Assignment 1

H.K.R.L. GUNASEKARA 180205H

Image

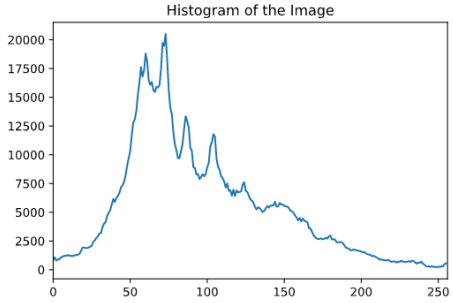
Q1)

a) Histogram Computation

```
1 img=cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_GRAYSCALE)
2 hist =cv.calcHist([img],[0],None,[256],[0,256])
3 plt.plot(hist)
4 plt.xlim([0,256])
5 plt.show()
```

Original Image

Histogram of the Image

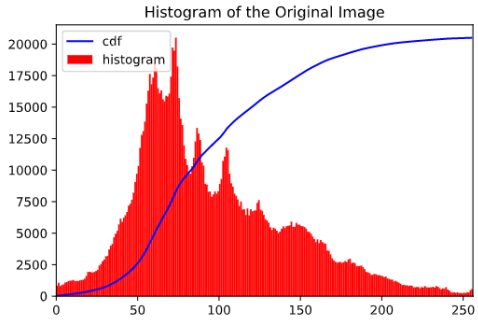


b) Histogram Equalization

```
1 img=cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_GRAYSCALE)
2 hist,bins =np.histogram(img.ravel(),256,[0,256])
3 cdf =hist.cumsum()
4 cdf_normalized=cdf*hist.max()/cdf.max()
5 equ=cv.equalizeHist(img)
6 hist,bins =np.histogram(equ.ravel(),256,[0,256])
7 cdf = hist.cumsum()
8 cdf normalized=cdf*hist.max()/cdf.max()
```

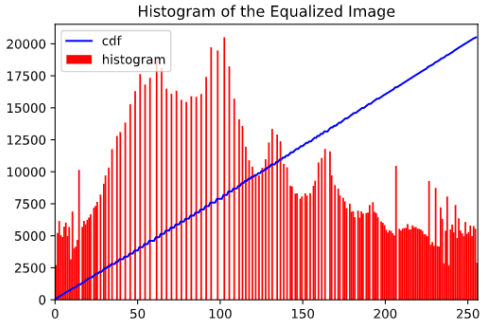
Original Image

Histogram of the Original Image



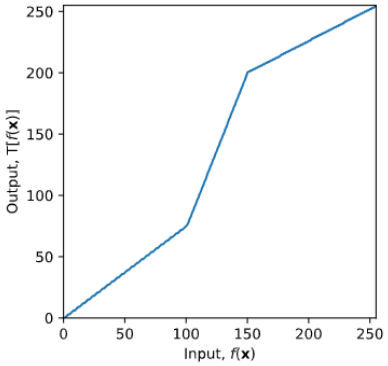
Histogram Equalized Image

Histogram of the Equalized Image



c)Intensity Transformation

```
c=np.array([(100,75),(150,200)])
t1=np.linspace(0,c[0,1],c[0,0]+1-0).astype('uint8')
t2=np.linspace(c[0,1]+1,c[1,1],c[1,0]-c[0,0]).astype('uint8')
t3=np.linspace(c[1,1]+1,255,255-c[1,0]).astype('uint8')
transform=np.concatenate((t1,t2),axis=0).astype('uint8')
transform=np.concatenate((transform,t3),axis=0).astype('uint8')
image_transformed=cv.LUT(img_orig,transform)
```



Intensity Transformation function

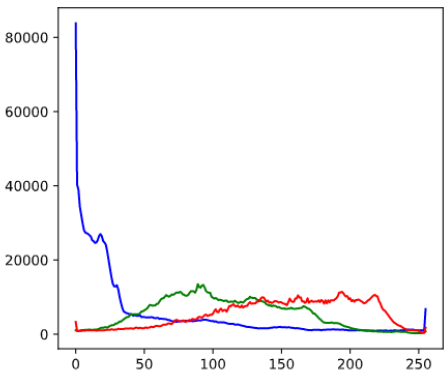
Original Image

Intensity Transformed Image

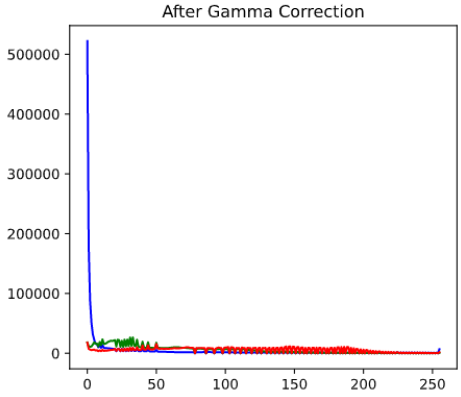
d)Gamma Correction

```
img_orig = cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_COLOR)
gamma =2#Changing the Gamma
table=np.array([(i/255.0)**(gamma)*255 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)
```

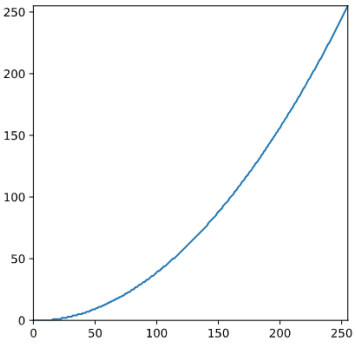
Before Gamma Correction



After Gamma Correction



Gamma Plot



$\gamma = 2$

Before Gamma Correction

After Gamma correction

e) Gaussian Smoothing

```
1 gaussian_kernal=cv.getGaussianKernel(15,3)
2 imgc = cv.filter2D(img_orig , -1 ,gaussian_kernal)
```

kernel size =15 x 15

σ= 3

Original

Gausian smoothed

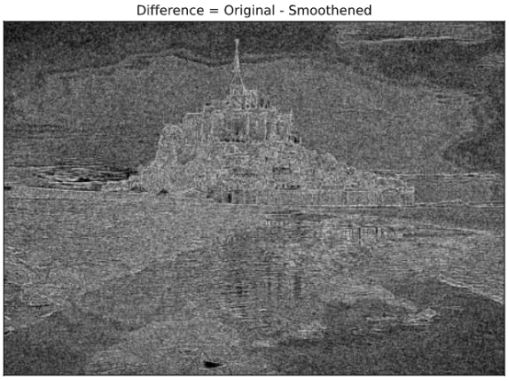
f) Unsharp Masking

```
1 img_orig = cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_GRAYSCALE)
2 gaussian_kernal=gkern(5,3)
3 img_smoothened = cv.filter2D(img_orig , -1 ,gaussian_kernal)
4 differenece=img_orig-img_smoothened
5 img_unsharpened=img_orig+differenece
```

Original

Smoothened

Difference = Original - Smoothened

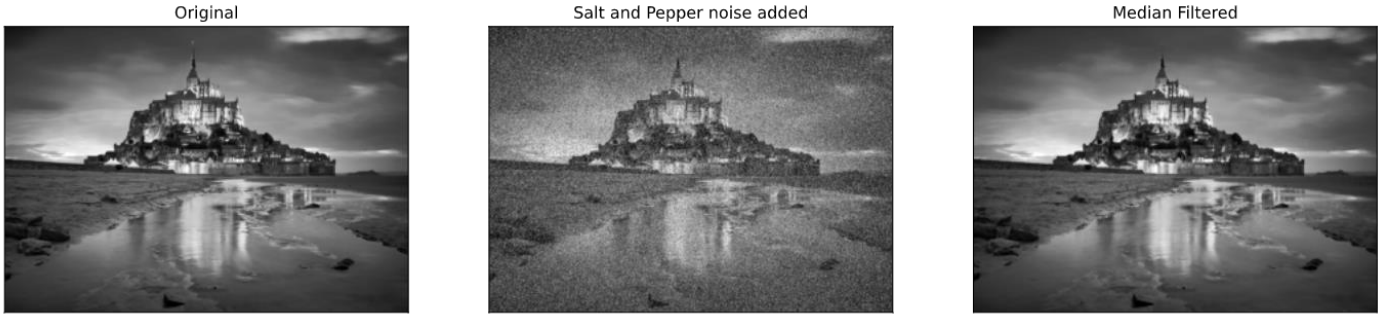


Unsharpened = Original + Difference

g) Median Filtering

```
1  img_orig = cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_GRAYSCALE)
2  def addsalt_pepper(img, SNR):
3      Img_ = img.copy()
4      h, w = Img_.shape
5      mask = np.random.choice((0,1, 2), size=(h, w), p=[SNR,(1 - SNR) / 2., (1 - SNR) / 2.])
6      Img_[mask == 1] = 255
7      Img_[mask == 2] = 0
8  return Img_
9  img_saltandpepper=addsalt_pepper(img_orig, 0.7)
10 img_median=cv.medianBlur(img_saltandpepper,5)
```

kernel size = 5 x 5



---

h) Bilateral filtering

```
1 img_orig = cv.imread('../images/le-mont-saint.jpg',cv.IMREAD_COLOR)
2 img_orig=cv.cvtColor(img_orig,cv.COLOR_BGR2RGB)
3 img_bilateral=cv.bilateralFilter(img_orig,15,200,80)
```

In filters like Gaussian Filter the filtering is done by 2D convolution using a gaussian kernel and this method blurs everything regardless of the pixels being on an edge or not, which leads to blurred edges. On the other hand, Bilateral Filtering can be used to reduce the noise while preserving the edges.

Bilateral filter is represented as a weighted average of nearby pixels which is somewhat similar to Gaussian Filter but, the difference is Bilateral filter considers the difference in neighboring pixel values which allows it to preserve edges while smoothing. In other words for a pixel to influence another not only the locality of the pixel but also the value of pixels must be similar too.

Bilateral Filter is given as follows,

Spatial Weight Range Weight

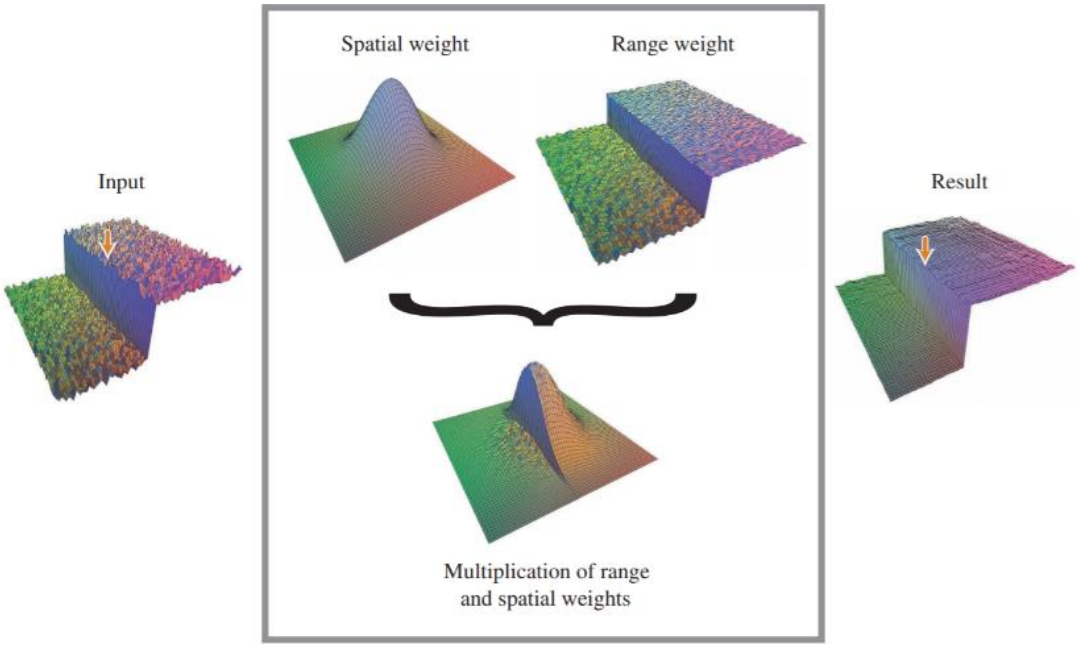
$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) \, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|) \, I_{\mathbf{q}},$$

normalization factor $W_{\mathbf{p}}$ ensures pixel weights sum to 1.0:

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in \mathcal{S}} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) \, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|).$$

$G_{\sigma s}$ = Spatial Gaussian weighting that decreases the influence of distant pixels

$G_{\sigma r}$ = Range Gaussian that decreases the influence of pixels q when their intensity values differ from I_p



Reference: Bilateral Filtering: Theory and Applications By Sylvain Paris, Pierre Kornprobst, Jack Tumblin and Fr´edo Durand

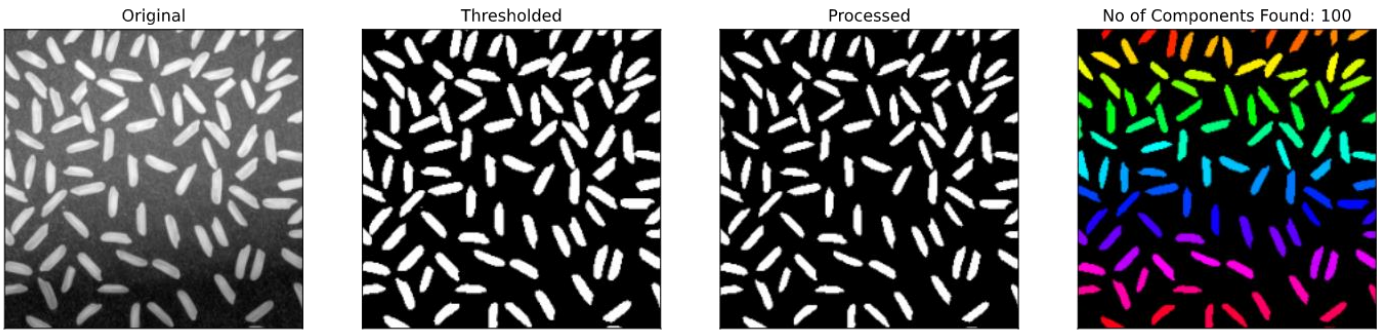
---

Q2)

1) First the Original Image is thresholded using adaptive thresholding in order to avoid problems like illumination gradient.
2) Then the image is eroded so that rice is clearly separated from each other.
3) Using Connected Components, the image is labeled.
4) Then the labeled image is represented as a HSV image and converted to RGB in order to show the components in a color map.

```
img_orig = cv.imread('../Assignments/a01images/rice.png',0)
img_thresholded= cv.adaptiveThreshold(img_orig,255.0,cv.ADAPTIVE_THRESH_MEAN_C,cv.THRESH_BINARY,51,20.0)#Thresholding
kern=np.ones((2,2),np.uint8)
img_processed=cv.erode(img_thresholded,kern)
np.set_printoptions(threshold=sys.maxsize)
num_labels, labels = cv.connectedComponents(img_processed) # Map component labels to hue values
label_hue = np.uint8(179*labels/np.max(labels))
blank_ch = 255*np.ones_like(label_hue)
labeled_img = cv.merge([label_hue, blank_ch, blank_ch]) # Converting cvt to BGR
labeled_img = cv.cvtColor(labeled_img, cv.COLOR_HSV2RGB
labeled_img[label_hue==0] = 0 # To set the background to black
```

100 Rice Grains were Found.



Q3)

Nearest neighbour Interpolation

```
def nearest_neighbour_interploation(img,scale):
    height,width,color =img.shape
    m=round(height*scale)
    n=round(width*scale)
    rescaled =np.zeros((m,n,color),np.uint8)
    for c in range(color):
        for i in range(m):
            for j in range(n):
                #Finding corresponding pixel in the image
                x = int(i/scale)
                y = int(j/scale)
                x=min(x,height-1)
                y=min(y,width-1)
                rescaled[i][j][c]=img[x,y][c]
    return rescaled
```

SSD calculation

```
def SSD(imageA, imageB):
    SSD = np.sum((imageA- imageB) ** 2)
    SSDperpixel = SSD/imageA.size
return SSD,SSDperpixel
```

Bilinear interpolation

```
def bilinear_interpolation(img, scale):
    height,width,color= img.shape
    m = round(height*scale)
    n = round(width*scale)
    resized = np.zeros((nh,nw,c),np.uint8)
    for c in range(color):
        for i in range(m):
            x1 = max(int(floor(i/scale)),1)
            x2 = int(ceil(i/scale))
            q = (i%x1)/scale
            for j in range(n):
                y1 = max(int(floor(j/factor)),1)
                y2 = int(ceil(j/scale))
                p = (j%y1)/scale
                t_left = img[x1-1][y1-1][c]#Finding the 4 corners
                t_right = img[x2-1][y1-1][c]
                b_left = img[x1-1][y2-1][c]
                b_right = img[x2-1][y2-1][c]
                resized[i][j][c] = q*(1- p)*t_right + (1-q)*(1-p)*t_left + q*p*b_right + (1-q)*p*b_left
     return resized
```

Original

Bilinear Interpolation

Nearest Neighbour Interpolation

SSD Calculation results

Image No	Bilinear Interpolation		Nearest Neighbor Interpolation	
	SSD	SSD per pixel	SSD	SSD per pixel
01	296924308	47.7	194613476	31.3
02	143889055	20.8	82266716	11.9
04	2092298628	84.0	1959248854	78.7
05	392457484	56.7	349589925	50.5
06	244925128	39.3	190070298	30.5
07	397038965	32.3	343623385	27.9
09	192366740	33.3	121813060	21.1