Assignment #2

HARNEET SINGH - 400110275

COMPENG 4TN4: Image Processing

January 31, 2022

Theory

1. Histogram Equalization

Given histogram:

Intensity Level / Bin Number	Number of Pixels
0	2
1	2
2	4
3	8
4	16
5	32
6	64
7	128

To find the equalized histogram, we need to compute PDF, CDF and then, number of pixels in the new gray level/bin. Here, L = number of possible intensity levels, therefore L = 8.

Bin Number	# of Pixels	PDF	CDF	s_k=CDF*(L-1)	Round(s_k)
0	2	$\frac{2}{256} = 0.0078125$	0.078125	0.078125*7 = 0.055	0
1	2	$\frac{2}{256} = 0.0078125$	0.015625	0.015625*7 = 0.109	0
2	4	$\frac{4}{256} = 0.015625$	0.03125	0.03125*7 = 0.219	0
3	8	$\frac{8}{256} = 0.03125$	0.0625	0.0625*7 = 0.438	0
4	16	$\frac{16}{256} = 0.0625$	0.125	0.125*7 = 0.875	1
5	32	$\frac{32}{256} = 0.125$	0.25	0.25*7 = 1.75	2
6	64	$\frac{64}{256} = 0.25$	0.5	0.5*7 = 3.5	4
7	128	$\frac{128}{256} = 0.5$	1	1*7 = 7	7
Total Pixels =	256				

To find the equalized histogram, we need to look at the number of pixels at each intensity level (i.e. new gray level (s_k)):

Number of Pixels	s_k
2	0
2	0
4	0
8	0
16	1
32	2
64	4
125	7

Now, we get the re-mapped pixels at the new intensity levels:

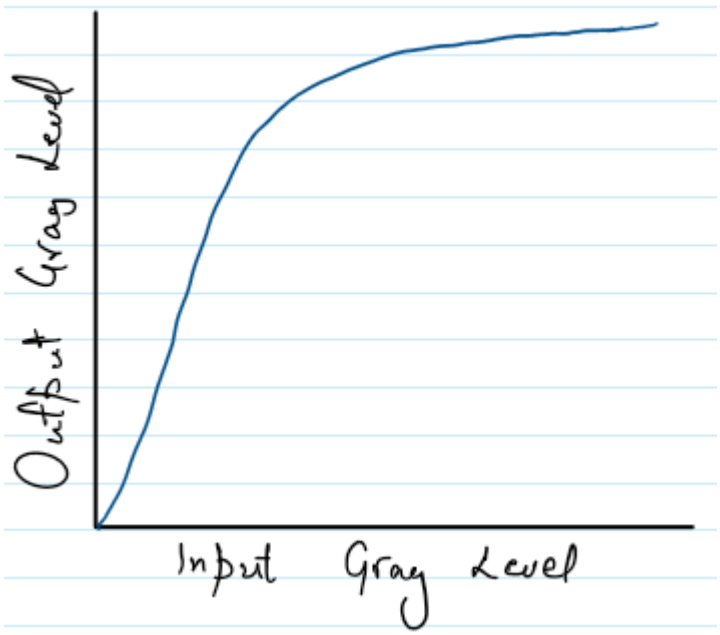
0: 2+2+4+8 = 16	4: 64
1: 16	5: 0
2: 32	6: 0
3: 0	7: 128

Equalized Output Histogram: (16, 16, 32, 0, 64, 0, 0, 128).

2. Transfer Function

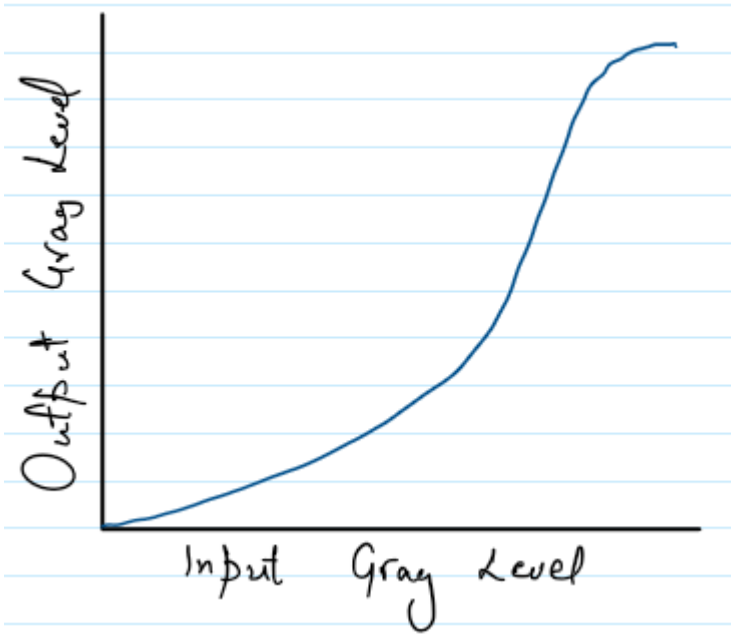
Following images are not to scale, but they offer an approximate transfer function between input and output image:

a.



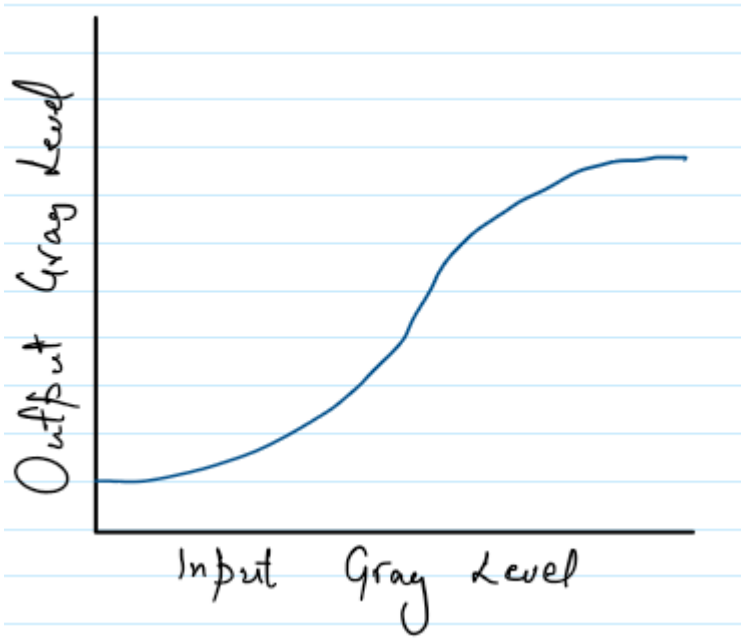
Majority of the pixels have increased intensity in the output i.e. the whiteness level has gone up. However, some pixels are still holding dark color such as grill of the bumper.

b.



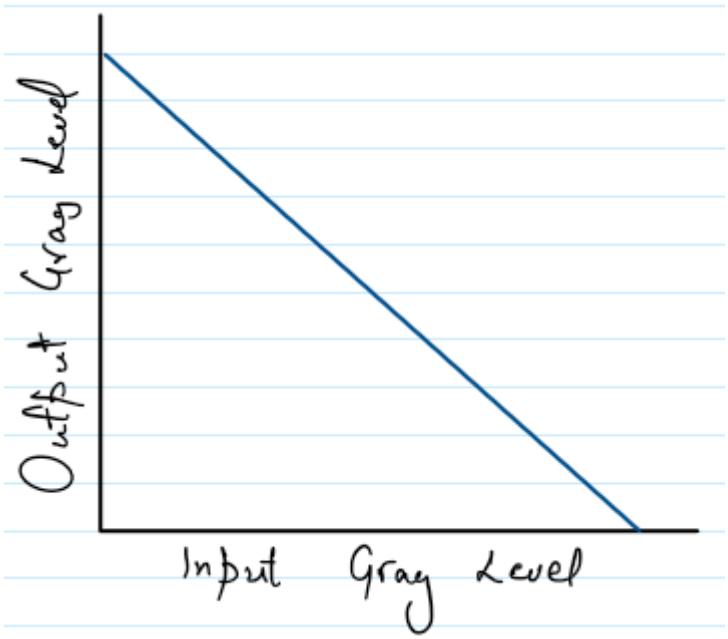
Gray to black levels have increased i.e. pixel values have reduced (tending toward 0). But, white levels can still be seen because white pixel values have not changed drastically (comparison based on car color).

c.



Both white and black levels in the output image have reduced, that is why most of the pixels now have gray intensity. Car in the output image is not as white as the input image and black color of the bumper is not as black as the input image's bumper, that is why this transfer function is not reaching either ends.

d.



This output image is a negative of the given input image. Another way to describe this transfer function is that black color has been swapped with white color and vice versa.

3. Filtering in Spatial Domain

a.

$$\text{Padded Image} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 4 & 0 \\ 0 & 9 & 1 & 2 & 9 & 0 \\ 0 & 4 & 6 & 7 & 3 & 0 \\ 0 & 3 & 8 & 5 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Because, the median kernel is 3x3 and each element is a 1. This means that we need to consider each element as we convolve the kernel with the image. Starting from the top-left corner, we can retrieve the median value of the row 2 and column 2 element of padded image as follows:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 5 \\ 0 & 9 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 3 & 5 & 0 & 9 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 3 & 5 & 9 \end{bmatrix}$$

Now that the sub-matrix has been unraveled and sorted in an ascending order, we can find the median by extracting the middle value i.e. 5^{th} element in this case, which is 0. So, the median value of 0 will replace row 2 and column 2 in the median filtered image.

As shown above, we can compute the remaining values of the median filtered image. I have used a python script to find all the values which is shown below, along with the answer:

Python Script: In the following code, image is read as a numpy array and padded before performing any processing operations. In the medianFilter(), kernel shape is passed to work on certain pixel values and numpy's median function is used to compute the median.

```
import cv2
import numpy as np

img = np.zeros((6,6), dtype = 'uint8')
givenimg = np.array([[3,5,8,4],
                     [9,1,2,9],
                     [4,6,7,3],
                     [3,8,5,4]])
img[1:5, 1:5] = givenimg
img = np.asmatrix(img)
kernel = np.ones((3,3), dtype = 'uint8')
shapeKernel = int(kernel.shape[0])
filteredImg = np.empty_like(givenimg, dtype = 'uint8')

def medianFilter(n):
    for k in range(n+1):
        for p in range(n+1):
            subMatrix = np.ravel(img[k:k+n, p:p+n])
            # print(subMatrix)
            medianValue = (np.median(subMatrix)).astype('uint8')
            # print(medianValue)
            filteredImg[k, p] = medianValue
    print(filteredImg)

medianFilter(shapeKernel)
```

Script Output:

```
[[0 2 2 0]
 [3 5 5 3]
 [3 5 5 3]
 [0 4 4 0]]
```

Therefore, median filter image is: $\begin{bmatrix} 0 & 2 & 2 & 0 \\ 3 & 5 & 5 & 3 \\ 3 & 5 & 5 & 3 \\ 0 & 4 & 4 & 0 \end{bmatrix}$

b.

Padded Image = $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 4 & 0 \\ 0 & 9 & 1 & 2 & 9 & 0 \\ 0 & 4 & 6 & 7 & 3 & 0 \\ 0 & 3 & 8 & 5 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Because, the median kernel is 3x3 and the corner elements are zero, these values will be ignored in the following computation. This means that we need to consider elements that have a value of 1 in the kernel while the image is convolved with the kernel. Starting from the top-left corner, we can retrieve the median value of the row 2 and column 2 element of padded image as follows:

$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 5 \\ 0 & 9 & 1 \end{bmatrix}$

We need to consider the following values only: $\begin{bmatrix} 0 & 0 & 3 & 5 & 9 \end{bmatrix}$

After ascending sorting, we get: $\begin{bmatrix} 0 & 0 & 3 & 5 & 9 \end{bmatrix}$

Now that sub-matrix has been unraveled and sorted in an ascending order, we can find the median by extracting the middle value i.e. 3^{rd} element in this case, which is 3. So, the median value of 3 will replace row 2 and column 2 in the median filtered image.

As shown above, we can compute the remaining values of the median filtered image. I have used a python script as shown below and answer is shown below:

Python Script: In the following code, image is read as a numpy array and padded before performing any processing operations. In the medianFilter(), kernel shape is passed to work on certain pixel values. Sub-matrix's values are adjusted before calling numpy's median function, this way we are only working on certain pixel values.

```python
import cv2
import numpy as np

kernel = np.array([[0,1,0],
                   [1,1,1],
                   [0,1,0]])
shapeKernel = int(kernel.shape[0])
img = np.zeros((6,6), dtype = 'uint8')
givenimg = np.array([[3,5,8,4],
                     [9,1,2,9],
                     [4,6,7,3],
                     [3,8,5,4]])
img[1:5, 1:5] = givenimg
img = np.asmatrix(img)
filteredImg = np.empty_like(givenimg, dtype = 'uint8')

def medianFilter(n):
    for k in range(n+1):
        for p in range(n+1):
            elements = []
            subMatrix = np.ravel(img[k:k+n, p:p+n])
            elements.append(subMatrix[1])
            elements.append(subMatrix[3])
            elements.append(subMatrix[4])
            elements.append(subMatrix[5])
            elements.append(subMatrix[7])

            medianValue = (np.median(elements)).astype('uint8')
            filteredImg[k, p] = medianValue
    print(filteredImg)

medianFilter(shapeKernel)
```

Script Output:

```
[[3 3 4 4]
 [3 5 7 3]
 [4 6 5 4]
 [3 5 5 3]]
```

Therefore, median filtered image is: $\begin{bmatrix} 3 & 3 & 4 & 4 \\ 3 & 5 & 7 & 3 \\ 4 & 6 & 5 & 4 \\ 3 & 5 & 5 & 3 \end{bmatrix}$

4. Fourier Domain

- (a) - H
- (b) - A
- (c) - G
- (d) - F
- (e) - B
- (f) - E
- (g) - C
- (h) - D

Explanation **:**

- (a) - H: Frequency spectrum shows dots everywhere on the spectrum and this is because, grain image has no pattern in the temporal domain. This is why no pattern can be observed in the frequency domain. In the frequency domain, this image presents both low and high frequencies.
- (b) - A: Three dots in the vertical direction shows that the time domain image has a single frequency, which fits true with the horizontally stripped black and white image. Two dots away from the center represent a single frequency and the middle dot is DC value of the image i.e. average value of all the pixels. Another correlation that is useful in this case is that if the edge is horizontal (spatial domain), the frequency component will show up perpendicular to the edge i.e. dots will be vertically aligned (frequency domain). Since both positive and negative values are shown in the frequency spectrum, we get to see two dots plus a DC component.
- (c) - G: Frequency spectrum shows a pattern i.e. all the dots are spread evenly (dots can be joined in concentric circles) and as we move away from the center, we can see a hexagonal shape of the dots (imagine interpolating lines between dots but equidistant from the center). This pattern is because of the honey comb pattern, this image has a hexagonal shapes which means that the frequency should be consistent and since hexagons are placed in a row (even at a titled angle), they can show both high and low frequencies. Also, we can notice slight tilt in the frequency spectrum which resembles the tilt in spatial domain.
- (d) - F: This frequency spectrum has high frequencies (away from the center), this pattern resembles the finger-print image in the time domain. This is because, we can notice high frequencies in the temporal domain and the black and white lines are spanning out in a spiral shape which can present some resemblance in the frequency domain (as can be seen in the frequency domain, although not perfectly circular).
- (e) - B: Similar reasoning as to part (b) - A, however in this case, the stripped lines are at an angle which is why we see that the three dots are at an angle as well. They still show similar distance away from the center as can be seen in part (b), which represents the frequency.
- (f) - E: In the frequency domain, we can see a pattern forming in all quadrants, but one interesting find is that the frequency along y-direction is higher than frequency along x-direction because of the distance between dots along both directions. We can easily connect this with the brick wall because both directions show consistency, hence the consistent pattern in the frequency domain. Higher frequency along y-direction can be witnessed in comparison to x-direction, because of the brick dimension in both directions (compare the spacing between white stripes in both directions).
- (g) - C: Frequency domain shows a uniform pattern in all directions and all of the dots away from the center show frequency component. In both X- and Y- directions, we encounter two frequencies (two dots in each quadrant) and this behavior can be connected with checkered board shown in C. Temporal domain also shows consistency and each side has four black square tiles, which means we can get two frequencies out of this image.
- (h) - D: Circular frequency shape shows that the image in temporal domain should have a single frequency value because of the distance away from the center. Because the frequency domain is a circle, we can expect that the image in the time domain should be symmetric in all quadrants and should possess only one frequency.

Implementation

1. Background Removal

Median filter relies on the neighbor values and usually, the values are not scaled in the filter which means that it is a good tool to remove random noise from an image. Median filter can efficiently remove salt and pepper noise and median filter can preserve edges. For example, in this 1D image: [151 27 157 15 10], if the pixel being processed is 157 then after median filtering, this value will be replaced by 27. Clearly, median filter can remove noise from the image as it removed random high pixel value. On the same note, it can smoothen the image thus attaining low pass filter property as well. Following these characteristics, median filter can remove random noise from a video because any object passing through the frame are there for a tiny amount of time and by comparing frame values, we can remove these objects (in essence, these objects are noise in the image).

Background Removal Code:

```python
import cv2
import numpy as np

cap = cv2.VideoCapture('video.mp4')
backgroundImage = np.empty(shape = (int(cap.get(4)), int(cap.get(3)), 3))
all_frames = np.empty(shape = (14, int(cap.get(4)), int(cap.get(3)), 3), dtype = 'uint8')
# print(len(all_frames))

def loadFramesAsNP():
    index = 0
    i = 0

    while cap.isOpened():
        index += 1
        ret, frame = cap.read()
        # print(frame.shape)

        if not ret:
            print(f"Can't receive frame. Exiting after {index} frames ...")
            break

        if index % 10 == 0:
            all_frames[i] = frame
            # print(all_frames[i].shape)
            # cv2.imshow('frames', all_frames[i])
            # cv2.waitKey(0)

            i += 1

        # cv2.imshow('frame', frame)
        if cv2.waitKey(1) == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

After loading the video, empty frames are created to store background image and frames of the video. To determine the median value of each pixel in the frame, all_frames is holding pixel values of all the frames. Instead of processing each frame of the video, loadFramesAsNP() is processing every tenth frame (albeit this would work on any number of frames, here every tenth frame is checked to improve computational efficiency). After, reading the frames, all windows are closed.

```
37  def findMedian():
38      medianValuedImage = np.median(all_frames, axis=0).astype(dtype = 'uint8')
39      cv2.imshow('background_image', medianValuedImage)
40
41      if cv2.waitKey(0) == ord('s'):
42          cv2.imwrite("impl1Background.jpg", medianValuedImage)
43          cv2.destroyAllWindows()
44      else:
45          cv2.destroyAllWindows()
46
47      return medianValuedImage
48
49  def VideoWithoutBackground(backgroundImage):
()50      cap = cv2.VideoCapture('video.mp4')
51      fourcc = cv2.VideoWriter_fourcc(*'XVID')
52      #important to get the framesize right, otherwise it won't work
53      outVideo = cv2.VideoWriter('impl1Video.avi', fourcc, 20.0, (int(cap.get(3)), int(cap.get(4))))
54
55      while cap.isOpened():
56          ret, frameNoBackground = cap.read()
57
58          if not ret:
59              print(f"Can't receive frame. Exiting after playing frames ...")
60              break
61
62          frameNoBackground = frameNoBackground.astype(np.int16)
63          withoutBackground = (np.abs(frameNoBackground - backgroundImage)).astype(np.uint8)
64          # Next line makes it work without casting to int16 and back to int8
65          # withoutBackground = cv2.absdiff(frame2, backgroundImage)
66
67          outVideo.write(withoutBackground)
68          cv2.imshow('frameNoBackground', withoutBackground)
69          if cv2.waitKey(1) == ord('q'):
70              break
71
72      outVideo.release()
73      cap.release()
74      cv2.destroyAllWindows()
```

findMedian() is to compute the median value of each pixel in all_frames numpy array and it is stored as an image and passed back as a return value to be used in the video. VideoWithoutBackground() is receiving background image and subtracting this image from all the frames to remove objects from the video. As the frames are being processed, output is being stored in outVideo using VideoWriter().

```
76
77  loadFramesAsNP()
78  backgroundImage = findMedian()
79  VideoWithoutBackground(backgroundImage)
```

Here, these functions are called to commence the program.

Background Image:

Output video (without background) will be available in the zip file.

2. Image Manipulation in FFT domain

Magnitude of the Fourier transform tells us about the structure of the image and phase determines the relationship between different frequency components. In essence, magnitude tells us which frequency components are present (or how much of each frequency is present in the image) and phase tells us where that frequency component is. Generally, magnitude of the FFT is displayed in frequency domain, because phase values are difficult to be linked with the image. However, both phase and magnitude are essential in re-creating the image from frequency domain to spatial domain. As can be seen in this question, if the phase is swapped between two images, we can witness that features of the other image dominate and magnitude component does little to nothing to show the original image, i.e. with Ronaldo's phase applied on Messi's image, we will see features of Ronaldo image in effect and vice versa.

Python Script:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

#read images
messiImg = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
ronaldoImg = cv2.imread('ronaldo.jpg', cv2.IMREAD_GRAYSCALE)

#convert messi's image to frequency spectrum
dftMessi = cv2.dft(np.float32(messiImg), flags = cv2.DFT_COMPLEX_OUTPUT )
dft_shiftMessi = np.fft.fftshift(dftMessi)
magnitudephaseMessi = list(cv2.cartToPolar(dft_shiftMessi[:,:,0], dft_shiftMessi[:,:,1]))

#convert messi's image to frequency spectrum
dftRonaldo = cv2.dft(np.float32(ronaldoImg), flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shiftRonaldo = np.fft.fftshift(dftRonaldo)
magnitudephaseRonaldo  = list(cv2.cartToPolar(dft_shiftRonaldo[:,:,0], dft_shiftRonaldo[:,:,1]))

#swap phases of both frequency spectrums
magnitudephaseMessi[1], magnitudephaseRonaldo[1] = magnitudephaseRonaldo[1], magnitudephaseMessi[1]

#convert back to cartesian domain and save the result
cartMessi = cv2.polarToCart(magnitudephaseMessi[0], magnitudephaseMessi[1])
f_ishiftMessi = np.fft.ifftshift(cv2.merge(cartMessi))
img_backMessi = cv2.idft(f_ishiftMessi, flags=cv2.DFT_REAL_OUTPUT | cv2.DFT_SCALE)
cv2.imwrite("Impl2MessiMag+RonaldoPhase.jpg", img_backMessi)

cartRonaldo = cv2.polarToCart(magnitudephaseRonaldo[0], magnitudephaseRonaldo[1])
f_ishiftRonaldo = np.fft.ifftshift(cv2.merge(cartRonaldo))
img_backRonaldo = cv2.idft(f_ishiftRonaldo, flags=cv2.DFT_REAL_OUTPUT | cv2.DFT_SCALE)
cv2.imwrite("Impl2RonaldoMag+MessiPhase.jpg", img_backRonaldo)
```

After reading the images, dft() is applied to compute the discrete fourier transform as complex numbers and subsequently, shifted to show low frequency values at the center of the spectrum. Phase is swapped after turning the array into a list. Finally, to retrieve the images, inverse fourier transform is applied and shift is undone.

Following image shows Messi's magnitude with Ronaldo's phase, as we can see phase of the ronaldo's image is dominating the magnitude values. Typically, phase holds more information about the image than it's magnitude. Similar results can be seen in the second output.

Script Output #1 - Messi Magnitude + Ronaldo Phase:

Script Output #2 - Ronaldo Magnitude + Messi Phase:

2.1 (Messi + LPF) & (Ronaldo + HPF)

Python Script:

Following code shows: images being read and createLPF() is used to create a circular low pass filter by setting values equal to 1 within the circle and zero outside of the circle. Range of frequency can be adjusted by altering the radius variable. As we increase the radius, we are allowing higher frequencies in the image.
fft2ifft() is used to convert the images to frequency spectrum, apply the mask/filter and turn the images back to the spatial domain.

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

messiImg = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
ronaldoImg = cv2.imread('ronaldo.jpg', cv2.IMREAD_GRAYSCALE)

# Creating a circular LPF for messi's image
def createLPF(img):
    rows, cols = img.shape
    centrePointRow, centrePointcol = int(rows / 2), int(cols / 2)
    center = [centrePointRow, centrePointcol]
    # below, third argument (2) is there to match DFT conversion
    lpfMask = np.zeros((rows, cols, 2), np.uint8)
    radius = 25
    x, y = np.ogrid[:rows, :cols]
    mask_area = (x - center[0]) ** 2 + (y - center[1]) ** 2 <= radius ** 2
    lpfMask[mask_area] = 1
    return lpfMask

def fft2ifft(img, mask):
    dft = cv2.dft(np.float32(img), flags = cv2.DFT_COMPLEX_OUTPUT)
    dft_shift = np.fft.fftshift(dft)
    # apply mask and inverse DFT
    fshift = dft_shift * mask
    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift, flags=cv2.DFT_REAL_OUTPUT | cv2.DFT_SCALE)
    # img_back = cv2.magnitude(img_back[:,:,0],img_back[:,:,1])
    return img_back
```

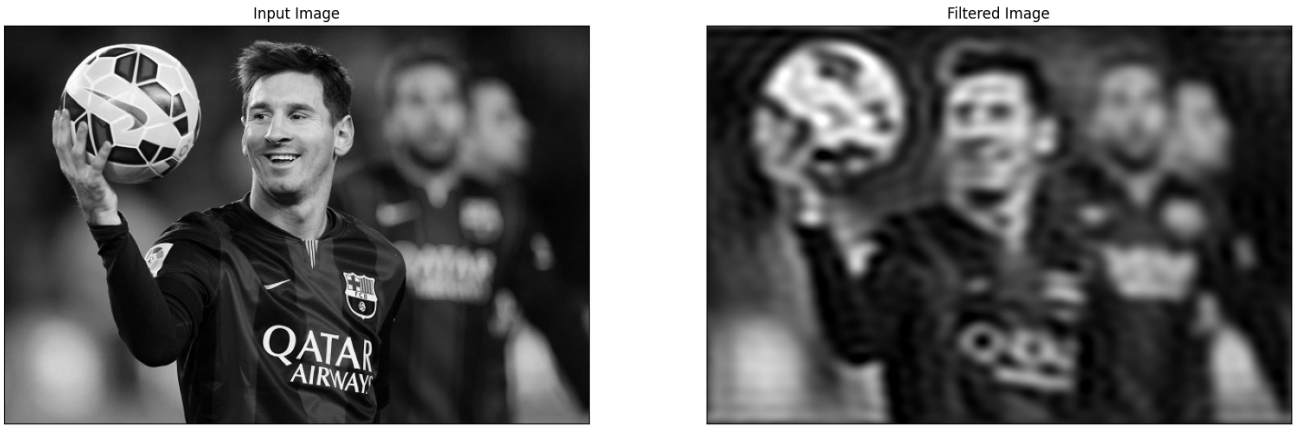
Following code shows two functions plotImg() and addimages(). plotImg() is used to plot the images using pyplot library and addimages() is achieving the image processing by adding high-passed ronaldo's image and low-passed messi's image.

```
def plotImg(filteredImg, img=None):
    fig = plt.figure(figsize=(1,1))
    if img is not None:
        plt.subplot(122), plt.imshow(img, cmap = 'gray', vmin=0, vmax=255)
        plt.title('Filtered Image'), plt.xticks([]), plt.yticks([])
        plt.subplot(121), plt.imshow(filteredImg, cmap = 'gray', vmin=0, vmax=255)
        plt.title('Input Image'), plt.xticks([]), plt.yticks([])
        plt.show()
    else:
        plt.subplot(111), plt.imshow(filteredImg, cmap = 'gray', vmin=0, vmax=255)
        plt.title('Filtered Image'), plt.xticks([]), plt.yticks([])
        plt.show()

def addimages(img1, img2):
    halfImg1 = np.multiply(img1, 0.5)
    halfImg2 = np.multiply(img2, 0.5)
    return halfImg1 + halfImg2

lpfMask = createLPF(messiImg)
hpfMask = 1 - lpfMask

filteredMessiImg = fft2ifft(messiImg, lpfMask)
cv2.imwrite("messi + LPF.jpg", filteredMessiImg)
plotImg(messiImg, filteredMessiImg)
filteredRonaldoImg = fft2ifft(ronaldoImg, hpfMask)
cv2.imwrite("ronaldo + HPF.jpg", filteredRonaldoImg)
plotImg(ronaldoImg, filteredRonaldoImg)

addedImages = addimages(filteredMessiImg, filteredRonaldoImg)
cv2.imwrite("messiLPF + ronaldoHPF.jpg", addedImages)
plotImg(addedImages)
```

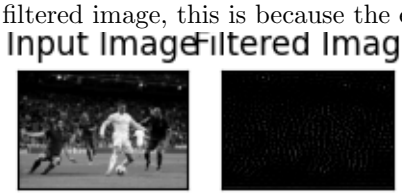
Results are shown below:

- Messi Given Image + Low Pass Filtered Output (Small): At a smaller size, both images look similar, however it has been rid of high frequencies and the pixel values have been reduced.

- Messi Given Image + Low Pass Filtered Output (Large): At a larger scale, human eye starts picking up the diminishing pixel values which contribute to sharp edges.



- Ronaldo Given Image + High Pass Filtered Output (Small): At a smaller size, we can barely see anything from the filtered image, this is because the edges are formed by lines of pixel (or, a few pixels) width.



- Ronaldo Given Image + High Pass Filtered Output (Large): At a larger scale, we can clearly see the edges and this represents high frequency signals in the image.

- Messi LPF + Ronaldo HPF (Small): (Intensity is halved before adding the two images)
  At a smaller scale, we can clearly see messi image and this is because small image size mimics having an image at a distance. Human eye is not able to pick up high frequency (sharp edges) from a distant object, our eyes and brain tries to blend information based on low frequencies.

- Messi LPF + Ronaldo HPF (Large)
  As we have increased the image size, we can see both high and low frequencies because human eye is not able to discern between the two images that are present in the image. Note that distance from the screen plays an important role when we are analyzing this image which is explained below.

In the above image, if we stay close to the screen, most of the time we will pick up ronaldo's image at first as human eye can perceive sharp edges from a close up view. However as we move away from the screen (say, 3 feet), messi's image becomes prominent because from a distance, human eye focuses on low frequencies of objects.

3. Remove Noise

3.1 Median Filter

Python Script:
Using OpenCV's function ran the median filtering process on the supplied image. Kernel size of 3 is selected that means each pixel will look at it's closest neighbors i.e. in total, 9 elements will be considered to compute median value. After reading the image, cv2.medianBlur() filter is used and next, image is shown and saved on the disk.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

kernelSize = 3

img3 = cv2.imread('img3.png')
medianImg3 = cv2.medianBlur(img3, kernelSize)

cv2.imshow("MedianFilteredImg3", medianImg3)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite("impl3Median.jpg", medianImg3)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()
```

Output:

We can notice that median filter has removed random noise as well as smoothened the image (compare the text 'University Hall' between input and median filtered image).

3.2 Low Pass Filter - Gaussian

Python Script:
Using OpenCV's function ran the Gaussian filtering process on the supplied image. In this case, we have two parameters that we can control, which are kernel size and sigma values for both axes. As before, kernel value determines how many neighboring pixels should be considered in the filtering process at each pixel spot. In this code, after reading the image, cv2.GaussianBlur() method is used and next, image is shown and saved on the disk.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

kernelSize = 55
sigmaX = 9
sigmaY = 9

img3 = cv2.imread('img3.png')
gaussianImg3 = cv2.GaussianBlur(img3, (kernelSize, kernelSize), sigmaX, sigmaY)

cv2.imshow("GaussianFilteredImg3", gaussianImg3)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite(f"impl3_2Gaussian_k={kernelSize}_sigmaX={sigmaX}.jpg", gaussianImg3)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()
```

Output with kernel size = 5 and sigma in X = 9: Neighboring 25 pixels are used to get average value for the pixel.

Output with kernel size = 55 and sigma in X = 9: Neighboring 3,025 pixel values are used to get average value for the pixel. A lot of neighboring pixels are used that is how the "Noise" word has faded away.

Output with kernel size = 0 and sigma = 5: Kernel size of 0 implies that the algorithm is automatically adjusting the kernel size by checking the sigma value.

Output with kernel size = 0 and sigma = 10:

At this sigma value, we can barely see the noise word, however we can increase the sigma value a bit more in an attempt to remove it completely.

3.3 Explanation

We noticed that median filter quickly removed random noise as well as smoothed the image (compare the text 'University Hall' between input and median filtered image). This is because median filter eliminates any large values (random noise), this is why median filter works better in this case. However, gaussian filter acts like an averaging filter, which uses all of its neighbor's pixel values (depending on kernel size) and then divides them to get a mean value. Essentially, gaussian filter does not eliminate any outliers, instead it uses them in the averaging process. With gaussian filter, as we are increasing the sigma value, the variance is increasing as well (width of the filter). This means that amount of smoothing effect is increasing as well and in turn, high frequency signals (noise) will be averaged out as well. As can be seen in the above example, as we increase the sigma value, we are getting rid of noise from the image and similar phenomenon happens when we increase the kernel size which can be controlled by sigma value as well.

4. Histogram Equalization

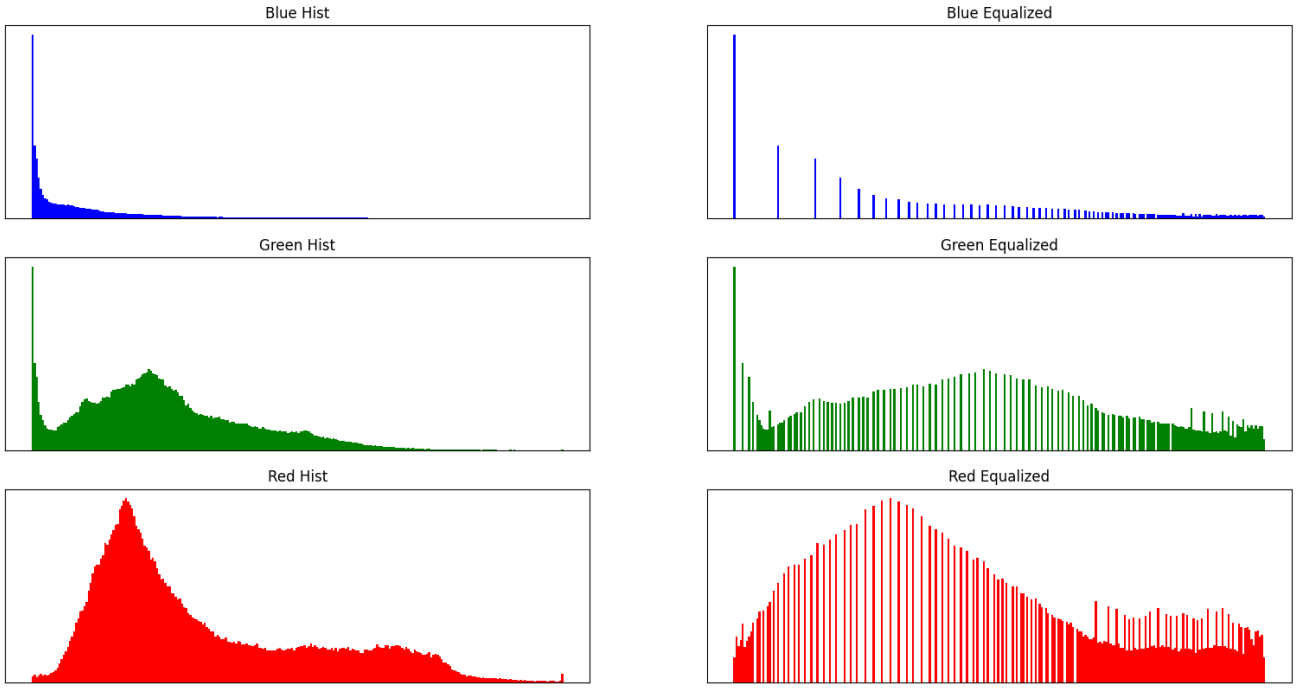
4.1 RGB Histogram Equalization

Python Script: Following code reads the image and then splits each channel into separate numpy array. Next, cv2.equalizeHist() is used to separately equalize the histograms and after that, these are merged back together to form a three channel image again. Finally, the image is displayed and histograms are plotted using pyplot.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img4 = cv2.imread('img4.png')
# cv2.imshow("original", img4)

b, g, r = cv2.split(img4)

bEqualized = cv2.equalizeHist(b)
gEqualized = cv2.equalizeHist(g)
rEqualized = cv2.equalizeHist(r)

mergedImg = cv2.merge((bEqualized, gEqualized, rEqualized))
cv2.imshow("Equalized Image", mergedImg)
if cv2.waitKey(0) == ord('s'):
    cv2.imwrite("Impl4_1_Equalized.png", mergedImg)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()

plt.subplot(321), plt.hist(b.ravel(), 256, color='blue')
plt.title(f'Blue Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(323), plt.hist(g.ravel(), 256, color='green')
plt.title(f'Green Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(325), plt.hist(r.ravel(), 256, color='red')
plt.title(f'Red Hist'), plt.xticks([]), plt.yticks([])

plt.subplot(322), plt.hist(bEqualized.ravel(), 256, color='blue')
plt.title(f'Blue Equalized'), plt.xticks([]), plt.yticks([])
plt.subplot(324), plt.hist(gEqualized.ravel(), 256, color='green')
plt.title(f'Green Equalized'), plt.xticks([]), plt.yticks([])
plt.subplot(326), plt.hist(rEqualized.ravel(), 256, color='red')
plt.title(f'Red Equalized'), plt.xticks([]), plt.yticks([])
plt.show()
```

Output - RGB Equalized Image:

Output - Histogram: Before on the left and After on the right



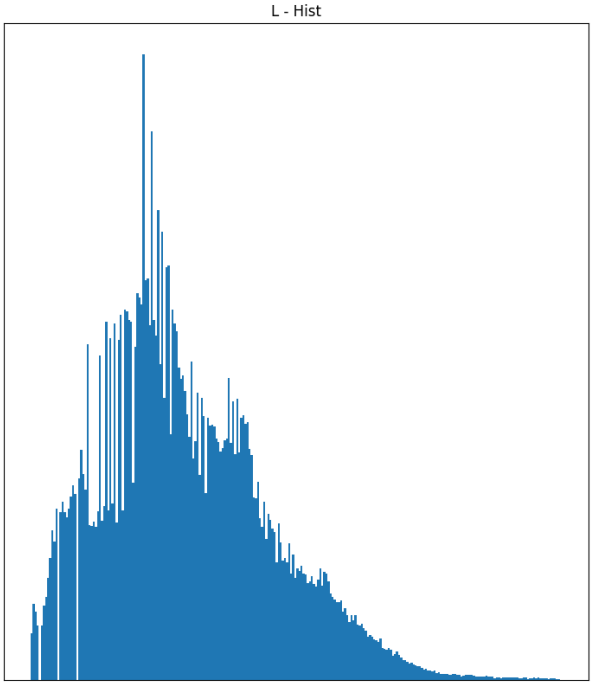
4.2 L - Histogram Equalization

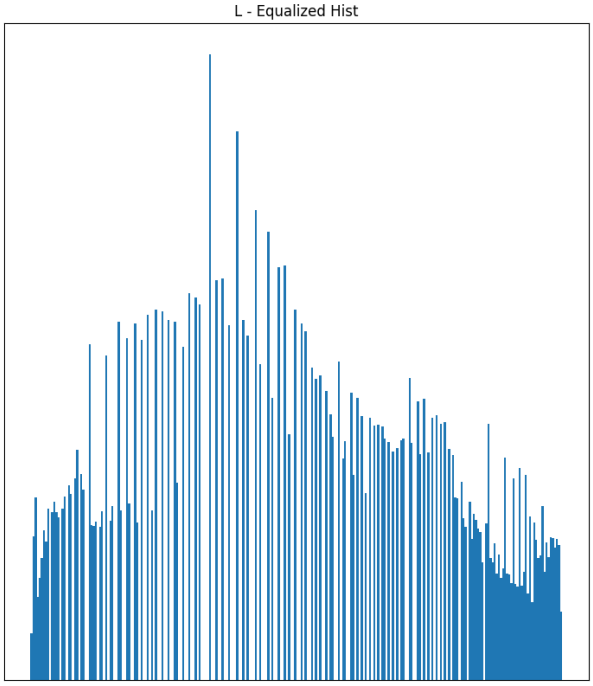
Python Script: Following code reads the image, converts to LAB color space, and then splits each channel into separate numpy array (l, a, b). Next, cv2.equalizeHist() is used to equalize the l-histogram and after that, equalized l, original a and b channels are merged back together to form a three channel image again. Finally, the image is converted back to BGR color space to be displayed and histograms are plotted using pyplot.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img4 = cv2.imread('img4.png')
cv2.imshow("original", img4)
labImg4 = cv2.cvtColor(img4, cv2.COLOR_BGR2LAB)

l, a, b = cv2.split(labImg4)

lEqualized = cv2.equalizeHist(l)

mergedLabImg = cv2.merge((lEqualized, a, b))
equalizedBgrImg = cv2.cvtColor(mergedLabImg, cv2.COLOR_LAB2BGR)
cv2.imshow("L-Equalized Image", equalizedBgrImg)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite("Impl4_2_L-Equalized.png", equalizedBgrImg)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()

plt.subplot(121), plt.hist(l.ravel(), 256)
plt.title(f'L - Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.hist(lEqualized.ravel(), 256)
plt.title(f'L - Equalized Hist'), plt.xticks([]), plt.yticks([])
plt.show()
```

Output - RGB Equalized Image:

Output - Histogram: Before on the left and After on the right





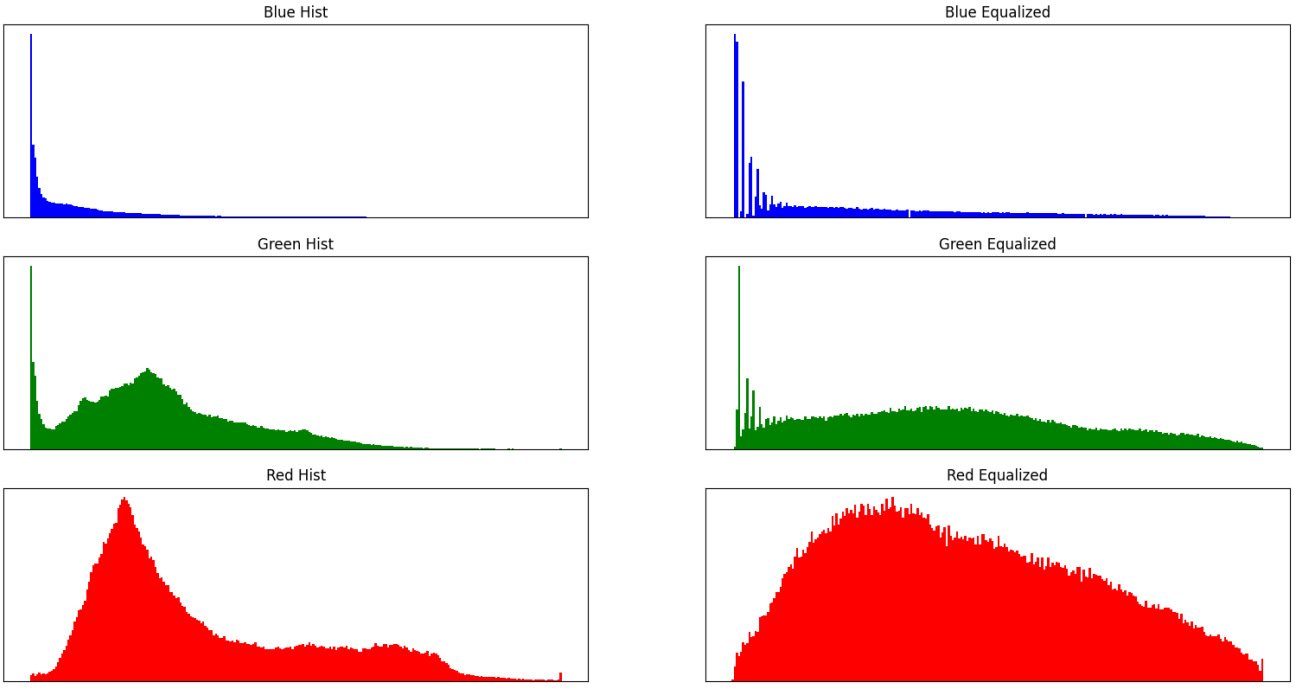
4.3.a RGB Histogram using CLAHE Equalization

Python Script: Following code reads the image and then splits each channel into separate numpy array. Next, OpenCV's CLAHE object is used to separately equalize the histograms and after that, these are merged back together to form a three channel image again. Finally, the image is displayed and histograms are plotted using pyplot. Cliplimit (contrast threshold) and tilegridsize (matrix size processed at each iteration) are choosen after trying a few different values.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img4 = cv2.imread('img4.png')
# cv2.imshow("original", img4)

b, g, r = cv2.split(img4)

claheObject = cv2.createCLAHE(clipLimit = 3.0, tileGridSize = (8,8))
bEqualized = claheObject.apply(b)
gEqualized = claheObject.apply(g)
rEqualized = claheObject.apply(r)

mergedImg = cv2.merge((bEqualized, gEqualized, rEqualized))
cv2.imshow("Equalized Image", mergedImg)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite("Impl4_3_a_Equalized.png", mergedImg)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()

plt.subplot(321), plt.hist(b.ravel(), 256, color='blue')
plt.title(f'Blue Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(323), plt.hist(g.ravel(), 256, color='green')
plt.title(f'Green Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(325), plt.hist(r.ravel(), 256, color='red')
plt.title(f'Red Hist'), plt.xticks([]), plt.yticks([])

plt.subplot(322), plt.hist(bEqualized.ravel(), 256, color='blue')
plt.title(f'Blue Equalized'), plt.xticks([]), plt.yticks([])
plt.subplot(324), plt.hist(gEqualized.ravel(), 256, color='green')
plt.title(f'Green Equalized'), plt.xticks([]), plt.yticks([])
plt.subplot(326), plt.hist(rEqualized.ravel(), 256, color='red')
plt.title(f'Red Equalized'), plt.xticks([]), plt.yticks([])
plt.show()
```

Output - RGB Equalized Image:

Output - Histogram: Before on the left and After on the right



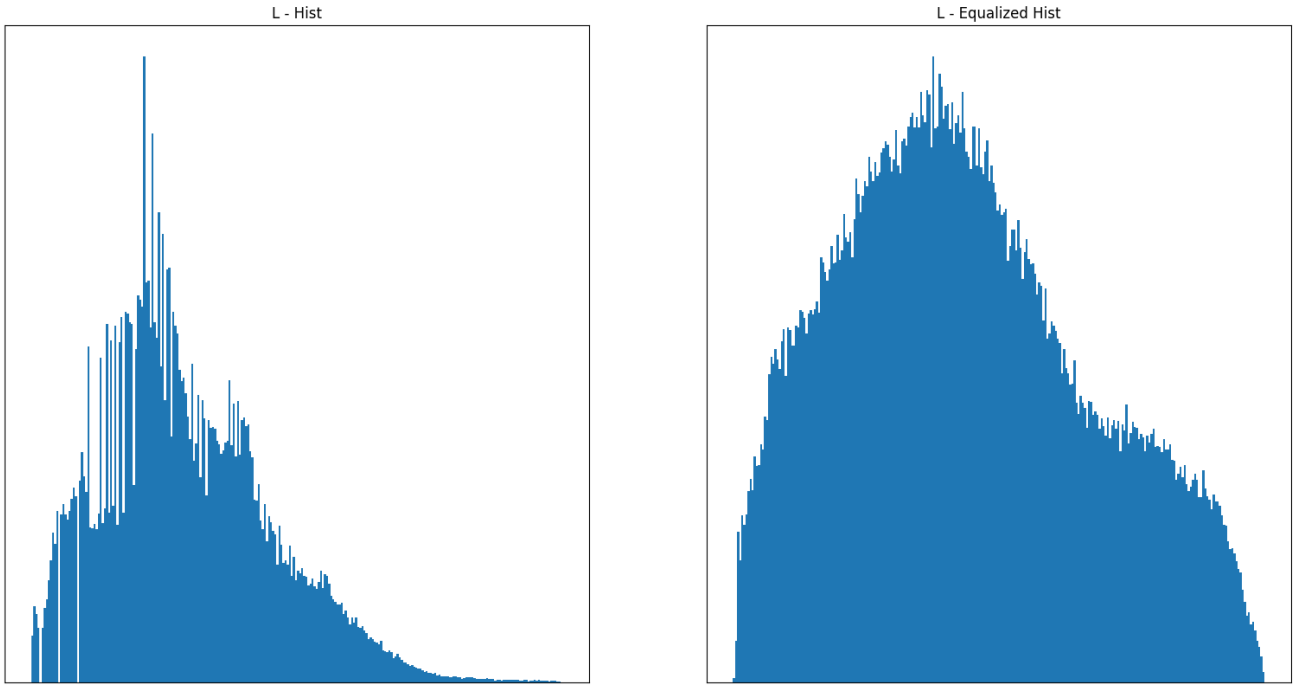
4.3.b L - Histogram using CLAHE Equalization

Python Script: Following code reads the image, converts to LAB color space, and then splits each channel into separate numpy array (l, a, b). Next, OpenCV's CLAHE object is used to equalize the l-histogram and after that, equalized l, original a and b channels are merged back together to form a three channel image again. Finally, the image is converted back to BGR color space to be displayed and histograms are plotted using pyplot. Cliplimit (contrast threshold) and tilegridsize (matrix size processed at each iteration) are choosen after trying a few different values.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img4 = cv2.imread('img4.png')
cv2.imshow("original", img4)
labImg4 = cv2.cvtColor(img4, cv2.COLOR_BGR2LAB)

l, a, b = cv2.split(labImg4)

claheObject = cv2.createCLAHE(clipLimit = 3.0, tileGridSize = (8,8))
lEqualized = claheObject.apply(l)

mergedLabImg = cv2.merge((lEqualized, a, b))
equalizedBgrImg = cv2.cvtColor(mergedLabImg, cv2.COLOR_LAB2BGR)
cv2.imshow("L-Equalized Image", equalizedBgrImg)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite("Impl4_3_b_L-Equalized.png", equalizedBgrImg)
    cv2.destroyAllWindows()
else:
    cv2.destroyAllWindows()

plt.subplot(121), plt.hist(l.ravel(), 256)
plt.title(f'L - Hist'), plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.hist(lEqualized.ravel(), 256)
plt.title(f'L - Equalized Hist'), plt.xticks([]), plt.yticks([])
plt.show()
```

Output - RGB Equalized Image:

Output - Histogram: Before on the left and After on the right



4.4 Explanation

4.1 With RGB histogram equalization, color of objects in the image has changed. Notice color of the log, it has turned bluish which is not ideal because we would like to preserve the color of the objects.
4.2 With L-equalization, we have made the image slightly brighter and kept most of it's natural color. However, as you can see in the output image, we are not seeing natural colors of the objects as they have become too bright, essentially contrastness of the image is not reflecting the true scene. One good outcome is that the brightness of the log has made it appear clearly in the image.

4.3.*a* With CLAHE RGB equalization, we can see that all the colors have enhanced. Notice a shriveled leaf next to mushroom on the far right which was not clearly visible in the input image. However, this has also led to color shift - notice the bigger mushroom in the middle has a slight bluish tan and this is why this is not a desired outcome.
4.3.4 With CLAHE L-equalization, we have managed to increase the light intensity of the image while keeping the colors intact. In this case, this would be the desired outcome.
In summary, L-channel adjusts the intensity of the image separately without affecting A- and B-channels which represent the colors of the image. Therefore, equalizing L-channel separately keeps the colors of the image consistent. However, with RGB equalization, we see color shifts which, in typical cases, is not a favorable outcome. This happens because each channel of R, G and B holds color intensities and by equalizing them, we are altering the colors of the image. Also, CLAHE produced better results because it is not trying to spread the histograms to its extremes, instead a tileGridGize variable is used to limit the equalization process to 8x8 pixels at a time. And, to control the contrastness of the image, clipLimit variable is used which acts as a threshold value for the contrast.

Bibliography

[1] Introduction to fourier transforms for image processing. https://www.cs.unm.edu/~brayer/vision/fourier.html.

[2] Module 5.6: Noise smoothing. https://nptel.ac.in/content/storage2/courses/117104069/chapter_8/8_16.html.

[3] Kundur Deepa. Magnitude and phase. https://www.comm.utoronto.ca/~dkundur/course_info/signals/notes/Kundur_FourierMagPhase.pdf.