1. **Write a program to read, save and display an image**

Background:

By using Opencv library we can use the function cv2.imread() to read an image, it takes the name of the image file or the URL as an argument.

To save an image using OpenCV library we can use the function cv2.imwrite() which takes the name of the output file and writes the image data to the specified file in the desired format, such as JPEG, PNG, BMP, etc.

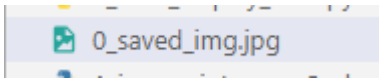
To display an image using OpenCV library, the function cv2.imshow() is used which takes the name of the window and shows the image in a graphical user interface window. We need the function cv2.waitKey() to specify how long the window should remain open and the function cv2.destroyAllWindows() to close all the windows.

Language: Python

Source Code:

```
import cv2
import numpy as np
img= cv2.imread('./Image_Processing/photo2.jpg')
image= cv2.resize(img,(400, 500))

outimage= cv2.imshow('MyImage',image)
cv2.imwrite('./Image_Processing/0_saved_img.jpg', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Conclusion:

In this lab work, we have learned how to convert a color (RGB) image into a grayscale image using OpenCV library. The process involves reading the original image using cv2.imread() function, saving the image using cv2.imwrite() function and displaying the image using cv2.imshow() function,

2. **Conversion of color image into gray-scale image.**

Background:

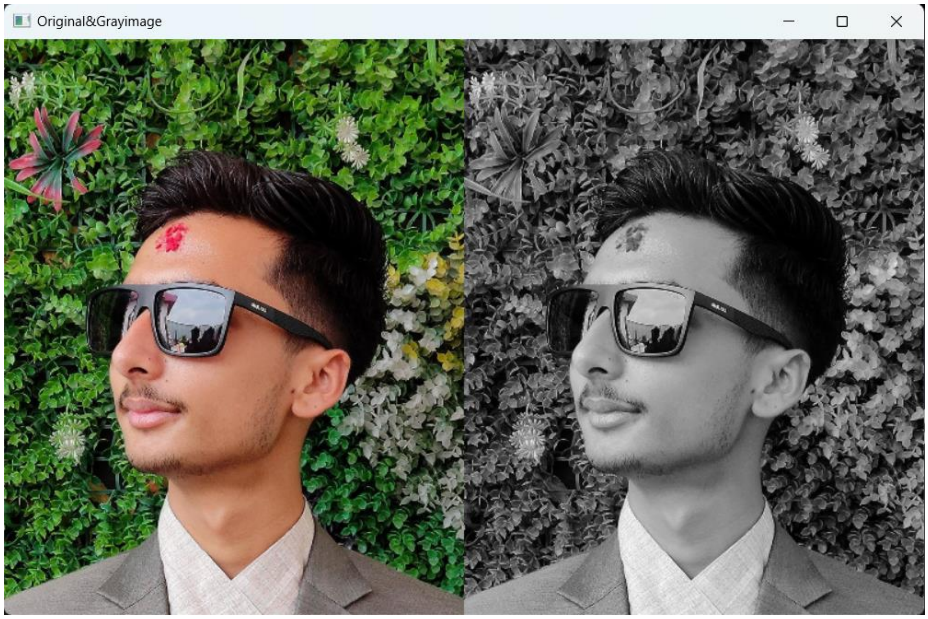
Grayscale images are digital images that have only shades of gray and no color. A grayscale image is created by converting a color image to black and white or by using a grayscale filter. Grayscale images are a kind of black-and-white or gray monochrome, are composed exclusively of shades of gray. The contrast ranges from black at the weakest intensity to white at the strongest.

Language: Python

Source Code:

```
import cv2
import numpy
import numpy as np

img= cv2.imread('./Image_Processing/photo2.jpg')
image= cv2.resize(img,(400, 500))

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Convertion Image into 3 channel for concatenation process
img2 = np.zeros_like(image)
img2[:,:,0] = gray_image
img2[:,:,1] = gray_image
img2[:,:,2] = gray_image

both_image = np.hstack([image, img2])
cv2.imshow('Original&Grayimage',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Conclusion:

In this lab work, we have learned how to convert a color (RGB) image into a grayscale image using OpenCV library. The process involves reading the original image using cv2.imread() function, converting it to grayscale using cv2.cvtColor() function with cv2.COLOR_BGR2GRAY flag, and displaying the grayscale image using cv2.imshow() function. In addition np.zeros_like() function is used to change the channel of image (i.e. 3 dimensional array)

3. Conversion of color image into black and white image

Background:

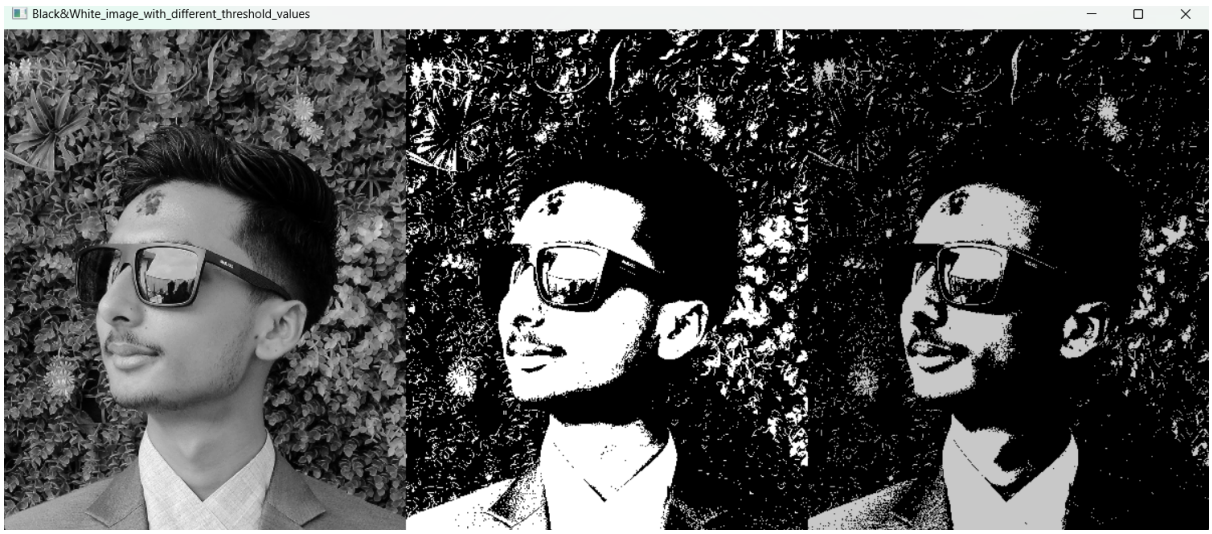
The image which consist of only black and white color is called BLACK AND WHITE IMAGE. The process of involves converting the color image into grayscale image and the grayscale image into binary, where each pixel is either black or white. Thresholding is the most common method for converting a grayscale image into a binary image.

Language: Python

Source Code:

```
import cv2
import numpy as np
img= cv2.imread('./Image_Processing/photo2.jpg')
img = cv2.resize(img,(400, 500))
image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
(thresh, bimage1) = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)
(thresh, bimage2) = cv2.threshold(image, 150, 200, cv2.THRESH_BINARY)
both_image = np.hstack([image, bimage1, bimage2])
cv2.imshow('Black&White_image_with_different_threshold_values',both_ima
ge)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Conclusion:

In this lab work, we have learned how to convert a color (RGB) image into a black and white image using OpenCV library. The process involves reading the original image using cv2.imread() function, then converting it to grayscale using cv2.cvtColor() function with cv2.COLOR_BGR2GRAY flag, and then converting grayscale image into black and white using cv2.threshold() function and displaying the grayscale image using cv2.imshow() function.

4. Conversion of an image into Digital Negative.

Background:

A digital negative refers to the inversion of an image, where each pixel's intensity is subtracted from the maximum intensity value. The negative of an image is achieved by replacing the intensity 'i' in the original image by 'i-1', i.e. the darkest pixels will become the brightest and the brightest pixels will become the darkest.

The transformation function used in creating a digital negative is typically expressed as:

$$s=T(r)=L-1-r$$

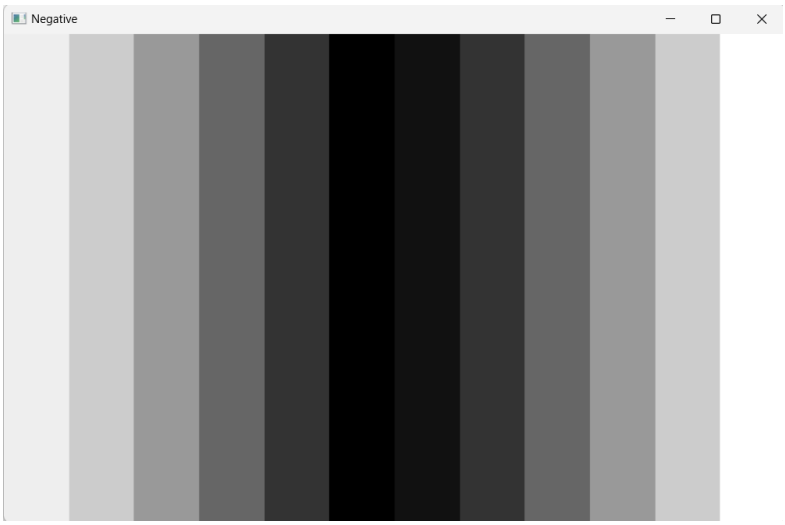
where L-1 is the maximum intensity value, s is the output pixel value, and r is the input pixel value

Language: Python

Source Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img= cv2.imread('./Image_Processing/gray.png')
image= cv2.resize(img,(400, 500))
# cv2.imshow('Original',image)
img_neg = 255-img
outimage= cv2.resize(img_neg,(400, 500))
both_image = np.hstack([image, outimage])
cv2.imshow('Negative',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



5. Histogram plot of an image.

Background:

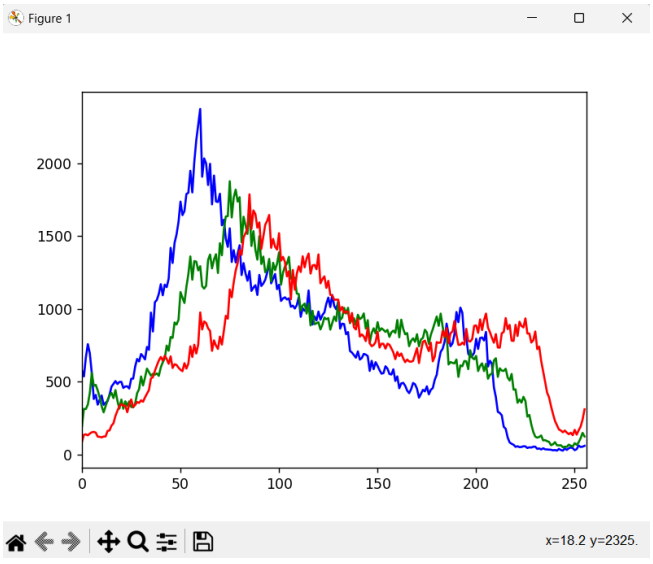
The histogram is used for graphical representation of a digital image. A graph is a plot by the number of pixels for each tonal value. The horizontal axis of the graph is used to represent tonal variations whereas the vertical axis is used to represent the number of pixels in that particular pixel. Black and dark areas are represented in the left side of the horizontal axis, medium grey color is represented in the middle, and the vertical axis represents the size of the area.

Language: Python

Source Code:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('./Image_Processing/meroimage.jpg')

# Calculate the histogram for each color channel
color = ('b', 'g', 'r')
for i, col in enumerate(color):
    histr = cv2.calcHist([img], [i], None, [256], [0, 256])
    plt.plot(histr, color=col)

# Set the range of the x-axis and display the plot
plt.xlim([0, 256])
plt.show()
```

Output:



6. Digital Negative with histogram.

Background:

The histogram is used for graphical representation of a digital image.An histogram transformation consists in applying a mathematical function to the intensity distribution. Generally, the transformations are used to improve the visual quality of an image, but are rarely needed inside an automatic processing. The transform "T" is applied on the pixel intensities to change their values:

$$j=T(i)$$

Language: Python

Source Code:

```
import cv2
import matplotlib.pyplot as plt
img_bgr = cv2.imread('./Image_Processing/meroimage.jpg',3)
plt.imshow(img_bgr)
plt.show()
# Histogram plotting of original image
color = ('b', 'g', 'r')
for i, col in enumerate(color):
    histr = cv2.calcHist([img_bgr],
                         [i], None,
                         [256],
                         [0, 256])
    plt.plot(histr, color = col)
    # Limit X - axis to 256
    plt.xlim([0, 256])
plt.show()
cv2.waitKey(0)
cv2.destroyAllWindows()
# Negate the original image
img_neg = 1 - img_bgr
plt.imshow(img_neg)
plt.show()
# Histogram plotting of
# negative transformed image
color = ('b', 'g', 'r')
for i, col in enumerate(color):

    histr = cv2.calcHist([img_neg],
                         [i], None,
                         [256],
                         [0, 256])
    plt.plot(histr, color = col)
    plt.xlim([0, 256])
plt.show()
```

Output:

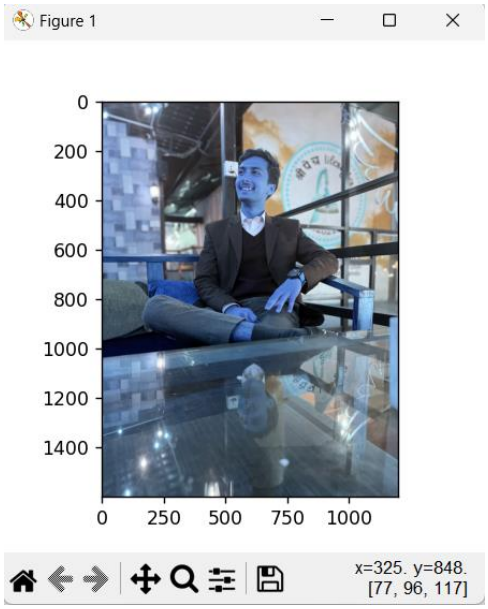


Figure 1
0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
x=325. y=848.
[77, 96, 117]

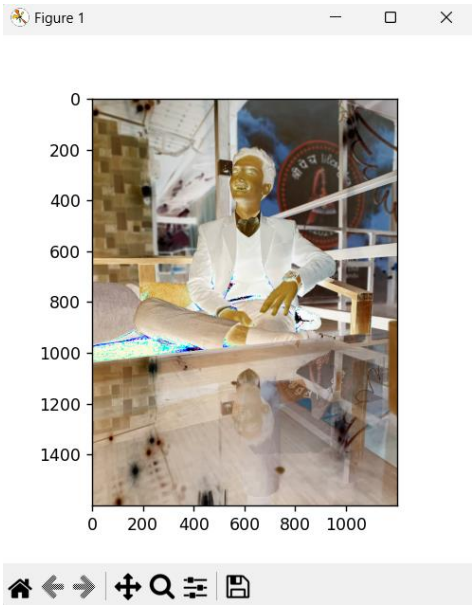


Figure 1
0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000

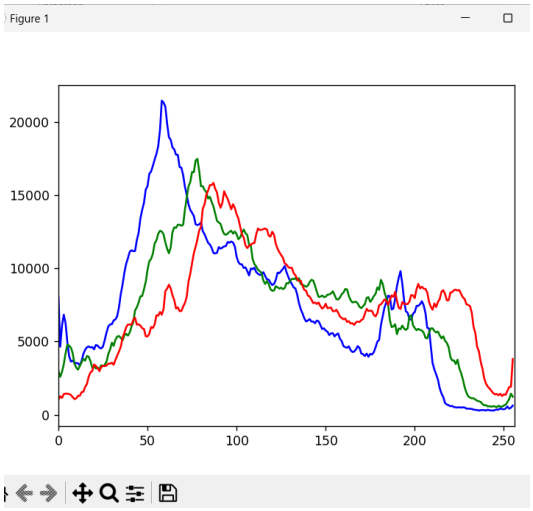


Figure 1
20000
15000
10000
5000
0
0
50
100
150
200
250

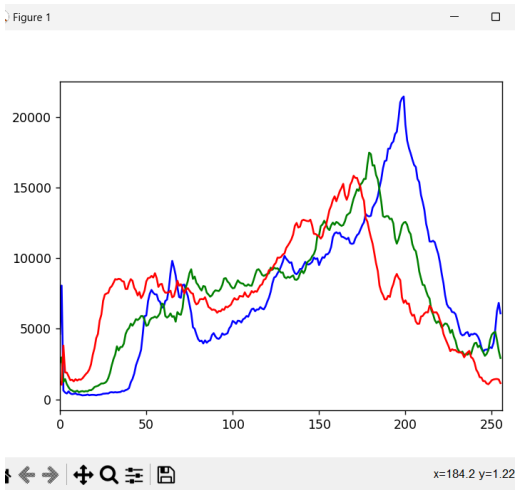


Figure 1
20000
15000
10000
5000
0
0
50
100
150
200
250
x=184.2 y=1.22

7. **Histogram equilization.**

Background:

Histogram equalization is a technique that redistributes the pixel intensities within an image to achieve a flatter or more uniform histogram. This essentially means spreading out the pixel values, stretching the contrast, and making even subtle details readily visible. It is used to enhance contrast.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/Screen.jpg', 0)
img = cv2.resize(img,(400, 500))
# cv2.imshow('input',img)

equ = cv2.equalizeHist(img)
both_image = np.hstack([img, equ])
cv2.imshow('output',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

8. Piecewise linear transformation.

Background:

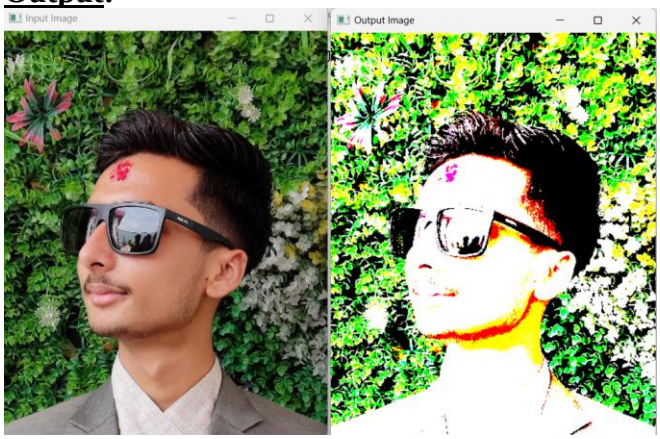
Piece-wise Linear Transformation is type of gray level transformation that is used for image enhancement. It is a spatial domain method. It is used for manipulation of an image so that the result is more suitable than the original for a specific application. Some commonly used piece-wise linear transformations are:Contrast Stretching, Clipping, Thresholding, Grey level slicing, Bit Extraction.

Language: Python

Source Code for contrast stretching:

```
import cv2
import numpy as np
def pixelVal(pix,r1,s1,r2,s2):
    if (0 <= pix and pix <= r1):
        return (s1/r1)*pix
    elif (r1 < pix and pix <= r2):
        return ((s2-s1)/(r2-r1)) * (pix-r1) + s1
    else:
        return ((255-s2)/(255-r2)) * (pix-r2) + s2
img = cv2.imread('./Image_Processing/photo2.jpg')
image= cv2.resize(img,(400, 500))
cv2.imshow('Input Image', image)
r1 = 70
s1 = 0
r2 = 255
s2 = 255
pixelVal_vec = np.vectorize(pixelVal)
cont_st_img = pixelVal_vec(img, r1, s1, r2, s2)
outimage= cv2.resize(cont_st_img,(400, 500))
cv2.imshow('Output Image', outimage)
cv2.waitKey(0)
```

Output:



9. Power Law Transformation.

Background:

power law transformation include nth power and nth root transformation. These transformations can be given by the expression: s=cr^γ

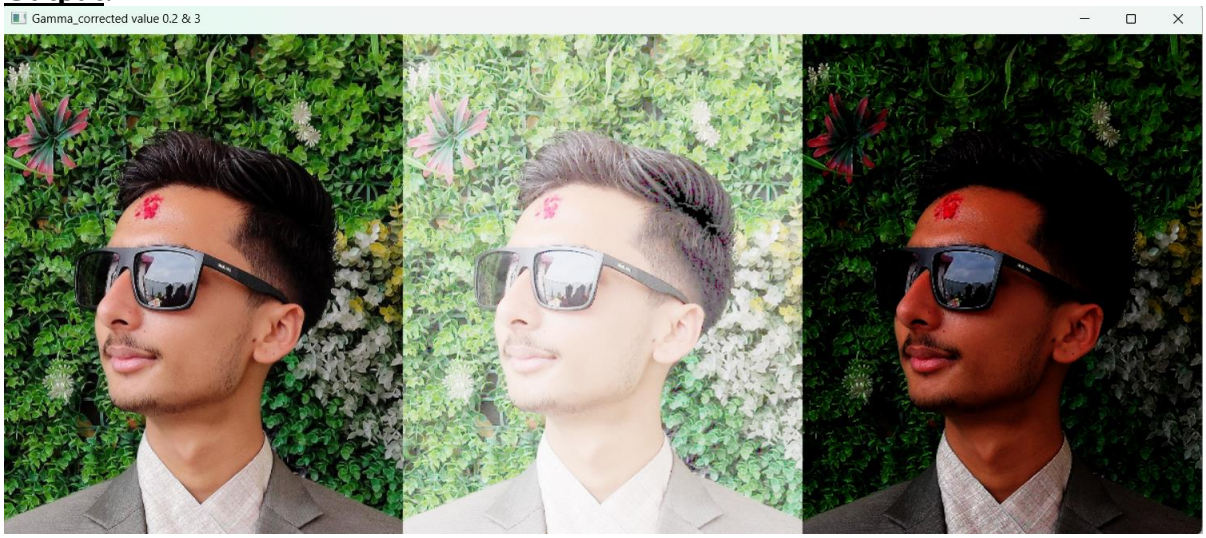
This symbol γ is called gamma, due to which this transformation is also known as gamma transformation.Variation in the value of γ varies the enhancement of the images. Different display devices / monitors have their own gamma correction, that's why display the image at different intensity in different device.This type of transformation is used for enhancing images for different type of display devices.

Language: Python

Source Code:

```
import cv2
import numpy as np
image= cv2.imread('./Image_processing/photo2.jpg')
image= cv2.resize(image,(400, 500))
# cv2.imshow('Original',image)
gamma1=0.2
gamma2= 3
gamma_corrected1 = np.array(255*(image / 255) ** gamma1, dtype =
'uint8')
gamma_corrected2 = np.array(255*(image / 255) ** gamma2, dtype =
'uint8')
both_image = np.hstack([image, gamma_corrected1, gamma_corrected2])
cv2.imshow('Gamma_corrected value 0.2 & 3',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



10. Log Transformation.

Background:

The log transformations can be defined by this formula: $s = c \log(r + 1)$.

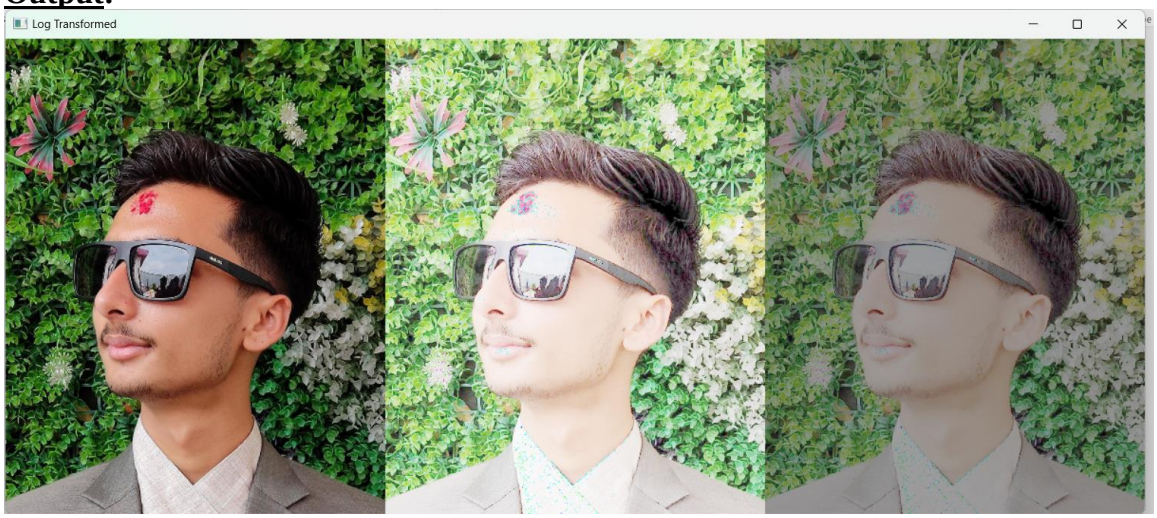
Where s and r are the pixel values of the output and the input image and c is a constant. The value 1 is added to each of the pixel value of the input image because if there is a pixel intensity of 0 in the image, then log (0) is equal to infinity. So 1 is added, to make the minimum value at least 1. During log transformation, the dark pixels in an image are expanded as compare to the higher pixel values. The higher pixel values are kind of compressed in log transformation. This result in following image enhancement.

Language: Python

Source Code:

```
import cv2
import numpy as np
img= cv2.imread('./Image_Processing/photo2.jpg')
image= cv2.resize(img,(400, 500))
# cv2.imshow('Original',image)
c1=(255/np.log(1+np.max(img)))
log_transformed1 = c1 * (np.log(img + 1))
c2=(160/np.log(1+np.max(img)))
log_transformed2 = c2 * (np.log(img + 1))
log_transformed1 = np.array(log_transformed1, dtype = np.uint8)
log_transformed2 = np.array(log_transformed2, dtype = np.uint8)
log_transformed1= cv2.resize(log_transformed1,(400, 500))
log_transformed2= cv2.resize(log_transformed2,(400, 500))
both_image = np.hstack([image, log_transformed1, log_transformed2])
cv2.imshow('Log Transformed',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



11. Implementation of weighted average filtering

Background:

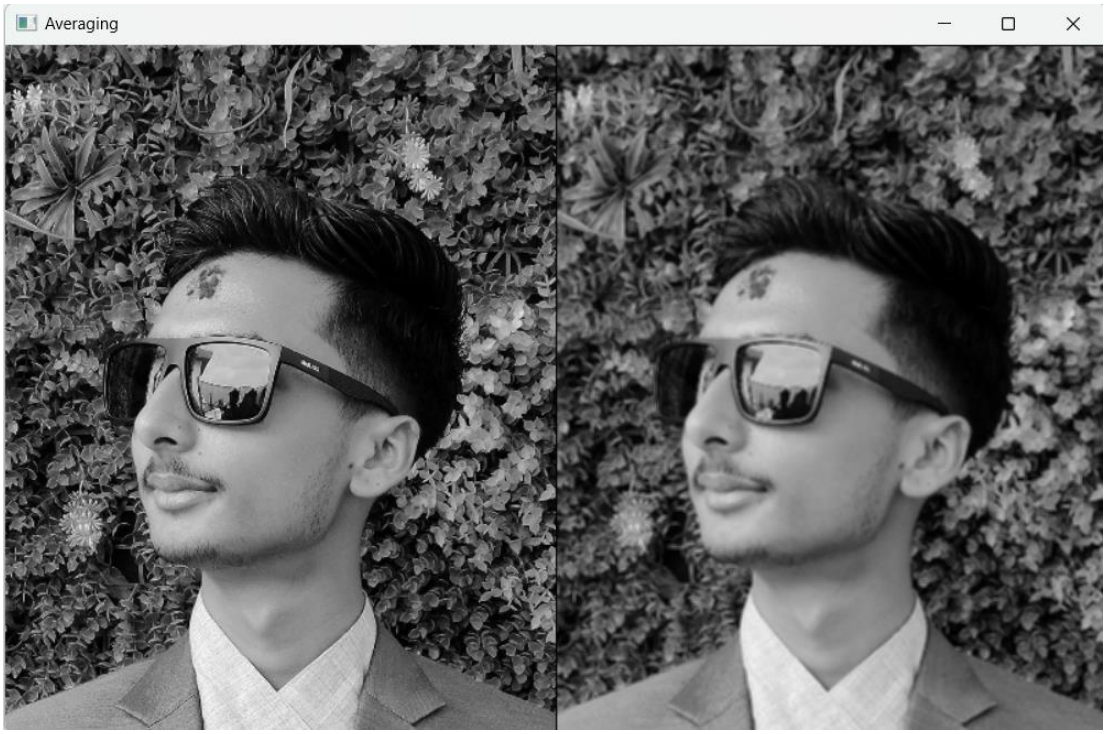
In weighted average filter pixels are multiplied by different numbers. This gives more weightage to some pixels at the expense of others. We give more weight and importance to the center value. Then, it is multiplied by a higher value than any other value in the mask. This gives this pixel a higher contribution and impact than the other pixels. It blurs the image and removes the noise from it. It is also used to reduce details, such as sharpness, edges, and so on.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/photo2.jpg')
img = cv2.resize(img,(400, 500))
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Obtain number of rows and columns
m, n = img.shape
# Develop Averaging filter(3, 3) mask
mask = np.ones([3, 3], dtype = int)
mask = mask / 9
# Convolve the 3X3 mask over the image
img_new = np.zeros([m, n])
for i in range(1, m-1):
    for j in range(1, n-1):
        temp = img[i-1, j-1]*mask[0, 0]+img[i-1, j]*mask[0, 1] \
        +img[i-1, j + 1]*mask[0, 2]+img[i, j-1]*mask[1, 0]+ img[i,
j]*mask[1, 1] \
        +img[i, j + 1]*mask[1, 2]+img[i + 1, j-1]*mask[2, 0]+img[i + 1,
j]*mask[2, 1] \
        +img[i + 1, j + 1]*mask[2, 2]

        img_new[i, j]= temp

img_new = img_new.astype(np.uint8)
both_image = np.hstack([img, img_new])
# Display the images
cv2.imshow('Averaging',both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



12. Implementation of median filtering.

Background:

The median filter is a type of nonlinear filters. It is very effective at removing impulse noise, the “salt and pepper” noise, in the image. The principle of the median filter is to replace the gray level of each pixel by the median of the gray levels in a neighborhood of the pixels, instead of using the average operation. Median filtering uses the kernel size, the pixel values, covered by the kernel, and the median level. If the kernel covers an even number of pixels, the average of two median values is used. Before beginning median filtering, zeros must be padded around the row edge and the column edge. Hence, edge distortion is introduced at image boundary.

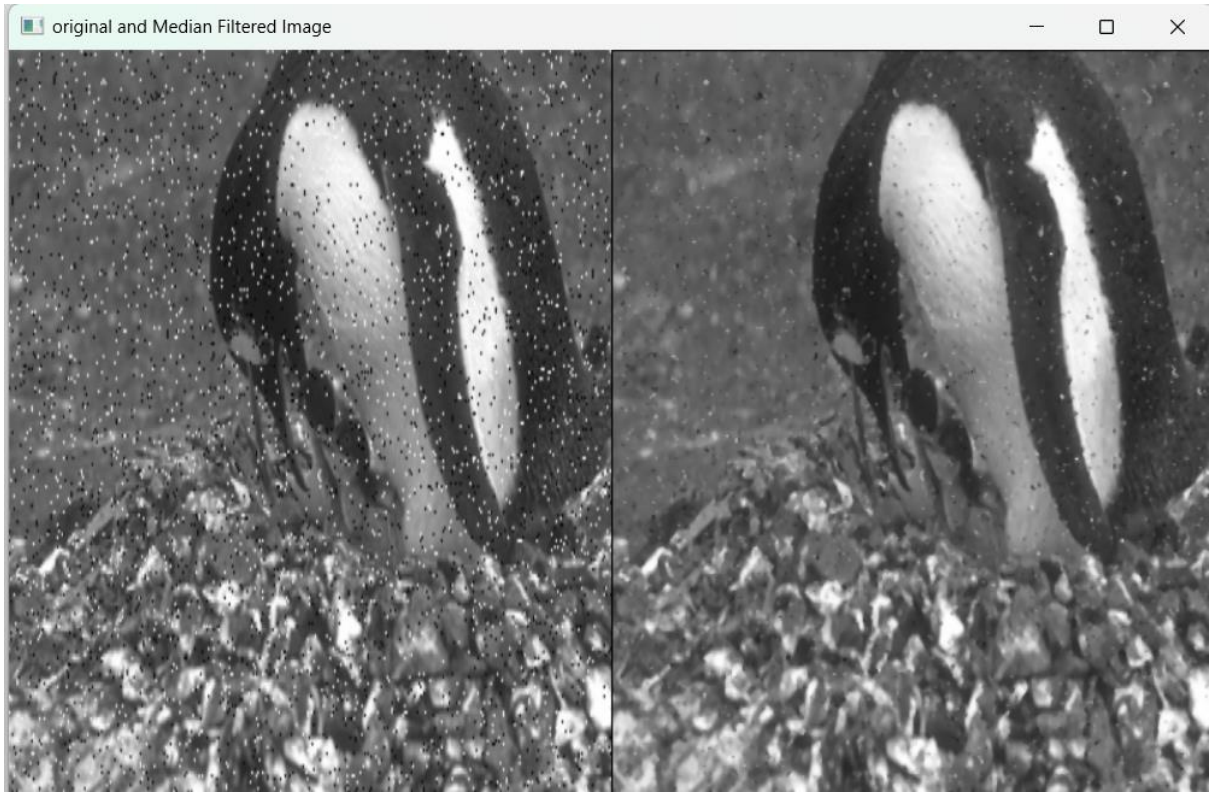
Language: Python

Source Code:

```
import cv2
import numpy as np
# Read the image (adjust path if needed)
img = cv2.imread('./Image_Processing/salt3.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_noisy1 = cv2.resize(img, (400, 500))
# Get image dimensions (accounting for color channels)
m, n = img_noisy1.shape
# Create a new image array for the filtered result
img_new1 = np.zeros([m, n])
# Iterate through the image, applying median filtering
for i in range(1, m - 1):
    for j in range(1, n - 1):
        temp = [
            img_noisy1[i - 1, j - 1],
            img_noisy1[i - 1, j],
            img_noisy1[i - 1, j + 1],
            img_noisy1[i, j - 1],
            img_noisy1[i, j],
            img_noisy1[i, j + 1],
            img_noisy1[i + 1, j - 1],
            img_noisy1[i + 1, j],
            img_noisy1[i + 1, j + 1]
        ]
        temp = np.sort(temp)
        img_new1[i, j] = temp[4]  # Assign the median value
# Convert to uint8 for display
img_new2 = img_new1.astype(np.uint8)
both_image = np.hstack([img_noisy1, img_new2])
```

```
cv2.imshow('original and Median Filtered Image', both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



13. Implementation of minimum filtering.

Background:

The minimum filter is one of the morphological filters .It is also called as dilation filter. The dark values present in an image are enhanced by the minimum filter. When the minimum filter is applied to a digital image it picks up the minimum value of the neighborhood pixels under the filter window and assigns it to the current pixel. A pixel with the minimum value means the darkest pixel from the window.

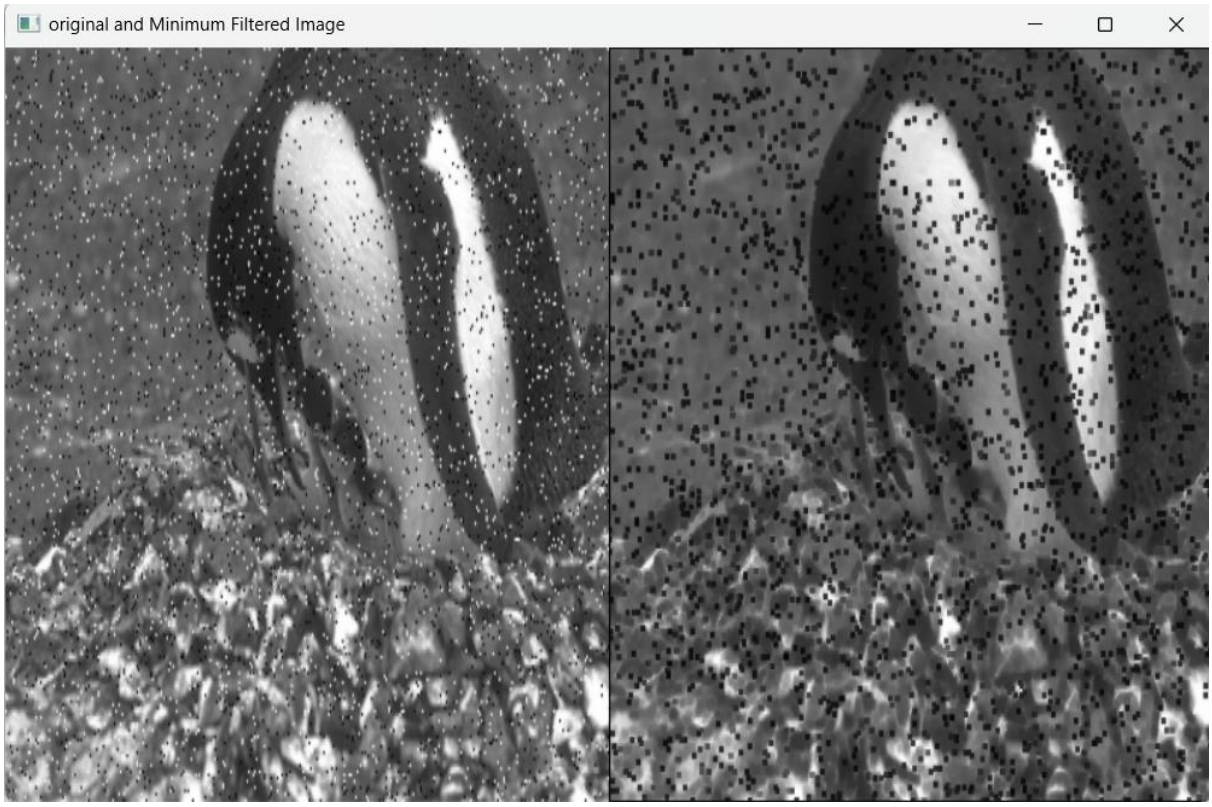
When minimum filter is applied, the object boundaries present in an image are extended.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('salt3.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_noisy1 = cv2.resize(img, (400, 500))
# Get image dimensions (accounting for color channels)
m, n = img_noisy1.shape  # Unpack height, width, and discard channels
# Create a new image array for the filtered result
img_new1 = np.zeros([m, n])
# Iterate through the image, applying minimum filtering
for i in range(1, m - 1):
    for j in range(1, n - 1):
        temp = [
            img_noisy1[i - 1, j - 1],
            img_noisy1[i - 1, j],
            img_noisy1[i - 1, j + 1],
            img_noisy1[i, j - 1],
            img_noisy1[i, j],
            img_noisy1[i, j + 1],
            img_noisy1[i + 1, j - 1],
            img_noisy1[i + 1, j],
            img_noisy1[i + 1, j + 1]
        ]
        temp = np.sort(temp)
        img_new1[i, j] = temp[0]  # Assign the minimum value
# Convert to uint8 for display
img_new2 = img_new1.astype(np.uint8)
both_image = np.hstack([img_noisy1, img_new2])

cv2.imshow('original and Minimum Filtered Image', both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



14. Implementation of Maximum filtering

Background:

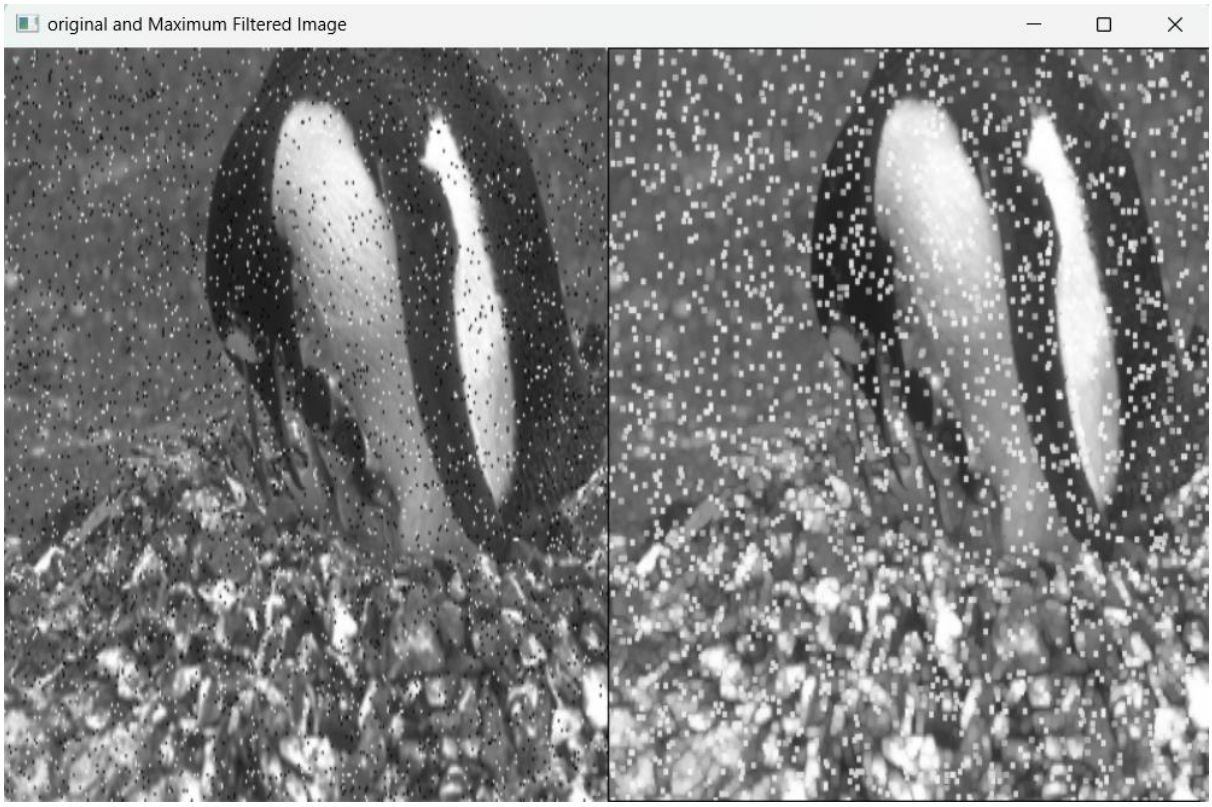
Maximum filter is also called erosion filter. It works opposite to minimum filter. When a maximum filter is applied, the darker objects present in the image are eroded. The maximum filter replaces each pixel value of a digital image with the maximum value(i.e., the value of the brightest pixel) of its neighborhoods. Applying the maximum filter removes the negative outlier noise present in a digital Image.it.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/salt3.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_noisy1 = cv2.resize(img, (400, 500))
# Get image dimensions (accounting for color channels)
m, n = img_noisy1.shape  # Unpack height, width, and discard channels
# Create a new image array for the filtered result
img_new1 = np.zeros([m, n])
# Iterate through the image, applying maximum filtering
for i in range(1, m - 1):
    for j in range(1, n - 1):
        temp = [
            img_noisy1[i - 1, j - 1],
            img_noisy1[i - 1, j],
            img_noisy1[i - 1, j + 1],
            img_noisy1[i, j - 1],
            img_noisy1[i, j],
            img_noisy1[i, j + 1],
            img_noisy1[i + 1, j - 1],
            img_noisy1[i + 1, j],
            img_noisy1[i + 1, j + 1]
        ]
        temp = np.sort(temp)
        img_new1[i, j] = temp[8]  # Assign the maximum value
# Convert to uint8 for display
img_new2 = img_new1.astype(np.uint8)
both_image = np.hstack([img_noisy1, img_new2])
cv2.imshow('original and Maximum Filtered Image', both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



15. **Implement Gaussian Blur.**

Background:

Gaussian Blur is linear spatial filter used for smoothing, noise reduction or blurring. Gaussian filtering is more effective at smoothing images. The kernel coefficients diminish with increasing distance from the kernel's center. Normally, the kernel matrix are considered symmetric and size of the filter is odd. The values inside the kernel are computed by the Gaussian kernel, Gaussian kernel in 2-D form is expressed as:

$$G_{2D}xy\sigma = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

In gaussian kernel, central pixels have a higher weighting than those on the periphery.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/photo2.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_noisy1 = cv2.resize(img, (400, 500))
# Get image dimensions (accounting for color channels)
m, n = img_noisy1.shape  # Unpack height, width, and discard channels
# Create a new image array for the filtered result
img_new1 = np.zeros([m, n])
# Create a Gaussian kernel
gaussian_kernel = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) / 16
# Iterate through the image, applying Gaussian blurring
for i in range(1, m - 1):
    for j in range(1, n - 1):
        temp = [
            img_noisy1[i - 1, j - 1] * gaussian_kernel[0, 0],
            img_noisy1[i - 1, j] * gaussian_kernel[0, 1],
            img_noisy1[i - 1, j + 1] * gaussian_kernel[0, 2],
            img_noisy1[i, j - 1] * gaussian_kernel[1, 0],
            img_noisy1[i, j] * gaussian_kernel[1, 1],
            img_noisy1[i, j + 1] * gaussian_kernel[1, 2],
            img_noisy1[i + 1, j - 1] * gaussian_kernel[2, 0],
            img_noisy1[i + 1, j] * gaussian_kernel[2, 1],
            img_noisy1[i + 1, j + 1] * gaussian_kernel[2, 2]
        ]
        img_new1[i, j] = sum(temp)  # Assign the blurred value
# Convert to uint8 for display
img_new2 = img_new1.astype(np.uint8)
```

```
both_image = np.hstack([img_noisy1, img_new2])
cv2.imshow('original and Gaussian Blurred Image', both_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

16. Implement dilation and erosion of an image.

Background:

Dilation and Erosion reshape objects in images and are vital for enhancing, extracting features, and reducing noise. These operations work on binary images, where each pixel is either black or white, using a small binary structure to define pixel neighborhoods. This structure can take various shapes, such as square, rectangle, circle, or diamond.

Dilation:

Dilation is a morphological operation that expands the boundaries of an object in an image. This is done by convolving the image with a structuring element, which determines the size and shape of the dilation. The output of the dilation operation is a new image where the pixels in the original image are expanded or dilated.

Erosion:

Erosion is a morphological operation that shrinks the boundaries of an object in an image. This is done by convolving the image with a structuring element, which determines the size and shape of the erosion. The output of the erosion operation is a new image where the pixels in the original image are eroded or shrunk.

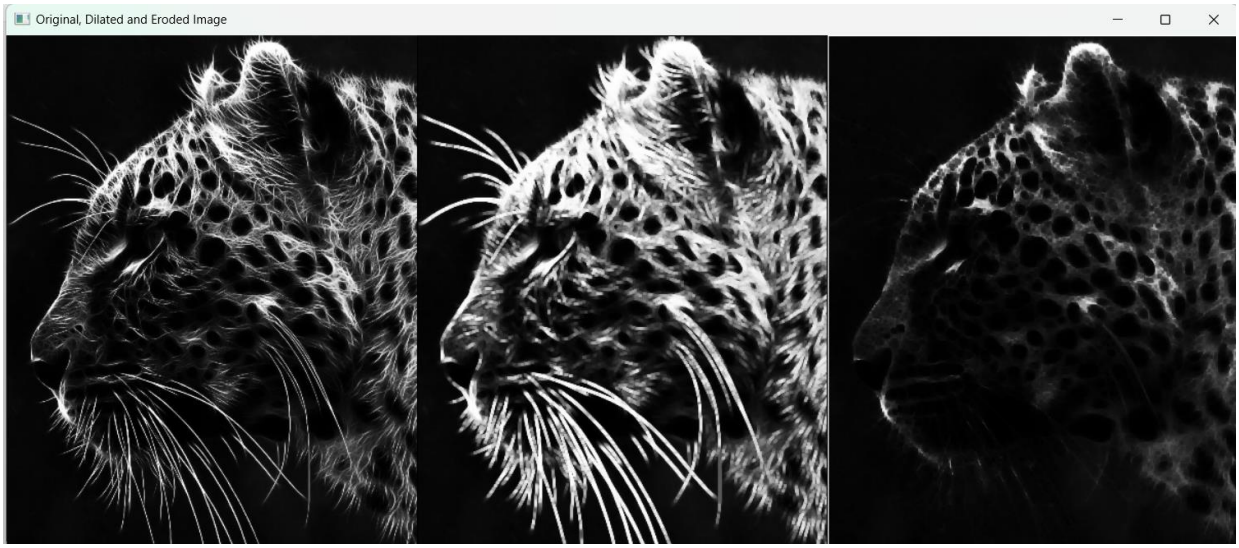
Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/anim.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.resize(img, (400, 500))
# Get image dimensions
m, n = img.shape
# Define the structuring element
se = [[1, 1, 1],
      [1, 1, 1],
      [1, 1, 1]]
# Initialize the output images
dilated_img = np.zeros([m, n], dtype=np.uint8)
eroded_img = np.ones([m, n], dtype=np.uint8) * 255
# Apply dilation
for i in range(1, m-1):
    for j in range(1, n-1):
        max_pixel = np.max(img[i-1:i+2, j-1:j+2])
        dilated_img[i, j] = max_pixel
# Apply erosion
for i in range(1, m-1):
```

```
        for j in range(1, n-1):
            min_pixel = np.min(img[i-1:i+2, j-1:j+2])
            eroded_img[i, j] = min_pixel
    # Save the output images
    out_image = np.hstack([img, dilated_img, eroded_img])
    cv2.imshow('Original, Dilated and Eroded Image', out_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Output:



17. Implement opening and closing of an image

Background:

Opening:

The opening operation erodes an image and then dilates the eroded image using the same structuring element for both operations, i.e.

$$AB = (A \ B) \ B$$

where A is the original image and B is the structuring element.The opening operation is used to remove regions of an object that cannot contain the structuring element, smooth objects contours, and breaks thin connections.

Closing:

The closing operation dilates an image and then erodes the dilated image using the same structuring element for both operations, i.e.

$$AB = (A \ B) \ B$$

where A is the original image and B is the structuring element. The closing operation fills holes that are smaller than the structuring element, joins narrow breaks, fills gaps in contours, and smoothes objects contours.

Opening and Closing are dual operations used in Digital Image Processing for restoring an eroded image. Opening is generally used to restore or recover the original image to the maximum possible extent. Closing is generally used to smoother the contour of the distorted image and fuse back the narrow breaks and long thin gulfs. Closing is also used for getting rid of the small holes of the obtained image. The combination of Opening and Closing is generally used to clean up artifacts in the segmented image before using the image for digital analysis.

Language: Python

Source Code:

```
import cv2
import numpy as np
img = cv2.imread('./Image_Processing/man.jpg', 0)
# Define the structuring element
se = np.ones((5,5), np.uint8)
# Apply opening operation (erosion followed by dilation)
opened_img = cv2.morphologyEx(img, cv2.MORPH_OPEN, se)
# Apply closing operation (dilation followed by erosion)
closed_img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, se)
```

```
# Display the images
out_image=np.hstack([img, opened_img, closed_img])
cv2.imshow('Original, Opened, and Closed Image', out_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

18. Run-Length Coding.

Background:

Run-length encoding (RLE) is a lossless compression method where sequences that display redundant data are stored as a single data value representing the repeated block and how many times it appears in the image. Later, during decompression, the image can be reconstructed exactly from the information. This type of compression works best with simple images and animations that have a lot of redundant pixels. It's useful for black and white images in particular. For complex images and animations, if there aren't many redundant sections, RLE can make the file size bigger rather than smaller. Thus it's important to understand the content and whether this algorithm will help or hinder.

Language: Python

Source Code:

```
def run_length_encoding(input_array):
    i = 0
    result = []
    while i < len(input_array):
        count = 1
        while i + 1 < len(input_array) and input_array[i] ==
input_array[i+1]:
            i += 1
            count += 1
        result.append((input_array[i], count))
        i += 1
    return result
data = [1,1,1,1,2,2,2,2,2,7,7,7,7,7,8,8,1,1,1]
encoded_data = run_length_encoding(data)
print(encoded_data)
```

Output:

```
[(1, 4), (2, 5), (7, 5), (8, 2), (1, 3)]
```

19. Huffman-coding.

Background:

Huffman coding is a method of data compression that is independent of the data type, that is, the data could represent an image, audio or spreadsheet. This compression scheme is used in JPEG and MPEG-2. Huffman coding works by looking at the data stream that makes up the file to be compressed. Those data bytes that occur most often are assigned a small code to represent them (certainly smaller then the data bytes being represented). Data bytes that occur the next most often have a slightly larger code to represent them. This continues until all of the unique pieces of data are assigned unique code words. For a given character distribution, by assigning short codes to frequently occurring characters and longer codes to infrequently occurring characters, Huffman's minimum redundancy encoding minimizes the average number of bytes required to represent the characters in a text.

Language: Python

Source Code:

```
import heapq
from collections import defaultdict
def calculate_frequency(chars, freq):
    return dict(zip(chars, freq))
def huffman_encode(frequency):
    heap = [[weight, [symbol, ""]] for symbol, weight in
frequency.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]),
p))
chars = ['a', 'b', 'c', 'd', 'e', 'f']
freq = [7, 8, 1, 40, 54, 68]
# Calculate frequency
frequency = calculate_frequency(chars, freq)
# Apply Huffman encoding
huff = huffman_encode(frequency)
print("Symbol".ljust(10) + "Weight".ljust(10) + "Huffman Code")
```

```
for p in huff:
    print(str(p[0]).ljust(10) + str(frequency[p[0]]).ljust(10) + p[1])
```

Output:

```
Symbol    Weight    Huffman Code
f         68        0
e         54        10
d         40        111
b         8         1100
a         7         11011
c         1         11010
```

20. Program to extract edge of an Image.

Background:

Edge detection includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. The points at which image brightness changes sharply are typically organized into a set of curved line segments termed edges. The same problem of finding discontinuities in 1D signals is known as step detection and the problem of finding signal discontinuities over time is known as change detection. Edge detection is a fundamental tool in image processing, machine vision and computer vision, particularly in the areas of feature detection and feature extraction.

Language: Python

Source Code:

```
import cv2
import numpy as np
image = cv2.imread('./Image_Processing/photo2.jpg',
cv2.IMREAD_GRAYSCALE)
image = cv2.resize(image,(400, 500))
edges = cv2.Canny(image, 100, 200)
output = np.hstack([image, edges])
cv2.imshow('Original Image and Edges', output)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output: