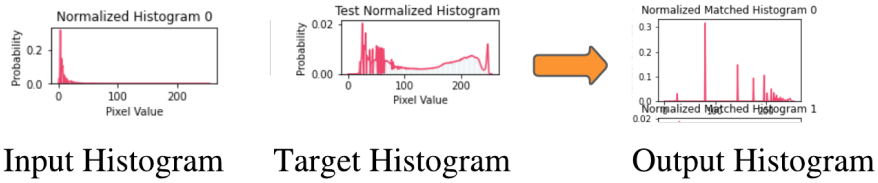


Figure 3: Image displaying results of a histogram matching process.

Note, the final output histogram is not an exact match of the target. This is because the mapping between histograms is not strictly single values and well-defined, as a result, the final map is an approximation based on least error between intensity values, $(h(i), h(i-1))$.

Procedure

Here two sets of gray scale images are considered each containing two input images and a unique target image. A histogram matching technique is preformed to adjust the contrast of each input image towards their set correspond target image. This process here includes an analysis of the following,

i) Loading raw image files,

ii) Converting images to gray scale,

iii) Generating histograms for each image,

iv) Normalizing each histogram

v) Generating the cumulative distribution function (CDF) from the normalized histogram of each image.

vi) Equalizing the images in each set based on the CDF histograms

vii) Mapping input histograms to their corresponding targets histogram using the CDF of each image.

Raw Image Files

Consider the following images belonging to each set.

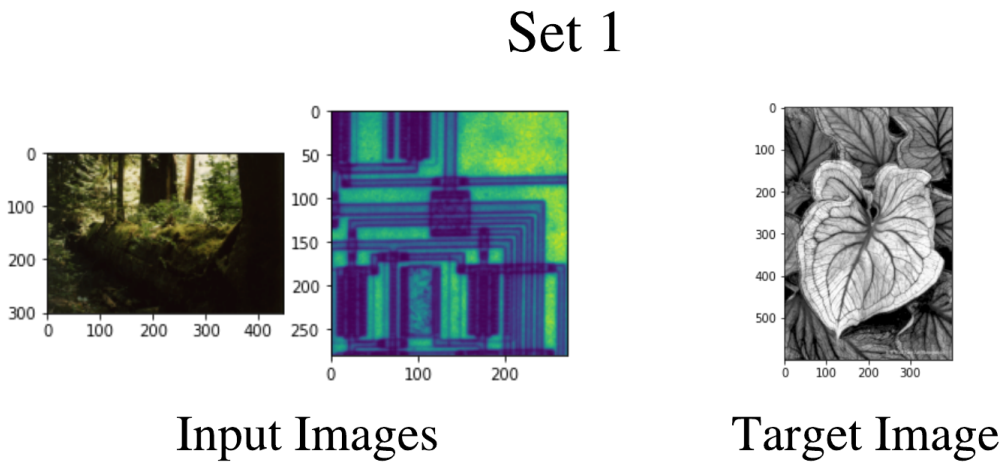


Figure 4: Set 1: raw images

Figure 5: Set 2 raw images

1.3 Converting To Gray Scale

Note, the scope of this analysis does not experimentally extend past gray scale images, and so all images in either set are converted to gray scale. This is typically done by one of two methods, either the RGB values for each pixel are averaged to make a single gray scale intensity (mean value method). Or as a means of better representing the color distribution detectable by the human eye, a luminosity method can be used which still uses a mean algorithm, however, changes the weights

impose on each RGB values. [2].

$$r_k = 0.299I_R + 0.587I_G + 0.114I_B$$

For image I, gray intensity values r_k and I_R, I_G, I_B being the red green and blue values at each pixel. Which is implemented by the following.

```
def RGB_to_Greyscale( image ) :
  temp = image.copy()
  grey = np.zeros(( len(temp), len(temp[0]) ), dtype = int )

  for i in range(0 , len( image ) ) :
    for j in range(0 , len( image[i] ) ) :
      Filter =  0.299*temp[i][j][0] + 0.587*temp[i][j][1] + 0.114*temp[i][j][2]
      grey[i][j] = int( Filter )

  return grey
```

And so the gray scale images for each set are.

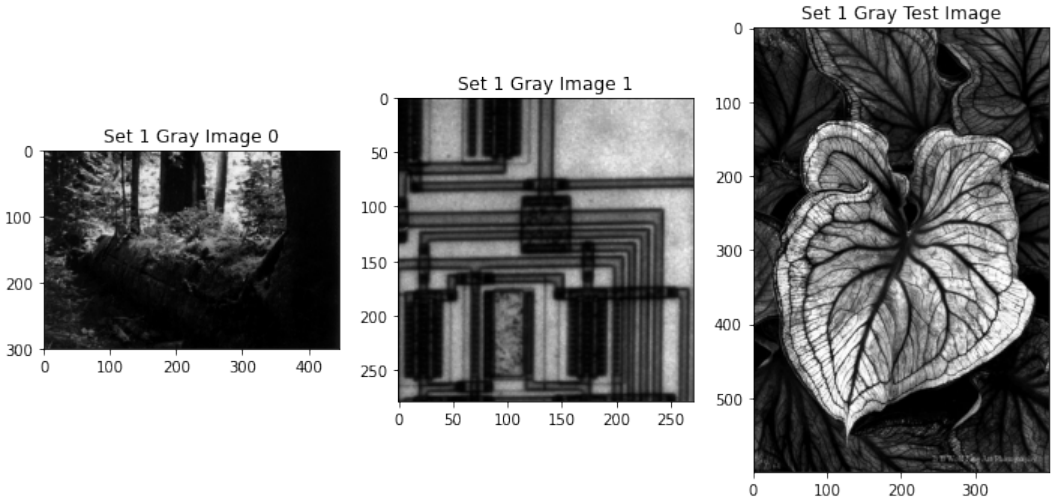


Figure 6: Set 1 gray scale images

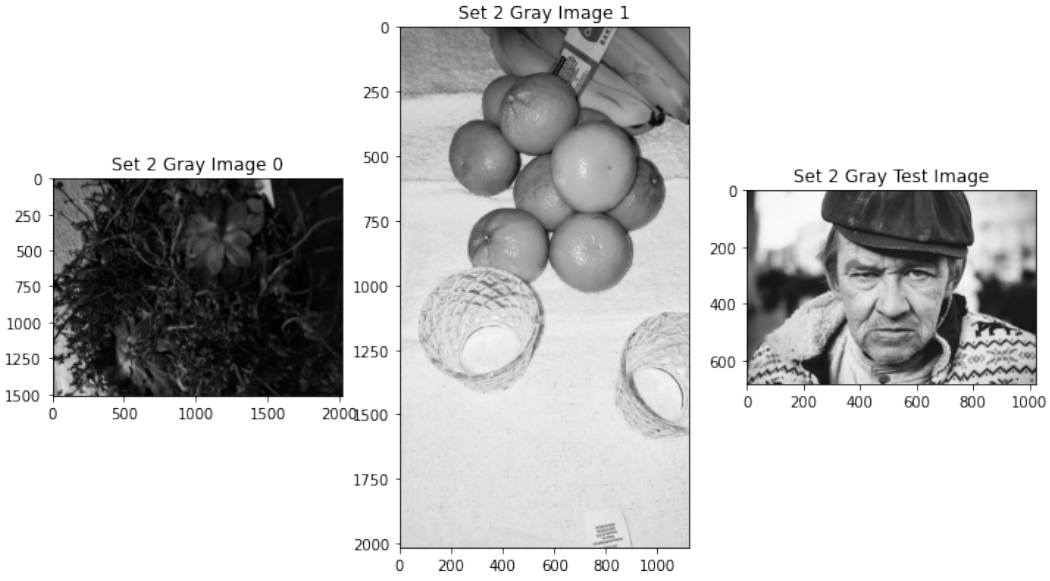


Figure 7: Set 2 gray scale images

1.4 Histogram Generation

Now each image is in gray scale form, so each pixel intensity value r_k will have a single input. And so by summing the amount of pixels belonging to every possible intensity value $[0, 255]$ a histogram can be generated for each image.

```
def generate_histogram( image ) :
  temp = image.copy()
  hist = []

  #Initialize an emply list on [0,255]
  for i in range( 0 , 256 ) :
    hist += [0]

  #Fill list with histogram values
  for i in range( 0 , len( image ) ) :
    for j in range( 0 , len( image[i] ) ) :
      intensity = int( image[i][j] )
      hist[intensity] += 1

  return hist
```

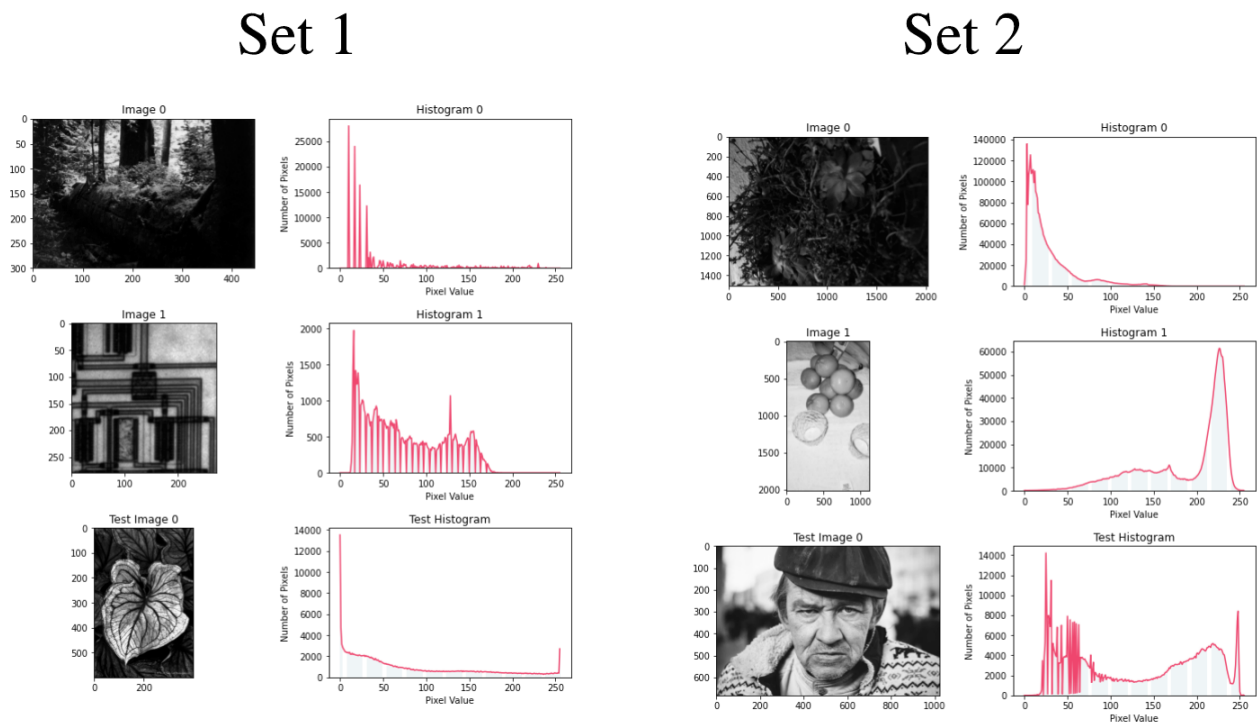


Figure 8: Set 1,2 images and corresponding histograms

1.5 Normalizing Histograms

Notice that each histogram generated from the gray scale images is on a different y-scale. Or because the number of pixels in each image is different, the maximum amount of pixels for each histogram is inconsistent. This may interfere with the mapping between histograms. As a result and by convention, the histograms are normalized. This also corresponds to the probability density function (pdf) that determines the probability of any pixel, x, in the image having a particular intensity value, i.

$$p_x(i) = p(x = i) = \frac{n_i}{n}$$

Where n_i is the number of pixels with intensity value i and n is the total number of pixels. This method can be implemented by the following,

```
def normalize_histogram( histogram ) :
  temp = histogram.copy()
  r = sum( temp )
  for i in range( 0 , len( temp ) ) :
    temp[ i ] = temp[ i ]/ r

  return temp
```

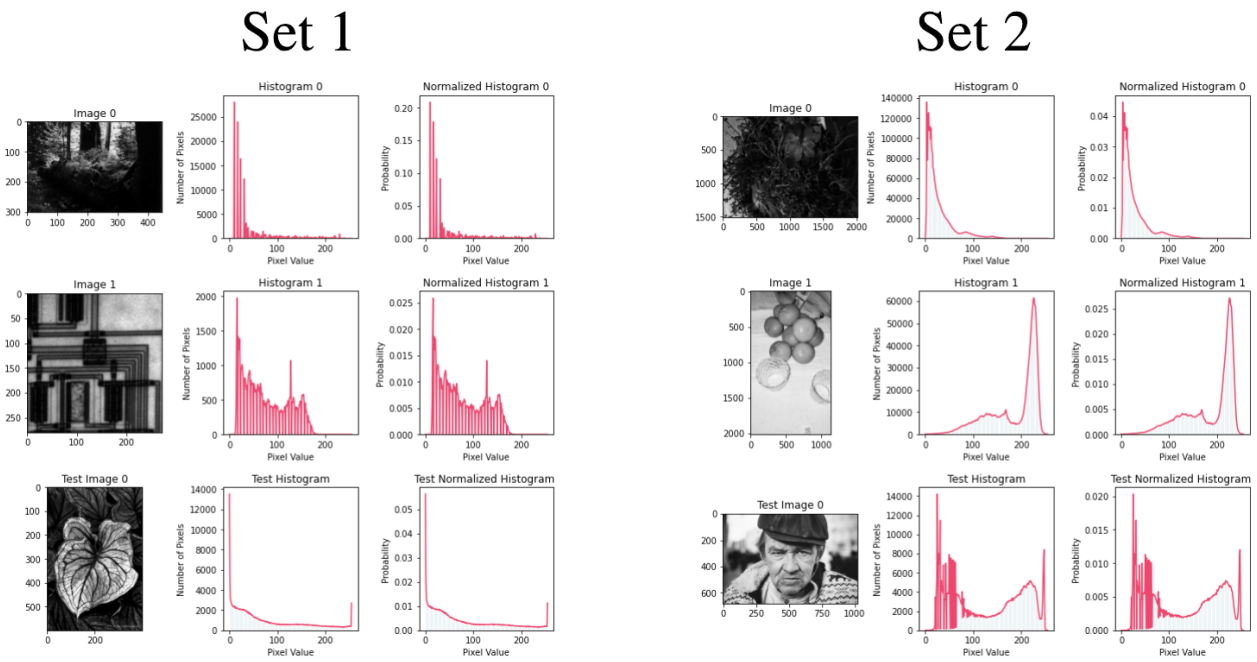


Figure 9: Normalized Histograms

1.6 Histogram Equalization

Histogram equalization is a process to map pixel values such that the number pixels at any intensity in the image is consistent across the image. For example, consider a low-contrast dark image where all the pixels are between intensity $[0, 50]$. Then, the image is not making use of its full processing range $[0, 255]$, and so may be enhanced by re-distributing the pixel intensity values to the entire range.

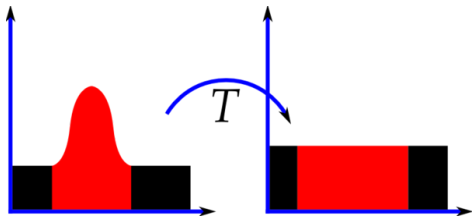


Figure 10: Concept art of histogram equalization [1]

Ultimately, this could result in a clearer image. Or an image with higher contrast making it easier to distinguish between darker and lighter regions of the original image.
Now, consider the normalized histogram of each image is already a pdf. And so, a technique for image equalization is to generate the cumulative distribution function (CDF) from the pdf, and map the result back to a new modified image.

1.6.1 Cumulative Distribution Function (CDF)

Consider the cumulative distribution function (CDF) describes the probability that given an intensity value i that any pixel in the image has intensity $r_k \leq i$.

$$cdf_x = \sum_{j=0}^{i} p(x = j)$$

Therefore, has some interesting properties in terms of image processing. First, by construction it must be monotonically increasing, and as a result, in terms of histogram equalization can create a injective map between an image and its CDF. Additionally, because it the integral of any pdf, $P(x = i)$, is such, $\int_{\infty}^{\infty} P(x = i)dx = 1$, it follows that the maximum value of any CDF must be 1, $\max_{i \in [0,255]} CDF = 1$. Which because the pdf can be mapped to span the image space, this maximum must happen and must also correspond to the largest intensity value. Which implies the map from an image to its CDF is surjective. Therefore, there is a bijection between an image histogram and its corresponding CDF. Here a CDF is implement by passing a normalized histogram.

```
def generate_CDF( histogram ) :
  CDF = np.zeros( [256] )

  CDF[0] = histogram[0]

  for i in range( 1 , len( histogram ) ) :
    CDF[i] = CDF[i-1] + histogram[i]

  return CDF
```

1.6.2 Generating an Equalized Image

Directly, the map between a CDF and its corresponding image can be generated by a look up table. Where intensity value of any pixel in the image, r_k, gets mapped to s_k in the CDF, where s_k is the probability that intensity any pixel has intensity less than k times 255. Or is the CDF evaluated at k times 255.

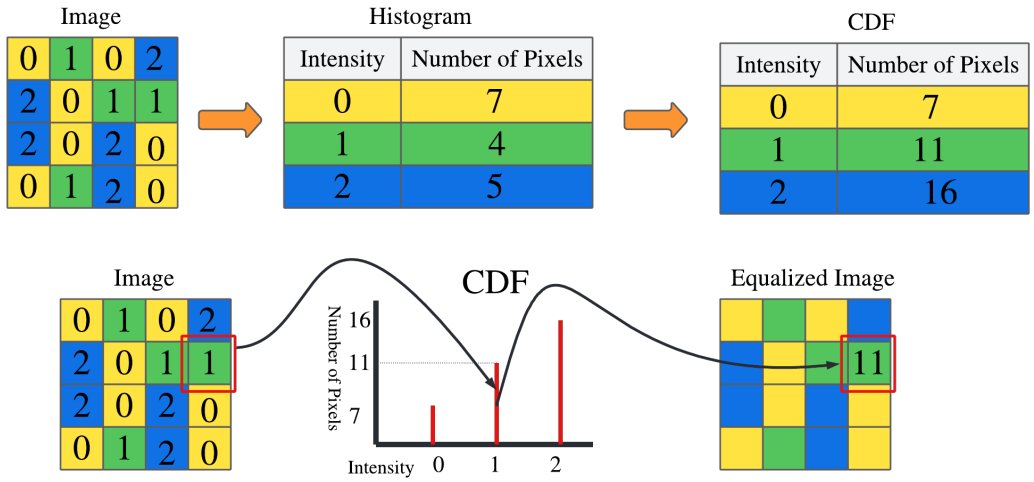


Figure 11: Concept art of histogram equalization

```
def histogram_equalization( image, CDF ) :
  temp_image = image.copy()
  equ_image = np.zeros(image.shape)

  for i in range( 0 , len( temp_image ) ) :
    for j in range( 0 , len( temp_image[i] ) ) :
      intensity = int( temp_image[i][j] )
      equ_image[i][j] = CDF[intensity] * 255

  return equ_image
```

Which results in images.

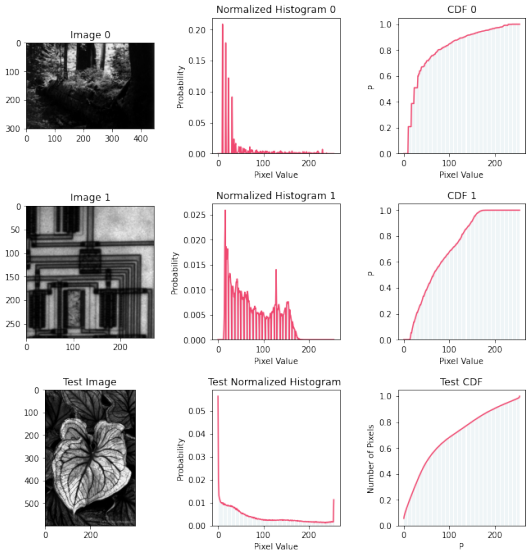


Figure 12: Image displaying gray scale images and their corresponding normalized histograms and CDF

Then mapping the CDF functions and the original images to a new equalized image.

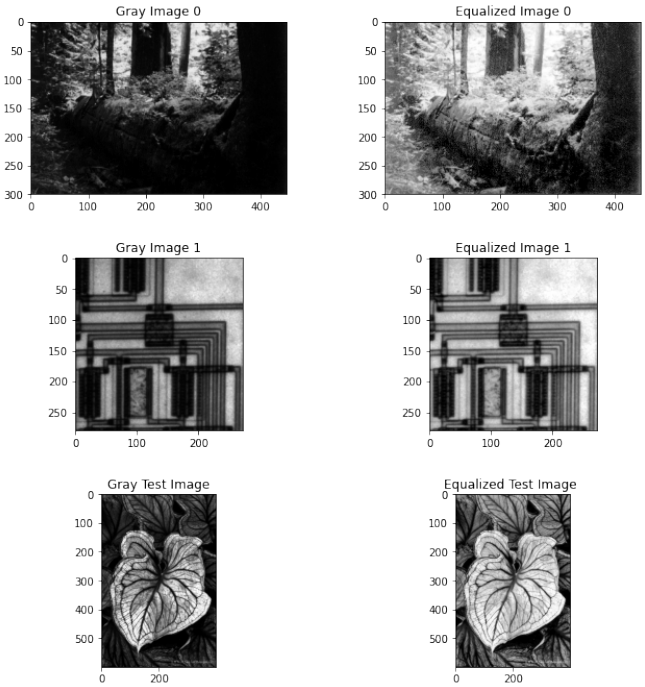


Figure 13: Image comparing the original gray scale images to their corresponding equalized images.

This is also applied to set 2.

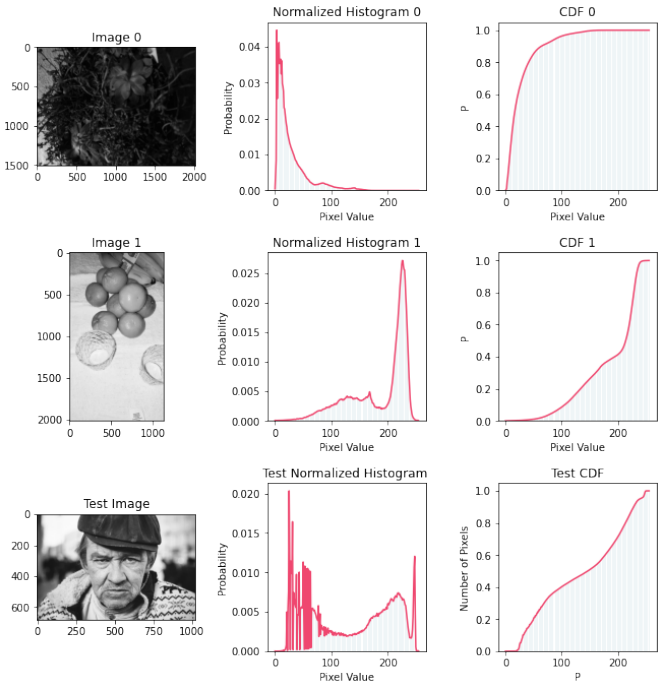


Figure 14: Image comparing the original gray scale images to their corresponding equalized images.

Which results in image comparison.

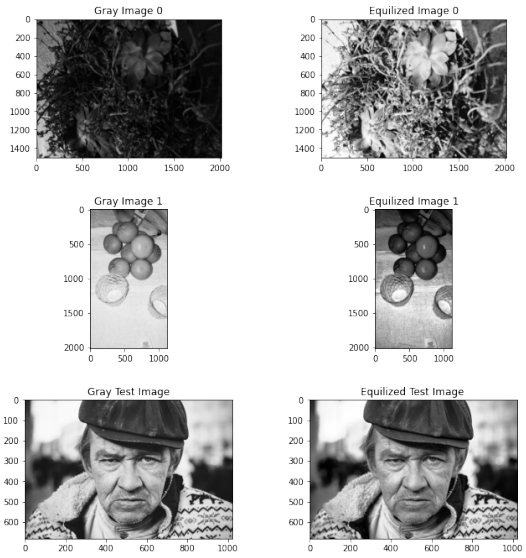


Figure 15: Image comparing the original gray scale images to their corresponding equalized images.

1.7 Histogram Matching

Histogram equalization is a good technique to expand the contrast in a image. However, if there is an intensity distribution in another image that is better or could make a good pair with any of the input images histogram matching can be sued to adjust the input image contrast towards that of the 'target' image. This is done by first equalizing both images. Then creating a map from the input image CDF to the target image CDF. Because the distribution across all intensity values may differ greatly between the images, the 1 to many map between CDF functions takes the input cumulative density at intensity i and maps it to the closest target image CDF value. $cdf_{input}(i) \approx cdf_{target}(i)$.

Then the intensity value i gets mapped back from the target cdf to the target image, and is set equal to the intensity corresponding to that target cdf value.

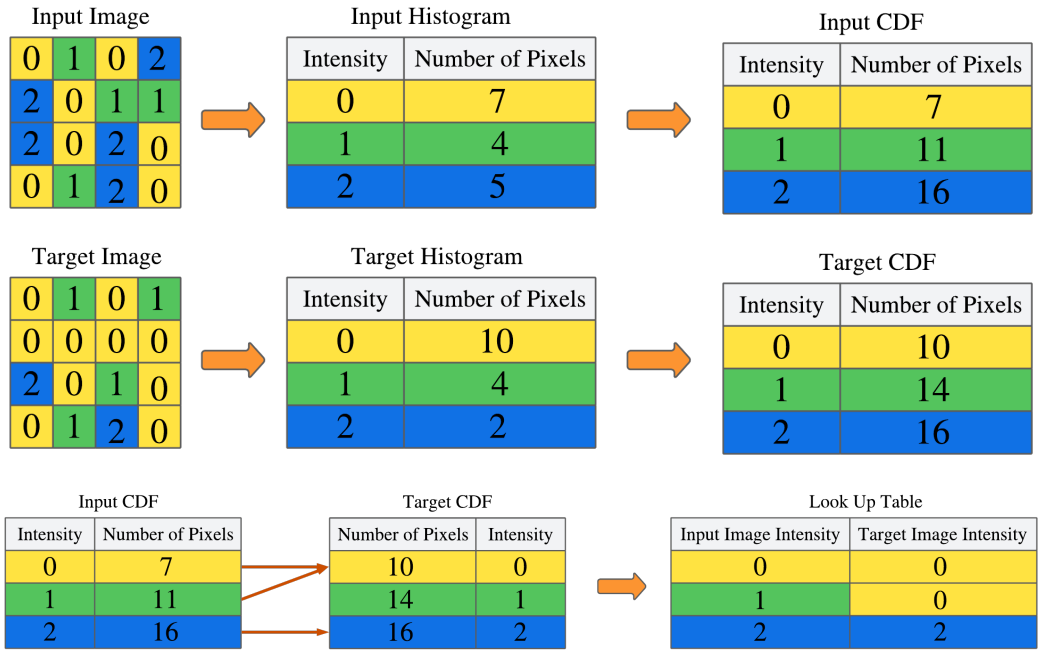


Figure 16: Histogram Matching Concept Art. Displaying the generation of the look up table map from input to target image.

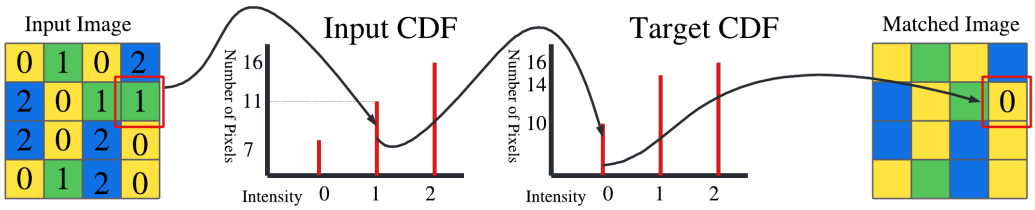


Figure 17: Histogram Matching Concept Art. Displaying the mapping process between input and target images.

```
def histogram_matching( image, histogram, target_histogram ) :
  A_image = image.copy()
  A_histogram = histogram.copy()
  target = target_histogram.copy()

  out_image = np.zeros(A_image.shape)

  cdf_A = generate_CDF( A_histogram )
  cdf_B = generate_CDF( target )

  for i in range( 0 , len( A_image ) ) :
    equalized_intensity = 0

    for j in range( 0 , len( A_image[i] ) ) :
      intensity = A_image[i][j]
      equalized_intensity = cdf_A[intensity]

      equiv_equal_intensity = 0

      for k in range( 0 , len( cdf_B ) ) :
        if cdf_B[k] > equalized_intensity :
          if abs( cdf_B[k] - equalized_intensity ) >= abs( cdf_B[k-1] - equalized_intensity ) :
            out_image[i][j] = int( k-1 )
          else :
```

```
                out_image[i][j] = int(k)

            break

    return out_image
```

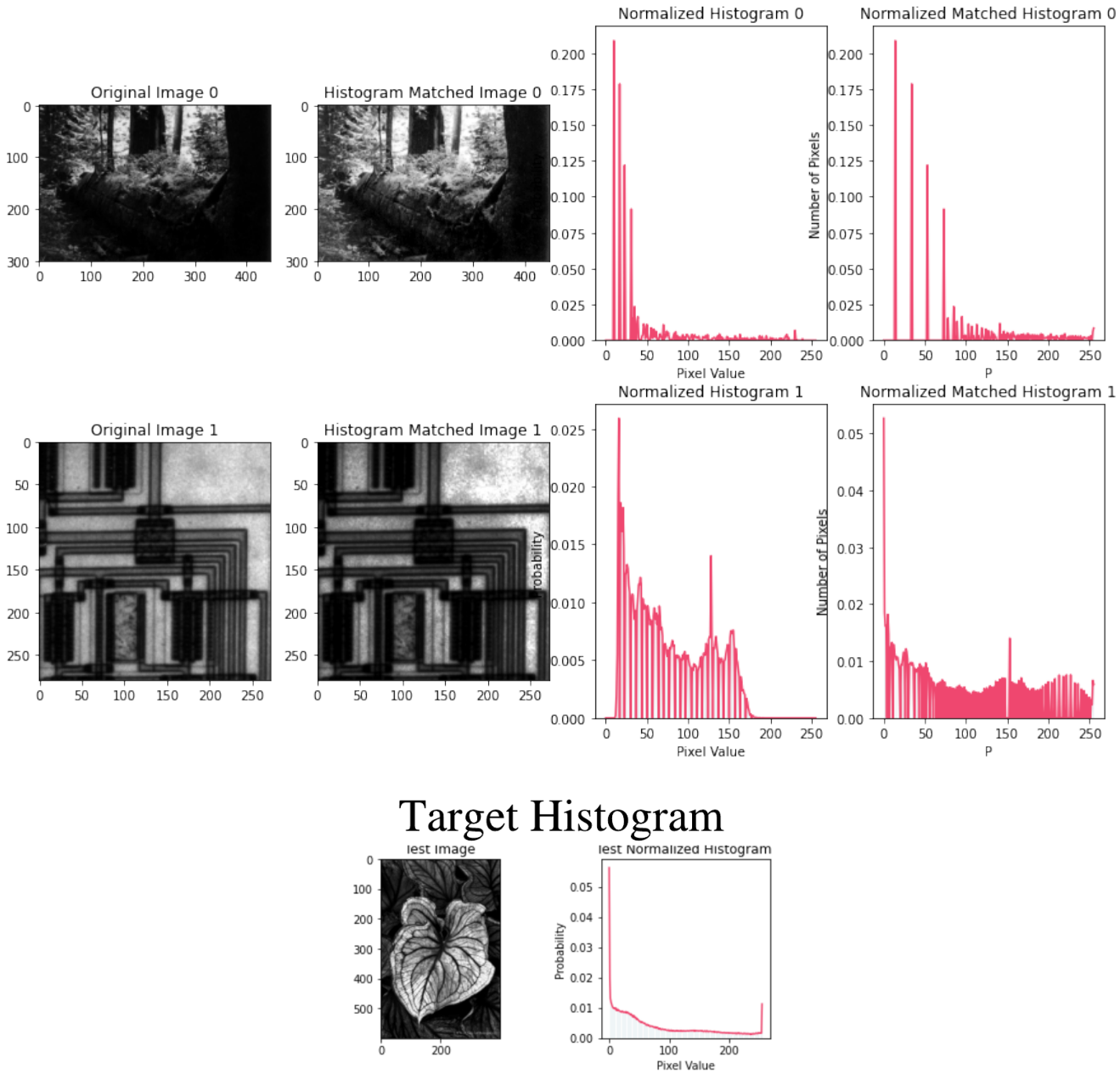


Figure 18: Set 1 image displaying gray scale images, updated images using matched histogram technique and their corresponding histograms.

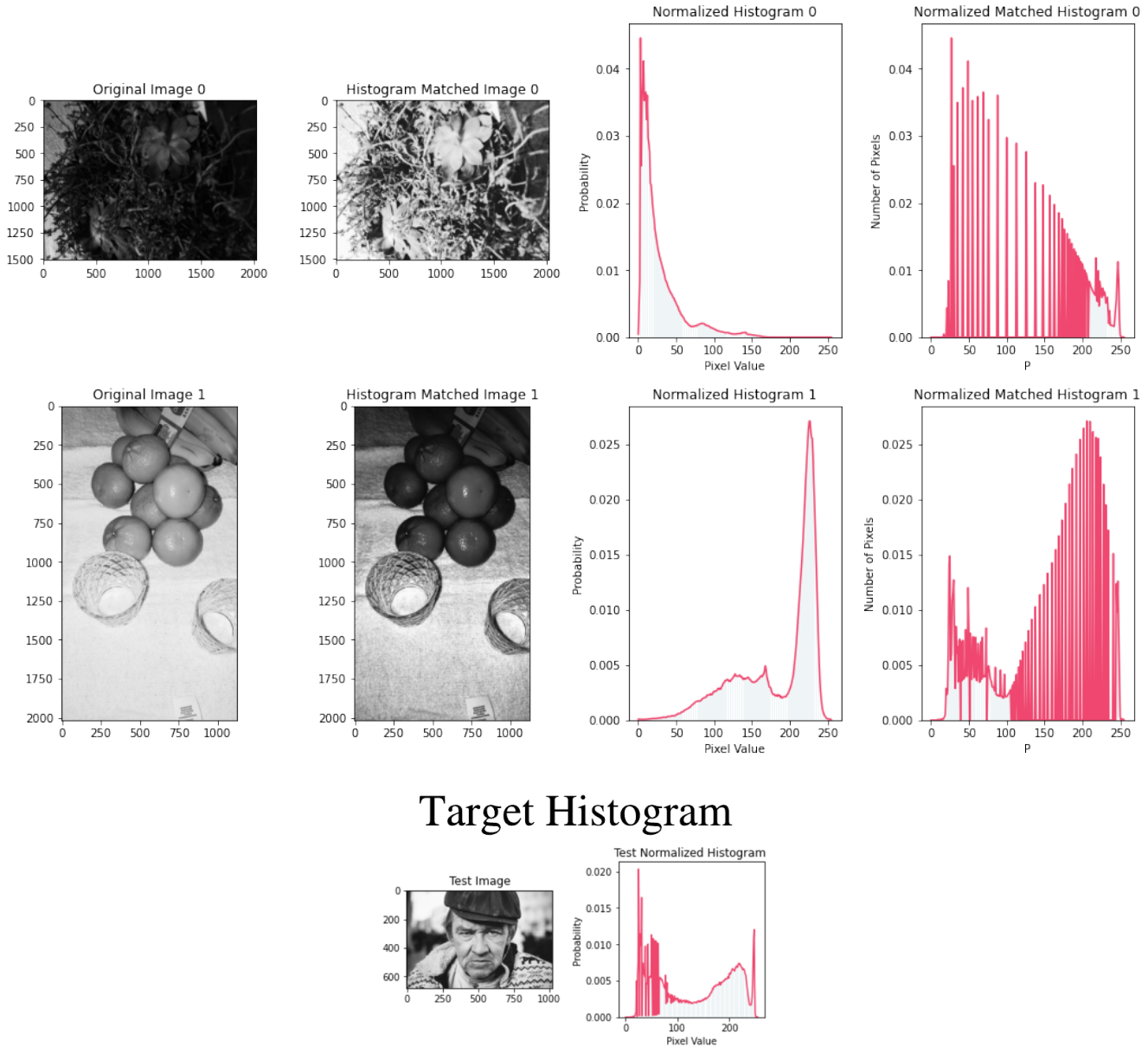


Figure 19: Set 1 image displaying gray scale images, updated images using matched histogram technique and their corresponding histograms.

1.8 Color Histogram Matching

This technique has been observed above for gray scale images; however, it is also applicable to color images by separating each RGB fields, red, blue, green images, preforming histogram matching on each sub-color image and combine the results at the end.

```
def separate_RGB_images( image ) :
  temp_R = np.zeros((len( image ), len( image[0])), dtype= int )
  temp_G = np.zeros((len( image ), len( image[0])), dtype= int  )
  temp_B = np.zeros((len( image ), len( image[0])), dtype= int  )

  for j in range(0 , len( image ) ) :
    for k in range(0 , len( image[j] ) ) :
      temp_R[j][k] = image[j][k][0]
      temp_G[j][k] = image[j][k][1]
      temp_B[j][k] = image[j][k][2]

  return temp_R, temp_G, temp_B
```

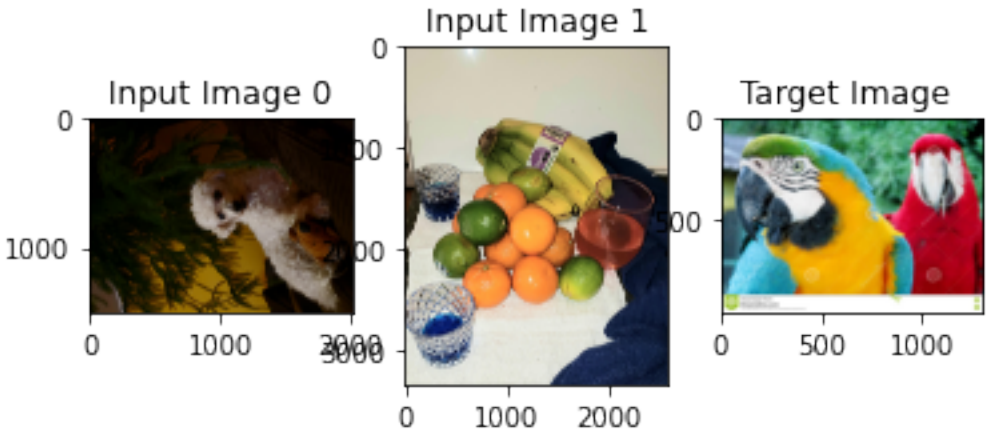


Figure 20: Set of color images including target image

Now creating a histogram and CDF for each image.

```
Red_set = []
Green_set = []
Blue_set = []


Red_t_set = []
Green_t_set = []
Blue_t_set = []


for i in range(0 , len( Color_set_raw_images )) :
  Red_set.append(  separate_RGB_images(  Color_set_raw_images[i] )[0]  )
  Green_set.append(  separate_RGB_images( Color_set_raw_images[i] )[1]  )
  Blue_set.append(  separate_RGB_images( Color_set_raw_images[i] )[2]  )

for i in range(0 , len( Color_set_raw_test_images )) :
  Red_t_set.append(  separate_RGB_images( Color_set_raw_test_images[0] )[0]  )
  Green_t_set.append(  separate_RGB_images( Color_set_raw_test_images[0] )[1]  )
  Blue_t_set.append(  separate_RGB_images( Color_set_raw_test_images[0] )[2]  )

Red_set_hist = []
Green_set_hist = []
Blue_set_hist = []


Red_t_set_hist = []
Green_t_set_hist = []
Blue_t_set_hist = []

for i in range(0 , len( Color_set_raw_images )) :
  Red_set_hist.append( normalize_histogram( generate_histogram( Red_set[i] ) ) )
  Green_set_hist.append( normalize_histogram( generate_histogram( Green_set[i] ) ) )
  Blue_set_hist.append( normalize_histogram( generate_histogram( Blue_set[i] ) ) )

Red_t_set_hist.append( normalize_histogram( generate_histogram( Red_t_set[0] ) ) )
Green_t_set_hist.append( normalize_histogram( generate_histogram(Green_t_set[0] ) ) )
Blue_t_set_hist.append( normalize_histogram( generate_histogram( Blue_t_set[0] ) ) )

Red_set_CDF = []
Green_set_CDF = []
Blue_set_CDF = []


Red_t_set_CDF = []
Green_t_set_CDF = []
Blue_t_set_CDF = []

for i in range(0 , len( Color_set_raw_images )) :
  Red_set_CDF.append( generate_CDF( Red_set_hist[i] ) )
```

```
50      Green_set_CDF.append( generate_CDF( Green_set_hist[i] ) )
51      Blue_set_CDF.append( generate_CDF( Blue_set_hist[i] ) )
52
53  Red_t_set_CDF.append( generate_CDF( Red_t_set_hist[0] ) )
54  Green_t_set_CDF.append( generate_CDF( Green_t_set_hist[0] ) )
55  Blue_t_set_CDF.append( generate_CDF( Blue_t_set_hist[0] ) )
```

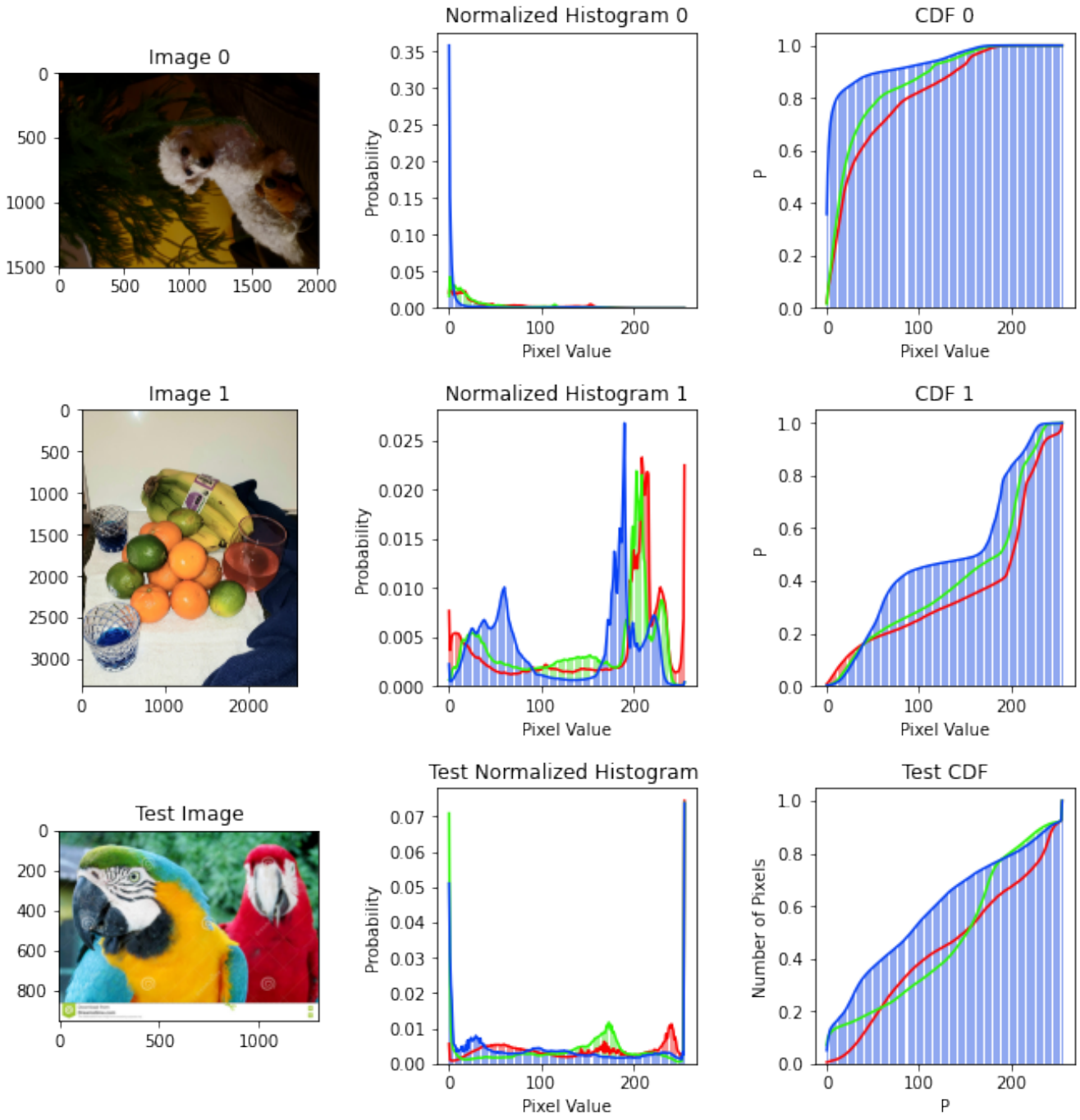
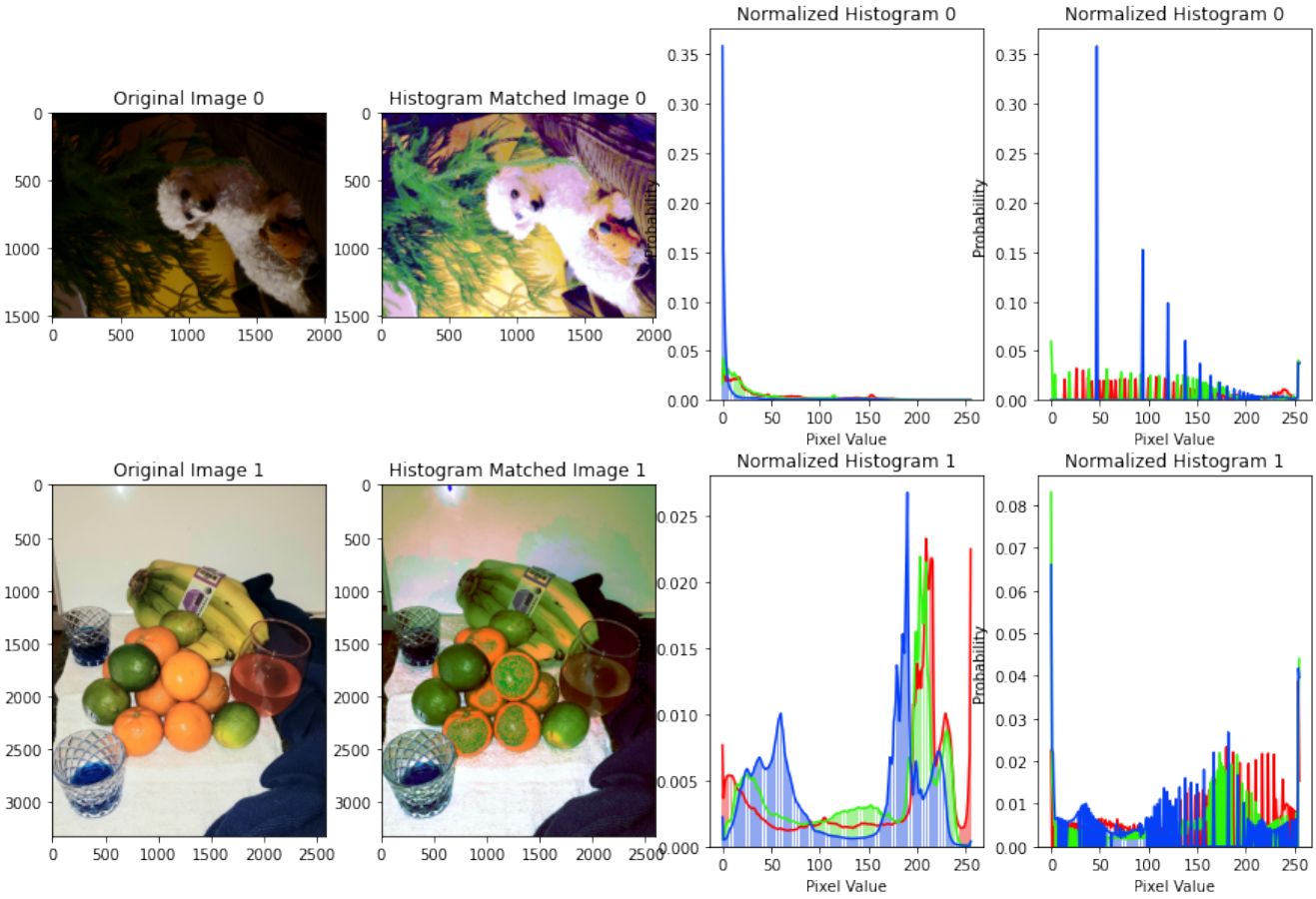


Figure 21: Image displaying color set images and their corresponding histograms and CDF's



Target Histogram

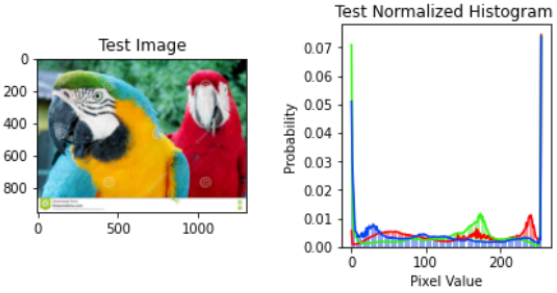


Figure 22: Image displaying the original images in the color set, their matched images and their corresponding histograms

```
def combine_RGB( red_image, green_image, blue_image ) :
  temp = np.zeros( (len(red_image) , len(green_image[0]) , 3), dtype=int )

  print(temp.shape)
  for j in range(0 , len( red_image ) ) :
    for k in range(0 , len( red_image[j] ) ) :
      temp[j][k][0] = red_image[j][k]
      temp[j][k][1] = green_image[j][k]
      temp[j][k][2] = blue_image[j][k]

  return temp



Red_set_match = []
```

```
Green_set_match = []
Blue_set_match = []

for i in range(0 , len( Color_set_raw_images ) ) :
  Red_set_match.append( histogram_matching( Red_set[i], Red_set_hist[i], Red_t_set_hist[0] ) )
  Green_set_match.append( histogram_matching( Green_set[i], Green_set_hist[i], Green_t_set_hist[0] ) )
  Blue_set_match.append( histogram_matching( Blue_set[i], Blue_set_hist[i], Blue_t_set_hist[0] ) )


Matched_images = []

for i in range(0 , len( Red_set ) ) :
  Matched_images.append( combine_RGB( Red_set_match[i], Green_set_match[i], Blue_set_match[i] ) )

Red_set_hist2 = []
Green_set_hist2 = []
Blue_set_hist2 = []

for i in range(0 , len( Color_set_raw_images )) :
  Red_set_hist2.append( normalize_histogram( generate_histogram( separate_RGB_images( Matched_images[i]
      )[0] ) ) )
  Green_set_hist2.append( normalize_histogram( generate_histogram( separate_RGB_images( Matched_images[
      i] )[1] ) ) )
  Blue_set_hist2.append( normalize_histogram( generate_histogram( separate_RGB_images( Matched_images[i
      ] )[2] ) ) )
```

2 Conclusion

Considered is a algorithm to apply histogram matching technique to gray scale images. More specifically, two sets are taken, each with two input images and a single target image. The first set has a very dark low-contrast image and a less dark higher contrast image. Both are compared to an image with both very light are very dark pixels. The result of the histogram matching adjusts the first image to be lighter with greater contrast. And the second to be darker with greater contrast. Set two had a first dark input image and a second lighter input image compared to a target image that has high contrast and very dark and very light pixels. After histogram matching, the darker image was adjusted to be much lighter. And the lighter input image was adjusted to be darker. Color histogram matching was also approached by separating the red blue and green components of images then preforming regular histogram matching. Critically, this may cause irregularities in the original color map. For example, when performing histogram matching it is usually desired for the intensity of colors to be changed and not the color its self. However, because each RGB histogram has a unique map to the matched image the resulting color may have a different RGB ratio and so could be a different color. Orange could be purple for example. There may be better ways to represent the color of the images that are less variant to histogram matching techniques that may better preserve the original colors of the input image. For example, representing the image as

3 References:

[1] automaticaddison [Posted on March 4, 2021] [Categories Computer Vision] image processing. Difference Between Histogram Equalization and Histogram Matching. https://automaticaddison.com/difference-between-histogram-equalization-and-histogram-matching/

[2] Image Processing 101 Chapter 1.3: Color Space Conversion

https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/#:~:text=The%20Average%20method%20takes%20the,B%20is%20greater%20than%20255.

4 Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mp
import PIL as pl
import cv2
from statistics import mean
import math

from scipy.spatial import distance as dist

import numpy as np
import argparse
import glob



from google.colab import drive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
from google.colab import drive
drive.mount('/content/drive')

# Saving Image Function
'''
Saves an input image to the input file.
'''

def save_toFile( image, filename ) :
  pass



# RGB to Greyscale Function (luminosity method)

'''
Function to convert a colored image to greyscale, intensity value taken by respecting human perseption of each RGB
    value differently. Specifically, preforming weighted averaging.


https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/#:~:text=The
    %20Average%20method%20takes%20the,B%20is%20greater%20than%20255.
'''

---



from traitlets.traitlets import List
def RGB_to_Greyscale( image ) :
  temp = image.copy()
  grey = np.zeros(( len(temp), len(temp[0]) ), dtype = int )

  for i in range(0 , len( image ) ) :
    for j in range(0 , len( image[i] ) ) :
      Filter =  0.299*temp[i][j][0] + 0.587*temp[i][j][1] + 0.114*temp[i][j][2]
      grey[i][j] = int( Filter )

  return grey

# Generating Histogram Function
'''
Function, given a grayscale image, will generate the histogram of that image.
'''

def generate_histogram( image ) :
  temp = image.copy()
  hist = []
```

```
68
69    #Initialize an emply list on [0,255]
70    for i in range( 0 , 256 ) :
71      hist += [0]
72
73    #Fill list with histogram values
74    for i in range( 0 , len( image ) ) :
75      for j in range( 0 , len( image[i] ) ) :
76        intensity = int( image[i][j] )
77        hist[intensity] += 1
78
79    return hist
80
81
82  # Normalize Histogram
83   '''
84  Given a histogram on the number of pixels r returns a normalized to [0,1] copy.
85   '''
86
87
88  def normalize_histogram( histogram ) :
89    temp = histogram.copy()
90    r = sum( temp )
91    for i in range( 0 , len( temp ) ) :
92      temp[i] = temp[i]/r
93
94    return temp
95
96  # Cumulative Distributive Function (CDF) Function
97
98   '''
99  Given a histogram, generates the CDF
100  '''
101
102 def generate_CDF( histogram ) :
103   CDF = np.zeros( [256] )
104
105   CDF[0] = histogram[0]
106
107   for i in range( 1 , len( histogram ) ) :
108     CDF[i] = CDF[i-1] + histogram[i]
109
110   return CDF
111
112 # Histogram Equalization
113  '''
114 Given a CDF and an image returns a modified image.
115  '''
116
117 def histogram_equalization( image, CDF ) :
118   temp_image = image.copy()
119   equ_image = np.zeros(image.shape)
120
121   for i in range( 0 , len( temp_image ) ) :
122     for j in range( 0 , len( temp_image[i] ) ) :
123       intensity = int( temp_image[i][j] )
124       equ_image[i][j] = CDF[intensity] * 255
125
126   return equ_image
127
128
129
130 # Histogram Matching
131  '''
132 Given two normalized histograms and a target histogram, returns a representative of the first histogram matched to
        the target.
133  '''
134
135 def histogram_matching( image, histogram, target_histogram ) :
136   A_image = image.copy()
137   A_histogram = histogram.copy()
138   target = target_histogram.copy()
139
```

```
140   out_image = np.zeros(A_image.shape)
141
142   cdf_A = generate_CDF( A_histogram )
143   cdf_B = generate_CDF( target )
144
145   for i in range( 0 , len( A_image ) ) :
146     equalized_intensity = 0
147
148     for j in range( 0 , len( A_image[i] ) ) :
149       intensity = A_image[i][j]
150       equalized_intensity = cdf_A[intensity]
151
152       equiv_equal_intensity = 0
153
154       for k in range( 0 , len( cdf_B ) ) :
155         if cdf_B[k] > equalized_intensity :
156           if abs( cdf_B[k] - equalized_intensity ) >= abs( cdf_B[k-1] - equalized_intensity ) :
157             out_image[i][j] = int( k-1 )
158           else :
159             out_image[i][j] = int( k )
160
161           break
162
163   return out_image
```