PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE - 411043

Department of Electronics & Telecommunication Engineering

CLASS	**: B.E. E &TC**	**SUBJECT: DIVP**
EXPT. NO.	**: 1**	**DATE: 07/08/2020**

TITLE : PERFORM BASIC OPERATIONS ON AN IMAGE

CO 1:	Apply the fundamentals of digital image processing to perform various operations on an image-enhancement in spatial domain/ frequency domain, image-restoration, image compression, video filtering and video compression on a given gray image. Examine the effect of varying the mask size and density of noise in an image and comment on the obtained results.
CO4:	Carry out experiments as an individual and in a team, comprehend and write a laboratory record and draw conclusions at a technical level.

AIM:

To implement:

1. Basic logical operations on an image (AND, XOR, OR)
2. Basic arithmetic operations on an image (Add, Subtract, Multiply, Divide)
3. Negative of an image
4. Log transformation
5. Power Law (Gamma Transformation)
6. Contrast stretching
7. Bit Plane slicing
8. Gray level slicing

SOFTWARES REQUIRED: Matlab 7.0 or above.

THEORY:

1.1 Image Enhancement.

The principal objective of enhancement is to process an image so that the result is more suitable than the original image for a specific application. Image enhancement approaches fall into two broad categories: spatial domain methods and frequency domain methods. The term spatial domain refers to the image plane itself, and approaches in this category are based on direct manipulation of pixels in an image. Frequency domain processing techniques are based on modifying the Fourier transform of an image.

1.1.1 Spatial Domain Transformations.

The term spatial domain refers to the aggregate of pixels composing an image. Spatial domain methods are procedures that operate directly on these pixels. Spatial domain processes will be denoted by the expression:

$$\mathbf{g(x, y) = T[f(x, y)]}$$

where f(x, y) is the input image, g(x, y) is the processed image, and T is an operator on f, defined over some neighborhood of (x, y). In addition, T can operate on a set of input images.

The principal approach in defining a neighborhood about a point (x, y) is to use a square or rectangular sub image area centered at (x, y), as Fig. 1.1 shows. The center of the sub image is moved from pixel to pixel starting, say, at the top left corner. The operator T is applied at each location (x, y) to yield the output, g at that location. The process utilizes only the pixels in the area of the image spanned by the neighborhood. Although other neighborhood shapes, such as approximations to a circle, sometimes are used, square and rectangular arrays are by far the most predominant because of their ease of implementation.

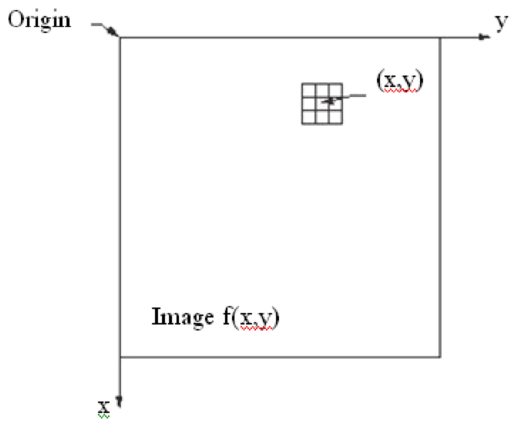


Fig 1.1: A 3x3 neighborhood about a point in an image f(x,y)

The simplest form of T is when the neighborhood is of size 1*1 (that is, a Single pixel). In this case, g depends only on the value of f at (x, y), and T becomes a gray-level (also called intensity or mapping) transformation function of the form:

$$s = T(r)$$

Where, for simplicity in notation, r and s are variables denoting, respectively, the gray level of f(x, y) and g(x, y) at any point (x, y).

1.2 Basic Gray Level Transformations:

1.2.1 Image Negative:

The negative of an image with gray levels in the range [0,L-1]is obtained by using the negative transformation, which is given by the expression, $\mathbf{s = L - 1 - r}$

Reversing the intensity levels of an image in this manner produces the equivalent of a photographic negative. This type of processing is particularly suited for enhancing white or gray detail embedded in dark regions of an image, especially when the black areas are dominant in size.

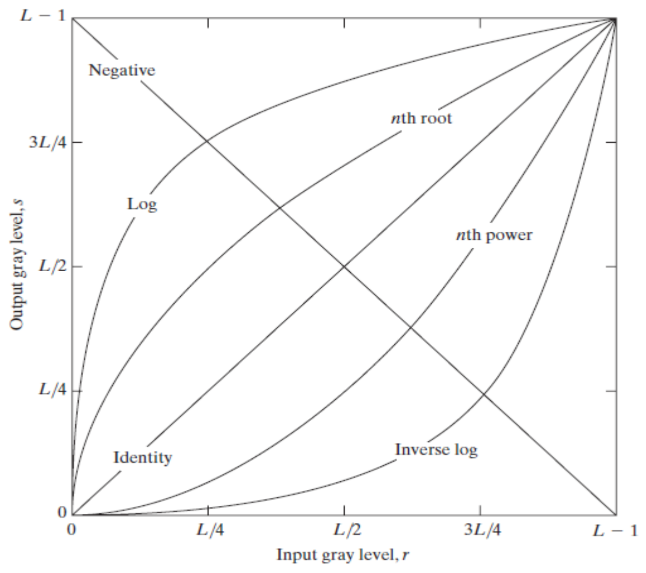


Fig 1.2: Some Basic Gray level Transformations

1.2.2 Log Transformation:

The general form of the log transformation shown in Fig. 1.2 is

$$S = c \log (1+r)$$

where c is a constant, and it is assumed that r >= 0.The shape of the log curve in Fig. 1.2 shows that this transformation maps a narrow range of low gray-level values in the input image into a wider range of output levels. The opposite is true of higher values of input levels. We would use a transformation of this type to expand the values of dark pixels in an image while compressing the higher-level values. The opposite is true of the inverse log transformation. Any curve having the general shape of the log functions shown in Fig.1.2 would accomplish this spreading/compressing of gray levels in an image. In fact, the power-law transformations discussed next are much more versatile for this purpose than the log transformation. However, the log function has the important characteristic that it compresses the dynamic range of images with large variations in pixel values. A classic illustration of an application in which pixel values have a large dynamic range is the Fourier spectrum. At the moment, we are concerned only with the image characteristics of spectra. It is not unusual to encounter spectrum values that range from 0 to or higher .While processing numbers such as these presents no problems for a computer, image display systems generally will not be able to reproduce faithfully such a wide range of intensity values. The net effect is that a significant degree of detail will be lost in the display of a typical Fourier spectrum. In such scenarios, log transformation is used.

1.2.3 Power - Law (Gamma) Transformation:

Power-law transformations have the basic form

$$s=cr^{\gamma}$$

Plots of s versus r for various values of γ are shown in Fig. 1.3 on the next page. Here c and γ are positive constants. As in the case of the log transformation, power-law curves with fractional values of gamma map a narrow range of dark input values into a wider range of output values, with the opposite being true for higher values of input levels. Unlike the log function, however, we notice here a family of possible transformation curves obtained simply by varying γ. As expected,we see in Fig. 1.3 that curves generated with values of γ >1 have exactly the opposite effect as those generated with values of γ<1. Finally, we note that eq. $s=cr^{\gamma}$ reduces to the identity transformation when c= γ =1. A variety of devices used for image capture, printing, and display respond according to a power law. By convention, the exponent in the power-law equation is referred to as gamma. The process used to correct the power-law response phenomena is called gamma correction. Gamma correction is important if displaying an image accurately on a computer screen is of concern.

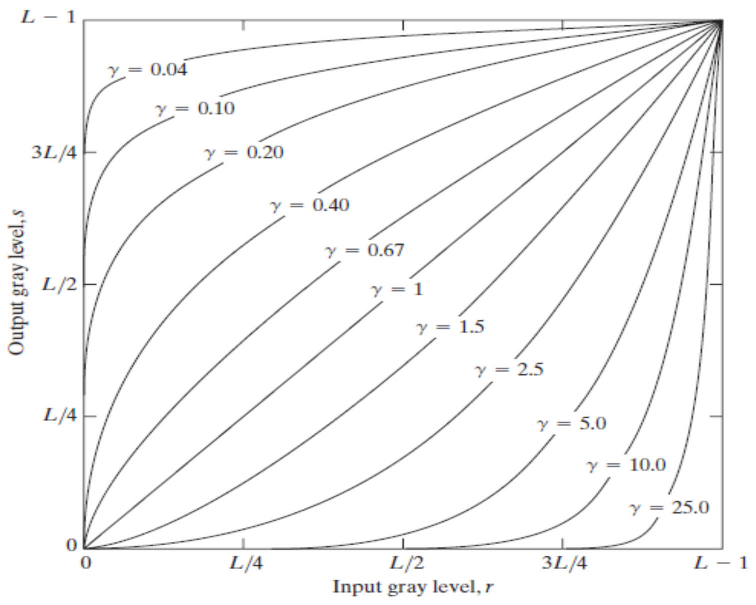


Fig 1.3: Plots of the curve $s=cr^{\gamma}$ **for various values of** γ **and c=1.**

Images that are not corrected properly can look either bleached out, or, what is more likely, too dark. Trying to reproduce colors accurately also requires some knowledge of gamma correction because varying the value of gamma correction changes not only the brightness, but also the ratios of red to green to blue.

1.2.4 Arithmetic and Logical Operations:

Logic operations operate on a pixel-by-pixel basis. We need only be concerned with the ability to implement the AND, OR, and NOT logic operators because these three operators are functionally complete. In other words, any other logic operator can be implemented by using only these three basic functions. When dealing with logic operations on gray-scale images, pixel values are processed as strings of binary numbers. For example, performing the NOT operation on a black, 8-bit pixel (a string of eight 0’s) produces a white pixel (a string of eight 1’s).

Intermediate values are processed the same way, changing all 1's to 0's and vice versa. Thus, the NOT logic operator performs the same function as the negative transformation. The AND and OR operations are used for masking; that is, for selecting subimages in an image. In the AND and OR image masks, light represents a binary 1 and dark represents a binary 0. Masking sometimes is referred to as region of interest (ROI) processing. In terms of enhancement, masking is used primarily to isolate an area for processing. This is done to highlight that area and differentiate it from the rest of the image. Logic operations also are used frequently in conjunction with morphological operations.

1.2.5 Gray-level Slicing:

Highlighting a specific range of gray levels in an image often is desired. Applications include enhancing features such as masses of water in satellite imagery and enhancing flaws in X-ray images. There are several ways of doing level slicing, but most of them are variations of two basic themes. One approach is to display a high value for all gray levels in the range of interest and a low value for all other gray levels. This transformation, shown in Fig. 1.4(a), produces a binary image.

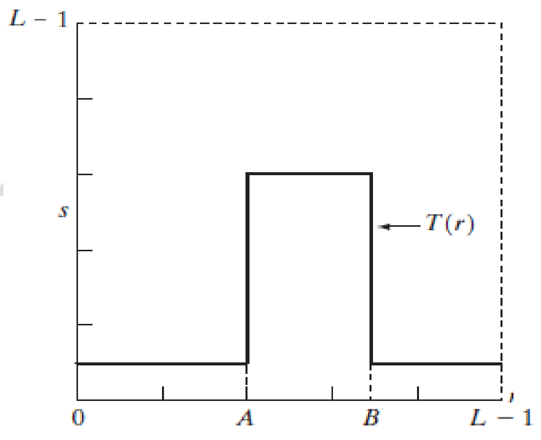


Fig 1.4(a)

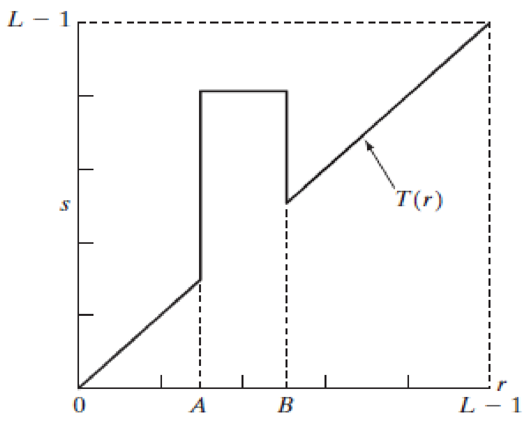


Fig 1.4(b)

The second approach, based on the transformation shown in Fig. 1.4(b), brightens the desired range of gray levels but preserves the background and gray-level tonalities in the image

1.2.6 Bit Plane Slicing:

Instead of highlighting gray-level ranges, highlighting the contribution made to total image appearance by specific bits might be desired. Suppose that each pixel in an image is represented by 8 bits. Imagine that the image is composed of eight 1-bit planes, ranging from bit-plane 0 for the leas`t significant bit to bit-plane 7 for the most significant bit. In terms of 8-bit bytes, plane 0 contains all the lowest order bits in the bytes comprising the pixels in the image and plane 7 contains all the high-order bits. Figure 1.5 illustrates these ideas. Note that the higher-order bits (especially the top four) contain the majority of the visually significant data. The other bit planes contribute to more subtle details in the image. Separating a digital image into its bit planes is useful for analyzing the relative importance played by each bit of the image, a process that aids in determining the adequacy of the number of bits used to quantize each pixel. Also, this type of decomposition is useful for image compression.

In terms of bit-plane extraction for an 8-bit image, it is not difficult to show that the (binary) image for bit-plane 7 can be obtained by processing the input image with a thresholding gray-level transformation function that (1) maps all levels in the image between 0 and 127 to one level (for example, 0); and (2) maps all levels between 129 and 255 to another (for example, 255).

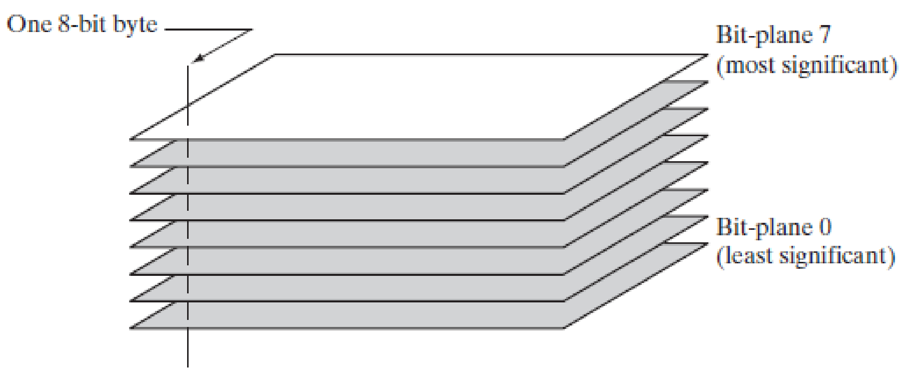


Figure 1.5: Bit Plane Representation of 8 bit Image.

1.2.7 Contrast Stretching:

One of the simplest piecewise linear functions (function whose pieces are linear) is a contrast stretching transformation. The form of the piecewise function can be arbitrary complex. A practical implementation of some important transformation can be formulated only as piecewise function. It increases the dynamic range of the gray levels in the image being processed. Low contrast images are resulted from poor illumination, lack of dynamic range in the imaging sensor, or even wrong setting of a lens aperture of image acquisition.

Transformation function is shown in Fig. 1.6. Locations of the points (r1, s1) and (r2, s2) used to control the shape of the transformation function. If r1=s1 and r2=s2 transformation is linear function and produces no changes in the gray level. If r1=r2, s1=0 and s2=L-1, transformation becomes a thresholding function that results a binary image. Intermediate values of (r1, s1) and (r2, s2) produce various degrees of spread in the gray levels of the output image, thus affecting its contrast.

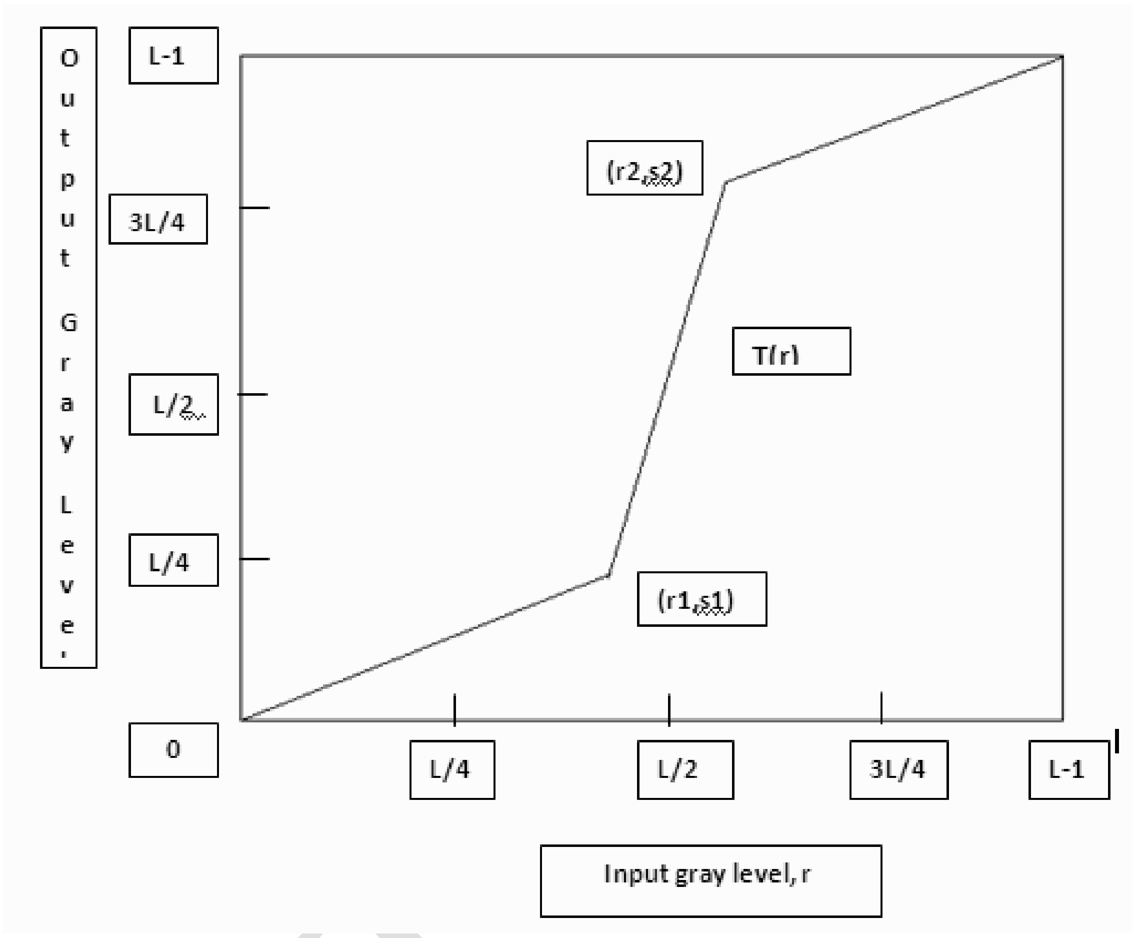


Fig 1.6: Contrast Stretching

1.3 Algorithms for Basic operations and gray transformations on Image:

1. Start
2. Take the input image from the user
3. Display the various operations
4. Ask the user for the operation to be performed
5. Display the histogram of the input image
6. Get the dimensions of the image
7. Pass the image and its dimensions to the respective functions

8. The function returns the new image
9. Display the histogram of the new image
10. Stop

1.4 Conclusion:

In this experiment, I have performed basic operations on the image such as: Simple arithmetic operations namely, Addition and Subtraction, Logical Operations namely, AND, OR and XOR. I have also performed Log transform, Power Law Transform, Negative, and Contrast Stretching on the image. In addition to the aforementioned, I have also performed Bit Plane Slicing and Grey Level Slicing on the image.

1.5 References:

i. Gonzalez R, Woods R, “Digital image processing”, Pearson Prentice Hall, 2008.

ii. Gonzalez R, Woods R, Steven E, “Digital Image Processing Using MATLAB®”, McGraw Hill Education,2010.

iii. Jayaraman S, Esakkirajan S and Veerakumar T,“Digital Image Processing” Tata McGraw Hill, 2010

iv. Joshi, Madhuri A. “Digital Image Processing: an algorithm approach”, PHI Learning Pvt. Ltd., 2006.

v. Pictures taken from: http://www.imageprocessingplace.com/root_files_V3/image_databases.html

(Course Teacher)

CLASS	**:**	**B.E (E &TC)**	**COURSE**	**:**	**DIVP**
AY	**:**	**2020-21 (SEM- I)**	**DATE**	**:**	**07/08/2020**
EXPT. NO.	**:**	**1**	**CLASS & ROLL NO**	**:**	**BE VIII 42410**
TITLE	**:**	**PERFORM BASIC OPERATIONS ON AN IMAGE**			

I. CODE:

1. Basic logical operations on an image (AND, XOR, OR)

2. Basic arithmetic operations on an image (Add, Subtract)
(COMBINED)

```
# -*- coding: utf-8 -*-
import numpy as np
import cv2
import matplotlib.pyplot as plt

#add, subtract
def arith_op(img1,img2):
   img3=cv2.add(img1,img2)
   img4=cv2.subtract(img1,img2)
   cv2.imshow('initial img1',img1)
   cv2.waitKey(50)
   cv2.imshow('initial img2',img2)
   cv2.waitKey(100)
   cv2.imshow('add result',img3)
   cv2.waitKey(150)
   cv2.imshow('sub result',img4)
   if cv2.waitKey(0) & 0xff == 27:
       cv2.destroyAllWindows()

#and, or, exor
def logic_op(img1,img2):
   img3=cv2.bitwise_and(img1,img2)
   img4=cv2.bitwise_or(img1,img2)
   img5=cv2.bitwise_xor(img1,img2)
   cv2.imshow('initial img1',img1)
   cv2.waitKey(500)
   cv2.imshow('initial img2',img2)
   cv2.waitKey(250)
   cv2.imshow('and result',img3)
   cv2.waitKey(100)
   cv2.imshow('or result',img4)
```

```
  cv2.waitKey(50)
  cv2.imshow('xor result',img5)
  cv2.waitKey(0)

def main():
   path1="C:/Users/Danish/Desktop/DIVP/bnw/bwsplit2.png"
   img1 = cv2.imread(path1,0)
   path2="C:/Users/Danish/Desktop/DIVP/bnw/bwsplit.png"
   img2 = cv2.imread(path2,0)
   arith_op(img1,img2)
   logic_op(img1,img2)
main()
```

3. Negative of an image

```
# -*- coding: utf-8 -*-
import cv2
import matplotlib.pyplot as plt

# Read the image
path="C:/Users/Danish/Desktop/DIVP/bnw/bwsplit2.png"
img = cv2.imread(path,0)
plt.imshow(img)
plt.show()

# Histogram plotting of original image
plt.hist(img.ravel(),255,[0,255])
plt.show()

# Negate the original image
img_neg = 1 - img
plt.imshow(img_neg)
plt.show()

# Histogram plotting of
# negative transformed image
plt.hist(img.ravel(),255,[0,255])
plt.show()
```

4. Log Transform

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
import matplotlib.pyplot as plt
def log_trans(img):
    #log transform
    c = 255 / (np.log(1 + np.max(img)))
    log_trans_img= c * np.log(1 + img)
    log_trans_img = np.array(log_trans_img,dtype = np.uint8)

    img_concat_horizontal=np.concatenate((img,log_trans_img),axis=1)
    cv2.imshow('concatenated_horizontal',img_concat_horizontal)
    cv2.waitKey(0)
    #histogram of original image
    hist_plot(img)
    #histogram of log tranformed image
    hist_plot(log_trans_img)

def hist_plot(img):
   #plot histogram
   plt.hist(img.ravel(),255,[0,255])
   plt.show()

def main():
   path="C:/Users/Danish/Desktop/DIVP/Images/humm.jpg"
   img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
   log_trans(img)
main()
```

5. Power Law (Gamma Transformation)

```
# -*- coding: utf-8 -*-
import numpy as np
import cv2
import matplotlib.pyplot as plt

def main():
    path="C:/Users/Danish/Desktop/DIVP/Images/crow.jpg"
    img = cv2.imread(path,0)

    for gamma in [0.1, 0.5, 1, 1.2, 2.2]:
       # Apply gamma correction.
       gamma_corrected = np.array(255*(img / 255) ** gamma, dtype = 'uint8')
       cv2.imshow('gamma transformed'+str(gamma), gamma_corrected)
       cv2.waitKey(0)
main()
```

6. Contrast Stretching

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
# read image
path="C:/Users/Danish/Desktop/DIVP/Images/eye2.jpg"
img = cv2.imread(path, 0)

# normalize float versions
norm_img1 = cv2.normalize(img, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX,
dtype=cv2.CV_32F)
norm_img2 = cv2.normalize(img, None, alpha=0, beta=1.2, norm_type=cv2.NORM_MINMAX,
dtype=cv2.CV_32F)

# scale to uint8
norm_img1 = (255*norm_img1).astype(np.uint8)
norm_img2 = np.clip(norm_img2, 0, 1)
norm_img2 = (255*norm_img2).astype(np.uint8)

# display input and both output images
cv2.imshow('original',img)
cv2.imshow('normalized1',norm_img1)
cv2.imshow('normalized2',norm_img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

7. Bit Plane Slicing

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
import matplotlib.pyplot as plt

def bit_plane_slicing(img):
    out = []
    for k in range(0, 7):
        #out = []
        # create an image for each k bit plane
        plane = np.full((img.shape[0], img.shape[1]), 2 ** k, np.uint8)
        # execute bitwise and operation
        res = cv2.bitwise_and(plane, img)
        # multiply ones (bit plane sliced) with 255 just for better visualization
        x = res * 255
        #out=out.append(x)
        plt.subplot(2,4,k+1)
        cv2.imshow("bit plane", x)
        cv2.waitKey()
def main():
  path="C:/Users/Danish/Desktop/DIVP/Images/flower.jpg"
  img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
  bit_plane_slicing(img)
main()
```

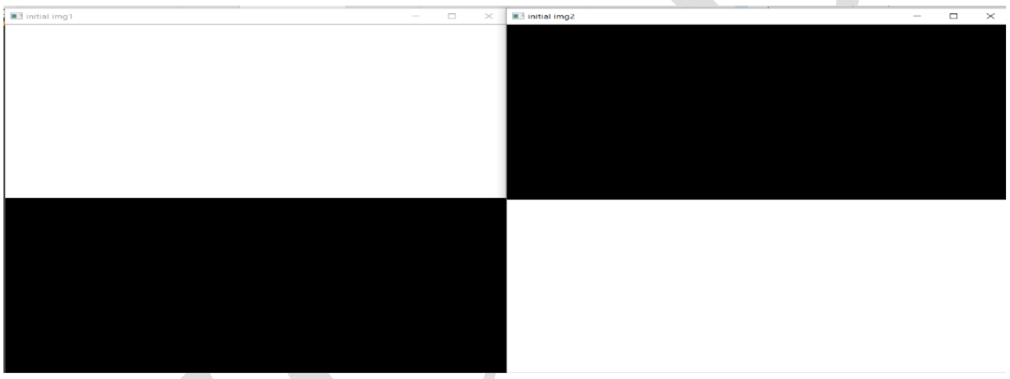
8. Gray Level Slicing

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
import matplotlib.pyplot as plt

def grey_level_slicing(img,min_r,max_r,wbg):
    # Find width and height of image
    row, column = img.shape
    # Create an zeros array to store the sliced image
    img1 = np.zeros((row,column),dtype = 'uint8')
    # Specify the min and max range
    min_range = min_r
    max_range = max_r

    #with background
    if(wbg == 1):
        for i in range(row):
            for j in range(column):
                if img[i,j]>min_range and img[i,j]<max_range:
                    #object of interest becomes white, background reamins as it is
                    img1[i,j] = 255
                else:
                    img1[i,j]=img[i,j]
        display_img(img, img1)
#without background
    elif(wbg == 2):
      # Loop over the input image and if pixel value lies in desired range set it to 255 otherwise set it to 0.
        for i in range(row):
            for j in range(column):
                if img[i,j]>min_range and img[i,j]<max_range:
                    #object of interest becomes white
                    img1[i,j] = 255
                else:
                    #background becomes black
                    img1[i,j] = 0
        display_img(img, img1)

    else:
        img2 = np.zeros((row,column),dtype = 'uint8')
        #with background
        for i in range(row):
            for j in range(column):
                if img[i,j]>min_range and img[i,j]<max_range:
                    #object of interest becomes white, background reamins as it is
                    img1[i,j] = 255
                else:
                    img1[i,j]=img[i,j]
        #without background
        for i in range(row):
            for j in range(column):
                if img[i,j]>min_range and img[i,j]<max_range:
                    #object of interest becomes white
                    img2[i,j] = 255
                else:
                    #background becomes black
                    img2[i,j] = 0
        display_img2(img, img1, img2)

def display_img(img,img1):
    # Display the image
    cv2.imshow('original image',img)
    cv2.waitKey(500)
    hist_plot(img)
    cv2.imshow('sliced image', img1)
    cv2.waitKey(0)
    hist_plot(img1)

def display_img2(img,img1,img2):
    # Display the image
    cv2.imshow('original image',img)
    cv2.waitKey(1000)
    hist_plot(img)
    cv2.imshow('with background', img1)
    cv2.waitKey(500)
    hist_plot(img1)
    cv2.imshow('without background', img2)
    cv2.waitKey(0)
    hist_plot(img2)

def hist_plot(img):
    #plot histogram
    plt.hist(img.ravel(),255,[0,255])
    plt.show()

def main():
    path="C:/Users/Danish/Desktop/DIVP/Images/flower.jpg"
    img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)

    hist_plot(img)
    range1=int(input("Enter the lower intensity range: "))
    range2=int(input("Enter the higher intensity range: "))
    back=int(input("Gray Level Slicing methods:\n1. With Background\n2. Without background\n3.
Both\n\nEnter your choice: "))
    grey_level_slicing(img,range1,range2,back)
main()
```

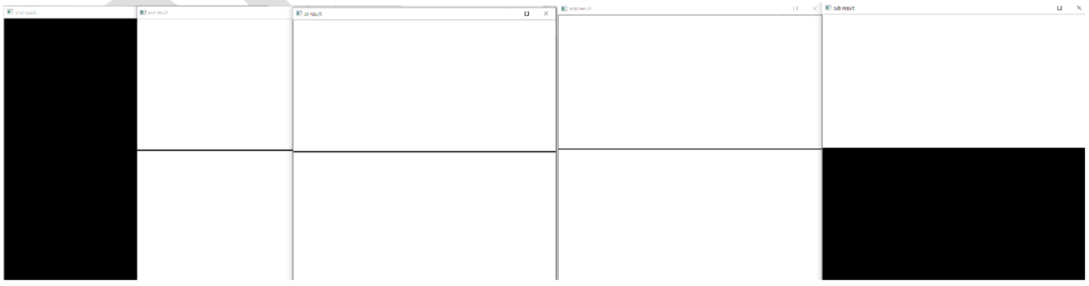
II. RESULTS:

1. Basic logical operations on an image (AND, XOR, OR)
2. Basic arithmetic operations on an image (Add, Subtract) (COMBINED)

INPUT IMAGE 1 INPUT IMAGE 2

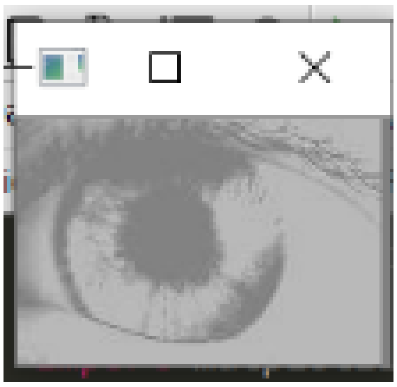


AND OR EXOR ADD SUB

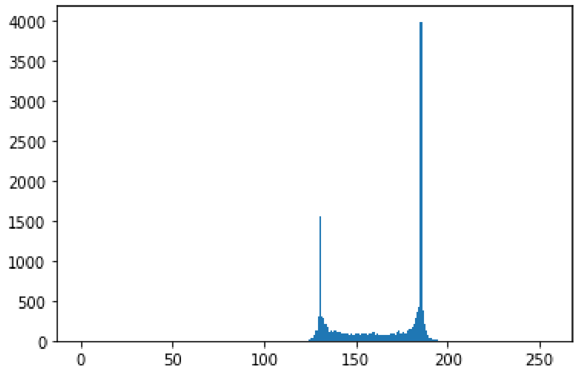


3. Negative of an image

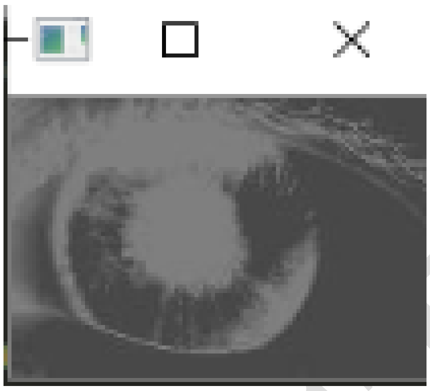
INPUT IMAGE



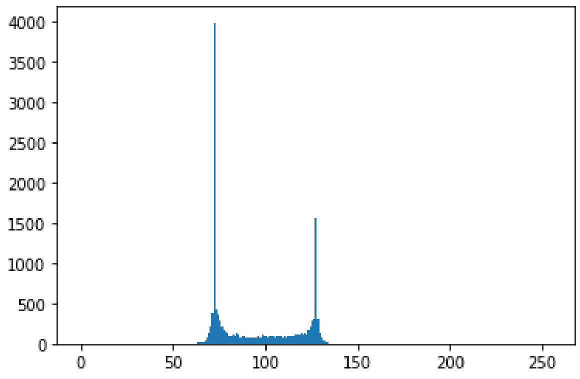
HISTOGRAM



OUTPUT IMAGE



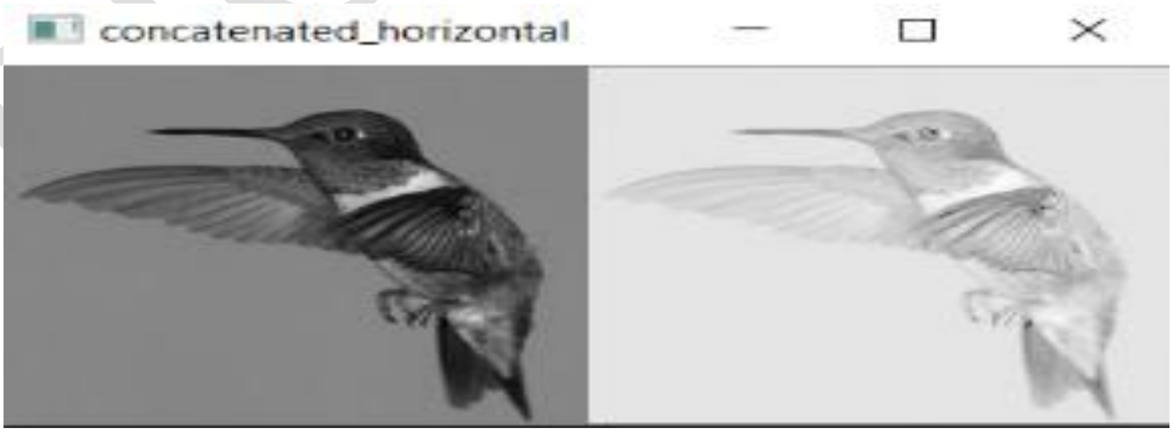
HISTOGRAM



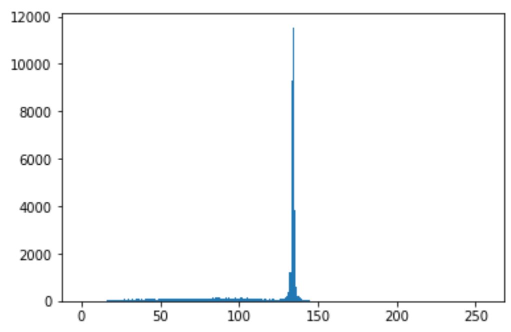
4. Log transformation

INPUT IMAGE

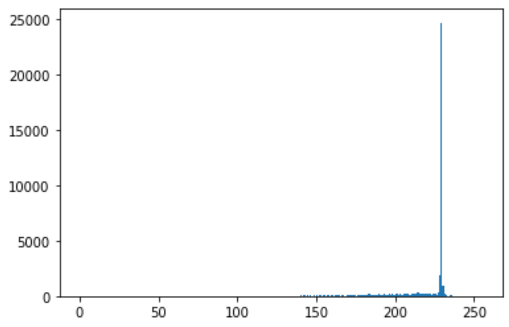
OUTPUT IMAGE



INPUT HISTOGRAM

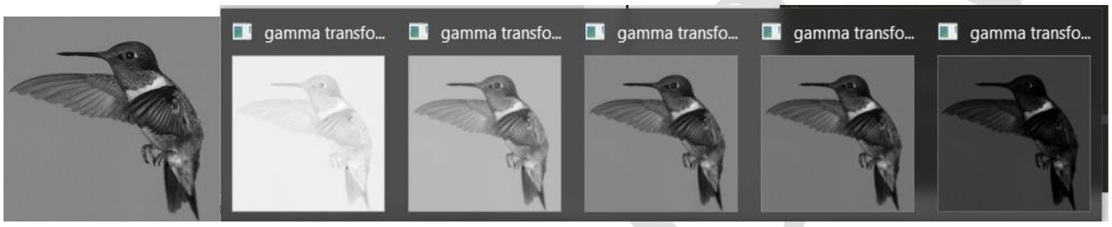


OUTPUT HITOGRAM



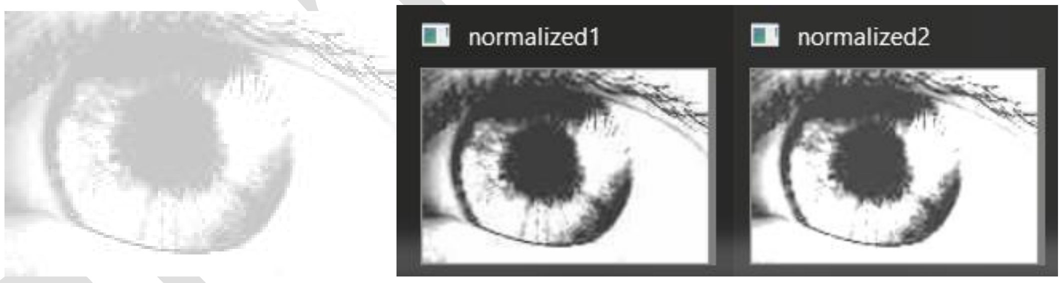
5. Power Law (Gamma Transformation)

Original Gamma: 0.1 0.5 1 1.2 2.2

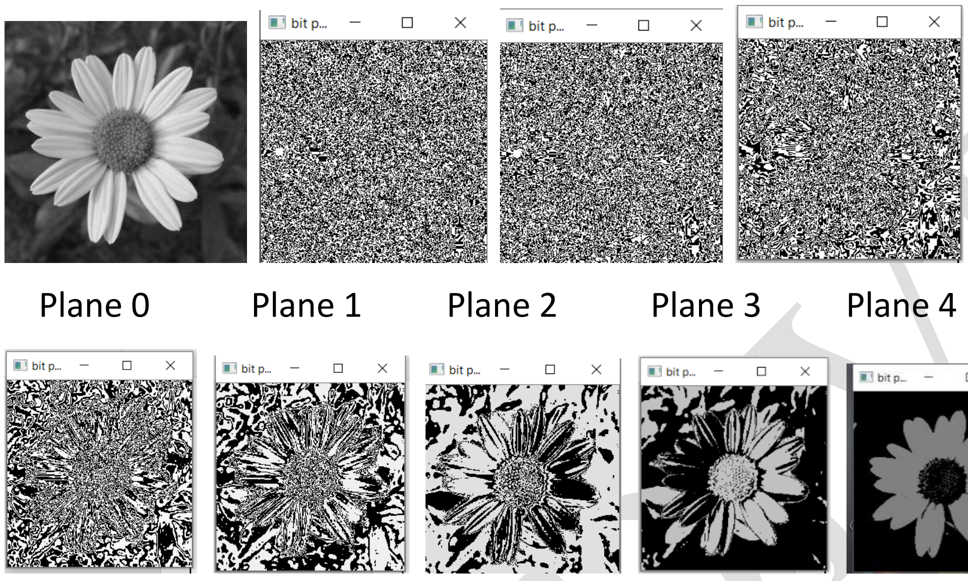


6. Contrast stretching

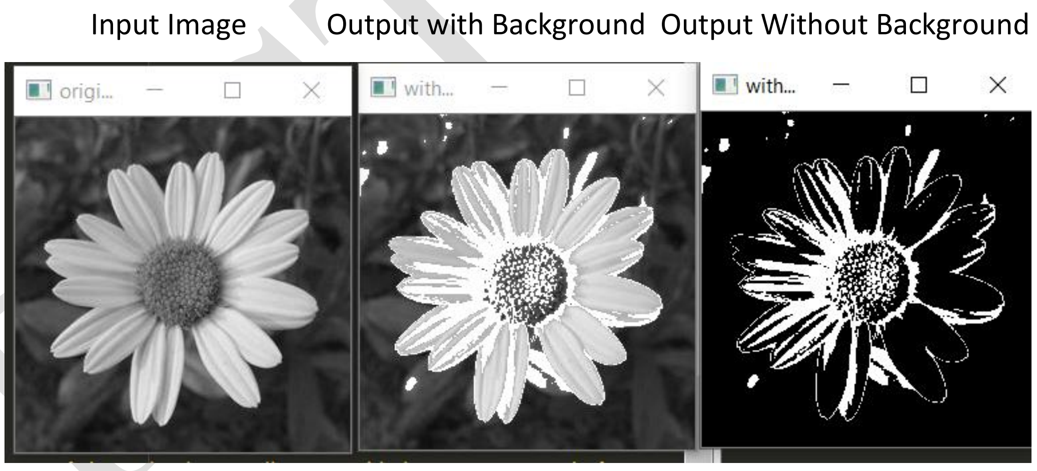
INPUT IMAGE Beta = 1 Beta = 1.2



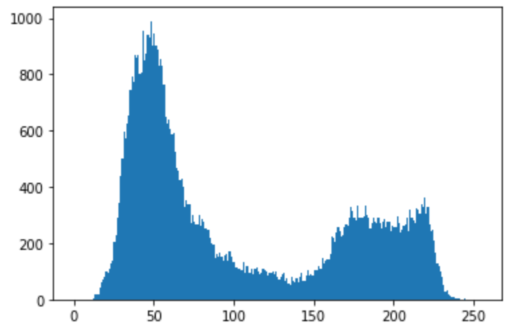
7. Bit Plane slicing



8. Gray level slicing



Original image histogram: Lower cut off at 100 and higher cutoff at 175


1000
800
600
400
200
0
0
50
100
150
200
250