Assignment 01 **180427N**

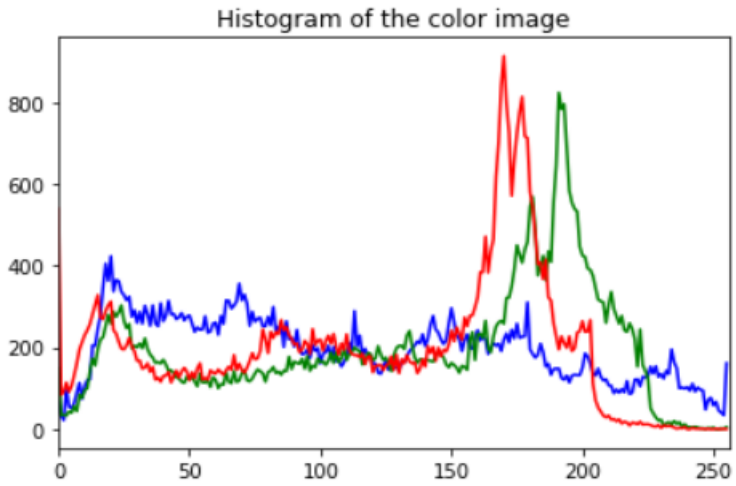
1. Necessary code for first question (plotting code is not included)

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

image = cv.imread('./images/k1.jpg', cv.IMREAD_COLOR)

# a .histogram of color image
img = cv.cvtColor(image, cv.COLOR_BGR2RGB)
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 of the color image")
plt.show()

# b .Histogram equalization
gray_img = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
equalized_img = cv.equalizeHist(gray_img)

plt.hist(gray_img.flatten(), 256, [0,256], color = 'r')
plt.title('Histogram of the Original Image'), plt.xlim([0, 256])
plt.show()
plt.hist(equalized_img.flatten(), 256, [0,256], color = 'r')
plt.title('Histogram of the Equalized Image'), plt.xlim([0, 256])
plt.show()
result = np.hstack((gray_img, equalized_img))
plt.axis('off'), plt.imshow(result, cmap = 'gray'), plt.show()
```

```
# c .Intensity transformation
c = np.array([(100,50), (150,200)])#coordinates where tranformation has changed

t1 = np.linspace(0, c[0,1], c[0,0]+1 -0) # valuse 0 to 50 with 101 points
t2 = np.linspace(c[0,1]+1, c[1,1], c[1,0] - c[0,0])
t3 = np.linspace(c[1,1]+1, 255, 255 - c[1,0])

transform = np.concatenate((t1,t2), axis = 0).astype('uint8')
transform = np.concatenate((transform,t3), axis = 0).astype('uint8')

plt.plot(transform) , plt.xlim(0,255), plt.ylim(0,255)
plt.show()
trans_img = cv.LUT(img, transform)

# d .Gamma correction
gamma = 2
tabel = np.array([(i/255)**gamma*255  for i in np.arange(0,256)]).astype(np.uint8)
trans = cv.LUT(img, tabel)
```

```
# e. Gaussian smmothing
sigma = 2
gaussian_kernel = cv.getGaussianKernel(9, sigma)
gau_smoothed_img = cv.sepFilter2D(gray_img, -1, gaussian_kernel, gaussian_kernel,
                  anchor = (-1, 1), delta = 0 , borderType = cv.BORDER_REPLICATE)

#f .Unsharp masking
diff = gray_img.astype('float32') - gau_smoothed_img.astype('float32')
sharpened_img = cv.addWeighted(gray_img.astype('float32') ,1.0, diff, 1.5,0)

def solt_pep_noisy(image):
    s_vs_p = 0.5
    amount = 0.2
    out = np.copy(image)
    num_salt = np.ceil(amount*image.size*s_vs_p) # no.on salts
    coors = [np.random.randint(0, i-1, int(num_salt)) for i in image.shape]
    out[coors] = 255
    num_pepper = np.ceil(amount*image.size*(1-s_vs_p)) # no.on pepper
    coors = [np.random.randint(0, i-1, int(num_salt)) for i in image.shape]
    out[coors] = 0
    return out

im_n = solt_pep_noisy(gray_img.astype('float32'))
#g.median filtering
median_blured_img = cv.medianBlur(im_n, 5)
#h.bilateral filtering
bilateral_filltered_img = cv.bilateralFilter(gray_img,5,75,75)
```

a. Histogram computation

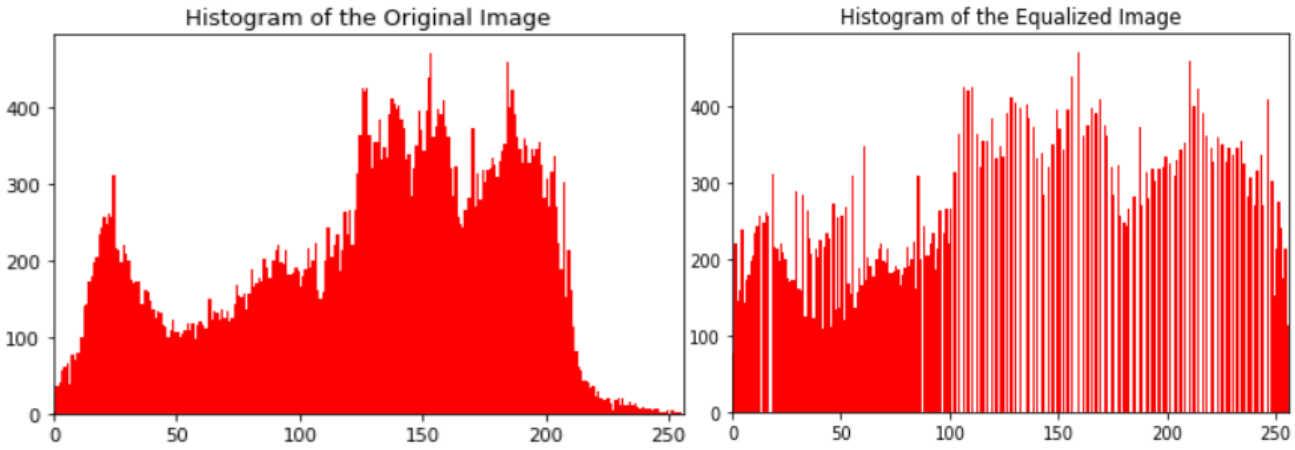


This is the histogram of selected image. Since it is a color image it shows three different intensity distribution. According to this distribution, the intensity is almost equally distributed in the region of 0 to 150. In the region of 150 to 200 the probability of blue channel is less compared to other channels. We can conclude most of the pixel values are in the region of 150 to 200.

b. Histogram equalization

Since it a gray level transformation, I have used gray version of selected image. The below two plots shows histogram of original gray level image and histogram equalized image. In the original image the probability of bright pixel (above 200) is low compared to other pixel values. However, after the equalization processes the histogram is almost flat. That means

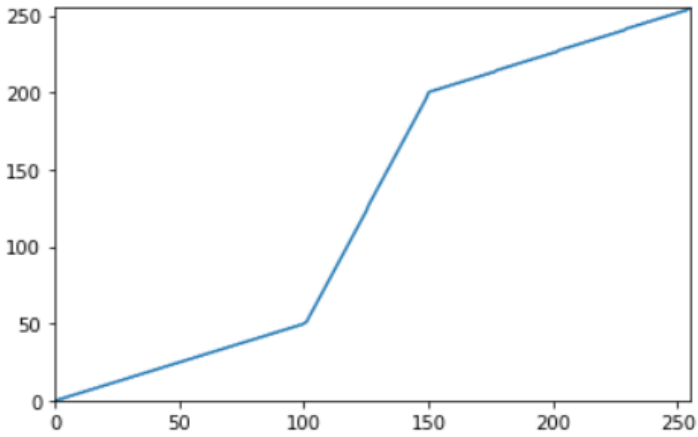
all pixel values are equally distributed. This equalization process results a vibrant image (high contrast).



Left – Original image

Right – after equalization (high contrast image)

c. Intensity Transformation

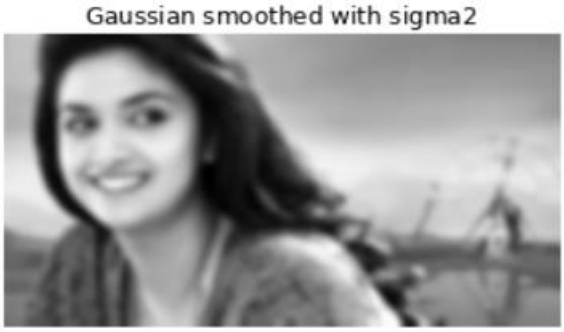


This plot depicts my intensity transformation function. According to this, bright pixels are even brightened, and darks pixels are even darkened. In the resulting intensity transformed image bright pixels are even brighten (face, white color in sky) and dark pixels are even darkened (hair, boarder at the wheel). This is what we expected from this transformation.

d. Gamma correction

It is a low power transformation through this we can transform the intensity level according to gamma value. Here I have used gamma as 2 (>1), so wide region of dark pixels maps to small region of dark pixels and small region of bright pixel map to wide region of bright pixels. Because of this reason the image darked after gamma (=2) correction. Darkness of the image will increase with increasing gamma. If gamma is less than one image will be brighter.

e. Gaussian smoothing



Smoothing is a low pass filtering process where it removes high frequency noise from an image. This image is gaussian smoothed image with standard deviation of 2. Although noise is reduced it introduces blurs at the edges(hair). Blurring is increased with increasing sigma (SD).

f. Unsharp masking

Sharpening is opposite to smoothing high pass filtering process. It is used to enhance high frequency component of an image such as edges and fine details. This picture shows sharpened image using unsharp masking method. We can see edges (hair, edge of the wheel) are sharpened compared to original image.

h. Median filtering

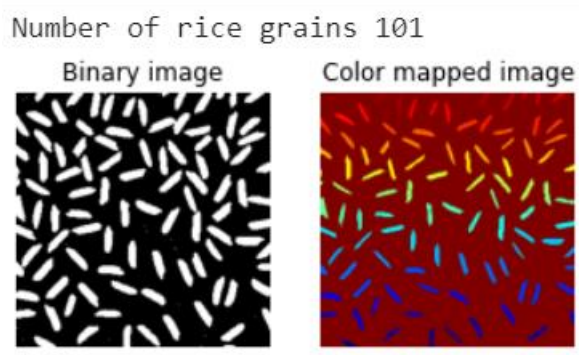
It is a non-linear filtering method used to reduce noise. For a reasonable clarification, I added some salt and pepper noise on original image and then used median filter to reduce that. As you can see this works well when we have noisy like salt and pepper. I used 5x5 kernel to this process. It reduces the noise and preserves the edges unlike gaussian filtering. So, this filtering method is widely used in many applications.

g. Bilateral filtering

It is a non-linear filter that reduces the noise, smooth the image with preserving edges. A pixel is replaced by weighted average of its neighbors. This weight depends on Euclidean distance as well as radiometric difference. This image is bilateral filtered image. We can see it smoothed the image and reduced the noise, but the edges were not blurred as it was in gaussian smoothing. However, this is not capable to reduce the noise as gaussian in some scenarios.

2.

```
# RICE counting
%matplotlib inline
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

rice_img = cv.imread('./a01images/rice.png', cv.IMREAD_GRAYSCALE)

# find local threshold based on a small region
output_adapthresh = cv.adaptiveThreshold (rice_img, 255.0,
        cv.ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY, 25, -20)

# morphologial erosion - helps to remove conjoined grains
kernel = np.ones((4,4),np.uint8)
output_erosion = cv.erode(output_adapthresh, kernel)

retval, labels, stats, centroids = cv.connectedComponentsWithStats(output_erosion)
print("Number of rice grains", retval)
#Appling a color map
color_mapped_img = cv.applyColorMap((labels/np.amax(labels)*255).astype('uint8'), cv.COLORMAP_JET)

fig, ax = plt.subplots(1,2, figsize = (5,5))
ax[0].imshow(output_adapthresh, cmap = 'gray'), ax[0].axis('off')
ax[1].imshow(color_mapped_img), ax[1].axis('off')
ax[0].set_title('Binary image'), ax[1].set_title('Color mapped image')
plt.show()
```



Through my code I have got 101 as the count of rice grains however it depends on how we have tuned the threshold value. Here I have used adaptive threshold that considers small regions to find. (In general, selection of threshold depends on lightning of the image).

3.

[3]:
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

def zooming_image(img, zoom_method, factor):
    rows, cols = int(factor*img.shape[0]),  int(factor*img.shape[1])
    zoomed_img = np.zeros((rows, cols, img.shape[2]), dtype=img.dtype)

    if zoom_method == "nearest-neighbor" :
        for i in range(0, rows):
            for j in range(0, cols):
                zoomed_img[i,j] = img[int(i/factor), int(j/factor)]

    if zoom_method == "bilinear interpolation":
        for i in range(0, rows):
            for j in range(0, cols):
                #original
                x, y = i/factor, j/factor
                x1 , y1 = int(x), int(y)
                x2, y2 = min(x1+1, img.shape[0]-1), min(y1+1, img.shape[1]-1)

                xf = x - x1
                yf = y - y1

                k = xf*img[x1, y1] + (1 - xf)*img[x2 ,y1]
                m = xf*img[x1, y2] + (1 - xf)*img[x2 , y2]
                zoomed_img[i,j] = yf*k + (1-yf )*m
    return zoomed_img
```

```
img = cv.imread('./a01images/im02small.png',cv.IMREAD_COLOR)
factor = 4
bilinear_zoomed_img = zooming_image(img, "bilinear interpolation", factor)
n_n_zoomed_img = zooming_image(img, "nearest-neighbor", factor)

orig = cv.imread('./a01images/im02.png',cv.IMREAD_COLOR)

mse_b = np.sum((orig.astype('float') - bilinear_zoomed_img.astype('float'))**2)/(orig.shape[0]*orig.shape[1])
mse_n = np.sum((orig.astype('float') - n_n_zoomed_img.astype('float'))**2)/(orig.shape[0]*orig.shape[1])
print("Bilinear MSE: {}, nearest-neighbor MSE: {}".format(mse_b, mse_n))
```

```
Bilinear MSE: 368.08457508680556, nearest-neighbor MSE: 79.33826215277777
```

The above code implements two types of zooming which are nearest neighbor method and bilinear interpolation method. Nearest neighbor is very simple and easy to implement it replicates each pixel by zooming factor in both row and column wise (same color is replicated). On the other hand, bilinear method is 2D linear interpolation where a particular pixel value is computed based on weighted average of its four neighbor pixels in original image. In this method new pixel value is introduced (different color). The computation time of a new pixel value is higher for the bilinear method than the nearest neighbor method.

To test the algorithm, I used mean square error (MSE) instead of SSD. When I tested each given image nearest neighbor method gave small MSE than bilinear method (One case is shown above). Therefore, nearest neighbor method works well for zooming. However, bilinear method produces smother image and reduces the distortion than nearest neighbor method.