EN2550 – Assignment 01

Name : Rathnayaka R.G.H.V.

Index No : 180529E

01.(a). Histogram Computation

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
```

```
#(1.a) Histogram Computation
img = cv.imread('im02small.png',cv.IMREAD_COLOR)
color = ('b','g','r')
for i,c in enumerate(color):
    hist = cv.calcHist([img],[i],None,[256],[0,256])
    plt.plot(hist,color = c)
    plt.xlim([0,256])
plt.title('Histogram')
plt.show ()
```

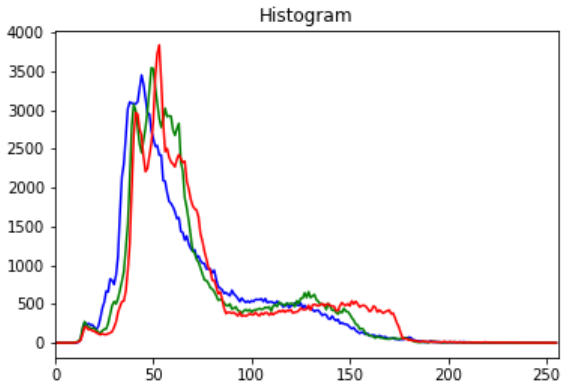


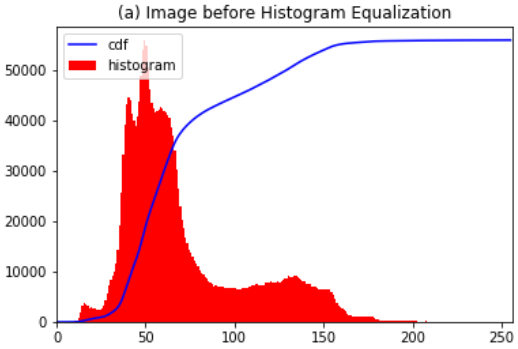
Figure (1)

This is a graphical illustration of the number of pixels in the image at each different intensity value found in the image plotted under each RGB plane. *Figure (1)* shows the histogram of the selected image.

(b). Histogram Equalization

```
#(1.b) Histogram Equalization
img = cv.imread('im02.png',cv.IMREAD_GRAYSCALE)
hist,bins = np.histogram(img.ravel(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf*hist.max()/ cdf.max()

equ = cv.equalizeHist(img)
hist,bins = np.histogram(equ.ravel(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf*hist.max()/ cdf.max()
```



Histogram equalization is a gray level transformation which adjust the contrast of an image by flattening out its histogram. It can be easily observed in *Figure (2)* that the histogram has got flattened after the histogram equalization process. This will result to reduce the darkness of the image in *Figure (3).* Image after histogram equalization has become lighter.

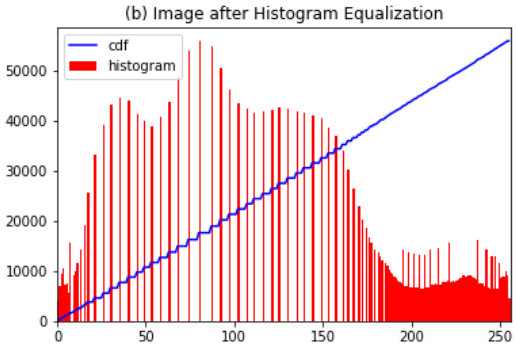


Figure (2)

Figure (3)

(c). Intensity Transformations

```
#(1.c) Intensity Transformations
img_orig = cv.imread('im02small.png',cv.IMREAD_GRAYSCALE)
transform = np.zeros(256)
transform[0] = 255
for i in range(1,256):
    transform[i] = ((i-128)**2)/(64)
tranform = np.uint(transform)
height = img_orig.shape[0]
width = img_orig.shape[1]
hist = np.zeros(256)
transformed_image = np.zeros([height,width])
for i in range(0,height):
    for j in range(0,width):
        transformed_image[i,j] = transform[img_orig[i,j]]
```

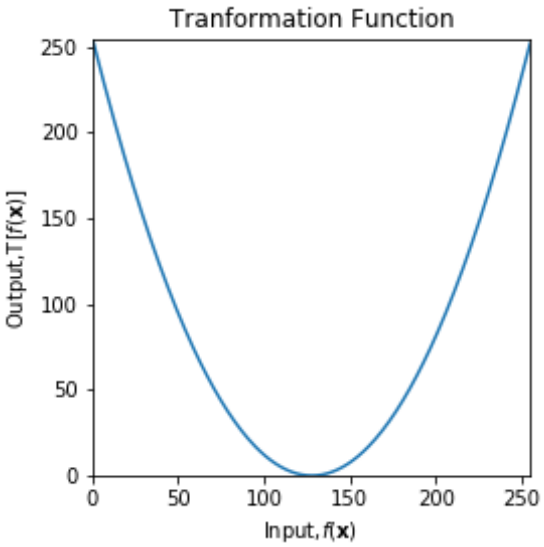


Figure (4)

Here, as the transformation function, it has been used a parabola shown under *Figure (4),* which is capable in making extreme intensity levels at both ends turn more darker and middle intensity levels turn more lighter. It is clearly depicted through *Figure (5).*

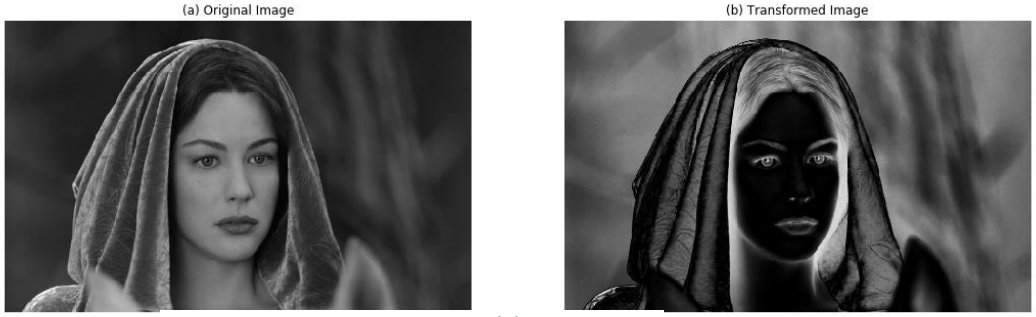


Figure (5)

(d). Gamma Correction

```
#(1.d) Gamma Correction
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
img_orig = cv.imread ('im02.png',cv.IMREAD_COLOR)
gamma = 2
table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')
img_gamma = cv.LUT(img_orig,table)
img_orig = cv.cvtColor(img_orig,cv.COLOR_BGR2RGB)
img_gamma = cv.cvtColor(img_gamma,cv.COLOR_BGR2RGB)
```

Figure (6)

Figure (6) shows the Gamma corrected image of the selected image. For this demonstration it has been selected **Gamma = 2**. Since Gamma > 1, the function maps a narrow range of bright pixels to a wider range of bright pixels. So, the resulting image has become somewhat darker than the original one. The variation of the transformation is given in *Figure (7).*

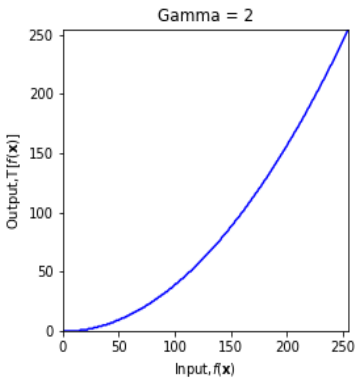


Figure (7)

(e). Gaussian Smoothing

```
#(1.e) Gaussian Smoothing
img = cv.imread('im02small.png',cv.IMREAD_GRAYSCALE)
gaussianKernel = cv.getGaussianKernel(5,2)
gaussianSmoothImage = cv.sepFilter2D(img,-1,gaussianKernel,gaussianKernel)
fig,axes=plt.subplots(1,2,sharex='all',sharey='all',figsize=(18,18))
```

The Gaussian smoothened image of the selected image is given under *Figure (8)* with a Kernel size 5, which means a **5 x 5 kernel with a sigma value of 2.**

Figure (8)

(f). Unsharp Masking

```
#(1.f) Unsharp Masking
alpha = 5
img = cv.imread('im02.png',cv.IMREAD_GRAYSCALE)
img = np.float32(img)
filteredImage = cv.GaussianBlur(img,(3,3),1)
edges = img - filteredImage
unsharpFilteredImage = img + alpha*edges
```

This is an image sharpening method which is used to highlight intensity transitions in the image. This is done in there steps as follow.

1. Blurring the original image using an averaging filter.
2. Substracting the blurred image from the original on and obtaining the mask.

3. Addition of mask to the original image.

Here, Gaussian kernel of 3 x 3 with an alpha of 2 is used as an averaging filter. It can be clearly observed that the edges have been sharpened in the output of *Figure (9).*

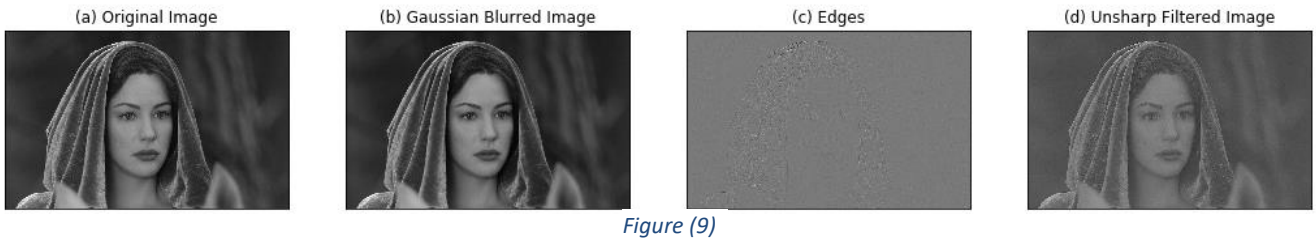


Figure (9)

(g). Median Filtering

```
#(1.g) Median Filtering
img = cv.imread('im03.png',cv.IMREAD_COLOR)
kernelSize = 5
saltpepper = noisy('s&p',img)
medianfiltered_image = cv.medianBlur(saltpepper,kernelSize)
img = cv.cvtColor(img,cv.COLOR_BGR2RGB)
medianfiltered_image = cv.cvtColor(medianfiltered_image,cv.COLOR_BGR2RGB)
```

Median filter is widely used in removing random noises like salt ad pepper noise from image. It has an excellent performance in noise reduction when comparing with other averaging filters. Median filters replace the value of central pixel by the median of the intensity values of pixels which are neighboring with it.

For this demonstration we are using a 5 x 5 median filter (**Kernel size = 5**). *Figure (10)* shows that most of the salt and pepper noise present in the original image have been vanished in the resulting image with the use of median filter.

Figure (10)

(h). Bilateral Filtering

```
#(1.h) Bilateral Filtering
img = cv.imread('im09.png',cv.IMREAD_COLOR)
d = 10
sigmaColor = 75
sigmaSpace = 75
bilateralfiltered_image = cv.bilateralFilter(img,d,sigmaColor,sigmaSpace)
img = cv.cvtColor(img,cv.COLOR_BGR2RGB)
bilateralfiltered_image = cv.cvtColor(bilateralfiltered_image,cv.COLOR_BGR2RGB)
```

Here also a Gaussian kernel is created by taking both distance from the middle pixel and intensity difference between pixels into the consideration. In this bilateral filtering we are eligible to change the weights which are being added by both factors by changing the two Sigma values named **SigmaColor and SigmaSpace.** As the both distance with neighboring pixels and intensity are taken into consideration in this filtering process, it is popular as a quite good version of filters.

Figure (11)

Due to this tonal weighting function in these bilateral functions very suitably used in preserving edges where a large total difference prevails while smoothing in more flat regions where a small tonal difference prevails. *Figure (11)* depicts how the bilateral filter will work on an image.

02.Counting Rice Grains

```
#(2) Counting Rice Grains
riceImage = cv.imread('rice.png',cv.IMREAD_GRAYSCALE)
kernel = np.ones((5,5),dtype="uint8")
gaussianBlurredImage = cv.GaussianBlur(riceImage,(7,7),2)
imageThreshold = cv.adaptiveThreshold(gaussianBlurredImage,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C,cv.THRESH_BINARY,9,0)
imageEroded = cv.erode(imageThreshold,kernel,iterations = 1)
imageDilate = cv.dilate(imageEroded,kernel,iterations = 1)
connectedOnes,labels = cv.connectedComponents(imageDilate)
riceGrains =connectedOnes - 1
print('The Total No: of Rice Grains =',riceGrains)
```

```
The Total No: of Rice Grains = 100
```

This is a binary segmentation technique where we are provided with two object types present in the image as rice grains and background. Adaptive thresholding can be used in binary segmentation as there are only two lighting conditions. Erode function will facilitate us to get rid of unwanted white noise prevail in the background. It will also help in detaching connected grains. Dilation will increase the white area in the image. According to the above algorithm the image contains **100 rice grains**. Images at each step of transformation are demonstrated in *Figure (12).*

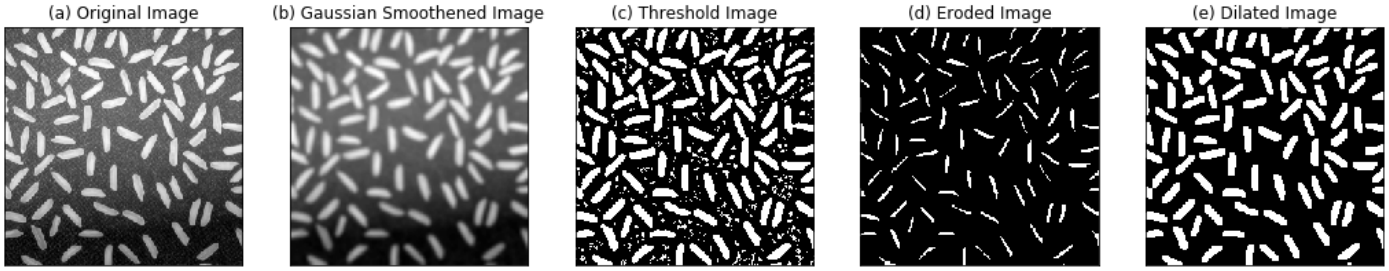


Figure (12)

03.(a). Zooming Images – Nearest-Neighbor

```
#(3.a) Nearest-Neighbor
def zoom(factor,img):
    height = img.shape[0]
    width = img.shape[1]
    zoomedImage = np.zeros((height*factor,width*factor,3),'uint8')
    for c in range(0,3):
        for x in range(0,height):
            for y in range(0,width):
                for i in range(x*factor,(x+1)*factor):
                    for j in range(y*factor,(y+1)*factor):
                        zoomedImage[i,j,c] = img[x,y,c]
    return zoomedImage
img = cv.imread('im08small.png',cv.IMREAD_COLOR)
factor = 4
zoomedImage = zoom(factor,img)
img = cv.cvtColor(img,cv.COLOR_BGR2RGB)
zoomedImage = cv.cvtColor(zoomedImage,cv.COLOR_BGR2RGB)
zoomedOriginal =cv.imread('im08.png',cv.IMREAD_COLOR)
error = 0.0
height1 = zoomedImage.shape[0]
width1 = zoomedImage.shape[1]
for x in range(0,height1):
    for y in range(0,width1):
        for c in range(0,factor-1):
            difference = zoomedImage[x,y,c] - zoomedOriginal[x,y,c]
            error += (difference)**2
print("SSD =",error)
```

The main key algorithm woven around this Nearest-Neighbor Zooming technique is, If the zooming factor is n, n x n square of pixels which have equal intensity for each and every pixel of original image are combined to make a zoomed image.

The resulted Sum of Squared Difference (SSD) from the selected image under Nearest-Neighbor is,

```
SSD = 209737952027.0
```

The original image before the zooming and the image after zooming is illustrated in *Figure (13)* below.

Figure (13)

(b). Zooming Images – Bilinear Interpolation

```
def onePlane(factor,p1,p2,p3,p4,c):
    zoomedSquare = np.zeros((factor,factor),np.float)
    topRow = np.linspace(p1[c],p2[c],factor)
    bottomRow = np.linspace(p3[c],p4[c],factor)
    for i in range(0,factor):
        zoomedSquare[i] = np.linspace(topRow[i],bottomRow[i],factor)
    return zoomedSquare
def getZoomedSquare(factorial,p1,p2,p3,p4):
    b = onePlane(factor,p1,p2,p3,p4,0)
    g = onePlane(factor,p1,p2,p3,p4,1)
    r = onePlane(factor,p1,p2,p3,p4,2)
    zoomedSquare = cv.merge((b,g,r))
    return zoomedSquare
img = cv.imread('im08small.png',cv.IMREAD_COLOR)
factor = 4
height = img.shape[0]
width = img.shape[1]
zoomedImage = np.zeros((height*factor,width*factor,3),'uint8')
for x in range(0,height - 1):
    for y in range(0,width - 1):
        zoomedSquare = getZoomedSquare(factor,img[x,y],img[x,y+1],img[x+1,y],img[x+1,y+1])
        for i in range(x*factor,(x+1)*factor):
            for j in range(y*factor,(y+1)*factor):
                zoomedImage[i,j] = zoomedSquare[i-x*factor,j-y*factor]
img = cv.cvtColor(img,cv.COLOR_BGR2RGB)
zoomedImage = cv.cvtColor(zoomedImage,cv.COLOR_BGR2RGB)
zoomedOriginal =cv.imread('im08.png',cv.IMREAD_COLOR)
error = 0.0
height1 =zoomedImage.shape[0]
width1 =zoomedImage.shape[1]
for x in range(0,height1):
    for y in range(0,width1):
        for c in range(0,factor-1):
            error = error + (zoomedImage[x,y,c] - zoomedOriginal[x,y,c])**2
print("SSD =",error)
```

The main key algorithm woven around this Bilinear Interpolation Zooming technique is, if the zooming factor is n, n x n square of pixels is created for each and every pixel in the original image in which intensity of corner edges is equal to the particular pixel in the original image and three neighborhood pixels which makes a square along with particular pixel. Afterward, the rest of the pixels of n x n square of pixels is filled using gradients. So, those n x n squares of pixels are combined to make the zoomed image.

The resulted Sum of Squared Difference (SSD) from the selected image under bilinear Interpolation is,

```
SSD = 221809406492.0
```

Figure (14)