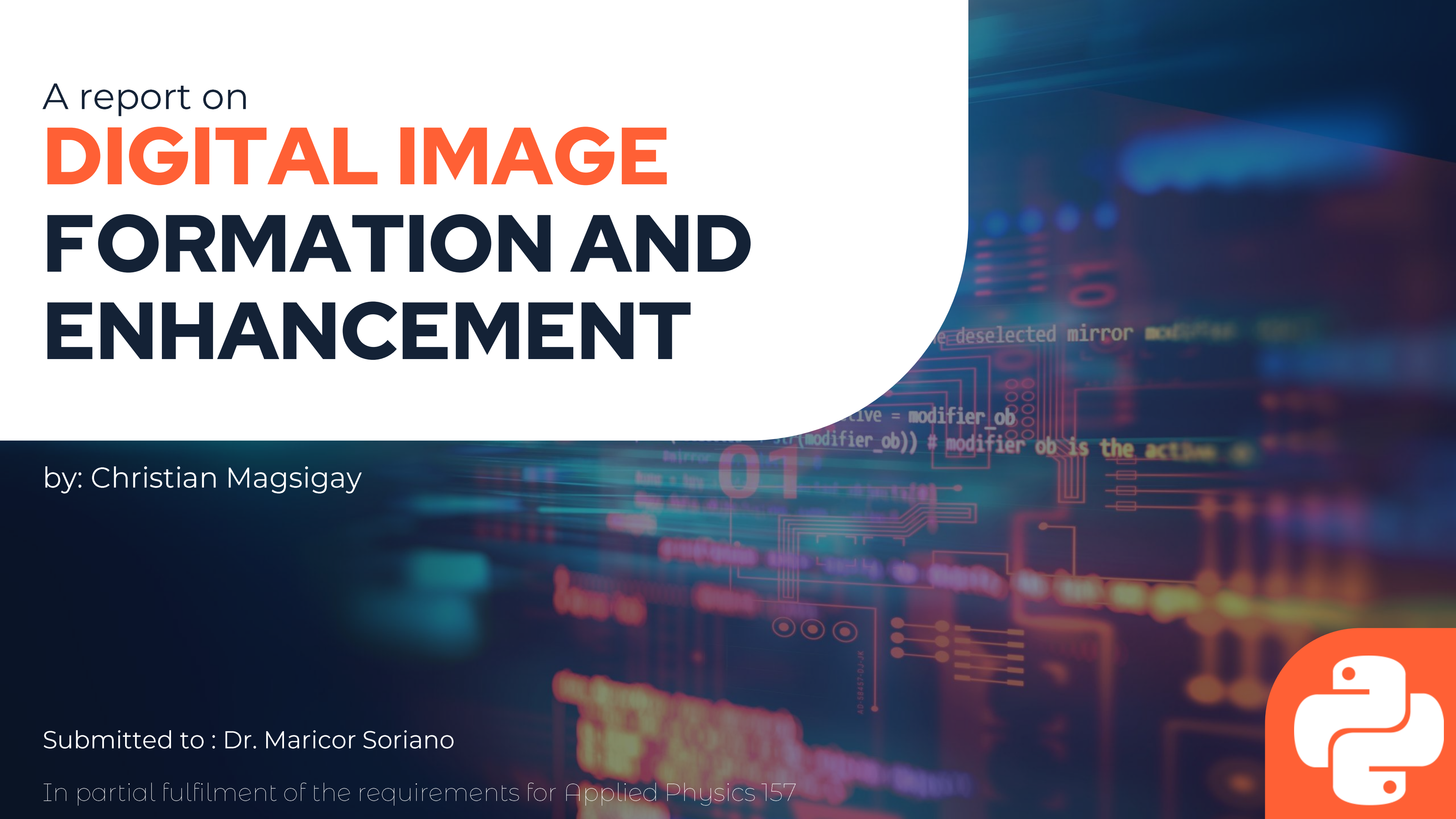
A report on

DIGITAL IMAGE FORMATION AND ENHANCEMENT

by: Christian Magsigay

Submitted to : Dr. Maricor Soriano

In partial fulfilment of the requirements for Applied Physics 157



Objectives:

- to mathematically create synthetic images
- to mathematically recreate a colored image
- to manipulate the I-O curve of a dark image
- to perform a histogram backprojection on a dark image
- to enhance an image by contrast stretching
- to perform different white balancing algorithms to restore faded photographs

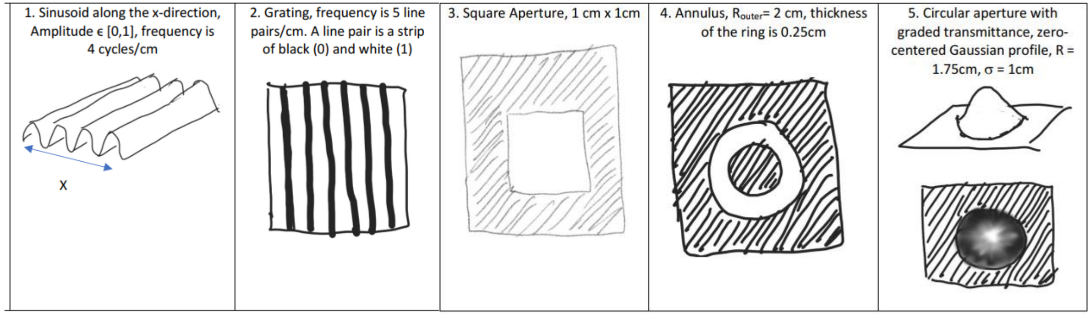
Results Analysis

All activities in this presentations were done using Python 3.0 and Adobe Photoshop CC. I chose these platforms because I am already familiar with them.

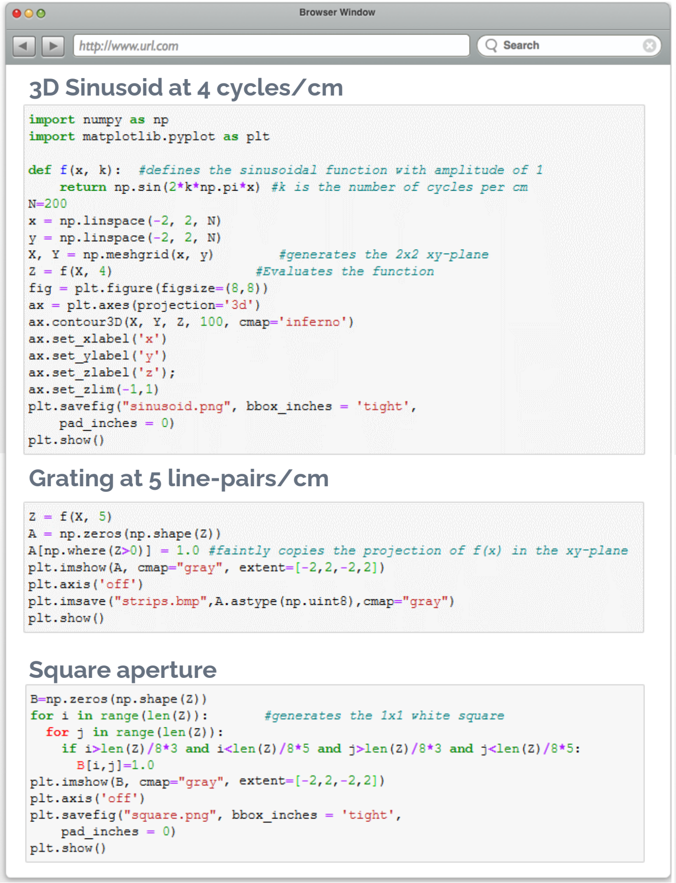
1

Creating Synthetic Images

We were tasked to create the following:



1. Sinusoid along the x-direction, Amplitude ϵ [0,1], frequency is 4 cycles/cm	2. Grating, frequency is 5 line pairs/cm. A line pair is a strip of black (0) and white (1)	3. Square Aperture, 1 cm x 1cm	4. Annulus, R_{outer}= 2 cm, thickness of the ring is 0.25cm	5. Circular aperture with graded transmittance, zero-centered Gaussian profile, R = 1.75cm, σ = 1cm



3D Sinusoid at 4 cycles/cm

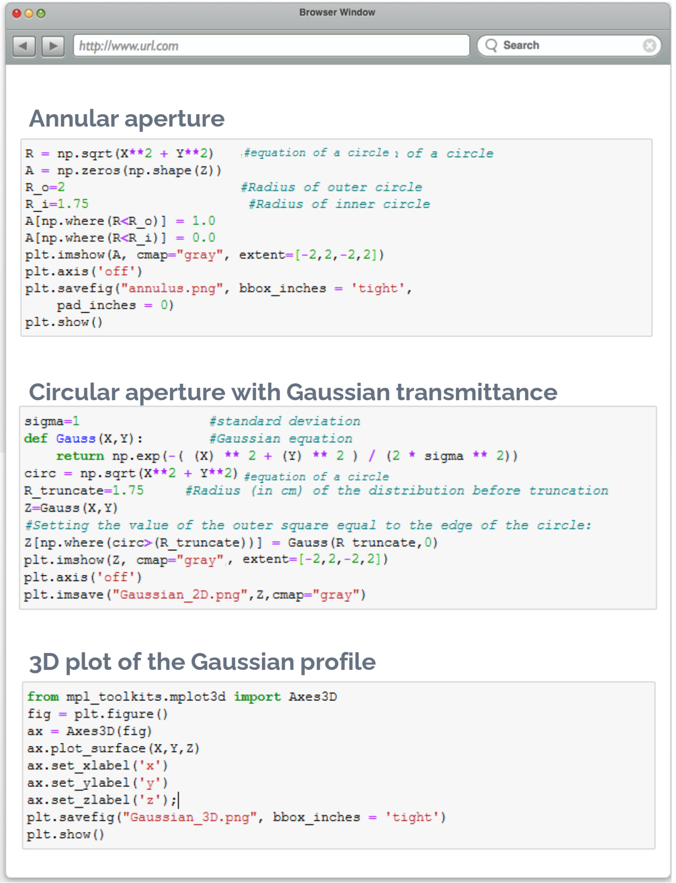
```
import numpy as np
import matplotlib.pyplot as plt

def f(x, k):  #defines the sinusoidal function with amplitude of 1
    return np.sin(2*k*np.pi*x)  #k is the number of cycles per cm
N=200
x = np.linspace(-2, 2, N)
y = np.linspace(-2, 2, N)
X, Y = np.meshgrid(x, y)          #generates the 2x2 xy-plane
Z = f(X, 4)                    #Evaluates the function
fig = plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 100, cmap='inferno')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
ax.set_zlim(-1,1)
plt.savefig("sinusoid.png", bbox_inches = 'tight',
    pad_inches = 0)
plt.show()
```

Grating at 5 line-pairs/cm

```
Z = f(X, 5)
A = np.zeros(np.shape(Z))
A[np.where(Z>0)] = 1.0 #faintly copies the projection of f(x) in the xy-plane
plt.imshow(A, cmap="gray", extent=[-2,2,-2,2])
plt.axis('off')
plt.imsave("strips.bmp",A.astype(np.uint8),cmap="gray")
plt.show()
```

Square aperture

```
B=np.zeros(np.shape(Z))
for i in range(len(Z)):        #generates the 1x1 white square
  for j in range(len(Z)):
    if i>len(Z)/8*3 and i<len(Z)/8*5 and j>len(Z)/8*3 and j<len(Z)/8*5:
      B[i,j]=1.0
plt.imshow(B, cmap="gray", extent=[-2,2,-2,2])
plt.axis('off')
plt.savefig("square.png", bbox_inches = 'tight',
    pad_inches = 0)
plt.show()
```



Annular aperture

```
R = np.sqrt(X**2 + Y**2)    ;#equation of a circle ; of a circle
A = np.zeros(np.shape(Z))
R_o=2                       #Radius of outer circle
R_i=1.75                     #Radius of inner circle
A[np.where(R<R_o)] = 1.0
A[np.where(R<R_i)] = 0.0
plt.imshow(A, cmap="gray", extent=[-2,2,-2,2])
plt.axis('off')
plt.savefig("annulus.png", bbox_inches = 'tight',
    pad_inches = 0)
plt.show()
```

Circular aperture with Gaussian transmittance

```
sigma=1                 #standard deviation
def Gauss(X,Y):         #Gaussian equation
    return np.exp(-( (X) ** 2 + (Y) ** 2 ) / (2 * sigma ** 2))
circ = np.sqrt(X**2 + Y**2) #equation of a circle
R_truncate=1.75     #Radius (in cm) of the distribution before truncation
Z=Gauss(X,Y)
#Setting the value of the outer square equal to the edge of the circle:
Z[np.where(circ>(R_truncate))] = Gauss(R truncate,0)
plt.imshow(Z, cmap="gray", extent=[-2,2,-2,2])
plt.axis('off')
plt.imsave("Gaussian_2D.png",Z,cmap="gray")
```

3D plot of the Gaussian profile

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(X,Y,Z)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
plt.savefig("Gaussian_3D.png", bbox_inches = 'tight')
plt.show()
```

Figure 1. Code for generating various synthetic images and some 3D plots.

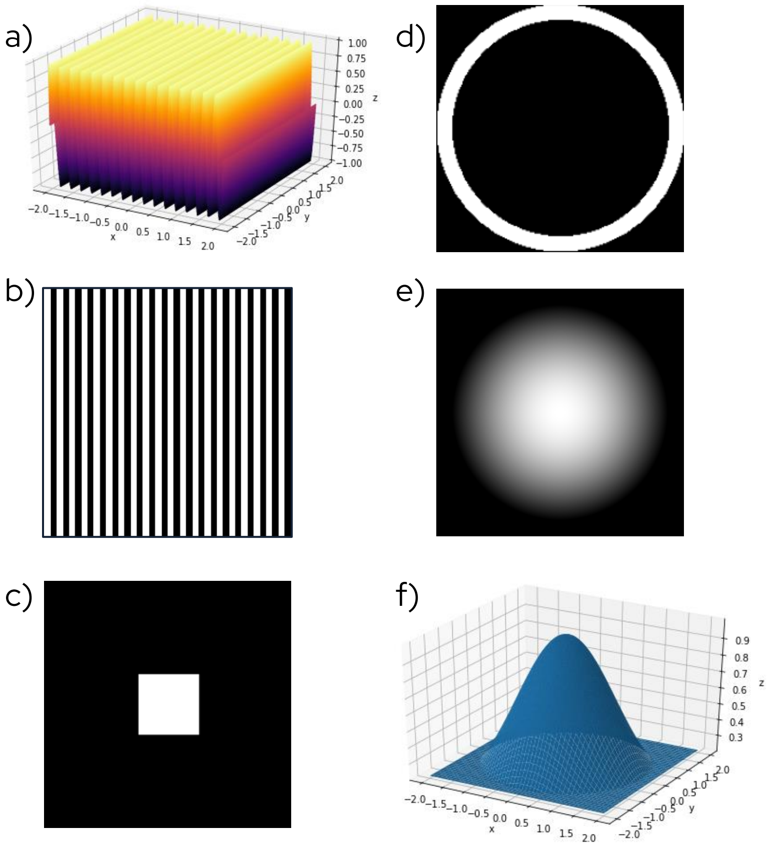


Figure 2. a) 3D plot of a sinusoid at 4 cycles / cm, b) grating at 5 line-pairs / cm, c) a 1cm x 1cm square aperture, d) an annular aperture with $R_{outer} = 2\text{cm}$ and with $R_{inner} = 1.75\text{cm}$, e) A circular aperture with Gaussian transmittance, f) the 3D plot of the Gaussian profile in e. All images have $\mathrm{X} \in [-2\text{cm}, 2\text{cm}]$ and $\mathrm{Y} \in [-2\text{cm}, 2\text{cm}]$.

The first image I generated was a 3D sinusoid. Initially, I made a user-defined function f(x,k) that would give me a sinusoidal plot that depends on x with an additional input k for the desired cycles per unit length. The amplitude was also set to 1. The variables x and y were constructed with N=200 samples to produce a 200x200 array. This number of samples will also be used for succeeding images. Both variables extend from -2 to 2 where 1 unit length represents 1 cm.

Then, a mesh grid with vectors X and Y was constructed using x and y to set up our XY-plane. In our predefined function f(x,k), X and 4 cycles/cm were inputted, and the result was our Z component. Finally, a 3D plot was generated using contour3D with X,Y, and Z as dimensions. The resulting plot has 16 cycles in total (figure 1a). .

The second image was a grating with 5 line-pairs per cm (figure 1b). To generate this, I used f(x,k) to evaluate Z, the z-component of a sinusoid along the x-axis with inputs x=X and k=5. An array filled with zeros *A* was generated with the same size as Z. I then defined a condition to set all the values of *A* to 1 at locations where the value of Z is greater than zero. A grating was then formed by displaying *A* as an image.

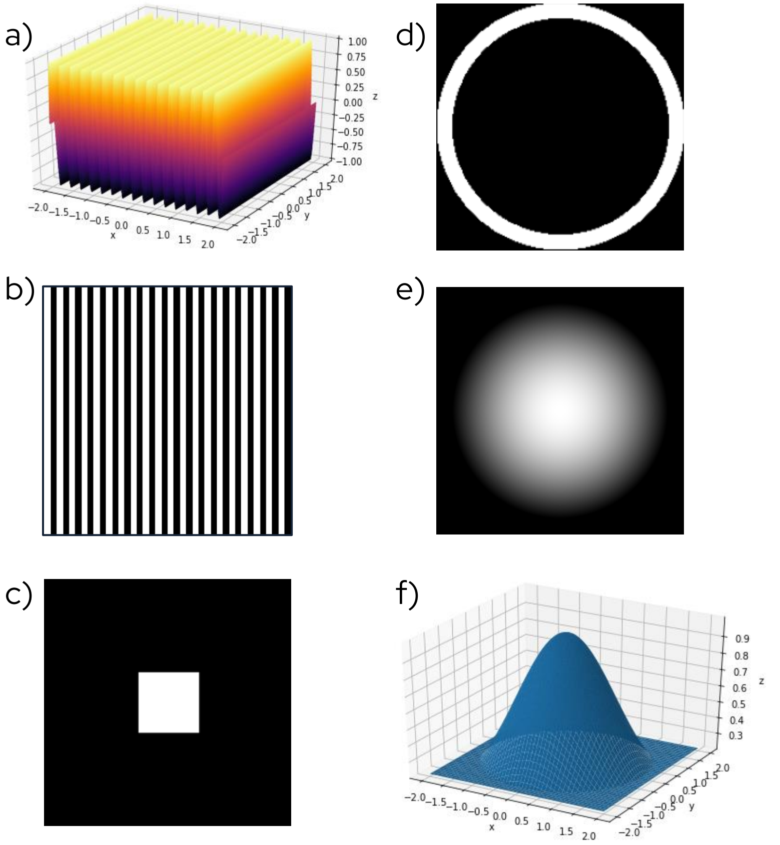


Figure 2. a) 3D plot of a sinusoid at 4 cycles / cm, b) grating at 5 line-pairs / cm, c) a 1cm x 1cm square aperture, d) an annular aperture with $R_{outer} = 2\text{cm}$ and with $R_{inner} = 1.75\text{cm}$, e) A circular aperture with Gaussian transmittance, f) the 3D plot of the Gaussian profile in e. All images have $X \in [-2\text{cm}, 2\text{cm}]$ and $Y \in [-2\text{cm}, 2\text{cm}]$.

For the square aperture (figure 1c), another array filled with zeros *B* was generated with the same shape as Z. To form the center square opening, a nested 'for loop' was used to set the targeted elements equal to 1, which was 37.5% to 62.5% of the length of both the height and width of the array.

For the annular aperture (figure 1d), I started by defining a circle *R* with the vectors X and Y generated from the previous mesh grid. Another array filled with zeros *A* was generated with the same size as the vectors. The outer and inner radii were inputted as *R_o* and *R_i*, respectively. I then set the certain elements of *A* equal to 1 where the values of *R* at those coordinates are less than *R_o*. This forms a circular aperture. To make it annular, I reverted the values at locations less than *R_i* back to 1.

Finally, I need to generate a circular aperture with Gaussian transmittance (figure 1e). So first, I defined the Gaussian function. I scaled it so that the maximum value is 1 and the standard deviation is also 1. I evaluated the function with vectors X and Y, and since we were given a specific radius for the aperture, the Gaussian was truncated at a radius of 1.75 cm. I arbitrarily set the region outside the circle to be the same value as the edge of the gaussian aperture to generate the desired 3D plot where only the circular aperture has a Gaussian profile, and the outside is completely flat (figure 1f).

Reflection

I found it interesting that arrays can be interpreted as images; how we can represent each element as a single pixel. I was amazed how we can do mathematical manipulations to pratically generate any image we want; may it be simple patterns like what we did in the activity or may be something more complex. It definitely changed the way I view matrices moving forward.

I have also observed that the more the number of samples we use the higher the image's resolution. Lastly, The images I have generated have satisfied the specifications stated in the module, hence my results are valid.

2 Recreating a colored image

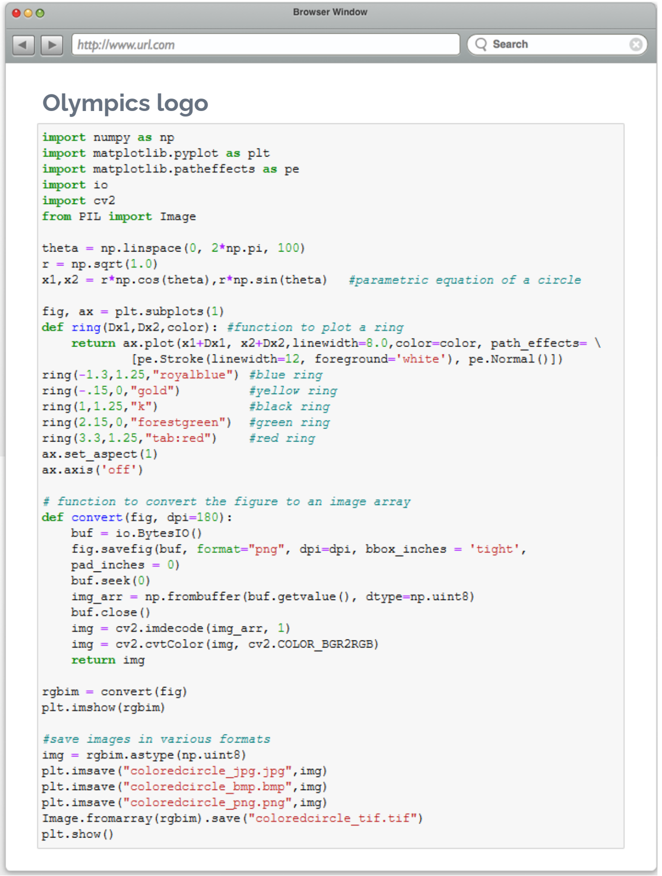
We were tasked to mathematically recreate the Olympics logo:

CODE

Main code

Figure to image conversion

Saving the image



Olympics logo

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patheffects as pe
import io
import cv2
from PIL import Image

theta = np.linspace(0, 2*np.pi, 100)
r = np.sqrt(1.0)
x1,x2 = r*np.cos(theta),r*np.sin(theta)   #parametric equation of a circle

fig, ax = plt.subplots(1)
def ring(Dx1,Dx2,color): #function to plot a ring
    return ax.plot(x1+Dx1, x2+Dx2,linewidth=8.0,color=color, path_effects= \
            [pe.Stroke(linewidth=12, foreground='white'), pe.Normal()])
ring(-1.3,1.25,"royalblue") #blue ring
ring(-.15,0,"gold")         #yellow ring
ring(1,1.25,"k")            #black ring
ring(2.15,0,"forestgreen")  #green ring
ring(3.3,1.25,"tab:red")    #red ring
ax.set_aspect(1)
ax.axis('off')

# function to convert the figure to an image array
def convert(fig, dpi=180):
    buf = io.BytesIO()
    fig.savefig(buf, format="png", dpi=dpi, bbox_inches = 'tight',
    pad_inches = 0)
    buf.seek(0)
    img_arr = np.frombuffer(buf.getvalue(), dtype=np.uint8)
    buf.close()
    img = cv2.imdecode(img_arr, 1)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return img

rgbim = convert(fig)
plt.imshow(rgbim)

#save images in various formats
img = rgbim.astype(np.uint8)
plt.imsave("coloredcircle_jpg.jpg",img)
plt.imsave("coloredcircle_bmp.bmp",img)
plt.imsave("coloredcircle_png.png",img)
Image.fromarray(rgbim).save("coloredcircle_tif.tif")
plt.show()
```

Figure 3. Code for a mathematically generated recreation of the Olympics logo.

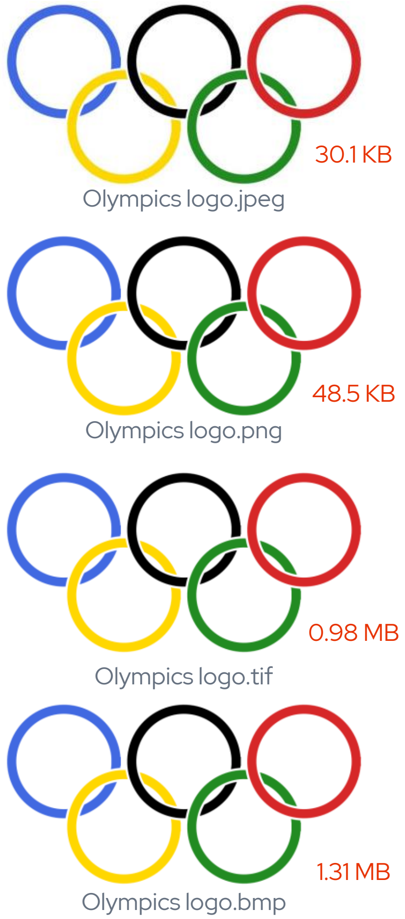


Figure 4. Mathematically recreated images of the Olympics logo in various formats and their corresponding sizes.

For this activity, I found the sample code confusing especially when adding rings and changing the colors. So, I deviated from it and did it entirely my way.

Instead of manipulating individual color channel arrays, I used plots instead. Initially, I defined a function that would plot me a ring. The input includes the x and y displacement and the color I want the ring to be. Using the function and by means of trial and error, I have generated and placed 5 rings to their designated places and assigned the closest colors to replicate the logo. I also added a white outline to each ring to faintly resemble the gaps. The problem with my approach is that it is not exactly an image array; it is a figure. So, to compensate, I defined another function that will convert my figure to an image array where I can also specify the output image's dpi (dots per inch).

Alas! I have generated an image array version of my figure at 180 dpi, and it is fairly similar to the original logo! What's left is to save the image in various formats which includes: jpeg, png, bmp, and tif.

Visually, all formats look quite the same until you zoom it in and see that jpeg lost more quality than the others. It can also be observed that jpeg has the least size among the formats and the bmp has the largest. Both tif and bmp formats produced the highest quality of the image, and it looks lossless, but I still prefer png because the quality is almost as good as the previous formats but with a significantly smaller size.

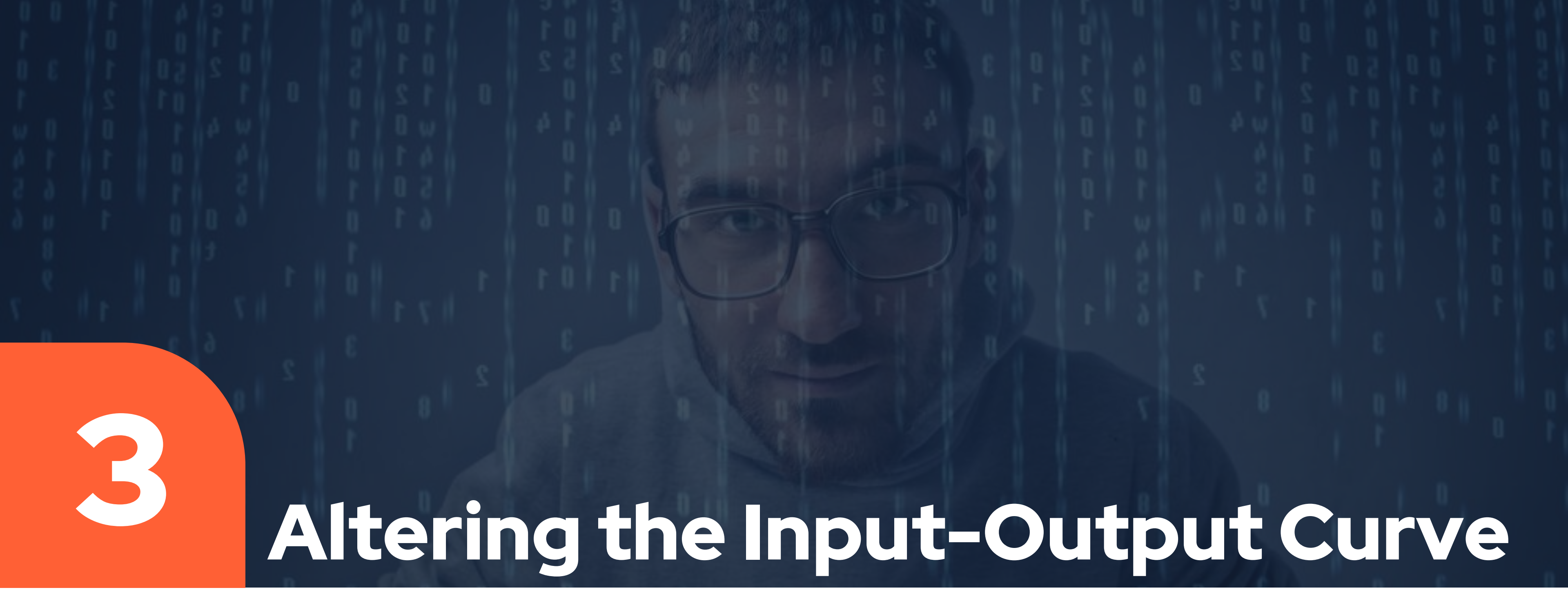
Reflection

The activity was really fun even if I got stuck when I tried to modify the sample code. Fortunately, I eventually thought of a new approach to satisfy the main objective. The algorithm I used to convert the figure to an image array was patterned in a code shared in StackOverflow [1]. The resulting image array resembles to the original Olympics logo; hence the result is valid.

Even if I ended up not using the RGB channels, I still learned a lot about them and how they produce a variety of colors by tweaking each individual RGB value. I also learned about the differences between image formats, and it gave me ideas on when to use or not use them.

References:

[1] "Matplotlib figure to image as a numpy array." Stack Overflow, 11 May. 2021, stackoverflow.com/questions/35355930/matplotlib-figure-to-image-as-a-numpy-array.



3

Altering the Input-Output Curve

We were tasked to manipulate an image's I-O curve.

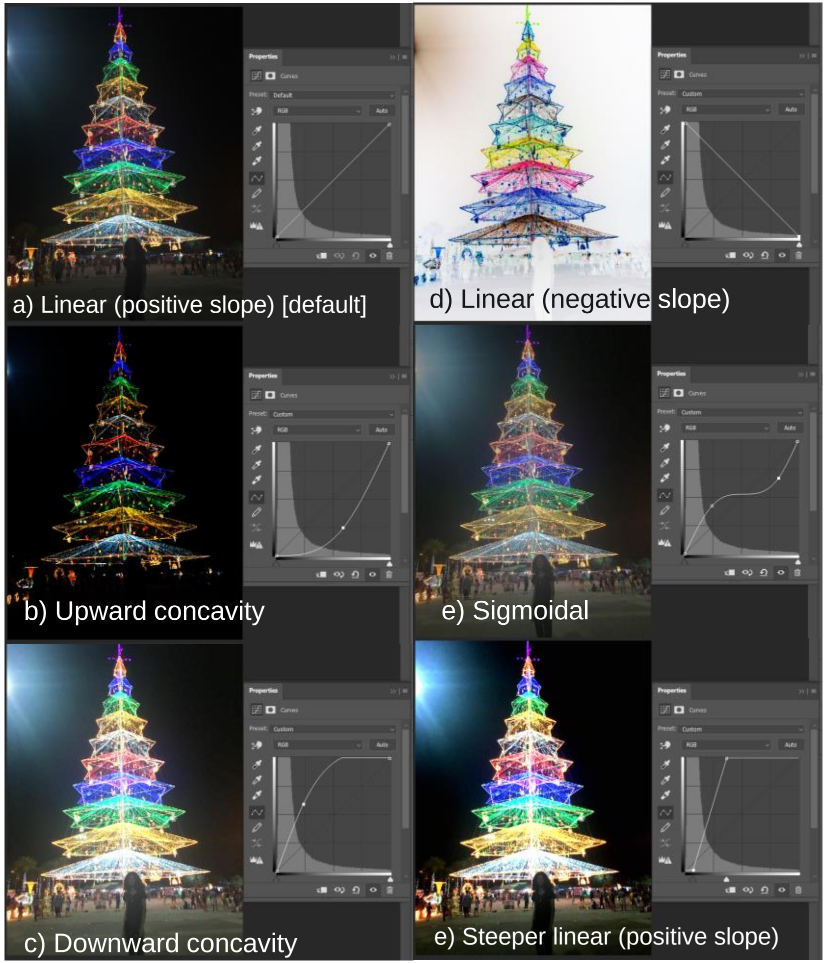

a) Linear (positive slope) [default]
d) Linear (negative slope)
b) Upward concavity
e) Sigmoidal
c) Downward concavity
e) Steeper linear (positive slope)

Figure 5. Manipulating the input-output curve of a dark image in Adobe Photoshop CC.

Adobe Photoshop CC allows us to manipulate our photos using the input-output curve, as seen in figure 5. Initially, a dark photo was imported in Photoshop, and its default I-O curve is linear with a positive slope. I observed that if we manipulate the curve to have an upward concavity, the image is darker, but the contrast increases. Conversely, if the curve has a downward concavity, the image looks brighter, but the contrast decreases; in this setting, the darker details from the original photo are now more visible.

Additionally, a linear negative slope inverts all the colors of the image. A sigmoidal-looking curve makes the image brighter but reduces the contrast of the highlights. Lastly, with a steeper linear positive slope, the image is brighter and has more contrast; this setting is the best one because it revealed the darker details of the image as well as improved the vibrancy of the colors.

Take note that I manipulated the RGB curve, which adjusts all the color channels, but we can also manipulate only the red, green, and blue curves if we want to fine-tune each color value.

Reflection

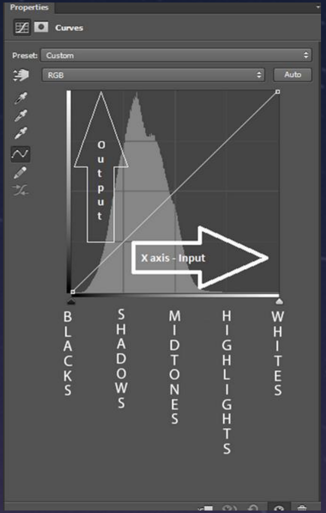


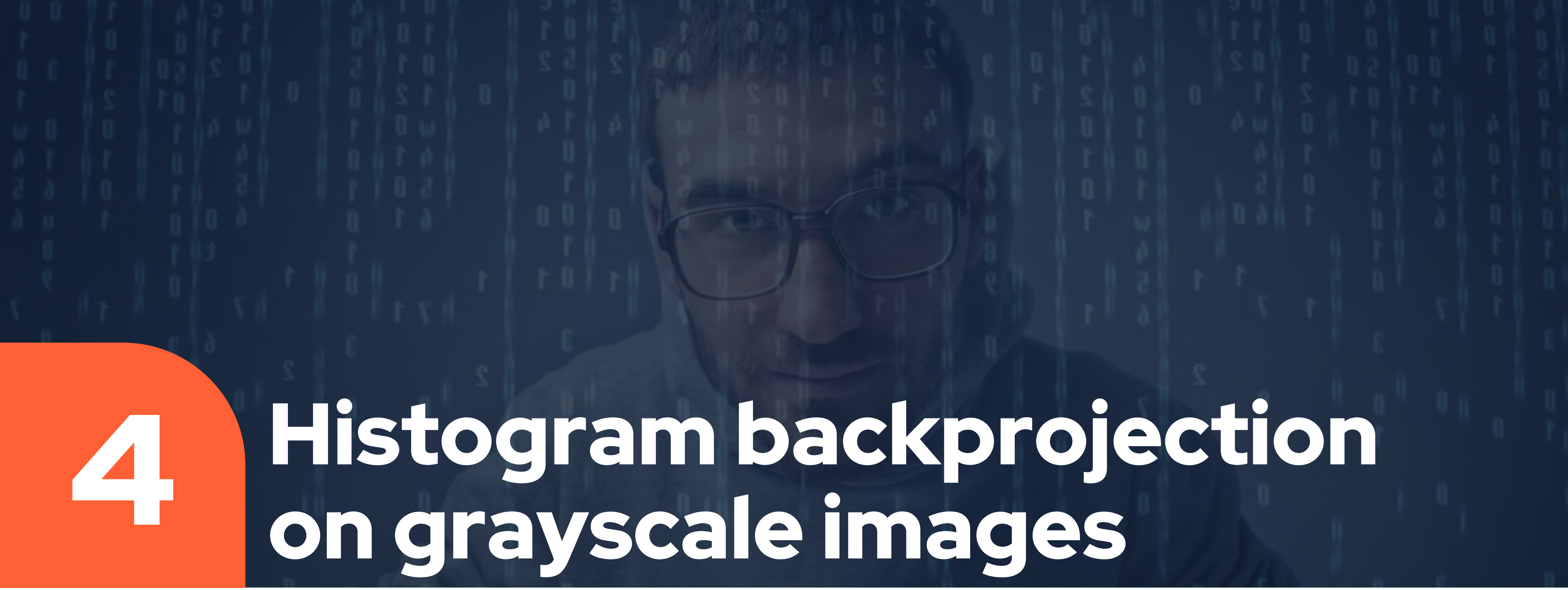
Figure 6. Sample image curve [1]

I have used the Curves feature of Photoshop many times before, but only now did I know how it works and what it represents. The graph in the background is actually the histogram of the image. The X and Y axis of the curve represent the blacks, shadows, midtones, highlights, and whites of the image, as seen in figure 6. If you think about this concept and look back to the previous slides, the results actually make sense. This will definitely help me have more control when I edit photos in the future.

I also used curves in manipulating the color channels of my image; for example, when my image is too orange, I manipulate the red curve to reduce the intensity of the warm colors.

References:

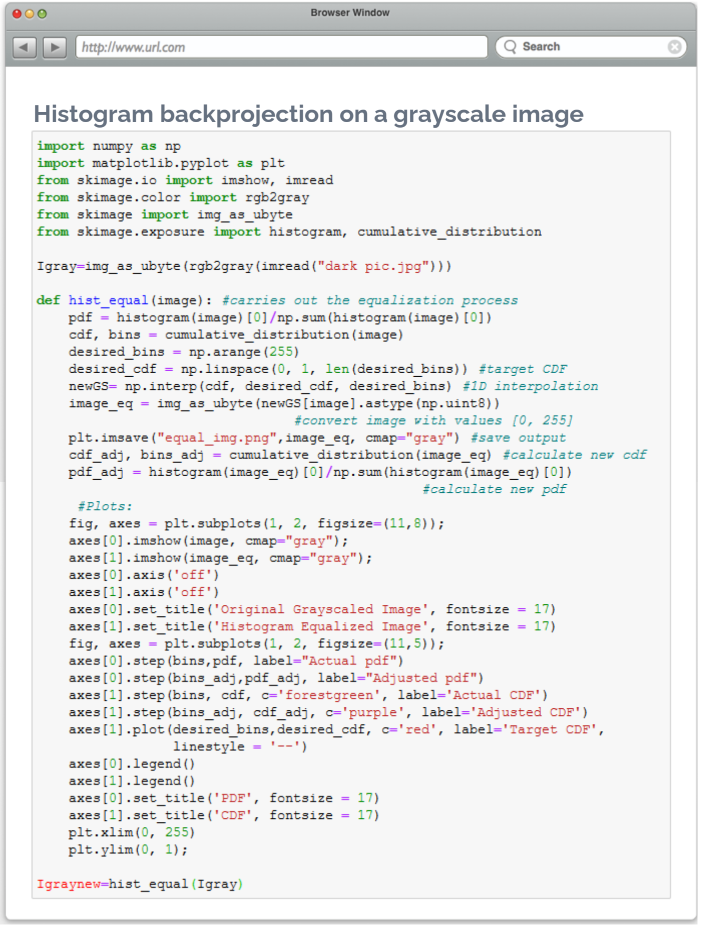
[1] "How to Use Curves in Photoshop." Photography Life, 12 Mar. 2019, photographylife.com/how-to-use-curves-in-photoshop.



4 Histogram backprojection on grayscale images

We were tasked to generate the histogram equalized image of a dark grayscaled photo.

CODE



Histogram backprojection on a grayscale image

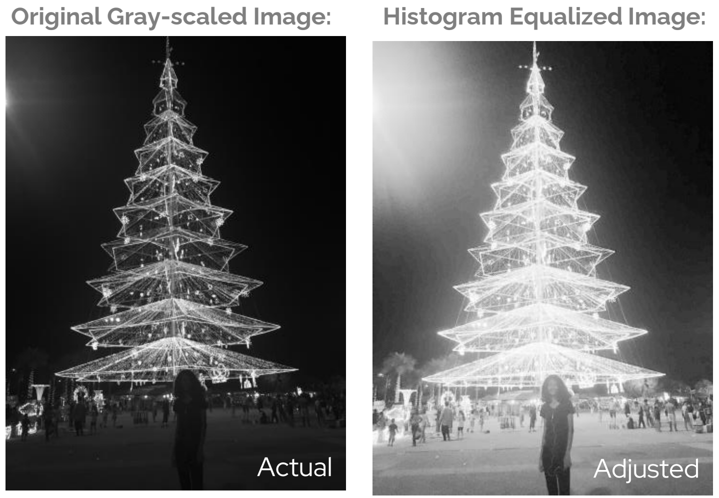
```
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imshow, imread
from skimage.color import rgb2gray
from skimage import img_as_ubyte
from skimage.exposure import histogram, cumulative_distribution

Igray=img_as_ubyte(rgb2gray(imread("dark pic.jpg")))

def hist_equal(image): #carries out the equalization process
    pdf = histogram(image)[0]/np.sum(histogram(image)[0])
    cdf, bins = cumulative_distribution(image)
    desired_bins = np.arange(255)
    desired_cdf = np.linspace(0, 1, len(desired_bins)) #target CDF
    newGS= np.interp(cdf, desired_cdf, desired_bins) #1D interpolation
    image_eq = img_as_ubyte(newGS[image].astype(np.uint8))
                                #convert image with values [0, 255]
    plt.imsave("equal_img.png",image_eq, cmap="gray") #save output
    cdf_adj, bins_adj = cumulative_distribution(image_eq) #calculate new cdf
    pdf_adj = histogram(image_eq)[0]/np.sum(histogram(image_eq)[0])
                                                #calculate new pdf
     #Plots:
    fig, axes = plt.subplots(1, 2, figsize=(11,8));
    axes[0].imshow(image, cmap="gray");
    axes[1].imshow(image_eq, cmap="gray");
    axes[0].axis('off')
    axes[1].axis('off')
    axes[0].set_title('Original Grayscaled Image', fontsize = 17)
    axes[1].set_title('Histogram Equalized Image', fontsize = 17)
    fig, axes = plt.subplots(1, 2, figsize=(11,5));
    axes[0].step(bins,pdf, label="Actual pdf")
    axes[0].step(bins_adj,pdf_adj, label="Adjusted pdf")
    axes[1].step(bins, cdf, c='forestgreen', label='Actual CDF')
    axes[1].step(bins_adj, cdf_adj, c='purple', label='Adjusted CDF')
    axes[1].plot(desired_bins,desired_cdf, c='red', label='Target CDF',
                 linestyle = '--')
    axes[0].legend()
    axes[1].legend()
    axes[0].set_title('PDF', fontsize = 17)
    axes[1].set_title('CDF', fontsize = 17)
    plt.xlim(0, 255)
    plt.ylim(0, 1);

Igraynew=hist_equal(Igray)
```

Main code

Plotting preferences

Executing the user-defined function to the image

Figure 7. Code for histogram backprojection on a grayscale image



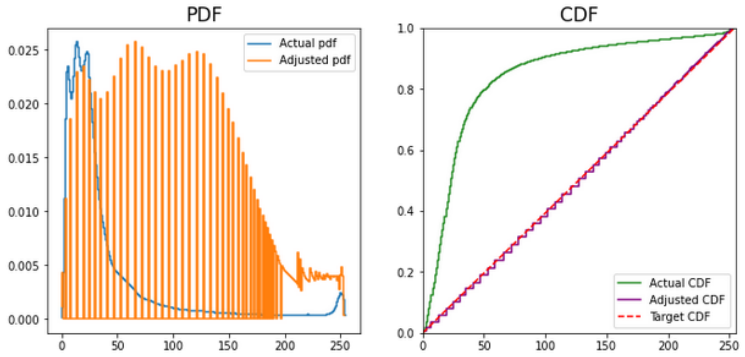


Figure 8. The original gray-scaled image vs the histogram equalized image with their corresponding actual and adjusted PDF and CDF plots.

Initially, I chose a photo I took three years ago, which was of my sister in front of a brightly lit Christmas tree, and it was taken against the light; we can hardly see any details besides the tree.

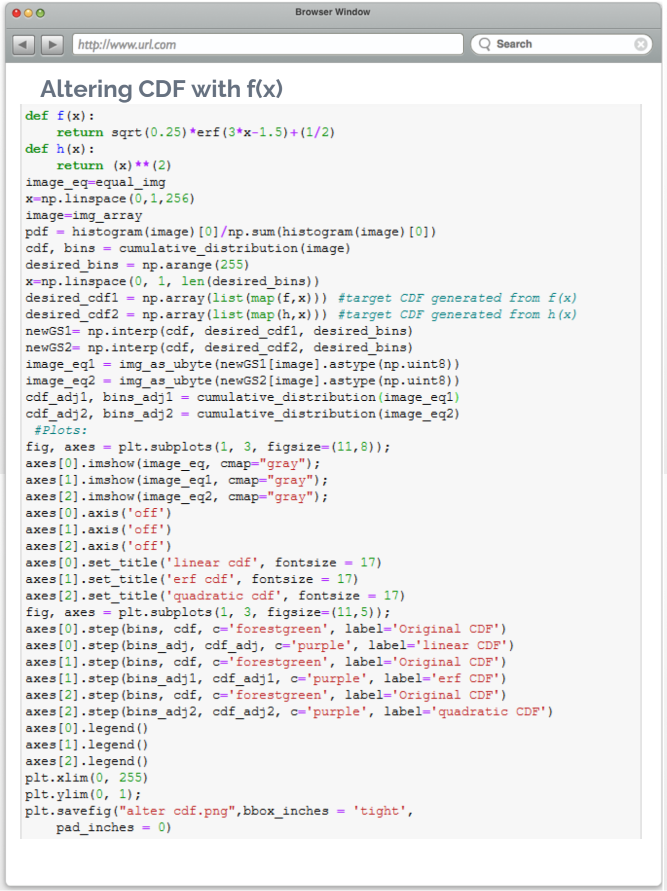
The first step was to convert the dark image into grayscale. Then, the probability distribution function (PDF) and the cumulative distribution function (CDF) were determined. Afterward, for the equalization process, a linearly increasing array was generated, which will be the target trend of the new CDF. The array was then used to backproject its values to the original CDF of the image. The last step was to transform the CDF back to an image array, and the result was a histogram equalized image.

This new image brightened dark areas revealing details we can't see before. The algorithm also improved the overall contrast of the image. If we take the PDF and CDF of the new image; we can see that the distribution was changed if we compare it with the original as seen in figure 8. The new PDF is now more spread out throughout the 0-255 range, and the new CDF was transformed into a step function that is well-fitted with the target CDF.

Defined equations

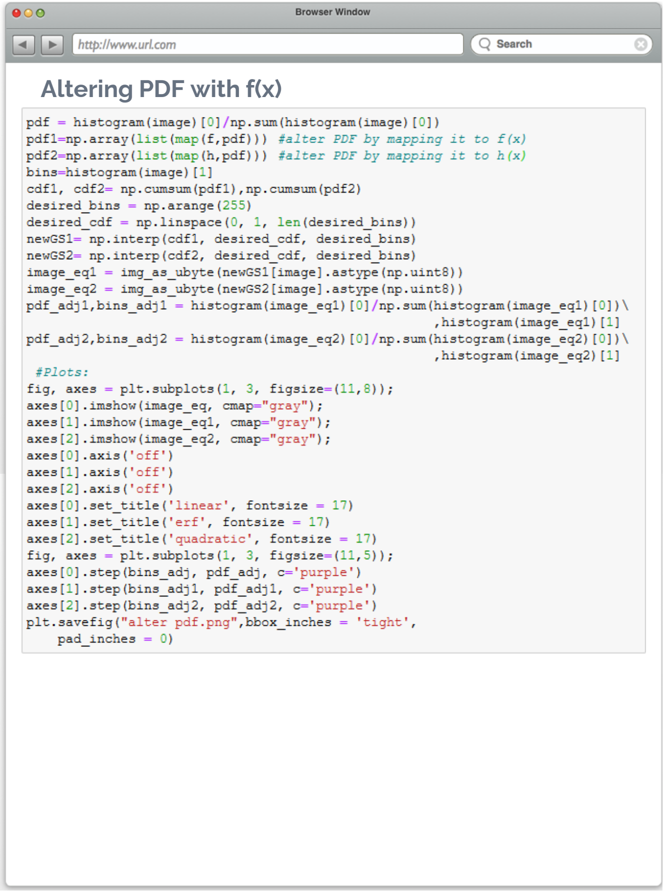
Main code

Plotting preferences



Altering CDF with f(x)

```
def f(x):
    return sqrt(0.25)*erf(3*x-1.5)+(1/2)
def h(x):
    return (x)**(2)
image_eq=equal_img
x=np.linspace(0,1,256)
image=img_array
pdf = histogram(image)[0]/np.sum(histogram(image)[0])
cdf, bins = cumulative_distribution(image)
desired_bins = np.arange(255)
x=np.linspace(0, 1, len(desired_bins))
desired_cdf1 = np.array(list(map(f,x))) #target CDF generated from f(x)
desired_cdf2 = np.array(list(map(h,x))) #target CDF generated from h(x)
newGS1= np.interp(cdf, desired_cdf1, desired_bins)
newGS2= np.interp(cdf, desired_cdf2, desired_bins)
image_eq1 = img_as_ubyte(newGS1[image].astype(np.uint8))
image_eq2 = img_as_ubyte(newGS2[image].astype(np.uint8))
cdf_adj1, bins_adj1 = cumulative_distribution(image_eq1)
cdf_adj2, bins_adj2 = cumulative_distribution(image_eq2)
 #Plots:
fig, axes = plt.subplots(1, 3, figsize=(11,8));
axes[0].imshow(image_eq, cmap="gray");
axes[1].imshow(image_eq1, cmap="gray");
axes[2].imshow(image_eq2, cmap="gray");
axes[0].axis('off')
axes[1].axis('off')
axes[2].axis('off')
axes[0].set_title('linear cdf', fontsize = 17)
axes[1].set_title('erf cdf', fontsize = 17)
axes[2].set_title('quadratic cdf', fontsize = 17)
fig, axes = plt.subplots(1, 3, figsize=(11,5));
axes[0].step(bins, cdf, c='forestgreen', label='Original CDF')
axes[0].step(bins_adj, cdf_adj, c='purple', label='linear CDF')
axes[1].step(bins, cdf, c='forestgreen', label='Original CDF')
axes[1].step(bins_adj1, cdf_adj1, c='purple', label='erf CDF')
axes[2].step(bins, cdf, c='forestgreen', label='Original CDF')
axes[2].step(bins_adj2, cdf_adj2, c='purple', label='quadratic CDF')
axes[0].legend()
axes[1].legend()
axes[2].legend()
plt.xlim(0, 255)
plt.ylim(0, 1);
plt.savefig("alter cdf.png",bbox_inches = 'tight',
    pad_inches = 0)
```



Altering PDF with f(x)

```
pdf = histogram(image)[0]/np.sum(histogram(image)[0])
pdf1=np.array(list(map(f,pdf))) #alter PDF by mapping it to f(x)
pdf2=np.array(list(map(h,pdf))) #alter PDF by mapping it to h(x)
bins=histogram(image)[1]
cdf1, cdf2= np.cumsum(pdf1),np.cumsum(pdf2)
desired_bins = np.arange(255)
desired_cdf = np.linspace(0, 1, len(desired_bins))
newGS1= np.interp(cdf1, desired_cdf, desired_bins)
newGS2= np.interp(cdf2, desired_cdf, desired_bins)
image_eq1 = img_as_ubyte(newGS1[image].astype(np.uint8))
image_eq2 = img_as_ubyte(newGS2[image].astype(np.uint8))
pdf_adj1,bins_adj1 = histogram(image_eq1)[0]/np.sum(histogram(image_eq1)[0])\
                                                    ,histogram(image_eq1)[1]
pdf_adj2,bins_adj2 = histogram(image_eq2)[0]/np.sum(histogram(image_eq2)[0])\
                                                    ,histogram(image_eq2)[1]
 #Plots:
fig, axes = plt.subplots(1, 3, figsize=(11,8));
axes[0].imshow(image_eq, cmap="gray");
axes[1].imshow(image_eq1, cmap="gray");
axes[2].imshow(image_eq2, cmap="gray");
axes[0].axis('off')
axes[1].axis('off')
axes[2].axis('off')
axes[0].set_title('linear', fontsize = 17)
axes[1].set_title('erf', fontsize = 17)
axes[2].set_title('quadratic', fontsize = 17)
fig, axes = plt.subplots(1, 3, figsize=(11,5));
axes[0].step(bins_adj, pdf_adj, c='purple')
axes[1].step(bins_adj1, pdf_adj1, c='purple')
axes[2].step(bins_adj2, pdf_adj2, c='purple')
plt.savefig("alter pdf.png",bbox_inches = 'tight',
    pad_inches = 0)
```

Main code

Plotting preferences

Figure 9. Code for altering the CDF and PDF of an image by mapping it into predefined equations.

Image after altering CDF:

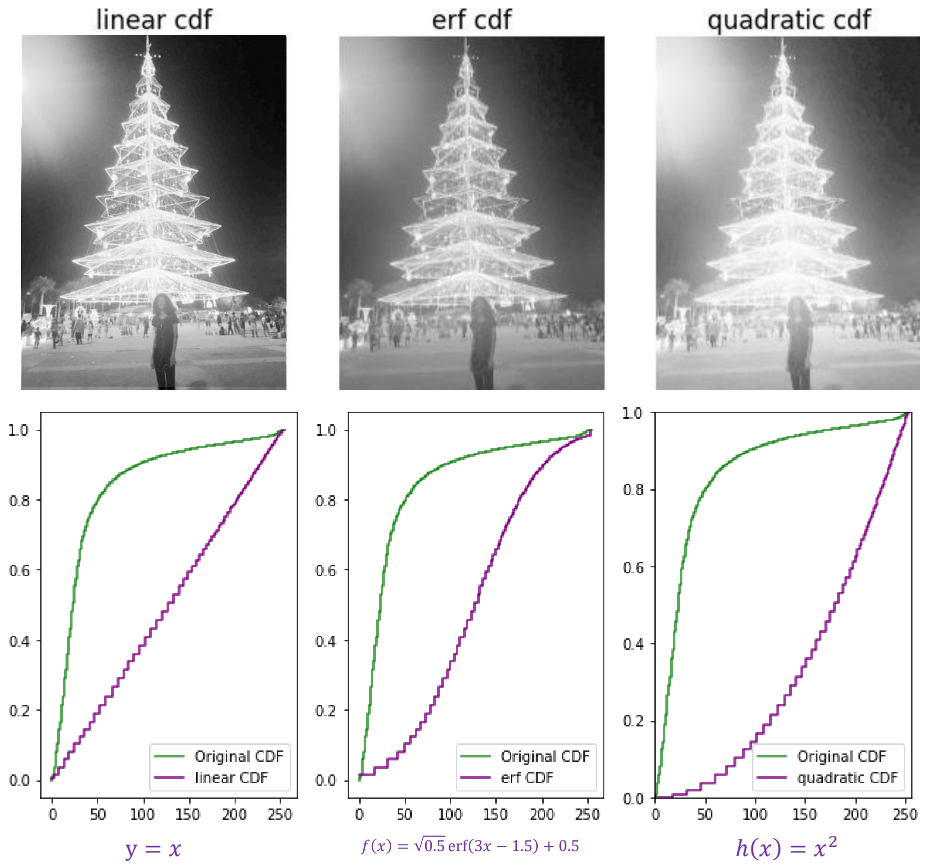


Figure 10. Resulting images and the plot of the new CDF after altering the original CDF by an erf and quadratic function.

Now, we examine the effect of altering the CDF to the resulting image. The majority of the process in this section is the same as the histogram equalization algorithm. The only significant difference is that instead of interpolating the CDF with a linearly increasing array, we use nonlinear functions instead, particularly an erf and a quadratic function.

As seen in figure 10, the CDFs were transformed according to the defined functions. We can see there that with an erf CDF, the resulting image is brighter but with some reduction of contrast if we compare it with the histogram equalized image. Similarly, the quadratic CDF made the image much brighter, and it also reduced its contrast.

Image after altering PDF:

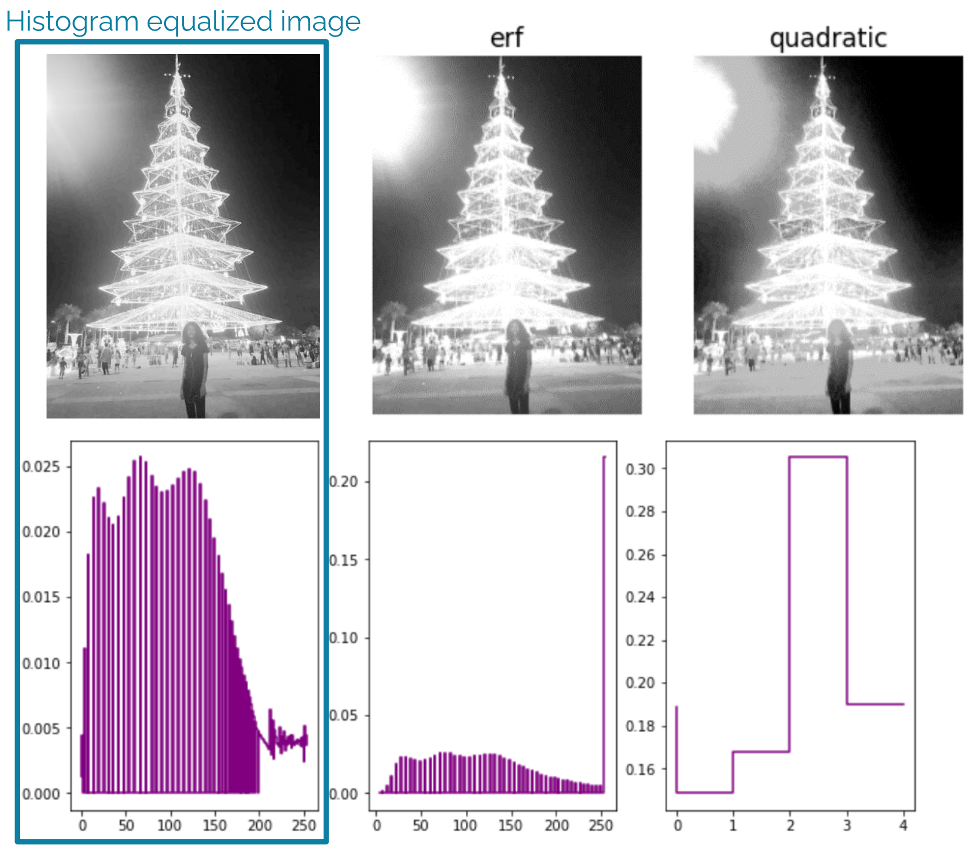


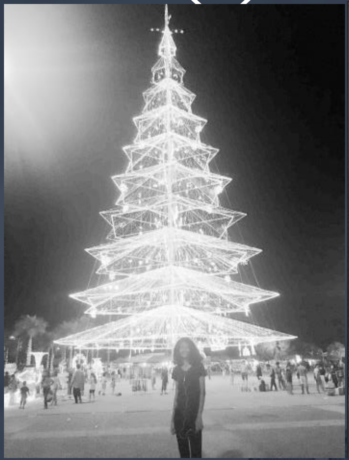
Figure 11. Resulting images and the plot of the new PDF after altering the original PDF by mapping with an erf and quadratic function.

This section now explores the effects if we alter the PDF of the image. Like the previous section, the majority of the process is the same as the histogram equalization algorithm. The only major difference is that the PDFs were initially mapped with the erf and quadratic functions we defined earlier before they underwent the equalization process.

As seen in figure 11, the PDF that was altered by the erf function is a tad brighter than the equalized image, and it also has more contrast. The PDF altered by the quadratic function, on the other hand, had the highest contrast but the gradient between white and black is now rougher with fewer graded variations.

Reflection

Histogram equalized image:



Equalized image using Photoshop:

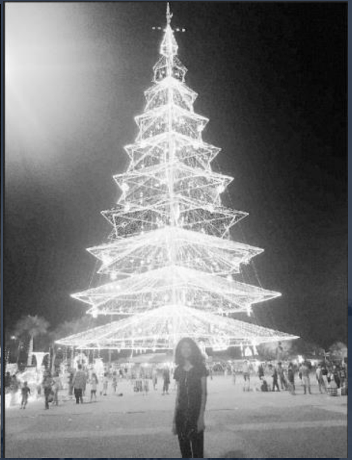


Figure 12. The histogram equalized image from this activity vs. the equalized image using Adobe Photoshop.

I learned so much from this activity. Being able to apply math in image equalization or other image manipulation blew my mind. It definitely made me appreciate the editing tools that I may have taken for granted through the years. I never realized that the underlying principles were intricate and elegant math equations hiding behind a simple option like the 'equalize' adjustment in Photoshop.

If we refer to figure 12, our result is similar to the equalized image, which was edited in Adobe Photoshop. This validates the result of this activity.

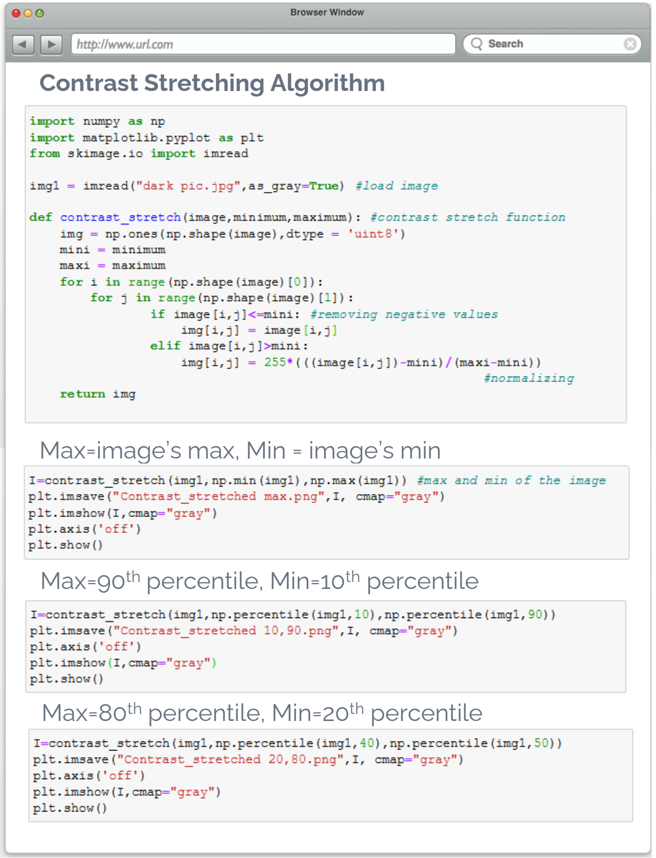
Representing an image as a distribution is also a new thing for me. It gave me a new perspective on photo editing. I usually see these plots in the side panel of my editing software. Now that I have general knowledge about them, maybe I can use them as an advantage to further improve my skills in graphic design and photo editing.

5

Contrast Enhancement

We were tasked to enhance a grayscale image by contrast stretching.

CODE



Contrast Stretching Algorithm

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread

img1 = imread("dark pic.jpg",as_gray=True) #load image

def contrast_stretch(image,minimum,maximum): #contrast stretch function
    img = np.ones(np.shape(image),dtype = 'uint8')
    mini = minimum
    maxi = maximum
    for i in range(np.shape(image)[0]):
        for j in range(np.shape(image)[1]):
                if image[i,j]<=mini: #removing negative values
                    img[i,j] = image[i,j]
                elif image[i,j]>mini:
                    img[i,j] = 255*(((image[i,j])-mini)/(maxi-mini))
                                                            #normalizing
    return img
```

Max=image's max, Min = image's min

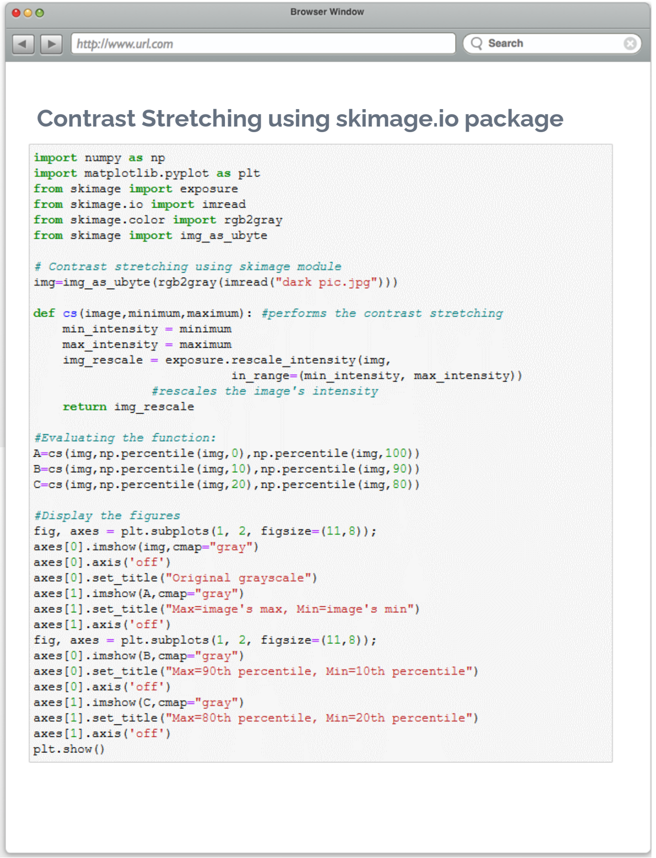
```python
I=contrast_stretch(img1,np.min(img1),np.max(img1)) #max and min of the image
plt.imsave("Contrast_stretched max.png",I, cmap="gray")
plt.imshow(I,cmap="gray")
plt.axis('off')
plt.show()
```

Max=90th percentile, Min=10th percentile

```python
I=contrast_stretch(img1,np.percentile(img1,10),np.percentile(img1,90))
plt.imsave("Contrast_stretched 10,90.png",I, cmap="gray")
plt.axis('off')
plt.imshow(I,cmap="gray")
plt.show()
```

Max=80th percentile, Min=20th percentile

```python
I=contrast_stretch(img1,np.percentile(img1,40),np.percentile(img1,50))
plt.imsave("Contrast_stretched 20,80.png",I, cmap="gray")
plt.axis('off')
plt.imshow(I,cmap="gray")
plt.show()
```



Contrast Stretching using skimage.io package

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import exposure
from skimage.io import imread
from skimage.color import rgb2gray
from skimage import img_as_ubyte

# Contrast stretching using skimage module
img=img_as_ubyte(rgb2gray(imread("dark pic.jpg")))

def cs(image,minimum,maximum): #performs the contrast stretching
    min_intensity = minimum
    max_intensity = maximum
    img_rescale = exposure.rescale_intensity(img,
                           in_range=(min_intensity, max_intensity))
                #rescales the image's intensity
    return img_rescale

#Evaluating the function:
A=cs(img,np.percentile(img,0),np.percentile(img,100))
B=cs(img,np.percentile(img,10),np.percentile(img,90))
C=cs(img,np.percentile(img,20),np.percentile(img,80))

#Display the figures
fig, axes = plt.subplots(1, 2, figsize=(11,8));
axes[0].imshow(img,cmap="gray")
axes[0].axis('off')
axes[0].set_title("Original grayscale")
axes[1].imshow(A,cmap="gray")
axes[1].set_title("Max=image's max, Min=image's min")
axes[1].axis('off')
fig, axes = plt.subplots(1, 2, figsize=(11,8));
axes[0].imshow(B,cmap="gray")
axes[0].set_title("Max=90th percentile, Min=10th percentile")
axes[0].axis('off')
axes[1].imshow(C,cmap="gray")
axes[1].set_title("Max=80th percentile, Min=20th percentile")
axes[1].axis('off')
plt.show()
```

Figure 13. Code for contrast stretching

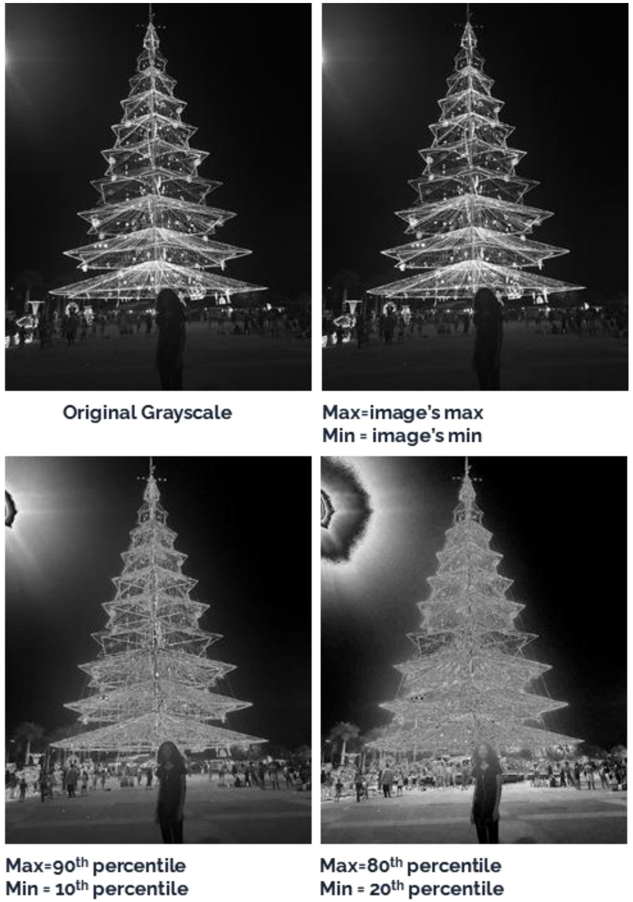


Figure 14. Resulting images after contrast stretching

This contrast stretching algorithm aims to improve the contrast of an image by stretching the intensity values to fill the entire dynamic range, which is dictated by the lower and upper values of the image. The minimum value of the input image is mapped to 0, and the maximum value is mapped to 255. All the values in between are reassigned new intensity values by:

$$I_{new} = \frac{I_{old} - I_{min}}{I_{max} - I_{min}}$$

The formula did nothing to improve the gray-scaled image since the photo already had a minimum value of 0 and a maximum of 255. I also tried to improve the image by setting the minimum and maximum intensities to certain percentile values. As I clipped both ends of the distribution by using the percentile values, some parts of the image were improved, but it also has problems like the inversion of intensities where what was supposed to be white turned into black. The contrast is also lower than anticipated, and it made the image duller.

Hence, this specific algorithm I carried out by solely relying on the given equation works only with min-max contrast stretching and not so well with percentile contrast stretching.

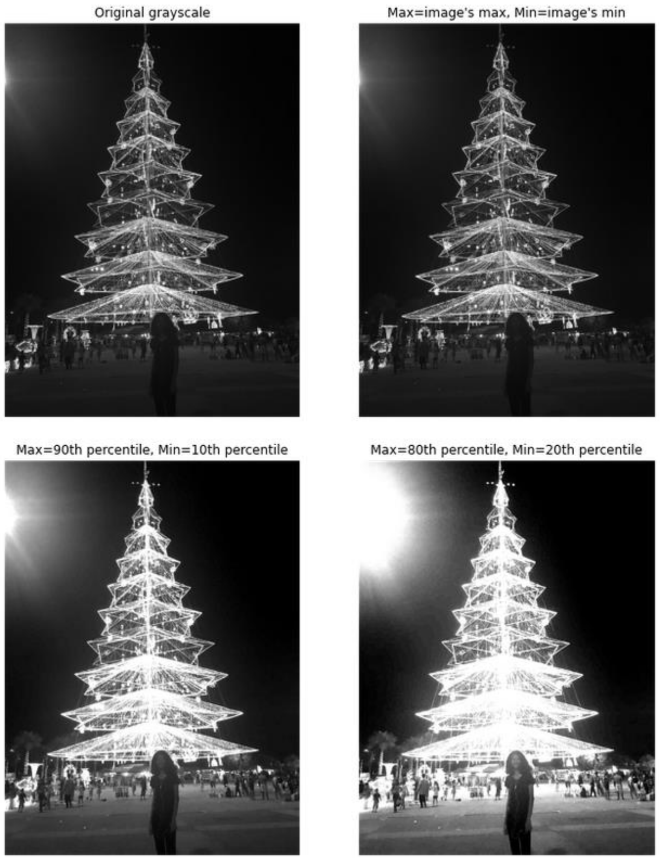
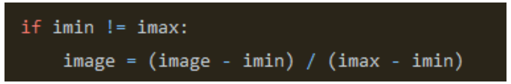

Original grayscale
Max=image's max, Min=image's min
Max=90th percentile, Min=10th percentile
Max=80th percentile, Min=20th percentile

Figure 15. Resulting images after contrast stretching with skimage.io package.

To address the issues encountered during the clipping at certain percentile values, I used the rescale_intenstity function in skimage.io package to adjust the contrast of the image by stretching it to a certain percentile value at both ends of the image's intensity distribution. The backbone of the said function is the same equation we used previously:



```
if imin != imax:
    image = (image - imin) / (imax - imin)
```

Figure 16. Snippet of the source code of the rescale_intensity function in skimage.io package

The major difference is that this function comprehensively retains the values of the intensities beyond the specified percentile intervals. Hence there are no unintentional color inversions and other distortions like those we saw in the previous slide.

From figure 15, we can see that as we decrease the distance between the chosen values for the minimum and maximum intensity, the contrast of the image increases. Overall, we can say that the percentile contrast stretching is also an effective technique in revealing more details in our darkened photos.

Also, the resulting images using the original's maximum and minimum values for both contrast stretching algorithms are identical. This gives us confidence that the resulting images in this activity are correct.

Reflection

In this activity, I struggled to filter out unwanted distortions when I carry out the percentile contrast stretching algorithm. So, I decided to seek help from packages to address this issue. The source code can be accessed in Github [1].

This method is a nice way to enhance grayscale images. I was able to brighten up dark areas in the image and reveal underlying details. Although it does have limitations, when you increase the contrast you also loose details in the lighter portions of the image. The histogram equalization process is a better way to address that problem.

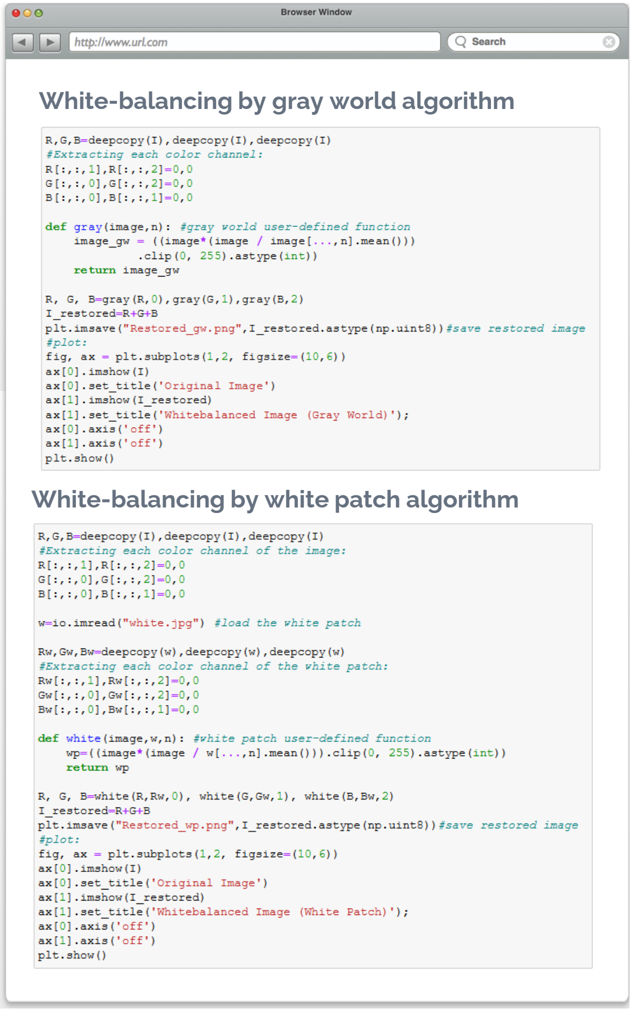
References:

[1] scikit-image. "scikit-image." GitHub, 13 May. 2021, github.com/scikit-image/scikit-image/blob/main/skimage/exposure/exposure.py#L313-L428.

6 Restoring Faded Colored Photographs

We were tasked to perform various white balancing algorithms to restore an old photograph.

CODE

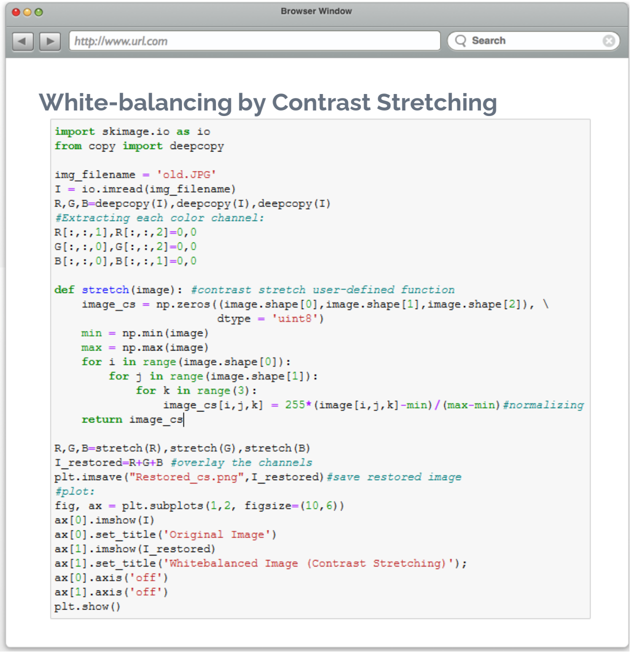


White-balancing by gray world algorithm

```
R,G,B=deepcopy(I),deepcopy(I),deepcopy(I)
#Extracting each color channel:
R[:,:,1],R[:,:,2]=0,0
G[:,:,0],G[:,:,2]=0,0
B[:,:,0],B[:,:,1]=0,0

def gray(image,n): #gray world user-defined function
    image_gw = ((image*(image / image[...,n].mean()))
               .clip(0, 255).astype(int))
    return image_gw

R, G, B=gray(R,0),gray(G,1),gray(B,2)
I_restored=R+G+B
plt.imsave("Restored_gw.png",I_restored.astype(np.uint8))#save restored image
#plot:
fig, ax = plt.subplots(1,2, figsize=(10,6))
ax[0].imshow(I)
ax[0].set_title('Original Image')
ax[1].imshow(I_restored)
ax[1].set_title('Whitebalanced Image (Gray World)');
ax[0].axis('off')
ax[1].axis('off')
plt.show()
```

White-balancing by white patch algorithm

```
R,G,B=deepcopy(I),deepcopy(I),deepcopy(I)
#Extracting each color channel of the image:
R[:,:,1],R[:,:,2]=0,0
G[:,:,0],G[:,:,2]=0,0
B[:,:,0],B[:,:,1]=0,0

w=io.imread("white.jpg") #load the white patch

Rw,Gw,Bw=deepcopy(w),deepcopy(w),deepcopy(w)
#Extracting each color channel of the white patch:
Rw[:,:,1],Rw[:,:,2]=0,0
Gw[:,:,0],Gw[:,:,2]=0,0
Bw[:,:,0],Bw[:,:,1]=0,0

def white(image,w,n): #white patch user-defined function
    wp=((image*(image / w[...,n].mean())).clip(0, 255).astype(int))
    return wp

R, G, B=white(R,Rw,0), white(G,Gw,1), white(B,Bw,2)
I_restored=R+G+B
plt.imsave("Restored_wp.png",I_restored.astype(np.uint8))#save restored image
#plot:
fig, ax = plt.subplots(1,2, figsize=(10,6))
ax[0].imshow(I)
ax[0].set_title('Original Image')
ax[1].imshow(I_restored)
ax[1].set_title('Whitebalanced Image (White Patch)');
ax[0].axis('off')
ax[1].axis('off')
plt.show()
```



White-balancing by Contrast Stretching

```
import skimage.io as io
from copy import deepcopy

img_filename = 'old.JPG'
I = io.imread(img_filename)
R,G,B=deepcopy(I),deepcopy(I),deepcopy(I)
#Extracting each color channel:
R[:,:,1],R[:,:,2]=0,0
G[:,:,0],G[:,:,2]=0,0
B[:,:,0],B[:,:,1]=0,0

def stretch(image): #contrast stretch user-defined function
    image_cs = np.zeros((image.shape[0],image.shape[1],image.shape[2]), \
                         dtype = 'uint8')
    min = np.min(image)
    max = np.max(image)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            for k in range(3):
                image_cs[i,j,k] = 255*(image[i,j,k]-min)/(max-min) #normalizing
    return image_cs

R,G,B=stretch(R),stretch(G),stretch(B)
I_restored=R+G+B #overlay the channels
plt.imsave("Restored_cs.png",I_restored) #save restored image
#plot:
fig, ax = plt.subplots(1,2, figsize=(10,6))
ax[0].imshow(I)
ax[0].set_title('Original Image')
ax[1].imshow(I_restored)
ax[1].set_title('Whitebalanced Image (Contrast Stretching)');
ax[0].axis('off')
ax[1].axis('off')
plt.show()
```

Figure 17. Codes for various white-balancing algorithms.

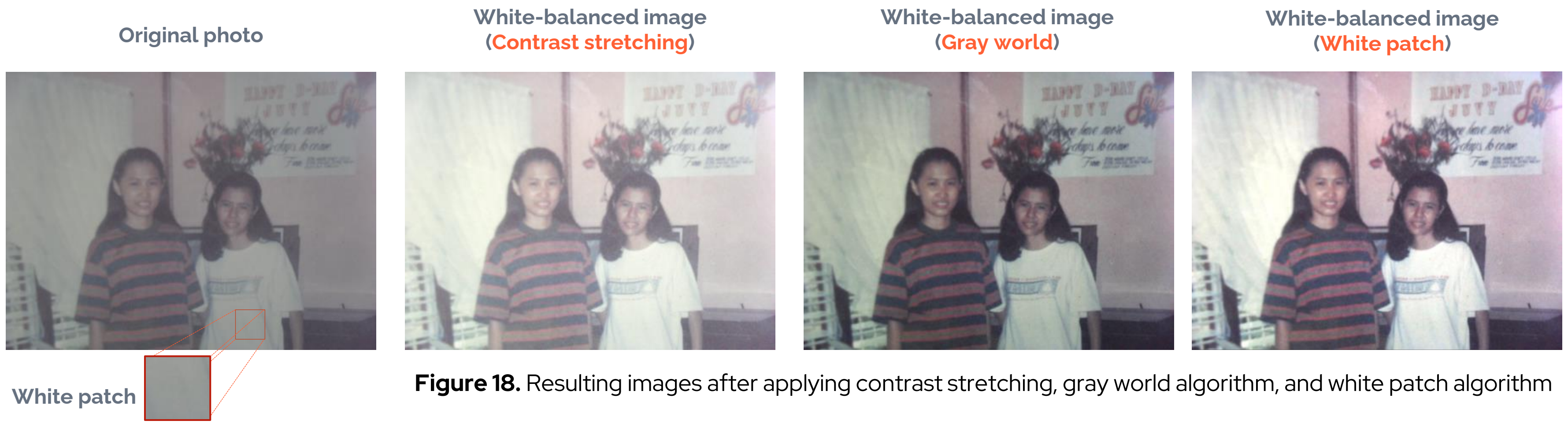

Original photo
White-balanced image (Contrast stretching)
White-balanced image (Gray world)
White-balanced image (White patch)
White patch

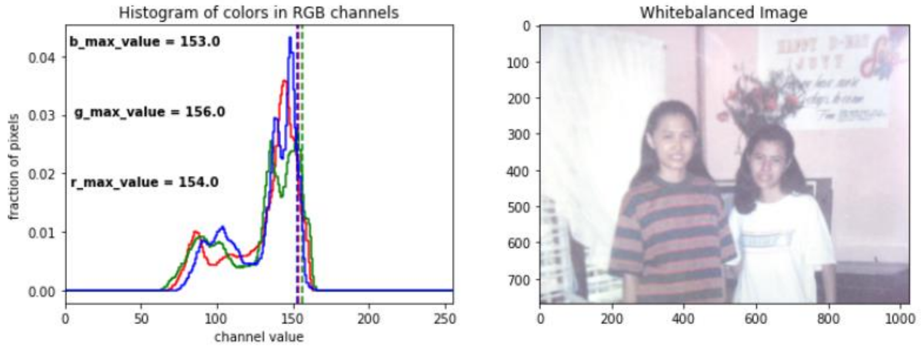
Figure 18. Resulting images after applying contrast stretching, gray world algorithm, and white patch algorithm

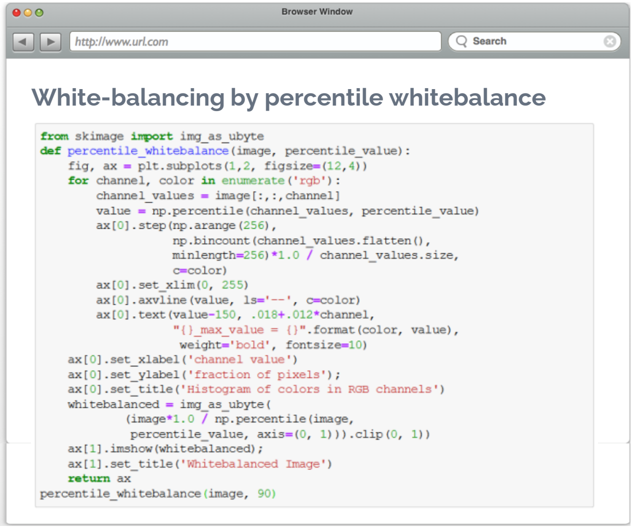
Figure 18 presents the resulting images after undergoing different white-balancing algorithms in an attempt to restore the original old photo.

For **contrast stretching**, the algorithm is the same as the one we used in the previous section, the one about min-max contrast stretching. I modified it to account for three dimensions instead of two since we are no longer manipulating a black-and-white photo and the three dimensions correspond to the RGB channels. First, the three channels were separated, and the algorithm was performed on each channel. Then, the channels were overlayed to produce the contrast-stretched image. The result is a very bright image. It did improve the colors of the old photo, but it looks too bright for my taste. The brightness made the photo slightly washed out.

For the **gray world** algorithm, the RGB channels were divided by their individual averages. Then, the resulting channels were overlayed to one another. The result is a more vibrant photo than the original. It corrected the grayed colors of the old photo and made it brighter but not as overkill as the previous algorithm. It also has more contrast.

The **white patch** algorithm resembles the code for the gray world, but instead of dividing by the channels' averages, we divide by the average of the channels from a 'white patch.' This is a portion of the image that we know is white. The result is the best among the three. The colors are vibrant, and the whites actually look white. The image is also brighter and has more contrast than the original.





White-balancing by percentile whitebalance

```python
from skimage import img_as_ubyte
def percentile_whitebalance(image, percentile_value):
    fig, ax = plt.subplots(1,2, figsize=(12,4))
    for channel, color in enumerate('rgb'):
        channel_values = image[:,:,channel]
        value = np.percentile(channel_values, percentile_value)
        ax[0].step(np.arange(256),
                   np.bincount(channel_values.flatten(),
                   minlength=256)*1.0 / channel_values.size,
                   c=color)
        ax[0].set_xlim(0, 255)
        ax[0].axvline(value, ls='--', c=color)
        ax[0].text(value-150, .018+.012*channel,
                   "{}_max_value = {}".format(color, value),
                    weight='bold', fontsize=10)
    ax[0].set_xlabel('channel value')
    ax[0].set_ylabel('fraction of pixels');
    ax[0].set_title('Histogram of colors in RGB channels')
    whitebalanced = img_as_ubyte(
            (image*1.0 / np.percentile(image,
             percentile_value, axis=(0, 1))).clip(0, 1))
    ax[1].imshow(whitebalanced);
    ax[1].set_title('Whitebalanced Image')
    return ax
percentile_whitebalance(image, 90)
```

Figure 19. Resulting image after percentile whitebalance algorithm and corresponding plot and code [1].

Additional:

Here's another white-balancing algorithm called **percentile whitebalance**. It undergoes first an equalization process reminiscent of the histogram equalization presented earlier. Then the next process is like the gray world algorithm, but instead of dividing by the mean of each channel, it uses the value of the specified percentile of each RGB channel, which was set to 90.

The resulting image looks like what we got from the contrast stretching algorithm. Although the image is not as good as the result from the white patch, it is still good to know that we have various algorithms to choose from to obtain the best-restored image possible.

Reference:
[1] Manansala, Jephraim. "Image Processing with Python: Color Correction using White Balancing Methods | by Jephraim Manansala | Medium." Medium, 3 Mar. 2021, jephraim-manansala.medium.com/image-processing-with-python-color-correction-using-white-balancing-6c6c749886de.

Reflection

This activity was fun. It made me appreciate the filters I have used in the past. I never imagined how profound the thought that went into these algorithms; it was terrific. These algorithms, in particular, are beneficial in restoring and correcting old images. I learned so much about how image restoration works, at least the basics.

All performed white-balancing techniques improved our old photo to some extent, therefore the results of this activity is reasonable.

The first three algorithms can still be improved. I doubt that all old/unbalanced photographs will look good as my output. Maybe if we incorporate some input functions to specify certain values, it may allow us more control in our resulting images.

120/100

I would give myself 120 because I understood all the concepts and were able to satisfy all the needed objectives of this activity. I also went beyond on certain topics and have pointed out their limitations.