PRACTICAL FILE

BE (CSE) 7th Semester

Digital Image Processing

Computer Science and Engineering
University Institute of Engineering and Technology
Panjab University, Chandigarh – 160014, INDIA
2020

Submitted By:

Jatin Ghai
UE183042
Group - 3
CSE - Section 1

Submitted To:

Ms. Sabrina

INDEX

Basic image processing

Reading an image:

```
import cv2
import copy
import matplotlib.pyplot as plt

#Read Image
img = cv2.imread ("C:/Users/jatin/Desktop/VideoStream/DIP LAB 1/RAW3.tif",1)
cv2.imshow("Normal Image", img)
cv2.waitKey(20000)
cv2.destroyAllWindows()
```

OUTPUT:

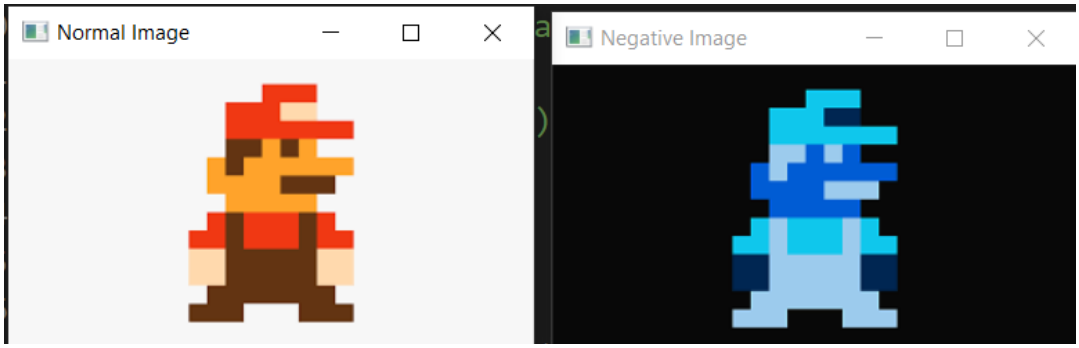
Negative of image:

Image negative is produced by subtracting each pixel from the maximum intensity value. e.g. for an 8-bit image, the max intensity value is $2^8 - 1 = 255$, thus each pixel is subtracted from 255 to produce the output image

```
import cv2
import copy
```

```
import matplotlib.pyplot as plt

#Read Image
img = cv2.imread ("C:/Users/jatin/Desktop/VideoStream/DIP LAB 1/RAW3.tif",1)

#Negative Image
negative = copy.deepcopy(img)
for x in range(len(img)):
    for y in range(len(img[x])):
        for z in range(len(img[x][y])):
            negative[x][y][z] = 255 - negative[x][y][z]

cv2.imshow("Normal Image", img)
cv2.imshow("Negative Image", negative)
cv2.waitKey(20000)
cv2.destroyAllWindows()
```

OUTPUT:



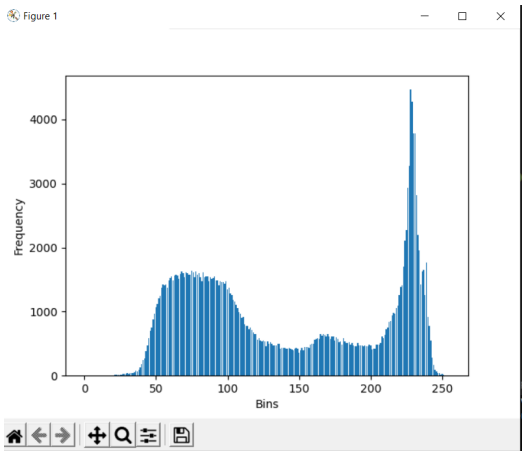
Histogram of image:

Histogram of an image, like other histograms also shows frequency. But an image histogram shows frequency of pixel intensity values.

```
import cv2
import copy
```

```
import matplotlib.pyplot as plt

#Read Image
img = cv2.imread ("C:/Users/jatin/Desktop/VideoStream/DIP LAB 1/RAW3.tif",1)

#histogram for red pixels
x_plt = [y for y in range(256)]
y_plt = [0  for y in range(256)]

for x in range(len(img)):
    for y in range(len(img[x])):
            y_plt[negative[x][y][0]] += 1


plt.bar(x_plt,y_plt,align='center')
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

OUTPUT:

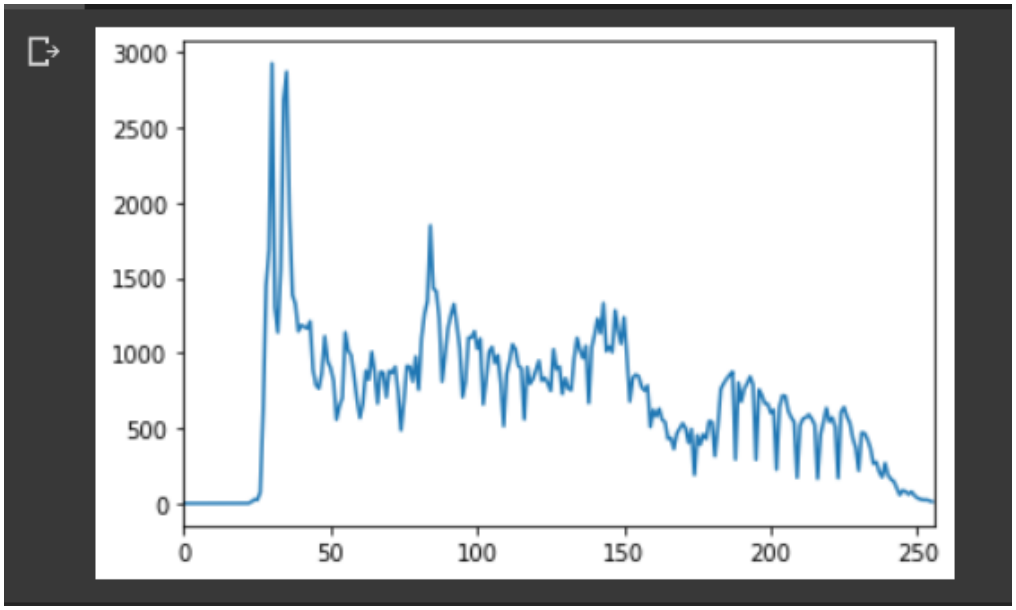


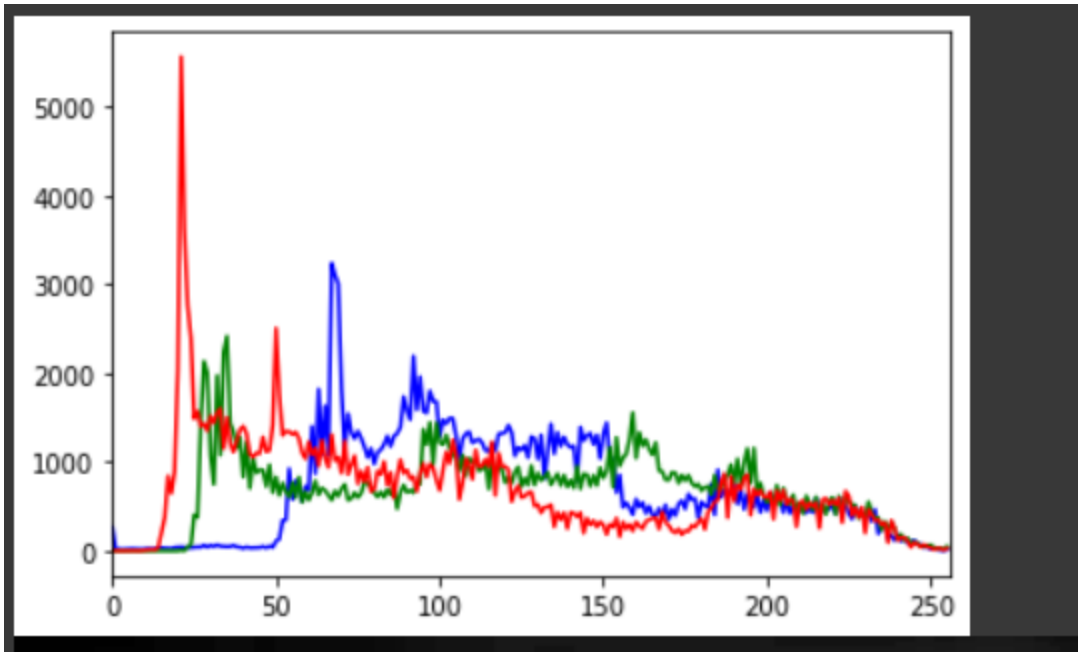
Histogram equalization

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. This method usually increases the global contrast of many images, especially when the image is represented by a narrow range of intensity values. Histogram equalization accomplishes this by effectively spreading out the highly populated intensity values which degrade image contrast.

```
#histogram Equalization
img = cv2.imread('/content/point1.jpg',0)
hist = cv2.calcHist([img],[0],None,[256],[0,256])
plt.imshow(img)
cum_dist_func = hist.cumsum()
cum_dist_func_form = cum_dist_func * hist.max()/cum_dist_func.max()

plt.plot(cum_dist_func_form)
plt.xlim([0,256])

plt.show()

pix = img.size
cum_dist_func_hist = cum_dist_func / pix
equalis_hist = np.floor(255 * cum_dist_func_hist)
equalis_img = [equalis_hist[i] for i in img.flatten()]
equalis_img = np.reshape(equalis_img, img.shape).astype(np.unit8)
plt.show()

hist_eq =  cv2.calcHist([equalis_img],[0],None,[256],[0,256])
form_hist = hist_eq.cumsum()
norm_equlais = form_hist * hist_eq.max() / form_hist.max()
plt.plot(norm_equlais)
plt.show()
```

OUTPUT:


3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250


5000
4000
3000
2000
1000
0
0
50
100
150
200
250

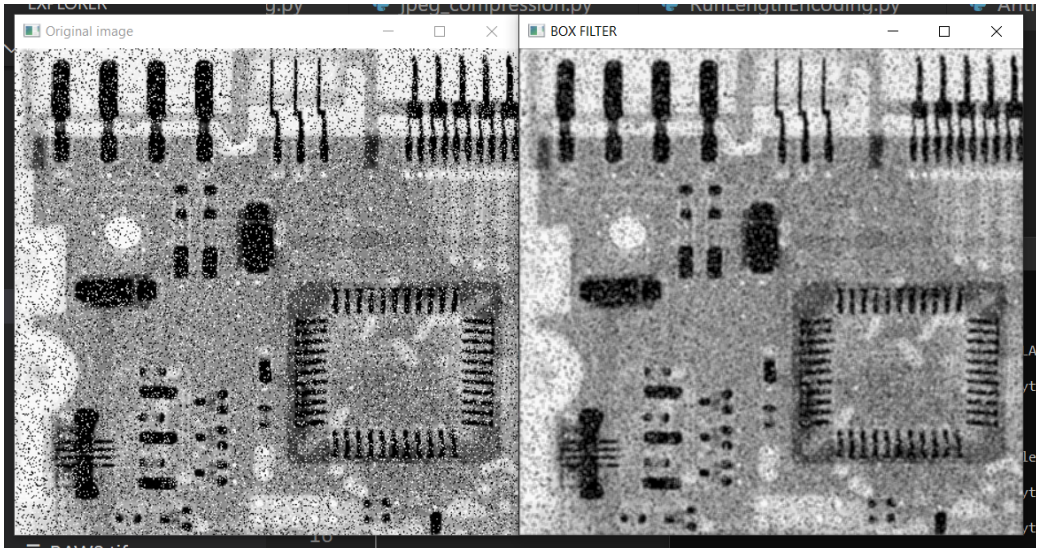
Spatial Domain Filtering

Box Filter

Average (or mean) filtering is a method of 'smoothing' images by reducing the amount of intensity variation between neighboring pixels.

```
def Box_filter(img):

    box_array = np.zeros([img.shape[0]+2, img.shape[1]+2], dtype = int)

    for i in range(1,img.shape[0]+1):
        for j in range(1,img.shape[1]+1):
            box_array[i][j] = img[i-1][j-1]

    box_filter = copy.deepcopy(img)

    for i in range(1,box_array.shape[0]-1):
        for j in range(1,box_array.shape[1]-1):
            box_filter[i-1][j-1] = (box_array[i-1][j-1] + box_array[i-1][j] + box_array[i-1][j+1] + box_array[i][j-1]
+ box_array[i][j] + box_array[i][j+1] + box_array[i+1][j-1] + box_array[i+1][j] + box_array[i+1][j+1]) / 9

    cv2.imshow("BOX FILTER", box_filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW4.tif",0)
Box_filter(img)
```

OUTPUT:



Gaussian Filter

Gaussian Filter (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function.

```
def Gaussian_filter(img):

  w = np.array([[4, 8, 4],
          [8, 16, 8],
          [4, 8, 4]], dtype = np.int16)
  div = 1  #sum of values in kernal



  array = np.pad(img, pad_width = ((1,1),(1,1)), mode = 'reflect')

  filter = np.full(img.shape,255, dtype = np.uint16)
  # print(img)
  for i in range(1,array.shape[0]-1):
    for j in range(1,array.shape[1]-1):
      filter[i-1][j-1] = np.sum(array[i-1:i+2, j-1:j+2] * w) / div
```

```
    cv2.imshow("GAUSSIAN FILTER", filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW5.tif",0)
Gaussian_filter(img)
```

OUTPUT:

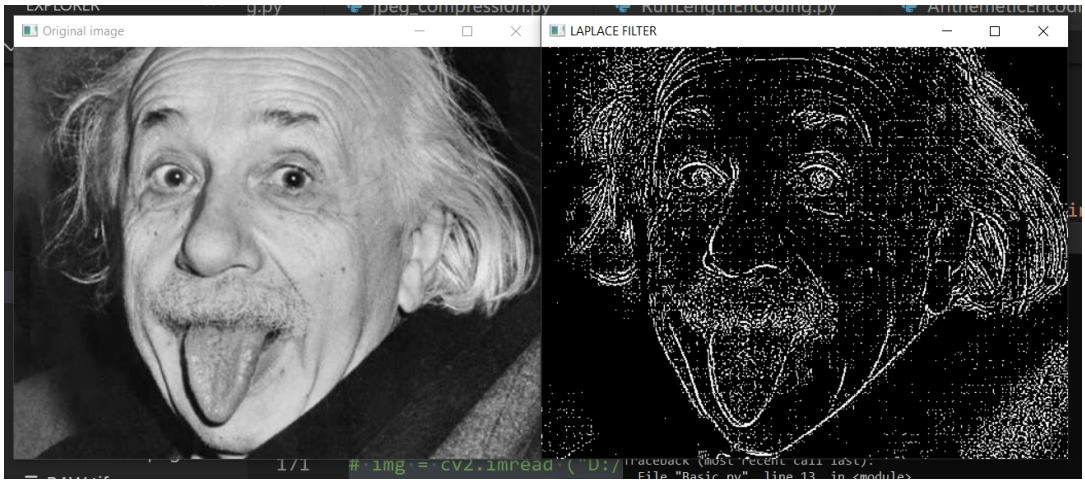
Laplacian Filter

A Laplacian filter is an edge detector used to compute the second derivatives of an image, measuring the rate at which the first derivatives change.

```
def Laplacian_filter(img):

    w = np.array([[0, -1, 0],
                  [-1, 4, -1],
                  [0, -1, 0]], dtype = np.int16)
    div = 16
```

```
    array = np.pad(img, pad_width = ((1,1),(1,1)), mode = 'reflect')

    filter = np.full(img.shape,255, dtype = np.uint8)

    for i in range(1,array.shape[0]-1):
        for j in range(1,array.shape[1]-1):
            filter[i-1][j-1] = np.sum(array[i-1:i+2, j-1:j+2] * w) / div

    # cv2.imshow("Original image",img)
    cv2.imshow("LAPLACE FILTER", filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/Einstein.jpg",0)
Laplacian_filter(img)
```

OUTPUT:

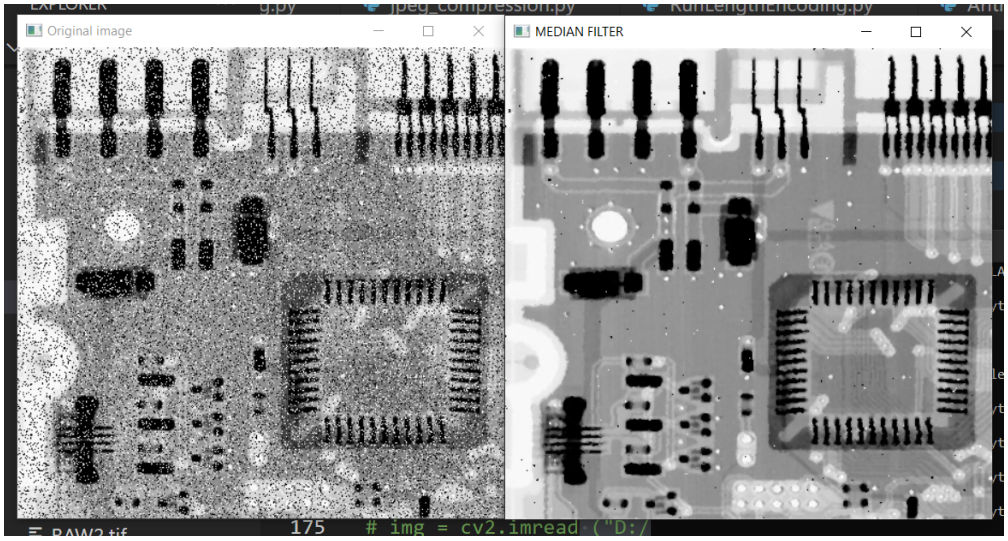


Median Filter

Median Filter is often used to remove noise from an image or signal. The median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighboring pixel values, it replaces it with the median of those values.

```
def Median_filter(img):

    array = np.zeros([img.shape[0]+2, img.shape[1]+2], dtype = int)



    for i in range(1,img.shape[0]+1):
        for j in range(1,img.shape[1]+1):
            array[i][j] = img[i-1][j-1]

    for j in range(1,img.shape[1]+1):
        array[0][j] = img[0][j-1]

    for j in range(1,img.shape[1]+1):
        array[img.shape[0]+1][j] = img[img.shape[0]-1][j-1]

    for i in range(1,img.shape[0]+1):
        array[i][0] = img[i-1][0]

    for i in range(1,img.shape[0]+1):
        array[i][img.shape[1]+1] = img[i-1][img.shape[1]-1]

    filter = copy.deepcopy(img)

    for i in range(1,array.shape[0]-1):
        for j in range(1,array.shape[1]-1):
                    temp = [array[i-1][j-1], array[i-1][j], array[i-1][j+1], array[i][j-1], array[i][j], array[i][j+1],
array[i+1][j-1], array[i+1][j], array[i+1][j+1]]
            temp = sorted(temp)
            filter[i-1][j-1] = temp[4]

    cv2.imshow("MEDIAN FILTER", filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW4.tif",0)
Median_filter(img)
```

OUTPUT:



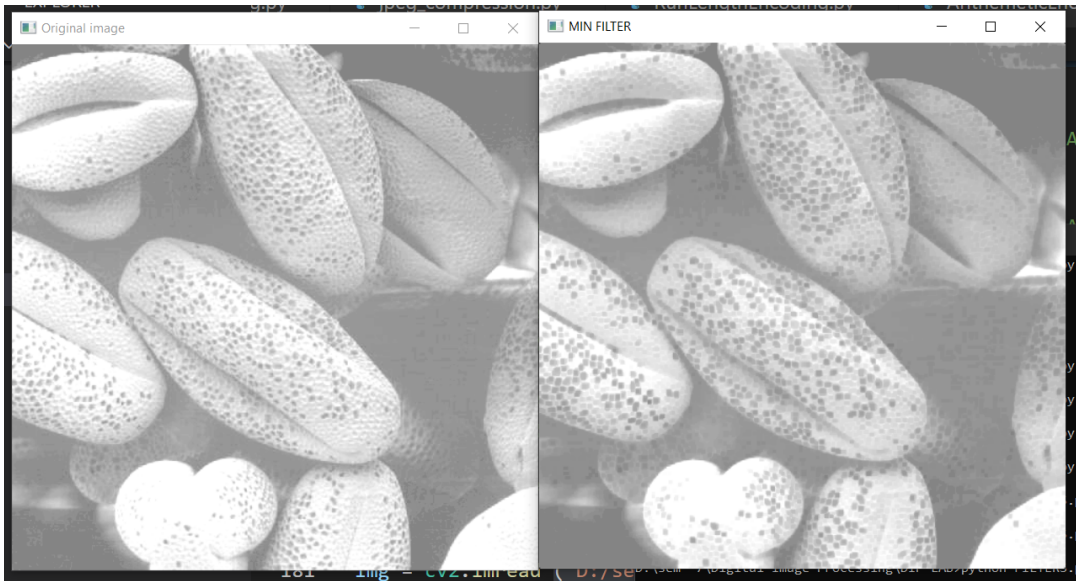
Min Filter

Min filter replaces the pixel value with the min of neighboring pixel values, it replaces it with the median of those values. This filter diminishes salt-like noise.

```
def Min_filter(img):

  array = np.zeros([img.shape[0]+2, img.shape[1]+2], dtype = int)
  for i in range(1,img.shape[0]+1):
    for j in range(1,img.shape[1]+1):
      array[i][j] = img[i-1][j-1]

  for j in range(1,img.shape[1]+1):
    array[0][j] = img[0][j-1]

  for j in range(1,img.shape[1]+1):
    array[img.shape[0]+1][j] = img[img.shape[0]-1][j-1]
  for i in range(1,img.shape[0]+1):
    array[i][0] = img[i-1][0]

  for i in range(1,img.shape[0]+1):
    array[i][img.shape[1]+1] = img[i-1][img.shape[1]-1]
```

```
    filter = copy.deepcopy(img)
    for i in range(1,array.shape[0]-1):
        for j in range(1,array.shape[1]-1):
                    temp = [array[i-1][j-1], array[i-1][j], array[i-1][j+1], array[i][j-1], array[i][j], array[i][j+1],
array[i+1][j-1], array[i+1][j], array[i+1][j+1]]
            filter[i-1][j-1] = min(temp)

    cv2.imshow("MIN FILTER", filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW3.tif",0)
Min_filter(img)
```

OUTPUT:



Max Filter

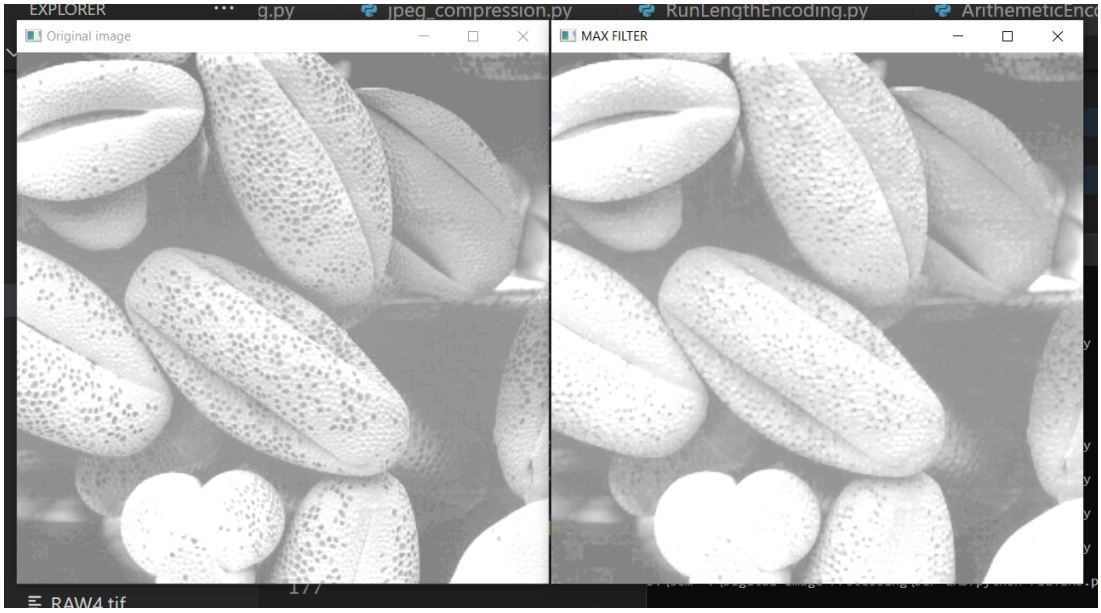
Max filter replaces the pixel value with the max of neighboring pixel values, it replaces it with the median of those values. The max filter is used to find the brightest point in an

image. It uses the maximum intensity value in a sub image area. This filter reduces the pepper noise because it has very low values of intensities.

```
def Max_filter(img):

    array = np.zeros([img.shape[0]+2, img.shape[1]+2], dtype = int)



    for i in range(1,img.shape[0]+1):
        for j in range(1,img.shape[1]+1):
            array[i][j] = img[i-1][j-1]

    for j in range(1,img.shape[1]+1):
        array[0][j] = img[0][j-1]

    for j in range(1,img.shape[1]+1):
        array[img.shape[0]+1][j] = img[img.shape[0]-1][j-1]

    for i in range(1,img.shape[0]+1):
        array[i][0] = img[i-1][0]

    for i in range(1,img.shape[0]+1):
        array[i][img.shape[1]+1] = img[i-1][img.shape[1]-1]

    filter = copy.deepcopy(img)

    for i in range(1,array.shape[0]-1):
        for j in range(1,array.shape[1]-1):
                    temp = [array[i-1][j-1], array[i-1][j], array[i-1][j+1], array[i][j-1], array[i][j], array[i][j+1],
array[i+1][j-1], array[i+1][j], array[i+1][j+1]]
            filter[i-1][j-1] = max(temp)

    cv2.imshow("MAX FILTER", filter)
    cv2.waitKey(0)

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW3.tif",0)
```

```
Max_filter(img)
```

OUTPUT:

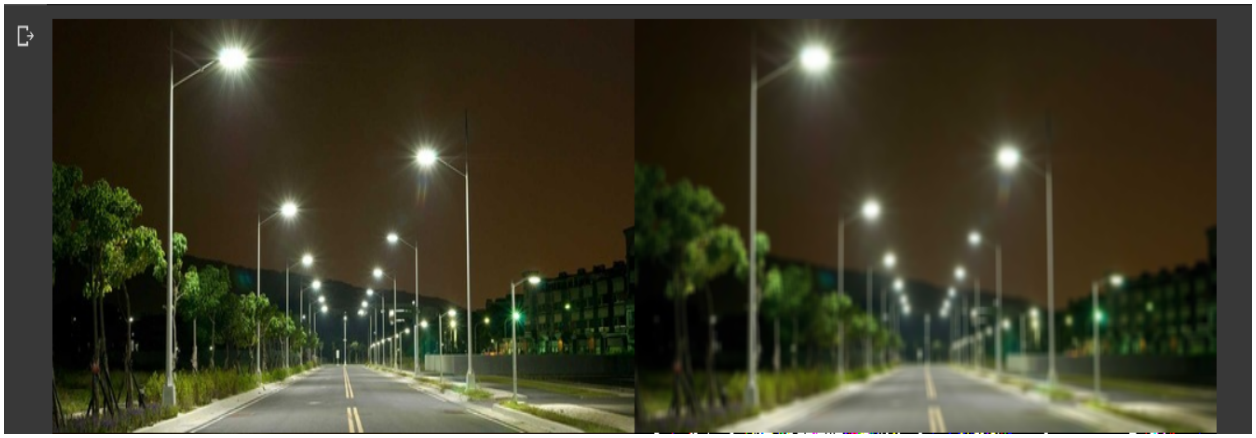


Frequency Domain Filters

High pass Filter:

```
img = cv2.imread('/content/point.jpeg',cv2.IMREAD_UNCHANGED)
gaussBlur = cv2.GaussianBlur(img,(5,5),cv2.BORDER_DEFAULT)
cv2_imshow(np.hstack((img,gaussBlur)))

highPass = img - gaussBlur
cv2_imshow(np.hstack((img, highPass)))
```

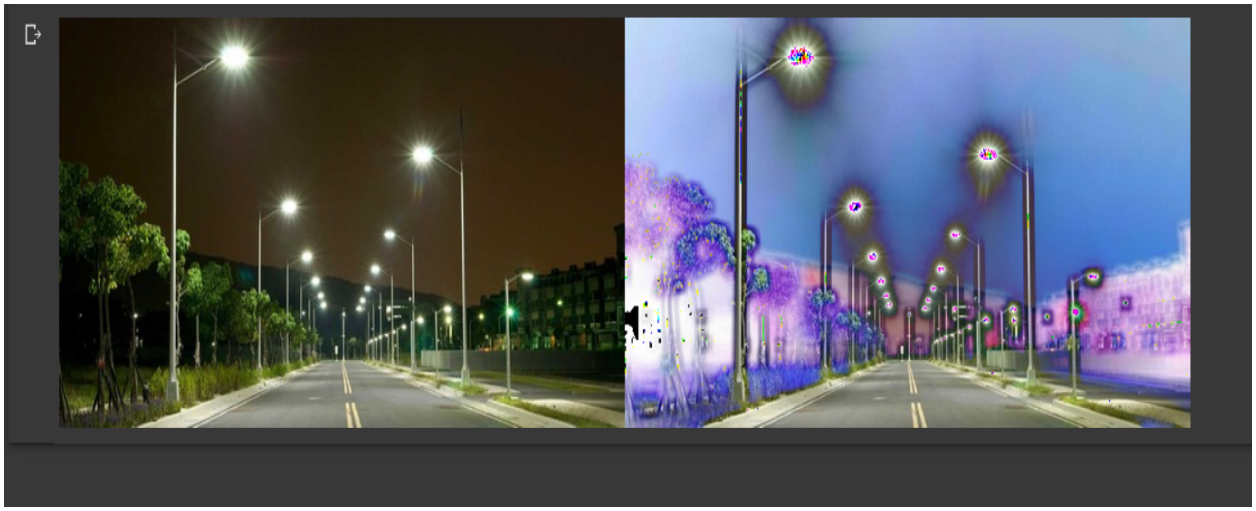
OUTPUT:



Low-Pass filter:

```
kernel = np.ones((10,10),np.float32)/25
lowPass = cv2.filter2D(img,-1, kernel)
lowPass = img - lowPass
cv2_imshow(np.hstack((img, lowPass)))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

OUTPUT:



Noise Models

Salt & Pepper Noise:

```
def s_p_noise(img,prob):
  res = img.copy()
  threshold = 1 - prob

  for i in range(img.shape[0]):
    for j in range(img.shape[1]):
      r = np.random.rand()
      if r < prob:
        res[i][j] = 0
      elif r > threshold:
        res[i][j] = 255
  # cv2_imshow(res)
  return res

img = cv2.imread('/content/point1.jpg')
s_p = s_p_noise(img,0.1)
Hori = np.concatenate((img,s_p), axis=1)
cv2_imshow(Hori)
```

Output:

Gaussian Noise:

```
#gaussian noise
def gaussian_noise(img,mean,var):
  sigma = var ** 0.5
  g = np.random.normal(mean,sigma,img.shape)
  nois_img = img + g
  Hori = np.concatenate((img,nois_img), axis=1)
  cv2_imshow(Hori)
img = cv2.imread('/content/point1.jpg',0)
gaussian_noise(img,2,98)
```

OUTPUT:

Exponential Noise:

```
#exponential noise
def exp_noise(img,s):
  rayl = np.random.rayleigh(s,img.shape)
  Hori = np.concatenate((img,img + rayl), axis=1)
  cv2_imshow(Hori)

img = cv2.imread('/content/point1.jpg',0)
exp_noise(img,30)
```

OUTPUT:

Rayleigh Noise:

```
#rayleigh noise
def rayl_noise(img,s):
  rayl = np.random.rayleigh(s,img.shape)
  Hori = np.concatenate((img,img + rayl), axis=1)
  cv2_imshow(Hori)

img = cv2.imread('/content/point1.jpg',0)
rayl_noise(img,25)
```

OUTPUT:

Image Compression

Image compression is a type of data compression applied to digital images, to reduce their cost for storage or transmission. Algorithms may take advantage of visual perception and the statistical properties of image to compress. Image compression may be lossy or lossless. Lossy compression that produces negligible differences may be called visually lossless.

Huffman Encoding:

Huffman coding is a lossless data compression technique. Huffman coding is based on the frequency of occurrence of a data item i.e. pixel in images. The technique is to use a lower number of bits to encode the data into binary codes that occur more frequently.

```
import numpy as np
import cv2

class PriorityQueue:

    def __init__(self, value, elements):
        self.value = value
        self.elements = elements
        self.left = None
        self.right = None

    def put(self, value, elements):
        if self.value >= value:
            if self.left == None:
                self.left = PriorityQueue(value, elements)
            else:
                self.left.put(value, elements)
        else:
            if self.right == None:
```

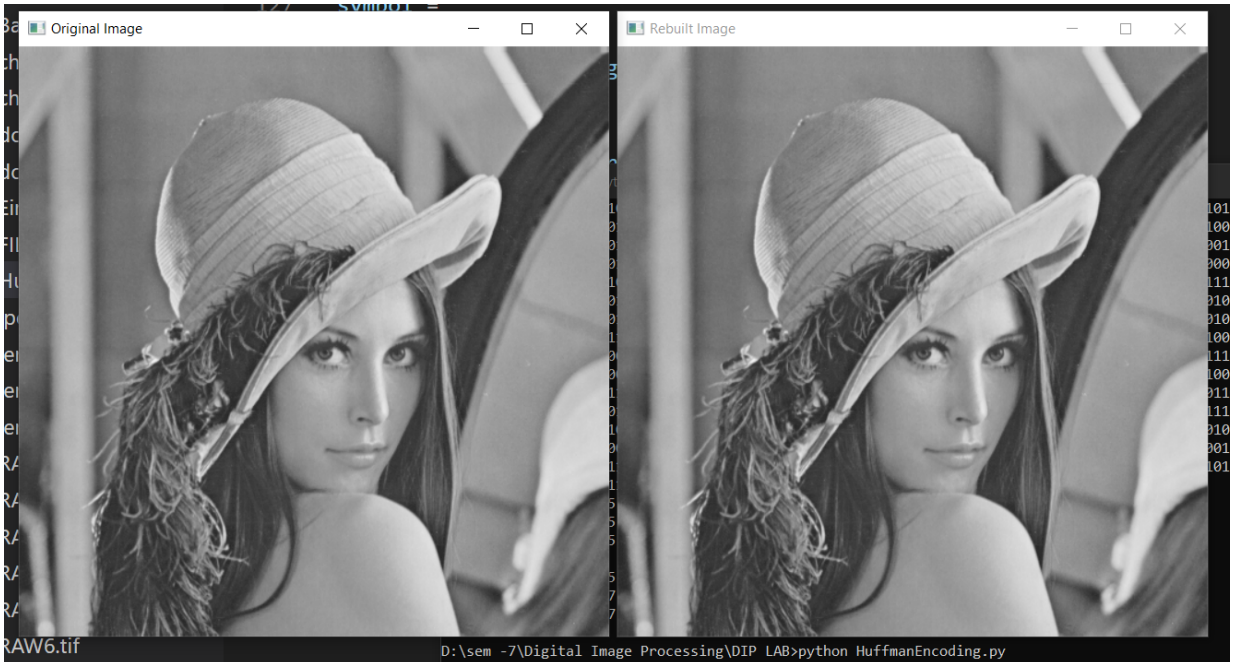
```
            self.right = PriorityQueue(value, elements)
        else:
            self.right.put(value, elements)

    def get(self):
        if self.left == None and self.right == None:
            ret = [self.value, self.elements,1]
            return ret

        elif self.left == None:
            ret = [self.value, self.elements,2]
            return ret

        else:
            obj = self.left.get()
            if obj[2] == 1:
                self.left = None
            if obj[2] == 2:
                self.left = self.left.right
            obj[2] = 0
            return obj

img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/lenna.png",0)
print(img)
prob = {}
symbol_table = {}
reverse_symbol_table = {}

#count occurrence of element in image array
for row in img:
    for element in row:
        if element in prob:
            prob[element] += 1
        else:
            prob[element] = 1
            symbol_table[element] ="
```

```
#enter values in priority_queue
q = None
for key,value in prob.items():
    if q == None:
        q = PriorityQueue(value, {key})
    else:
        q.put(value, {key})


#create huffman tree and assign values in reverse
while 1:
    item_a = q.get()
    if item_a[2] == 2:
        q = q.right

    #No more elements left
    if item_a[2] == 1:
        break

    item_b = q.get()
    if item_b[2] == 2:
        q = q.right

    for element in item_a[1]:
        symbol_table[element]+='1'

    for element in item_b[1]:
        symbol_table[element]+='0'

    #No more elements left
    if item_b[2] == 1:
        break

    q.put(item_a[0] + item_b[0], item_a[1].union(item_b[1]))

#reverse the assigned valued to get in correct format
for key,value in symbol_table.items():
```

```
    symbol_table[key] = value[::-1]

#calculating number of bits required per pixel
sum = 0
mx = 0
print("\nSymbol table")
for key,value in symbol_table.items():
    print(key, value)
    reverse_symbol_table[value] = key
    sum += prob[key] * len(value)
    if mx < len(value):
        mx = len(value)
print("\nBits per pixel")
print(sum/(len(img)*len(img[0])))
print(mx)
#encode
image_string = ''
for row in img:
    for element in row:
        image_string += symbol_table[element]

print("Encoded length ", len(image_string))
print(image_string)
#decode
image_array = np.zeros( len(img)* len(img[0]), dtype = np.uint8)
i = 0
symbol = ''

for ch in image_string:
    symbol += ch
     if symbol in reverse_symbol_table:
        image_array[i] = reverse_symbol_table[symbol]
        symbol = ''
        i += 1

    if len(symbol) > mx:
        print(i)
```

```
        break

image_array = image_array.reshape(len(img), len(img[0]))
cv2.imshow("Rebuilt Image",image_array)
cv2.waitKey(0)
print(image_array)
```

OUTPUT:



Run Length Encoding:

Run-length encoding (RLE) is a form of lossless data compression in which runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most efficient on data that contains many such runs.

```
import numpy as np
import cv2
```

```
img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/lenna.png",0)
print(img)

#run length encoding
img_string = ''

for line_no in range(len(img)):
    new_line = 0
    count = 0
    previous_pixel = 0
    for idx in range(len(img[0])):
        if new_line == 0:
            new_line =1
            previous_pixel = img[line_no][idx]
            count = 1

        elif previous_pixel != img[line_no][idx]:
            img_string += '('+ str(previous_pixel) + ',' + str(count) +')'
            count = 1
            previous_pixel = img[line_no][idx]

        else:
            count += 1

    img_string += '('+ str(previous_pixel) + ',' + str(count) +')'

print("length of image string", len(img_string))



#decode
image_array = np.zeros( len(img)* len(img[0]), dtype = np.uint8)
i = 0
datum = ''
for ch in img_string:

    if ch == ')':
```

```
        pixel = int(datum.split(',')[0])
        count = int(datum.split(',')[1])
        datum = ''

        for x in range(count):
            image_array[i] = pixel
            i+=1

    elif ch != '(':
        datum += ch

image_array = image_array.reshape(len(img), len(img[0]))
cv2.imshow("Rebuilt Image",image_array)
cv2.waitKey(0)
print(image_array)
```

OUTPUT:

Arithmetic Encoding:

An arithmetic coding algorithm encodes an entire file as a sequence of symbols into a single decimal number. The input symbols are processed one at each iteration. The initial interval [0, 1) (or [0, 1]) is successively divided into subintervals on each iteration according to the probability distribution. The subinterval that corresponds to the input symbol is selected for the next iteration. The interval derived at the end of this division process is used to decide the codeword for the entire sequence of symbols.

```
import numpy as np
import cv2

img = [[169, 169, 168, 175, 162,138], [169, 169, 168, 175, 162,138], [169, 169, 168, 175, 162,138]]

no_of_pixel = len(img[0]) *len(img)
prob = {}

#count occurrence of element in image array
for row in img:
    for element in row:
        if element in prob:
            prob[element] += 1
        else:
            prob[element] = 1

prev = 0
for key,val in prob.items():
    prob[key] = [prev, prev+(val/no_of_pixel)]
    prev += (val/no_of_pixel)


low = 0.0
high = 1.0

for row in img:
    for element in row:
```

```
        range = high - low

        high = low + range* prob[element][1]
        low = low + range* prob[element][0]

tag_val = (high+low)/2
# decode
length = len(img)*len(img[0])
image_array = np.zeros( len(img)* len(img[0]), dtype = np.uint8)
i = 0
while(length) :
    length -= 1
    for key, val in prob.items():

        if tag_val > val[0] and tag_val < val[1]:
            tag_val = (tag_val - val[0]) / (val[1] - val[0])
            image_array[i] = key
            i+=1
            break
image_array = image_array.reshape(len(img), len(img[0]))
print("original array")
print(img)
print("Rebuild Array")
print(image_array)
```

OUTPUT:

```
D:\sem -7\Digital Image Processing\DIP LAB>python ArithemeticEncoding.py
original array
[[169, 169, 168, 175, 162, 138], [169, 169, 168, 175, 162, 138], [169, 169, 168, 175, 162, 138]]
Rebuild Array
[[169 169 168 175 162 138]
 [169 169 168 175 162 138]
 [169 169 168 175 162 138]]
```

JPEG Compression

JPEG or **JPG** is a commonly used format for lossy compression for digital images, particularly for images produced by digital photography. The degree of compression can be adjusted, allowing a selectable trade-off between storage size and image quality. JPEG typically achieves 10:1 compression with little perceptible loss in image quality. Since its introduction in 1992, JPEG has been the most widely used image compression standard in the world

```
import cv2
import numpy as np
import copy
import scipy.fftpack

# reading gray image
img = cv2.imread ("D:/sem -7/Digital Image Processing/DIP LAB/RAW5.tif", 0)

image_array = np.zeros( [len(img), len(img[0])], dtype = int)

def zigzag(quantized):

    vector = np.zeros( [len(quantized)* len(quantized[0])], dtype = int )
    i = 0
    j = 0
    vector[0] = quantized[i][j]
    k=1
    j+=1
    while(k < len(quantized)* len(quantized[0])):

        while i < len(quantized) and j >= 0:
            vector[k] = quantized[i][j]
            k+=1
            j-=1
            i+=1

        j+=1
```

```
        i-=1

        if i == len(quantized) -1:
            j+=1
        else:
            i+=1

        while j < len(quantized[0]) and i >= 0:
            vector[k] = quantized[i][j]
            k+=1
            i-=1
            j+=1
        i+=1
        j-=1

        if j == len(quantized[0]) -1:
            i+=1
        else:
            j+=1

    return vector

def reversezigzag(vector):

    quantized= np.zeros( [8,8], dtype = int )
    i = 0
    j = 0
    quantized[i][j] = vector[0]
    k=1
    j+=1
    while(k < len(vector)):

        while i < len(quantized) and j >= 0:
            quantized[i][j] = vector[k]
            k+=1
            j-=1
            i+=1
```

```
        j+=1
        i-=1

        if i == len(quantized) -1:
            j+=1
        else:
            i+=1

        while j < len(quantized[0]) and i >= 0:
            quantized[i][j] = vector[k]
            k+=1
            i-=1
            j+=1
        i+=1
        j-=1

        if j == len(quantized[0]) -1:
            i+=1
        else:
            j+=1

    return quantized

#centralizing image pixels //img is of type uint8 so converted it to int first
image_array = img.astype(int) - 128

# quantization_Array = np.ones([8,8], dtype = int)
quantization_Array = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                               [12, 12, 14, 19, 26, 58, 60, 55],
                               [14, 13, 16, 24, 40, 57, 69, 56],
                               [14, 17, 22, 29, 51, 87, 80, 62],
                               [18, 22, 37, 56, 68, 109, 103, 77],
                               [24, 35, 55, 64, 81, 104, 113, 92],
                               [49, 64, 78, 87, 103, 121, 120, 101],
                               [72, 92, 95, 98, 112, 100, 103, 99]])
img_string = ''
```

```
# taking 8x8 blocks of img
for i in range(8,len(img)+1, 8):
    for j in range(8,len(img[0])+1, 8):

        # dct transformation (method for matrix)
        dct_transform = scipy.fftpack.dct( scipy.fftpack.dct( image_array[i-8:i, j-8:j].T, norm='ortho' ).T,
norm='ortho' )

        # quantization
        quantized = np.divide(dct_transform, quantization_Array)

        # rounding float numpy matrix (rounding gives better result than converting to int)
        quantized = np.round_(quantized)

        # zig zag traversal
        vector = zigzag(quantized)

        #run length encoding
        count = 1
        previous_pixel = vector[0]
        for idx in range(1,len(vector)):

            if previous_pixel != vector[idx]:
                img_string += '('+ str(previous_pixel) + ',' + str(count) +')'
                count = 1
                previous_pixel = vector[idx]
            else:
                count += 1
        img_string += '('+ str(previous_pixel) + ',' + str(count) +')'

# decode

# to store final image
decoded_img = np.zeros( [len(img), len(img[0])], dtype = int)
idx = 0 # to keep track of img_string
```

```
# decoding
for i in range(8,len(img)+1, 8):
    for j in range(8,len(img[0])+1, 8):

        # to open rle into 1 d vector
        vector = np.zeros( [64], dtype =int)
        cnt = 0
        datum = ''
        for ch in img_string[idx:]:
            idx += 1
            if ch == ')':
                pixel = int(datum.split(',')[0])
                count = int(datum.split(',')[1])
                datum = ''

                for x in range(count):
                    vector[cnt] = pixel
                    cnt +=1

            elif ch != '(':
                datum += ch

            # to break from for loop after taking 64 pixels
            if cnt == 64 :
                break

        # reading zigzag vector to 2d vector
        quantized = reversezigzag(vector)

        # multiplying with quantization array
        multiplied = np.multiply(quantized, quantization_Array)

        # applying inverse dct
        idct_transform = scipy.fftpack.idct( scipy.fftpack.idct(multiplied.T, norm='ortho' ).T, norm='ortho' )

        # storing idct result into original image as integer
        decoded_img[i-8:i, j-8:j] = idct_transform.astype(int)
```

```
# decentralizing array
decoded_img += 128

# converting to uint8 for display
decoded_img = decoded_img.astype(np.uint8)

cv2.imshow("Original Image",img)
cv2.imshow("Rebuilt Image",decoded_img)
cv2.waitKey(0)
```

OUTPUT: