**Task 1**

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

**Theory**

In image processing, the OpenCV library in Python serves as a powerful tool for reading, saving, and displaying images. OpenCV, an open-source computer vision and machine learning library, provides a comprehensive set of functions and tools for image manipulation. The process typically involves using the imread() function to read an image from a file, saving the processed image using imwrite(), and displaying it through imshow(). These functions enable efficient image handling, allowing for various operations such as resizing, filtering, and color space conversion. Additionally, OpenCV facilitates the integration of image processing algorithms, making it an invaluable resource for tasks like computer vision and pattern recognition. In the context of the lab exam, a well-structured Python program utilizing OpenCV can seamlessly execute the fundamental tasks of reading, saving, and displaying images.

**Source Code**

import cv2

# Read an image from file

image = cv2.imread('./final lab/img1.jpg')

# Check if the image was successfully loaded

if image is not None:

    # Display the image in a window

    cv2.imshow('Image', image)

    # Wait for a key press and close the window

    cv2.waitKey(0)

    cv2.destroyAllWindows()

    # Save the image to a new file

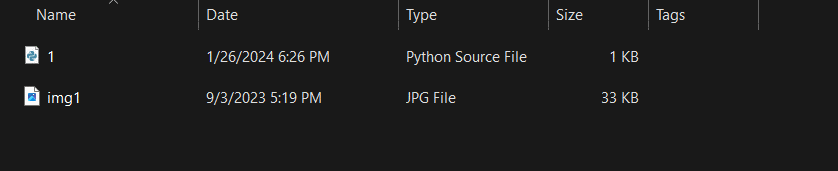
    cv2.imwrite('./final lab/processed\_image.jpg', image)

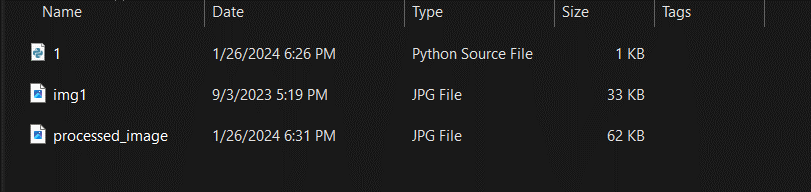
else:

    print('Image not found or could not be loaded.')

**Output**

Initially, we have a single file in the directory named ‘img1.jpg’, and after running the code, the directory also has another image file titled ‘processed\_img.jpg’.





**Conclusion**

Hence, in this way, we can use OpenCV library to read, display and save an image.

**Task 2**

**Write a program to convert an RGB image to a greyscale image.**

**Theory**

Converting a color image from RGB to grayscale is a fundamental operation in image processing, often used to simplify data for subsequent analysis. In the context of the OpenCV library in Python, this task is seamlessly accomplished through the cv2.cvtColor() function. RGB images consist of three color channels (Red, Green, Blue), and the grayscale conversion involves blending these channels to create a single intensity channel. The cv2.cvtColor() function, when applied with the appropriate conversion code (cv2.COLOR\_RGB2GRAY), efficiently transforms the input RGB image into its grayscale equivalent. Grayscale images are particularly advantageous for image processing tasks as they eliminate color complexity, reducing computational overhead while retaining essential structural information. This operation is foundational in computer vision applications, enhancing the efficiency of subsequent algorithms and analyses by working with simpler, yet informative, grayscale representations of the original images.

Top of Form

**Source Code**

import cv2

image = cv2.imread("./final lab/img1.jpg")

cv2.imshow("real image", image)

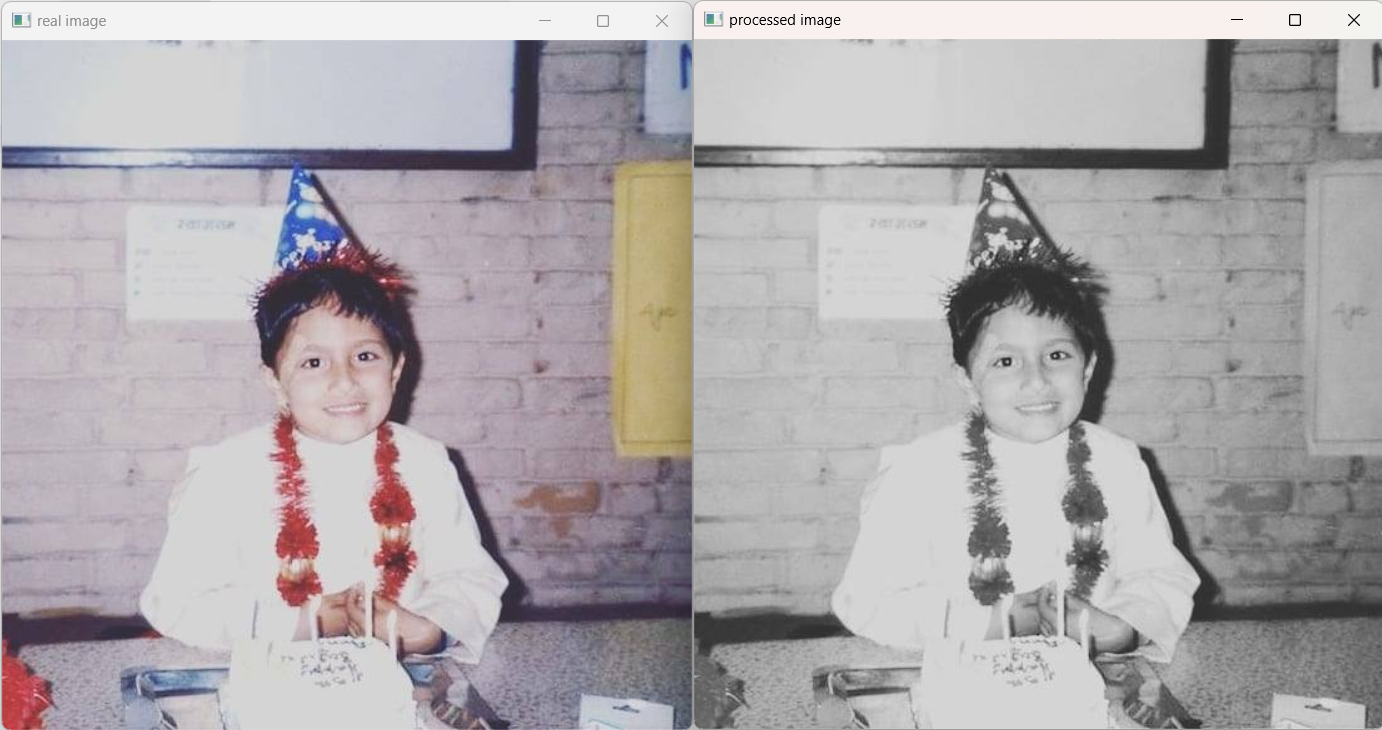
gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

cv2.imshow("processed image", gray\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to convert a RGB image to its corresponding greyscale image.

**Task 3**

**Write a program to convert an RGB image to Black and White image.**

**Theory**

Converting an RGB image to a black and white (binary) image is a common preprocessing step in image processing, often employed to extract essential features for further analysis. In Python, utilizing the OpenCV library, this transformation is achieved through the cv2.cvtColor() function, where the conversion code cv2.COLOR\_RGB2GRAY is employed to first convert the RGB image to grayscale. Subsequently, the cv2.threshold() function is applied to create a binary image by assigning pixels to either black or white based on a specified threshold value. This process enables the extraction of high-contrast representations, emphasizing the presence or absence of certain features in the original image. Black and white images are particularly useful in scenarios where color information is not crucial, streamlining computational tasks and enhancing the effectiveness of subsequent image processing algorithms. This programmatic approach showcases the versatility of OpenCV in efficiently handling such transformations, thereby providing a foundation for more advanced image analysis in various applications.

Top of Form

**Source Code**

import cv2

img= cv2.imread('./lab 1/img1.jpg')

# img = cv2.resize(img, (800, 600))

image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

(thresh, bimage) = cv2.threshold(image, 200, 225, cv2.THRESH\_BINARY)

cv2.imshow("image",bimage)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to convert and RGB image to black and white.

**Task 4**

**Write a program to create a digital negative of an image.**

**Theory**

Creating a digital negative of an image is a transformative process in image processing, commonly used for artistic and creative purposes. In the context of the OpenCV library in Python, this task involves inverting the pixel values of the original image, effectively converting dark areas to light and vice versa. The cv2.bitwise\_not() function is instrumental in achieving this inversion, facilitating the generation of a digital negative. By negating the pixel values across all color channels, the program produces an image where highlights become shadows and shadows become highlights, resulting in an intriguing and visually distinct representation of the original photograph. This operation is not only aesthetically interesting but also offers a unique perspective on image manipulation. The ease of implementation with OpenCV underscores its capability in providing powerful tools for creative image processing tasks.

Top of Form

Top of Form

**Source Code**

import cv2

import matplotlib.pyplot as plt

img = cv2.imread('./final lab/img2.png')

cv2.imshow('input image', img)

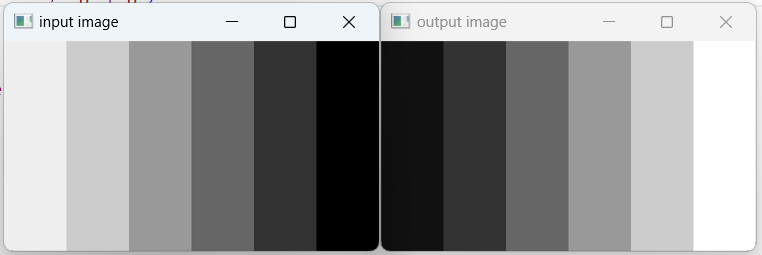
img\_neg = 255 - img

cv2.imshow('output image', img\_neg)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to create a digital negative of an image.

**Task 5**

**Write a program to create a histogram plot of an image.**

**Theory**

Generating a histogram plot of an image is a crucial step in understanding its pixel intensity distribution, providing valuable insights into the image's contrast, brightness, and overall tonal characteristics. In the realm of image processing with Python's OpenCV library, the cv2.calcHist() function plays a pivotal role in constructing histograms. By specifying the image and the channels of interest, this function computes the frequency distribution of pixel intensities, which can then be visualized using libraries like Matplotlib. The resulting histogram plot illustrates the distribution of pixel values across different intensity levels, offering a comprehensive overview of the image's tonal composition. Analyzing such histograms aids in making informed decisions during image enhancement or manipulation processes, contributing to the effective utilization of image processing techniques. The ability to easily generate histograms exemplifies the versatility and analytical capabilities that OpenCV brings to image processing tasks.

Top of Form

Top of Form

**Source Code**

import cv2

import matplotlib.pyplot as plt

def plot\_histogram(image\_path):

    # Read the image using OpenCV

    image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

    # Calculate histogram

    hist = cv2.calcHist([image], [0], None, [256], [0, 256])

    # Plot histogram

    plt.plot(hist, color='black')

    plt.xlabel('Pixel Intensity')

    plt.ylabel('Frequency')

    plt.title('Histogram')

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

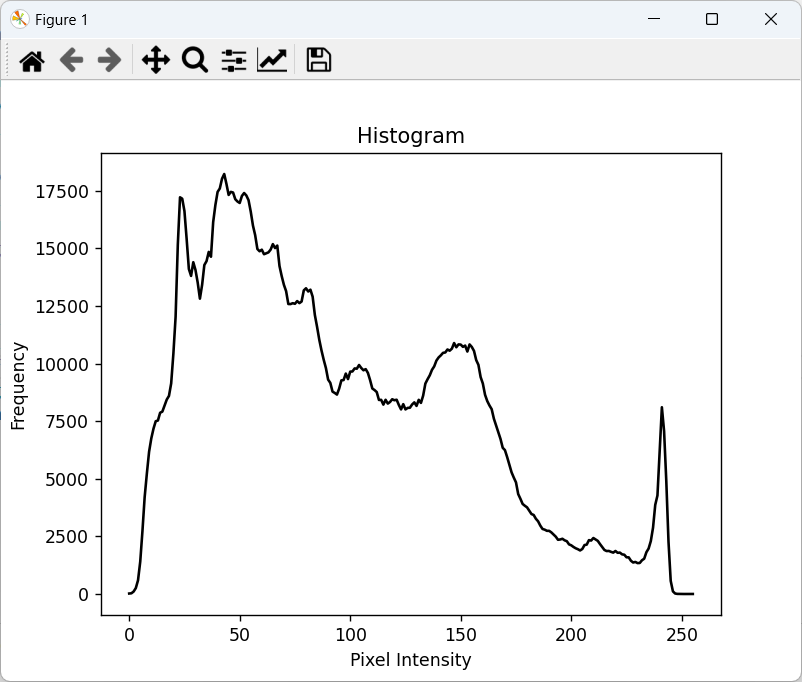
    # Replace 'your\_image.jpg' with the path to your image file

    image\_path = './final lab/1.jpg'

    # Plot the histogram

    plot\_histogram(image\_path)

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to plot a histogram of an image.

**Task 6**

**Write a program to find digital negative of an image with histogram.**

**Theory**

Creating a digital negative of an image with an accompanying histogram involves a two-fold process in image processing using Python's OpenCV library. Initially, the negative of the image is directly calculated by subtracting the pixel values from 1 for each pixel. Subsequently, the histogram of the resulting negative image is computed using cv2.calcHist(). This histogram provides a visual representation of the pixel intensity distribution in the digital negative, showcasing the changes induced by the inversion process. Analyzing the histogram aids in understanding the impact on contrast, brightness, and tonal characteristics, offering valuable insights into the transformed image's overall composition. This program thus combines the creative manipulation of pixel values with the analytical exploration of the resulting image's histogram, demonstrating the versatility and power of OpenCV in facilitating both image transformation and comprehensive visual analysis.

Top of Form

Top of Form

**Source Code**

import cv2

import matplotlib.pyplot as plt

# Read an image

img\_bgr = cv2.imread('./final lab/1.jpg', 3)

plt.imshow(img\_bgr)

plt.show()

# Histogram plotting of the 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()

# Negate the original image

img\_neg = 1 - img\_bgr

plt.imshow(img\_neg)

plt.show()

# Histogram plotting of

# the negative transformed image

color = ('b', 'g', 'r')

for i, col in enumerate(color):

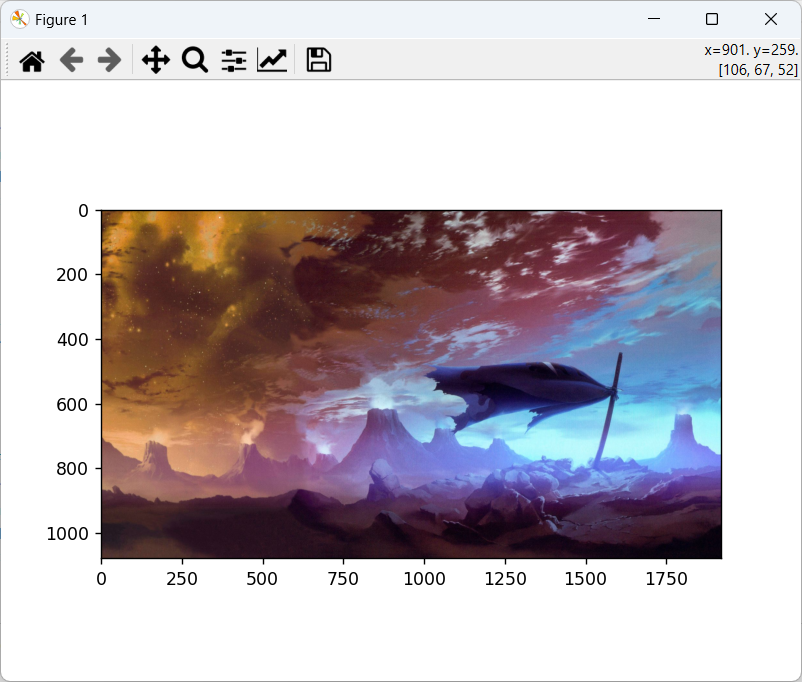
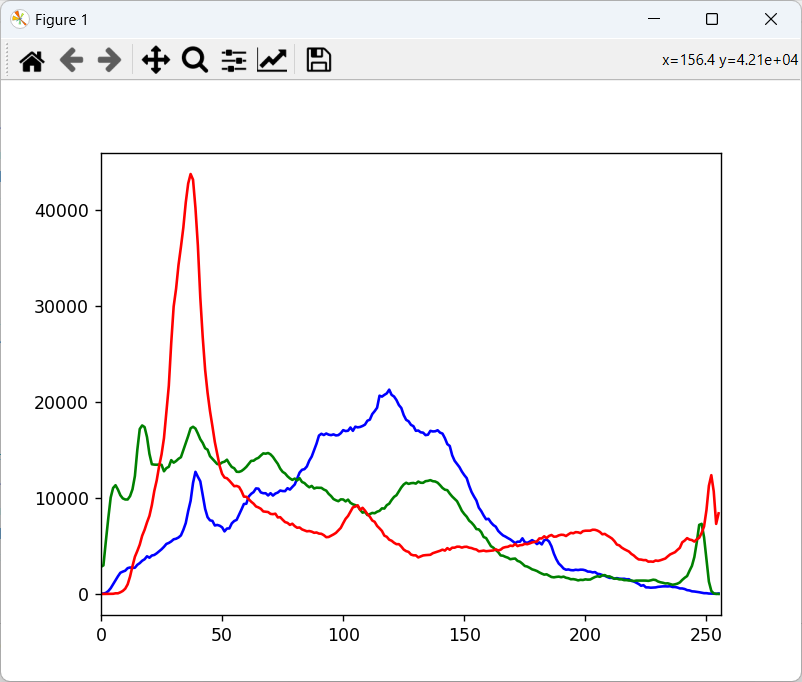
    histr = cv2.calcHist([img\_neg], [i], None, [256], [0, 256])

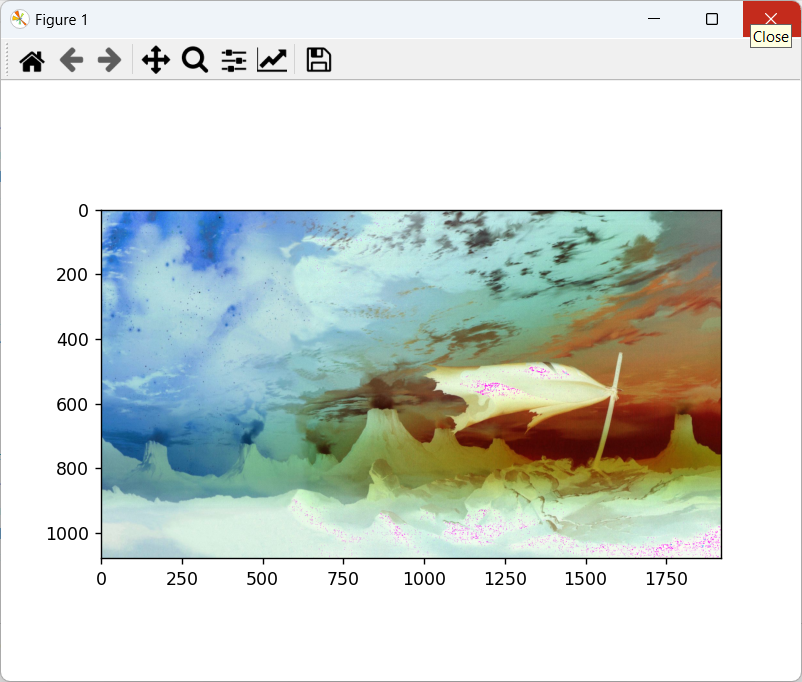
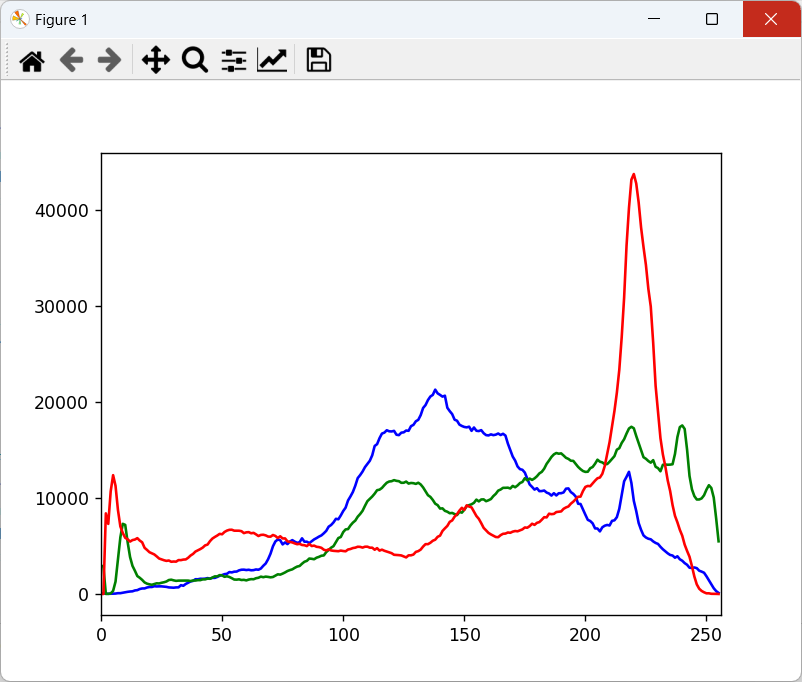
    plt.plot(histr, color=col)

    plt.xlim([0, 256])

plt.show()

**Output**

**Conclusion**

Hence, in this way, we can use OpenCV library get digital negative of an image and histogram for both original and negative image.

**Task 7**

**Write a program to perform histogram equalization.**

**Theory**

Histogram equalization is a fundamental technique in image processing employed to enhance the global contrast of an image by redistributing pixel intensities. In the realm of Python's OpenCV library, the cv2.equalizeHist() function plays a central role in achieving this transformation. This process involves calculating the cumulative distribution function (CDF) of pixel intensities and then mapping the original pixel values to their corresponding values in the equalized distribution. The result is an image with a more balanced and stretched intensity range, effectively improving visibility of details in both dark and bright regions. Histogram equalization is particularly useful in scenarios where images exhibit uneven contrast, and its implementation using OpenCV showcases the library's capability to efficiently perform complex image enhancement operations with a few lines of code, making it an essential tool for image processing tasks.Top of Form

Top of Form

**Source Code**

import cv2

# read an image using imread

img = cv2.imread('./final lab/2.jpg', 0)

cv2.imshow('input', img)

# creating a Histograms Equalization

# of an image using cv2.equalizeHist()

equ = cv2.equalizeHist(img)

# show image input vs output

cv2.imshow('output', equ)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform histogram equalization.

**Task 8**

**Write a program to perform piecewise linear transformation.**

**Theory**

Piecewise linear transformation is a technique in image processing used to adjust the contrast and brightness of different segments of an image independently. In this Python program utilizing the OpenCV library, a custom piecewise\_linear\_transform function is defined to perform such transformations. The function takes an input image, along with user-defined breakpoints and slopes corresponding to the linear segments. By iteratively applying linear transformations to different intensity ranges defined by the breakpoints, the function creates a transformed image with distinct contrast variations across specified regions. The example code demonstrates the application of this piecewise linear transformation on a grayscale image, adjusting the image contrast in segments defined by the breakpoints and slopes. The resulting transformed image showcases the flexibility and control offered by piecewise linear transformations in tailoring the visual characteristics of an image to specific requirements. The use of OpenCV, in conjunction with matplotlib for visualization, exemplifies the efficiency and ease with which complex image processing operations can be implemented in Python.

Top of Form

Top of Form

Top of Form

**Source Code**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def piecewise\_linear\_transform(image, breakpoints, slopes):

    transformed\_image = np.zeros\_like(image)

    # Apply piecewise linear transformation

    for i in range(len(breakpoints) - 1):

        mask = np.logical\_and(image >= breakpoints[i], image < breakpoints[i + 1])

        transformed\_image = np.where(mask, slopes[i] \* (image - breakpoints[i]), transformed\_image)

    # Handle the last segment

    mask = image >= breakpoints[-1]

    transformed\_image = np.where(mask, slopes[-1] \* (image - breakpoints[-1]), transformed\_image)

    return transformed\_image.astype(np.uint8)

# Read an example image

original\_image = cv2.imread('./final lab/2.jpg', cv2.IMREAD\_GRAYSCALE)

# Define breakpoints and slopes for the piecewise linear transformation

breakpoints = [0, 100, 150, 255]

slopes = [0, 2, 0.5, 1]

# Apply piecewise linear transformation

transformed\_image = piecewise\_linear\_transform(original\_image, breakpoints, slopes)

# Display the results

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(original\_image, cmap='gray')

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

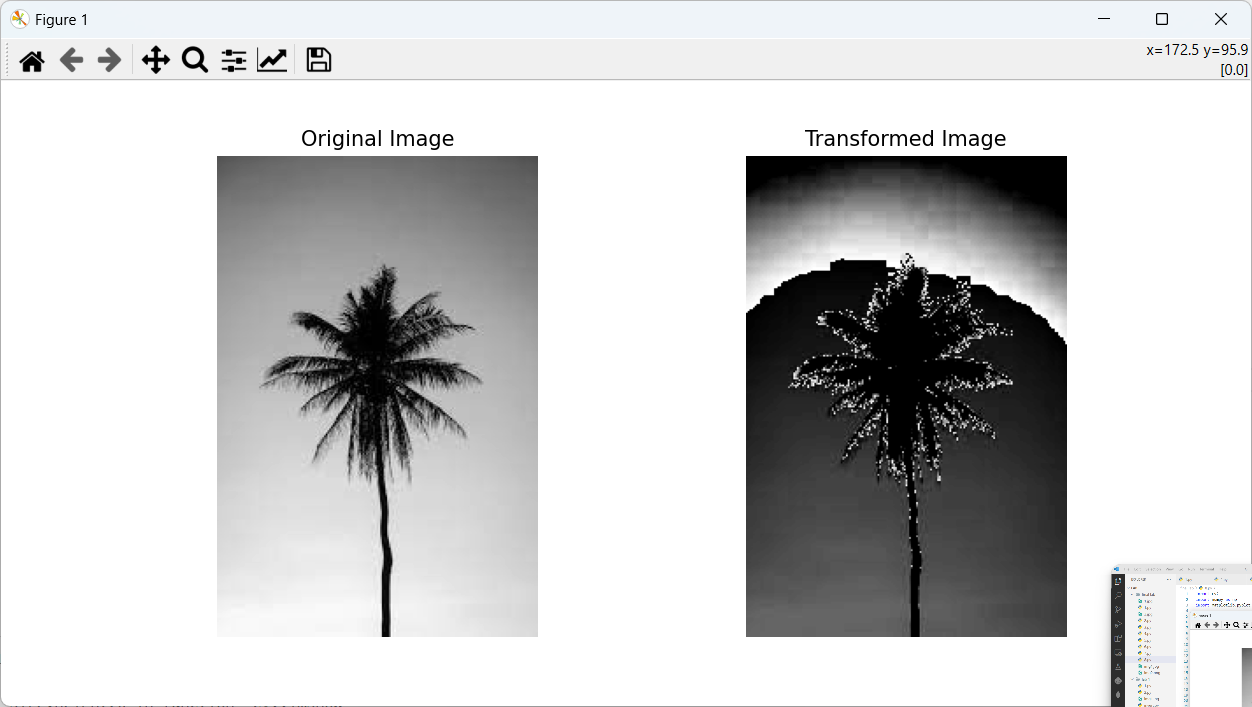
plt.imshow(transformed\_image, cmap='gray')

plt.title('Transformed Image')

plt.axis('off')

plt.show()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform piecewise linear transformation of an image.

**Task 9**

**Write a program to perform power law transformation.**

**Theory**

Power law transformation, also known as gamma correction, is a technique widely used in image processing to adjust the brightness and contrast of an image by applying a nonlinear mapping function. In this Python program using the OpenCV library, a power law transformation is implemented on an image loaded with cv2.imread(). The transformation is defined by the gamma parameter, which controls the shape of the mapping curve. A gamma value less than 1 enhances the contrast of the darker regions, while a value greater than 1 enhances the contrast of the brighter regions. The resulting image, s, is obtained by scaling pixel intensities according to the power law transformation. This technique is particularly useful for compensating for the non-linear response of display devices and cameras, ensuring better visibility and interpretation of image details. The program showcases the simplicity and effectiveness of implementing power law transformations using OpenCV in Python, providing a valuable tool for enhancing image quality in various applications.

Top of Form

Top of Form

**Source Code**

import cv2

import numpy as np

img = cv2.imread('./final lab/2.jpg')

gamma = 0.6

s = np.array(255 \* (img / 255) \*\* gamma, dtype='uint8')

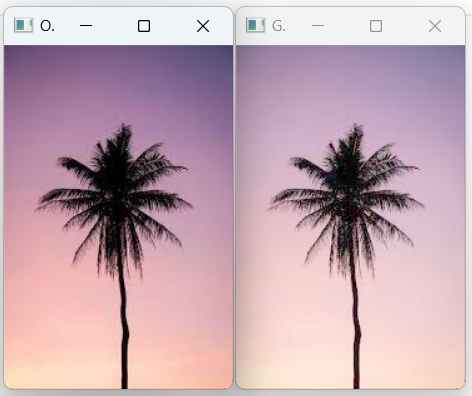
cv2.imshow("Original Image", img)

cv2.imshow("Gamma Corrected Image", s)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform power law transformation.

**Task 10**

**Write a program to perform log transformation.**

**Theory**

Logarithmic transformation is a widely employed technique in image processing for enhancing the visibility of details in low-intensity regions while compressing the dynamic range of high-intensity regions. In this Python program utilizing the OpenCV library, a log transformation is implemented on an image loaded with cv2.imread(). The transformation is achieved by applying a logarithmic mapping function to the pixel intensities, where the logarithm is computed element-wise across the image. The scaling factor 'c' ensures that the transformed values are within the valid intensity range (0 to 255). By adjusting the intensity values logarithmically, the resulting image, log\_trx\_img, effectively enhances the details in darker areas while controlling the intensities in brighter regions. This technique is valuable in scenarios where image features in both dark and bright regions need to be accentuated simultaneously. The simplicity of implementing log transformations using OpenCV in Python, as demonstrated in the provided code, underscores the utility of this method in improving image visualization and analysis.

Top of Form

Top of Form

**Source Code**

import cv2

import numpy as np

img = cv2.imread('./final lab/3.jpg')

cv2.imshow('input image', img)

c = (255 / np.log(1 + np.max(img)))

log\_trx\_img = c \* np.log(1 + img)

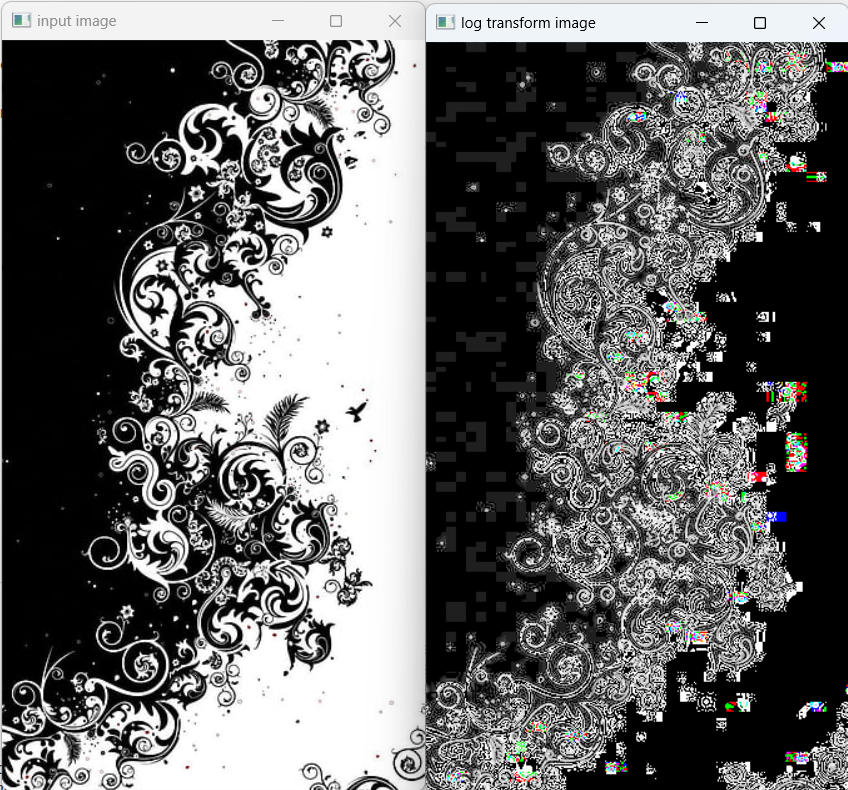
log\_trx\_img = np.array(log\_trx\_img, dtype='uint8')

cv2.imshow('log transform image', log\_trx\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform log transformation.

**Task 11**

**Write a program to implement weighted averaging filter.**

**Theory**

The weighted averaging filter, also known as the weighted mean filter, is a fundamental image processing technique employed for smoothing and noise reduction. In this Python program utilizing the OpenCV library, a weighted\_average\_filter function is implemented. The filter operation involves applying a convolution with a weighted kernel to each pixel in the image. The weights in the kernel are assigned based on a predefined pattern, typically with higher weights assigned to the central pixel and decreasing weights towards the periphery. This program defines a square kernel with weights that increase linearly from the center to the edges. The resulting convolution is then applied to the input image using the cv2.filter2D() function. This weighted averaging operation effectively blurs the image while preserving its overall structure. The provided code demonstrates the application of the weighted averaging filter with a 3x3 kernel on a sample image, highlighting the simplicity and utility of this technique for image smoothing and noise reduction in various image processing applications.

Top of Form

Top of Form

**Source Code**

import numpy as np

import cv2

def weighted\_average\_filter(image, kernel\_size=3):

    # Ensure the kernel size is odd

    if kernel\_size % 2 == 0:

        raise ValueError("Kernel size must be an odd number")

    # Create a kernel with weights (center pixel has the highest weight)

    weights = np.arange(1, kernel\_size\*\*2 + 1).reshape((kernel\_size, kernel\_size))

    kernel = weights / np.sum(weights)

    # Apply the filter to each channel (if the image is color)

    if len(image.shape) == 3:

        result = np.zeros\_like(image, dtype=np.float32)

        for channel in range(image.shape[2]):

            result[:, :, channel] = cv2.filter2D(image[:, :, channel], -1, kernel)

    else:

        result = cv2.filter2D(image, -1, kernel)

    return result.astype(np.uint8)

input\_image = cv2.imread("./final lab/3.jpg")

# Apply weighted average filtering with a 3x3 kernel

output\_image = weighted\_average\_filter(input\_image, kernel\_size=3)

# Display the results

cv2.imshow("Original Image", input\_image)

cv2.imshow("Filtered Image", output\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to implement weighted averaging filter.

**Task 12**

**Write a program to implement median filter.**

**Theory**

The median filter is a widely used image processing technique for noise reduction, particularly effective in preserving edges and fine details. In this Python program utilizing the OpenCV library, a median\_filter function is implemented to perform a median filtering operation on an input image. The filter operation involves sliding a square-shaped kernel over each pixel in the image and replacing the pixel's intensity with the median value of the pixel's neighborhood. This process is particularly effective in suppressing salt-and-pepper noise, as the median value is less sensitive to extreme outliers compared to the mean. The provided code employs a kernel size of 3x3, and the resulting filtered image demonstrates the capability of the median filter in reducing noise while preserving the essential features of the original image. The simplicity and effectiveness of the median filter, as demonstrated in the code, make it a valuable tool in various image processing applications, especially in scenarios where robust noise reduction is essential.

Top of Form

Top of Form

Top of Form

**Source Code**

import numpy as np

import cv2

def median\_filter(image, kernel\_size=3):

    pad\_size = kernel\_size // 2

    padded\_image = cv2.copyMakeBorder(image, pad\_size, pad\_size, pad\_size, pad\_size, cv2.BORDER\_REFLECT)

    result = np.zeros\_like(image)

    for i in range(pad\_size, padded\_image.shape[0] - pad\_size):

        for j in range(pad\_size, padded\_image.shape[1] - pad\_size):

            neighborhood = padded\_image[i-pad\_size:i+pad\_size+1, j-pad\_size:j+pad\_size+1]

            result[i-pad\_size, j-pad\_size] = np.median(neighborhood)

    return result.astype(np.uint8)

input\_image = cv2.imread("me.jpg", cv2.IMREAD\_GRAYSCALE)

output\_image = median\_filter(input\_image, kernel\_size=3)

cv2.imshow("Original Image", input\_image)

cv2.imshow("Filtered Image", output\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to implement median filter.

**Task 13**

**Write a program to implement minimum filter.**

**Theory**

The minimum filter, often employed in image processing for noise reduction and morphological operations, is designed to replace each pixel's intensity with the minimum value within its neighborhood. In this Python program utilizing the OpenCV library, a min\_filter function is implemented. The function iterates over each pixel in the image, considering a local neighborhood defined by a square-shaped kernel. The pixel's intensity is then updated to be the minimum value within this neighborhood. This operation effectively suppresses high-intensity outliers and enhances the visibility of low-intensity features. The provided code uses a 3x3 kernel for minimum filtering, showcasing the simplicity and utility of the minimum filter in mitigating noise and preserving essential image details. The application of this filter is particularly valuable in scenarios where the emphasis is on highlighting low-intensity structures or regions of interest in an image.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

import numpy as np

def min\_filter(image, kernel\_size=3):

    # Ensure the kernel size is odd

    if kernel\_size % 2 == 0:

        raise ValueError("Kernel size must be an odd number")

    # Pad the image to handle border pixels

    pad\_size = kernel\_size // 2

    padded\_image = cv2.copyMakeBorder(image, pad\_size, pad\_size, pad\_size, pad\_size, cv2.BORDER\_REFLECT)

    # Apply minimum filtering

    result = np.zeros\_like(image)

    for i in range(pad\_size, padded\_image.shape[0] - pad\_size):

        for j in range(pad\_size, padded\_image.shape[1] - pad\_size):

            neighborhood = padded\_image[i-pad\_size:i+pad\_size+1, j-pad\_size:j+pad\_size+1]

            result[i-pad\_size, j-pad\_size] = np.min(neighborhood)

    return result.astype(np.uint8)

input\_image = cv2.imread("./final lab/3.jpg", cv2.IMREAD\_GRAYSCALE)

# Apply minimum filtering with a 3x3 kernel

output\_image = min\_filter(input\_image, kernel\_size=3)

# Display the results

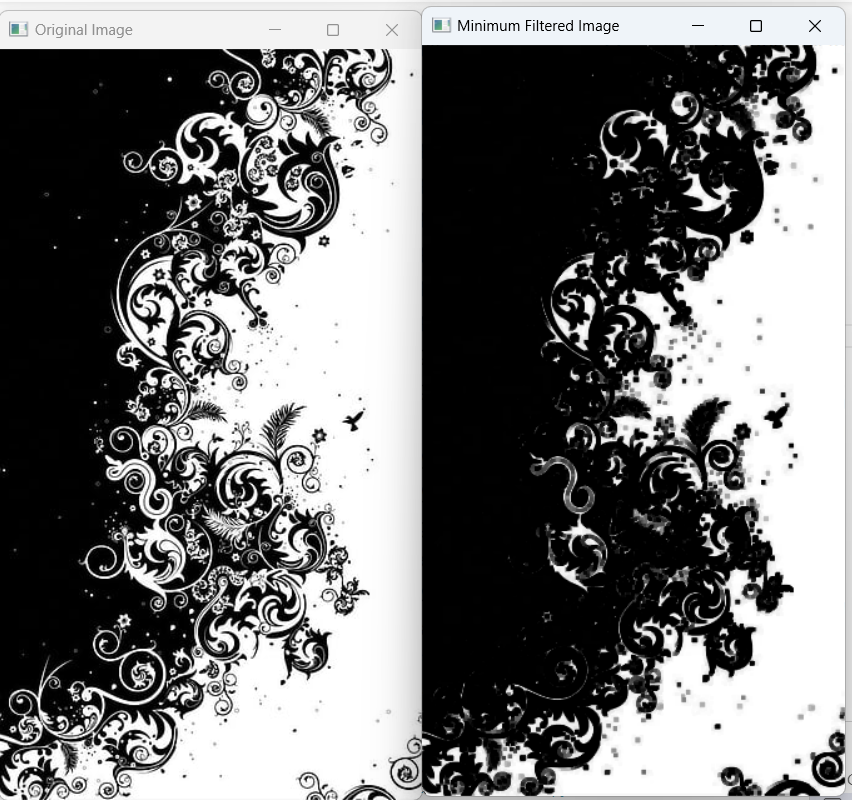
cv2.imshow("Original Image", input\_image)

cv2.imshow("Minimum Filtered Image", output\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to implement minimum filter.

**Task 14**

**Write a program to implement maximum filter.**

**Theory**

The maximum filter is a key element in image processing, widely utilized for morphological operations and noise reduction. In this Python program using the OpenCV library, a max\_filter function is implemented to apply maximum filtering on an input image. The function operates by iterating over each pixel and considering a local neighborhood defined by a square-shaped kernel. The pixel's intensity is then updated to be the maximum value within this neighborhood, effectively highlighting high-intensity features and mitigating the impact of low-intensity outliers. The code employs a 3x3 kernel for maximum filtering, illustrating the simplicity and effectiveness of the filter in enhancing the visibility of prominent image features. The application of the maximum filter is particularly valuable in scenarios where the emphasis is on accentuating high-intensity structures or regions of interest, making it a versatile tool in image processing applications.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

import numpy as np

def max\_filter(image, kernel\_size=3):

    # Ensure the kernel size is odd

    if kernel\_size % 2 == 0:

        raise ValueError("Kernel size must be an odd number")

    # Pad the image to handle border pixels

    pad\_size = kernel\_size // 2

    padded\_image = cv2.copyMakeBorder(image, pad\_size, pad\_size, pad\_size, pad\_size, cv2.BORDER\_REFLECT)

    # Apply maximum filtering

    result = np.zeros\_like(image)

    for i in range(pad\_size, padded\_image.shape[0] - pad\_size):

        for j in range(pad\_size, padded\_image.shape[1] - pad\_size):

            neighborhood = padded\_image[i-pad\_size:i+pad\_size+1, j-pad\_size:j+pad\_size+1]

            result[i-pad\_size, j-pad\_size] = np.max(neighborhood)

    return result.astype(np.uint8)

input\_image = cv2.imread("me.jpg", cv2.IMREAD\_GRAYSCALE)

# Apply maximum filtering with a 3x3 kernel

output\_image = max\_filter(input\_image, kernel\_size=3)

# Display the results

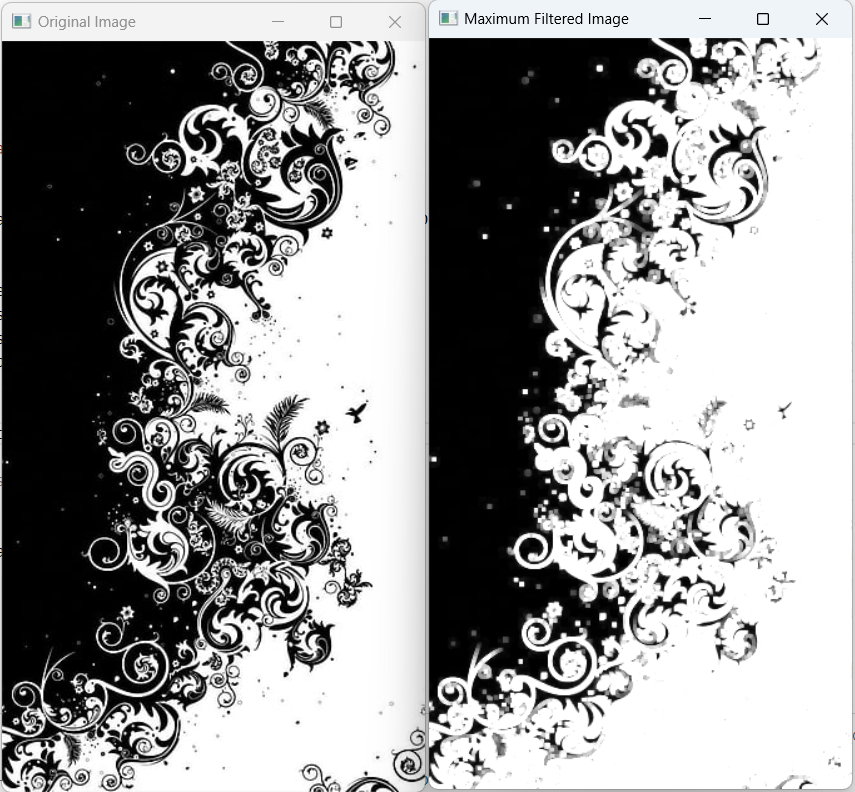
cv2.imshow("Original Image", input\_image)

cv2.imshow("Maximum Filtered Image", output\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to implement maximum filter.

**Task 15**

**Write a program to implement Gaussian blur filter.**

**Theory**

Gaussian blur is a widely used image processing technique employed for smoothing and noise reduction. In this Python program utilizing the OpenCV library, a GaussianBlur function is applied to an input image. The program starts by reading an image file, converting it to grayscale using cv2.cvtColor, and subsequently applying the Gaussian blur using cv2.GaussianBlur. The parameters of the GaussianBlur function include the source image, the kernel size (in this case, (9, 9)), and the standard deviation (set to 0 for automatic calculation based on the kernel size). The resulting blurred image effectively reduces high-frequency noise and sharpens the image by averaging pixel values in the vicinity of each pixel. The provided code showcases the simplicity of implementing Gaussian blur in Python using OpenCV, offering a valuable tool for pre-processing images in various applications, including edge detection and feature extraction. The program also demonstrates the intermediary steps of converting the image to grayscale, providing insights into the different representations of the image during the process.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

img = cv2.imread('me.jpg')

# using cvtcolor we can convert RGB image to GRAY

imggray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# using Gaussian blur function we can blur the image

imgblur = cv2.GaussianBlur(imggray, (9, 9), 0)

# to show the output image

cv2.imshow('output', img)

# to show the blur image

cv2.imshow('blur image', imgblur)

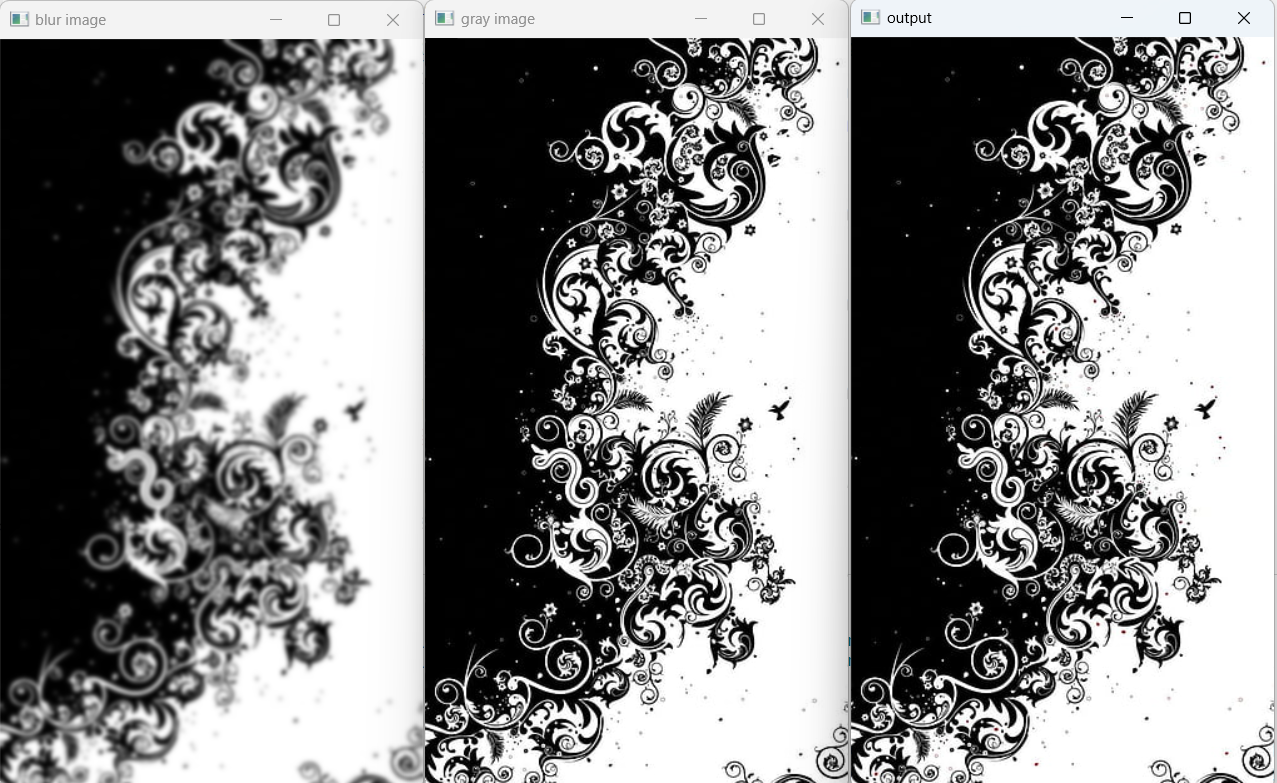
# to show the gray image

cv2.imshow('gray image', imggray)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to implement Gaussian Blur filter.

**Task 16**

**Write a program to implement dilation and erosion of an image.**

**Theory**

Dilation and erosion are fundamental morphological operations in image processing, widely used for shape analysis, noise reduction, and feature extraction. In this Python program utilizing the OpenCV library, an input image is read and subjected to erosion and dilation operations. Erosion involves the process of shrinking the boundaries of objects in an image by iteratively applying a structuring element (in this case, a 5x5 matrix of ones) to erode away the boundaries. Conversely, dilation expands the object boundaries by applying the structuring element to highlight regions of interest. These operations are achieved through cv2.erode() and cv2.dilate() functions, each executed with one iteration in the provided code. The resulting images, img\_erosion and img\_dilation, effectively demonstrate the impact of these operations on the original image. Erosion is beneficial for removing small noise and fine details, while dilation is useful for connecting broken structures and highlighting prominent features. This program showcases the versatility of OpenCV in implementing essential morphological operations for image analysis and manipulation.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

import numpy as np

# Reading the input image

img = cv2.imread('me.jpg', 0)

# Taking a matrix of size 5 as the kernel

kernel = np.ones((5, 5), np.uint8)

img\_erosion = cv2.erode(img, kernel, iterations=1)

img\_dilation = cv2.dilate(img, kernel, iterations=1)

cv2.imshow('Input', img)

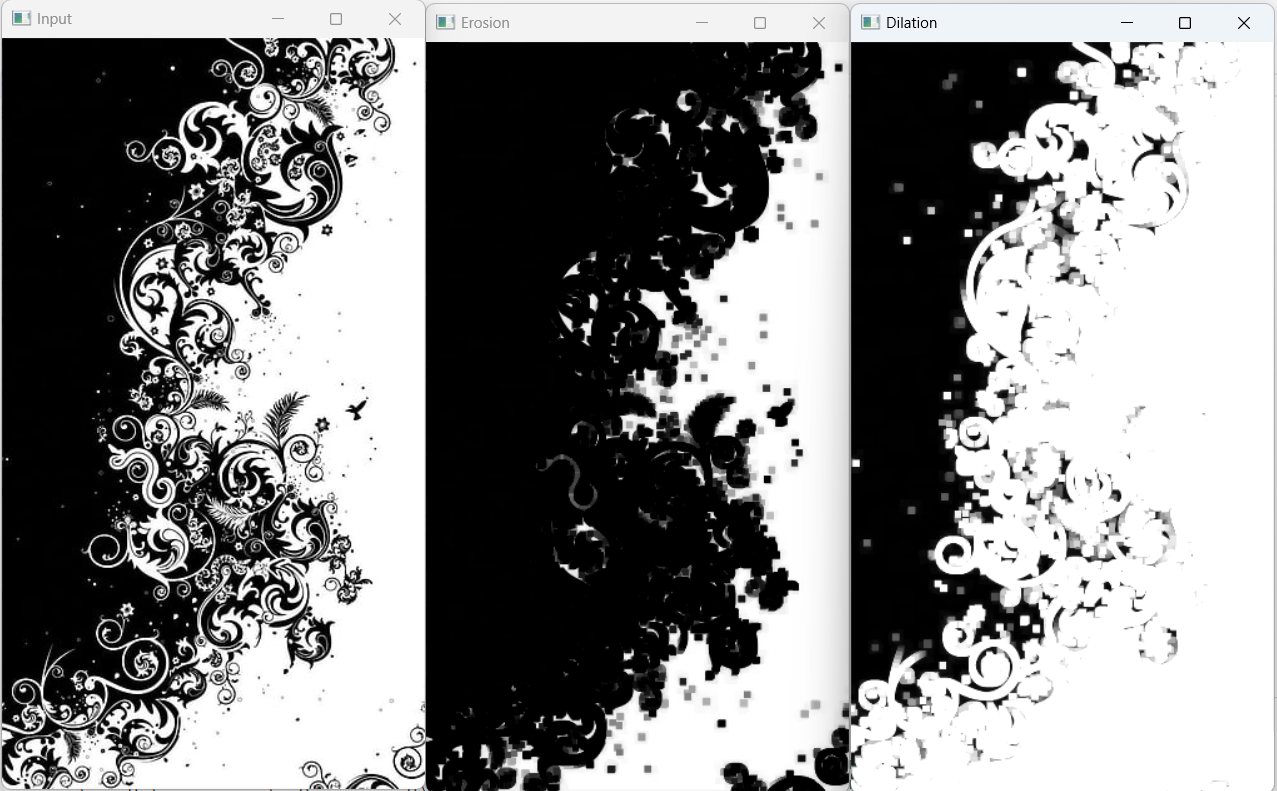
cv2.imshow('Erosion', img\_erosion)

cv2.imshow('Dilation', img\_dilation)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform erosion and dilation on an input image.

**Task 17**

**Write a program to implement opening and closing of an image.**

**Theory**

Opening and closing are essential morphological operations in image processing, commonly used for noise reduction, object separation, and feature extraction. In this Python program employing the OpenCV library, an input grayscale image is subjected to opening and closing operations using a rectangular kernel of size 5x5. The opening operation, implemented with cv2.morphologyEx() and cv2.MORPH\_OPEN, involves the sequential execution of erosion followed by dilation. This operation effectively removes small-scale noise and fine structures while preserving the overall shapes of larger objects. Conversely, the closing operation, implemented with cv2.morphologyEx() and cv2.MORPH\_CLOSE, consists of dilation followed by erosion. Closing is particularly useful in closing gaps between object structures and smoothing object boundaries. The resulting images, opening and closing, showcase the impact of these operations on the original image, emphasizing their role in image enhancement and segmentation. This program illustrates the practical application of opening and closing operations using OpenCV, highlighting their versatility in morphological transformations for effective image processing tasks.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

import numpy as np

# Read the image in grayscale

img = cv2.imread('me.jpg', 0)

kernel = np.ones((5, 5), np.uint8)

# Opening operation

opening = cv2.morphologyEx(img, cv2.MORPH\_OPEN, kernel)

# Closing operation

closing = cv2.morphologyEx(img, cv2.MORPH\_CLOSE, kernel)

# Display the original, opening, and closing images

cv2.imshow('Original Image', img)

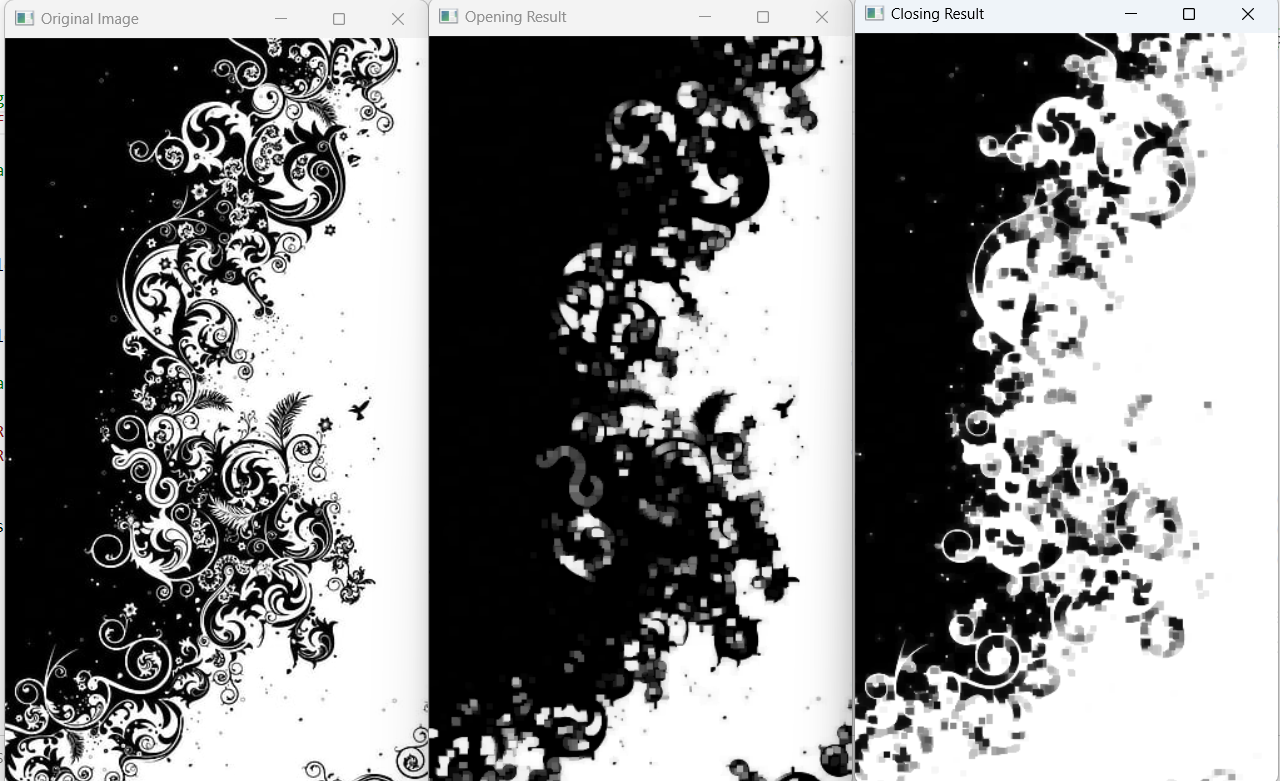
cv2.imshow('Opening Result', opening)

cv2.imshow('Closing Result', closing)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**



**Conclusion**

Hence, in this way, we can use OpenCV library to perform opening and closing of an image.

**Task 18**

**Write a program to calculate the run length code.**

**Theory**

Run-Length Encoding (RLE) is a simple and efficient method for compressing data by representing consecutive repeated elements with a count and the repeated element itself. In the context of image processing, Run-Length Coding can be applied to compress image data by encoding consecutive runs of identical pixel values. In this Python program, a function named 'encode' is defined to perform Run-Length Encoding on a given message string. The function iterates through the message and counts the consecutive occurrences of each character, creating a compact representation of the message by storing the count followed by the character. The provided example encodes a string representing pixel values in an image. This approach can be extended to encode pixel values in an image, reducing redundancy and efficiently representing the data. The simplicity and effectiveness of Run-Length Encoding make it a valuable technique for data compression in various applications, including image processing.Top of Form

Top of FormTop of Form

**Source Code**

def encode(message):

    encoded\_message = ""

    i = 0

    while i <= len(message) - 1:

        count = 1

        ch = message[i]

        j = i

        while j < len(message) - 1:

            if message[j] == message[j + 1]:

                count = count + 1

                j = j + 1

            else:

                break

        encoded\_message = encoded\_message + str(count) + ch

        i = j + 1

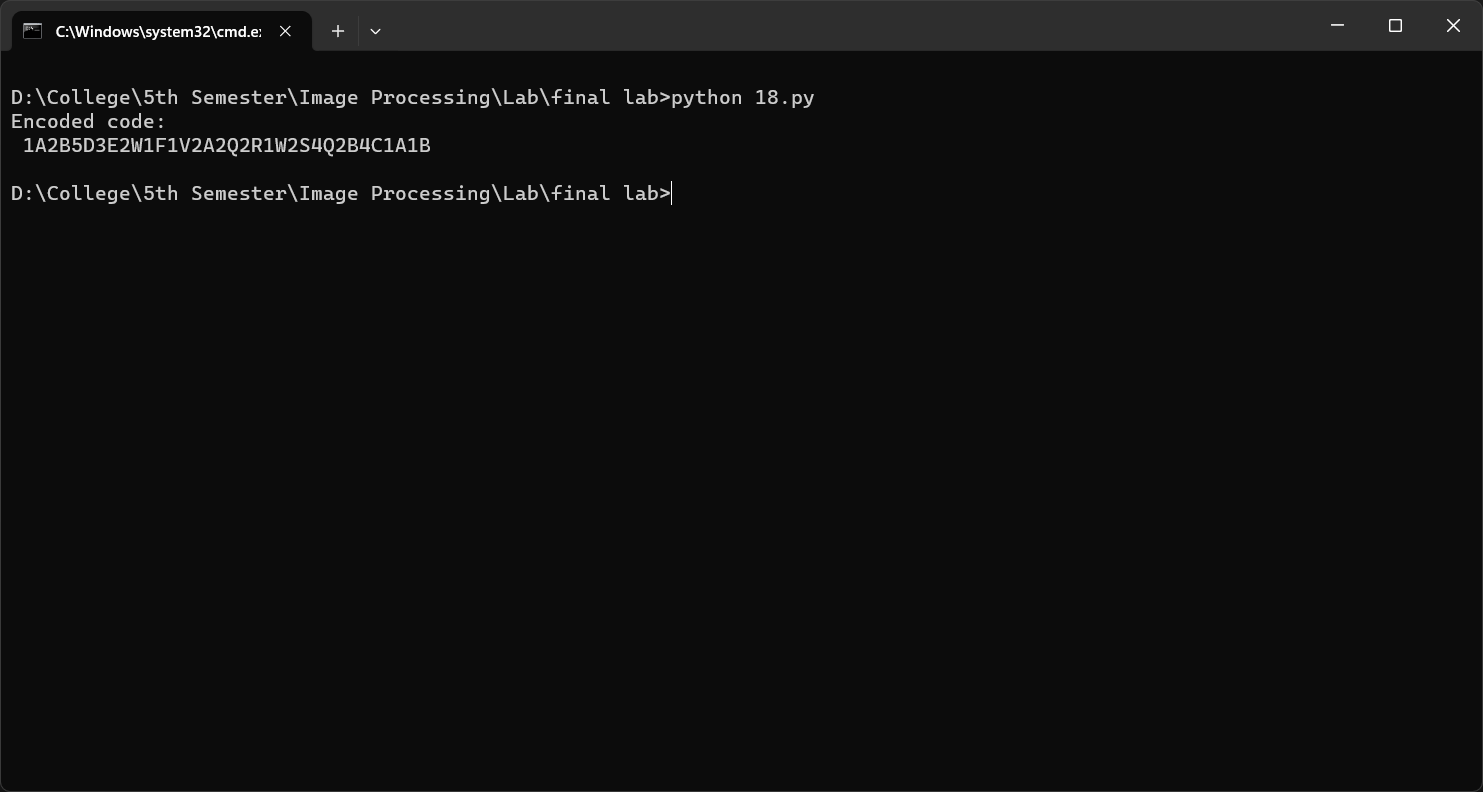
    return encoded\_message

# Provide different values for message and test your program

encoded\_message = encode("ABBDDDDDEEEWWFVAAQQRRWSSQQQQBBCCCCAB")

print("Encoded code:\n", encoded\_message)

**Output**



**Conclusion**

Hence, in this way, we can calculate run length code of a given code.

**Task 19**

**Write a program to calculate the Huffman code.**

**Theory**

Huffman coding is a widely used technique in information theory and image processing for lossless data compression. The fundamental idea is to assign shorter binary codes to more frequently occurring symbols, resulting in an efficient representation of the data. In this Python program, a Huffman tree is constructed for a set of characters and their corresponding frequencies. Each character is represented by a leaf node in the tree, and the binary code for each character is determined by traversing the tree from the root to the leaf. The program utilizes a priority queue to efficiently build the Huffman tree. The resulting Huffman codes are printed, showcasing the binary representation of each character based on their frequencies. This program illustrates the application of Huffman coding for efficient data representation and compression, making it a valuable tool in various image processing and data compression applications.

Top of Form

Top of FormTop of Form

**Source Code**

import heapq

class node:

    def \_\_init\_\_(self, freq, symbol, left=None, right=None):

        self.freq = freq

        self.symbol = symbol

        self.left = left

        self.right = right

        self.huff = ''

    def \_\_lt\_\_(self, nxt):

        return self.freq < nxt.freq

def printNodes(node, val=''):

    newVal = val + str(node.huff)

    if node.left:

        printNodes(node.left, newVal)

    if node.right:

        printNodes(node.right, newVal)

    if not node.left and not node.right:

        print(f"{node.symbol} -> {newVal}")

# characters for huffman tree

chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters

freq = [5, 9, 12, 13, 16, 45]

nodes = []

for x in range(len(chars)):

    heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:

    left = heapq.heappop(nodes)

    right = heapq.heappop(nodes)

    left.huff = 0

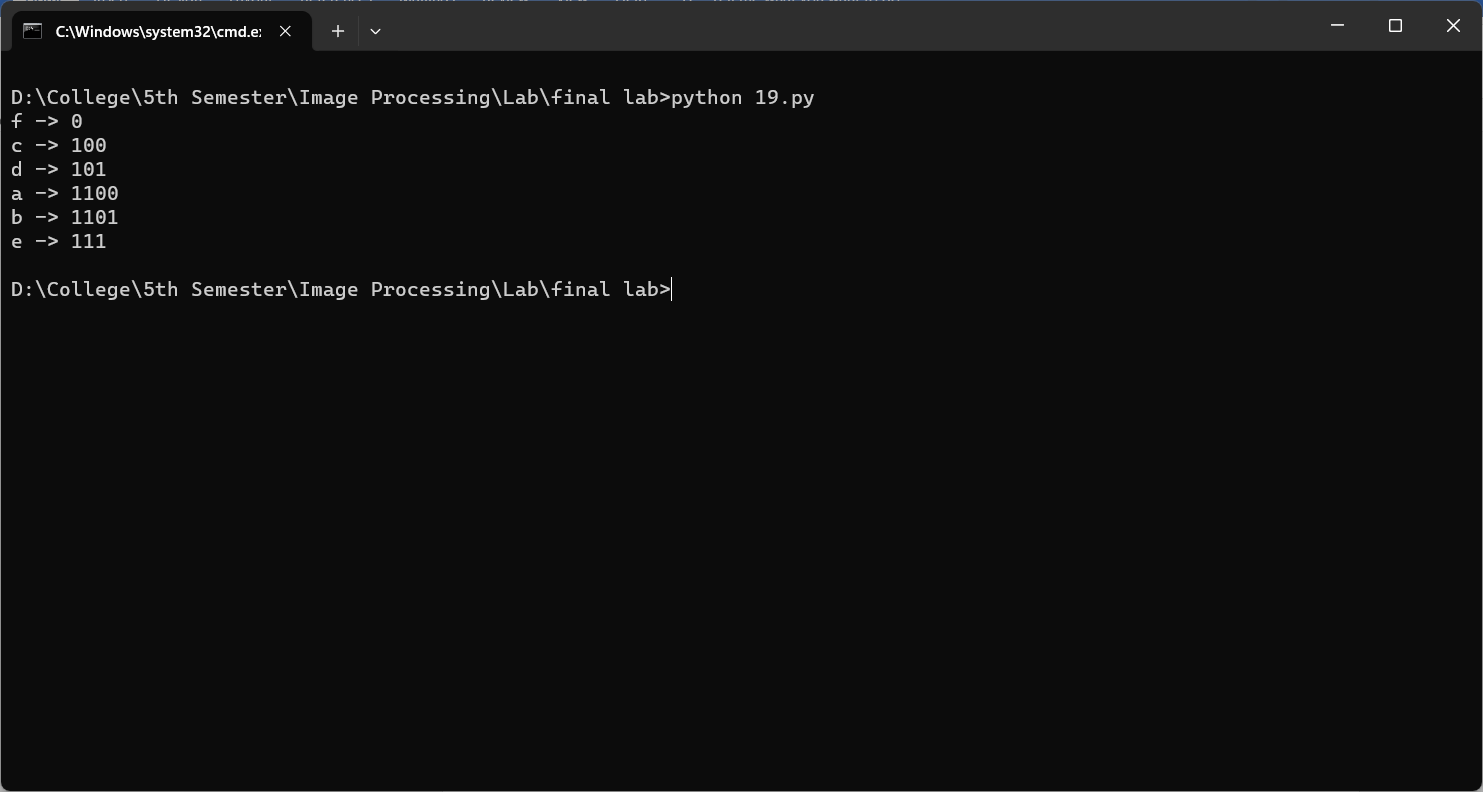
    right.huff = 1

    newNode = node(left.freq + right.freq, left.symbol + right.symbol, left, right)

    heapq.heappush(nodes, newNode)

printNodes(nodes[0])

**Output**



**Conclusion**

Hence, in this way, we can calculate Huffman code of a given code.

**Task 20**

**Write a program to extract edge of an image.**

**Theory**

Edge extraction is a fundamental process in image processing that aims to identify and highlight the boundaries of objects within an image. In this Python program utilizing the OpenCV library, the Canny edge detector is applied to extract edges from a grayscale image. The Canny edge detector is an edge detection algorithm known for its ability to accurately identify edges while minimizing noise. The program first loads the image using cv2.imread() in grayscale mode, then applies the Canny edge detector with specified threshold values (50 and 150 in this case). The resulting edge-detected image is displayed alongside the original image using the matplotlib library. The Canny edge detector works by detecting areas with rapid intensity changes, effectively highlighting edges in the image. This program demonstrates the effectiveness of Canny edge detection in extracting crucial features for subsequent image analysis and computer vision applications.

Top of Form

Top of FormTop of Form

**Source Code**

import cv2

import numpy as np

from matplotlib import pyplot as plt

# Load the image

image = cv2.imread('me.jpg', cv2.IMREAD\_GRAYSCALE)

# Apply the Canny edge detector

edges = cv2.Canny(image, 50, 150)

# Display the original and edge-detected images

plt.subplot(121), plt.imshow(image, cmap='gray')

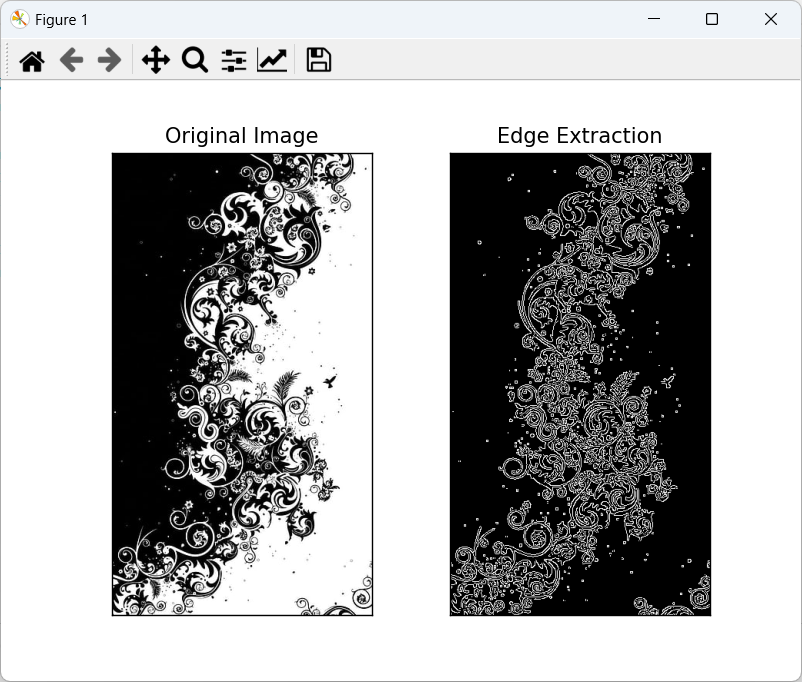
plt.title('Original Image'), plt.xticks([]), plt.yticks([])

plt.subplot(122), plt.imshow(edges, cmap='gray')

plt.title('Edge Extraction'), plt.xticks([]), plt.yticks([])

plt.show()

**Output**



**Conclusion**

Hence, in this way, we can extract edges of an image.