**LAB 10**

**Title: Log Transformation**

**Background:**

Log transformation is a technique that enhances the contrast of images by mapping the pixel intensities to their logarithmic values. It expands the darker pixels while compressing the brighter ones, making details in dark regions more visible. It's a non-linear transformation, meaning it doesn't preserve the original relationships between pixel values.

The general formula for the log transformation of a pixel intensity value s is given by:

s = c \* log(r + 1)

where s = new pixel value in the output image.

c = constant that controls the degree of enhancement (typically chosen to fit the output within the desired range, like 0-255 for 8-bit images).

1 is added to r to avoid undefined values for zero-intensity pixels.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

import numpy as np

img=cv2.imread('me.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:**

It's primarily used for enhancing images with low-intensity details. It's not suitable for all images, as it can overly compress bright regions in some cases. Careful choice of the constant c is crucial for achieving the desired enhancement.

**LAB 1**

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

**Background:**

**Reading an Image:**

The Pillow library provides a convenient method, Image.open(), to read an image from a file. This method returns an Image object, representing the image data.

**Saving an Image:**

Using the Image.save() method, we can save an image to a specified file path. This method allows us to store the image in various formats, such as JPEG, PNG, etc.

**Displaying an Image:**

The Image.show() method is employed to display the image. This function opens the default image viewer associated with the operating system.

**Tools Used:**

Visual Studio Code

**Source Code:**

from PIL import Image

# Read image

input\_image\_path = 'me.jpg'

image = Image.open(input\_image\_path)

# Display image

image.show()

# Save image

output\_image\_path = 'output\_image.jpg'

image.save(output\_image\_path)

print(f"Image saved successfully at {output\_image\_path}")

**Output:**

****

****

**Conclusion:**

In conclusion, this script serves as a practical introduction to basic image processing tasks.

**LAB 4**

**Title: Digital Negative**

**Background:**

A Digital Negative, often referred to as DNG, is a file format designed to store raw image data captured by digital cameras. It serves as a standardized, open format that aims to provide a more universal and flexible option for raw image files. DNG files contain unprocessed sensor data, metadata, and other relevant information, allowing photographers and software developers to work with high-quality images while preserving the original data.

**Tools Used:**

Visual studio code

**Source Code:**

#image negative

import cv2

import matplotlib.pyplot as plt

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

cv2.imshow('input image',img)

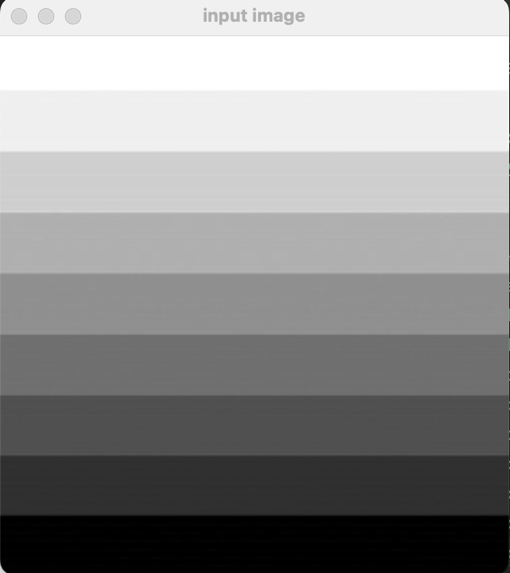
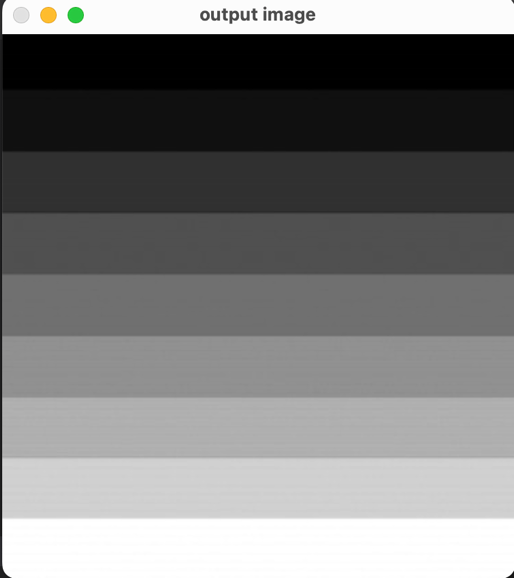
img\_neg=255-img

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

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output:**

** **

**Conclusion:**

It ensures compatibility, preserves unprocessed sensor data, includes metadata, supports lossless compression, and promotes long-term accessibility. DNG serves as a reliable choice for photographers and digital artists, fostering interoperability and future-proofing in the handling of raw image files.

**LAB 5**

**Title: Histogram plot of an image**

**Background:**

A histogram plot of an image is a visual representation of the distribution of pixel intensities. It helps analyze brightness, contrast, and color composition. Peaks and valleys in the histogram indicate areas of high and low intensity. Understanding the histogram is essential for quality assessment, image enhancement, segmentation, thresholding, and color correction in image processing.

**Tools Used:**

Visual studio code

**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\_\_":

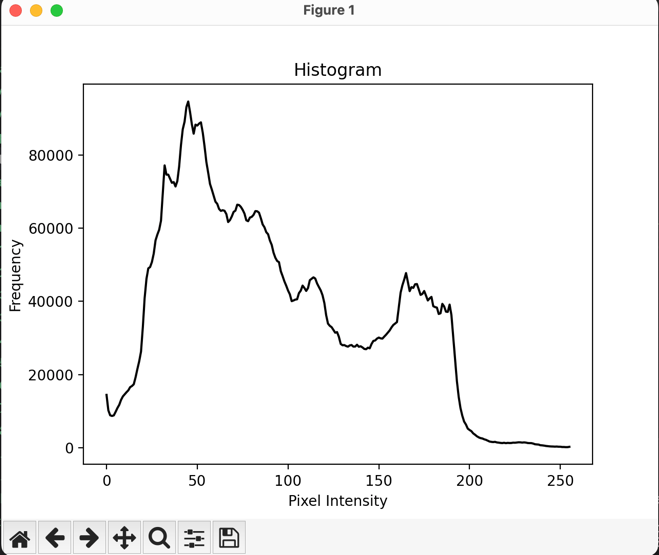
# Replace 'your\_image.jpg' with the path to your image file

image\_path = 'your\_image.jpg'

# Plot the histogram

plot\_histogram(image\_path)

**Output:**

****

**Conclusion:**

In conclusion, the histogram plot of an image serves as a valuable tool for understanding the distribution of pixel intensities within that image. By visualizing the frequency of different intensity levels, one can make informed decisions regarding image quality, enhancement, segmentation, and color correction.

**LAB 6**

**Title: Digital Negative with histogram**

**Background:**

Digital Negative (DNG) is an open standard file format by Adobe for storing raw image data. It ensures interoperability, preserves unprocessed sensor data, and includes metadata. When combined with a histogram analysis, DNG allows for a visual representation of pixel intensity distribution. This provides insights into image quality, dynamic range, and aids in making informed adjustments during post-processing. The DNG format, coupled with histogram analysis, empowers photographers and digital artists to work with high-quality raw image data in a standardized and flexible manner.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

import matplotlib.pyplot as plt

# Read an image

img\_bgr = cv2.imread('me.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()

# 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],

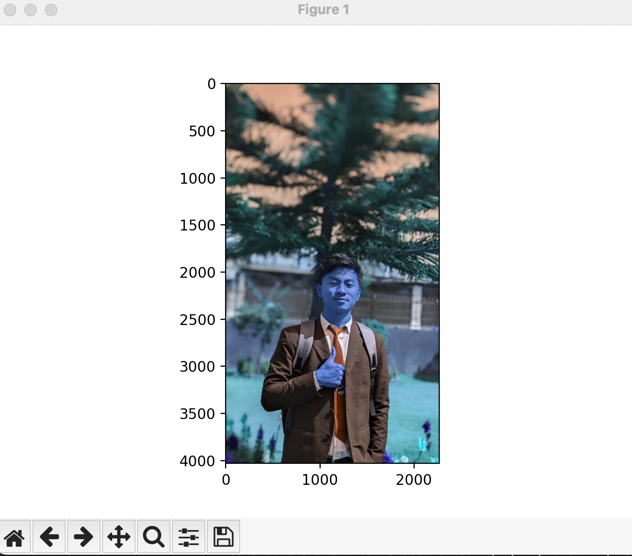
[0, 256])

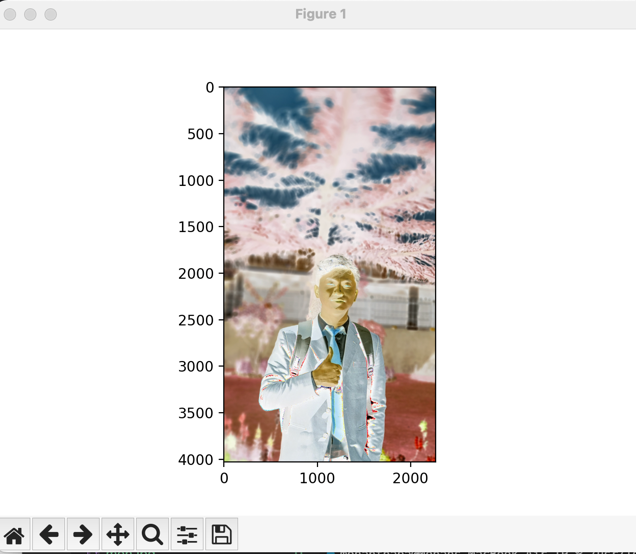
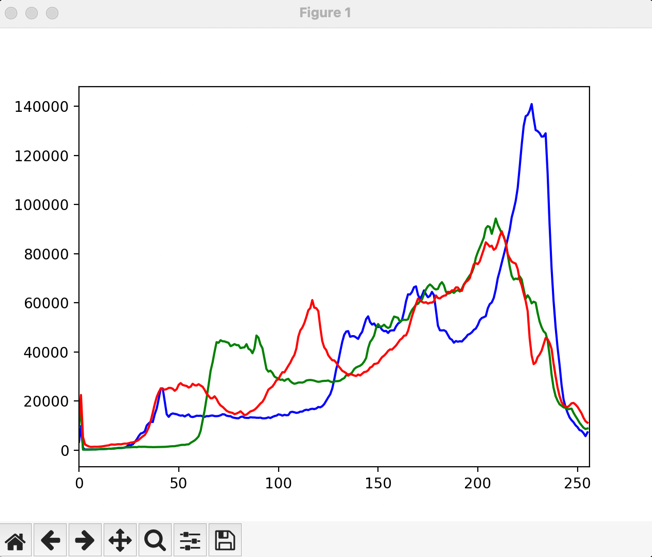
plt.plot(histr, color = col)

plt.xlim([0, 256])

plt.show()

**Ouput:**

****



**Conclusion:**

In conclusion, combining the Digital Negative (DNG) file format with histogram analysis enhances the understanding and utilization of raw image data. DNG's standardized approach ensures accessibility and interoperability, while histogram analysis provides a visual representation of pixel intensity distribution.

**LAB 7**

**Title: Histogram equalization**

**Background:**

Histogram equalization is an image processing technique that enhances contrast by redistributing pixel intensities, resulting in a more uniform histogram. It improves visibility, adapts to image content, and finds applications in medical imaging, computer vision, and photography. The process involves calculating a histogram, creating a Cumulative Distribution Function (CDF), and using it to map original pixel values to new values for contrast enhancement.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

# import Numpy

import numpy as np

# read a image using imread

img = cv2.imread('hi.jpeg',0)

cv2.imshow('input',img)

# creating a Histograms Equalization

# of a image using cv2.equalizeHist()

equ = cv2.equalizeHist(img)

# stacking images side-by-side

#res = np.hstack((img, equ))

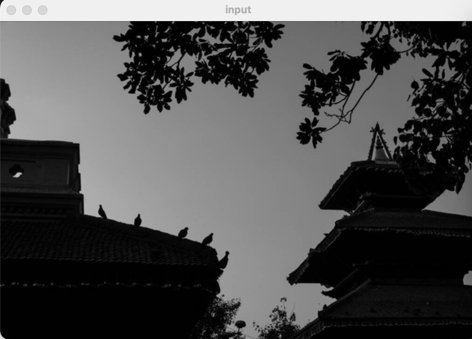
# show image input vs output

cv2.imshow('output',equ)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output:**



**Conclusion:**Histogram equalization is a powerful technique for improving the contrast and brightness of images.

**LAB 8**

**Title: Piecewise linear transformation**

**Background:**

Piecewise linear transformation involves dividing the input space into intervals and applying linear functions to each interval. This technique is useful for approximating non-linear relationships in a simple and flexible way. It finds applications in signal processing, computer graphics, and data analysis, allowing for effective mapping between different value ranges. **Tools Used:**

Visual studio code

**Source Code:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

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

# Initialize the transformed image

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('me.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)

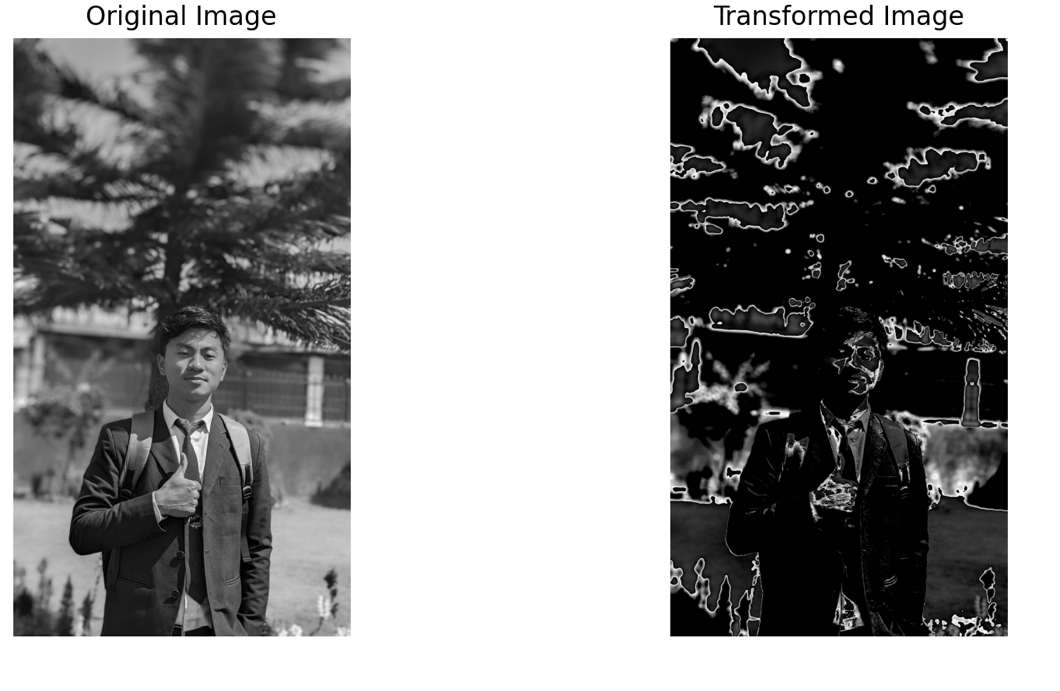
plt.imshow(transformed\_image, cmap='gray')

plt.title('Transformed Image')

plt.axis('off')

plt.show()

**Output:**



**Conclusion:**

Piecewise linear transformation provides a flexible way to enhance or modify the contrast of an image by customizing the transformation in different intensity ranges. It allows for targeted adjustments, making it suitable for scenarios where global transformations may not be sufficient. Experimenting with different breakpoints and slopes enables fine-tuning based on the characteristics of the image and the desired output.

**LAB 9**

**Title: Power Law transformation**

**Background:**

Power Law Transformation, also known as gamma correction or gamma adjustment, is a type of image enhancement technique used to modify the intensity values of an image using a power-law function. The transformation is expressed as:

s=c\*t^ *γ*

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

import numpy as np

img=cv2.imread('me.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:**

Power Law Transformation is a valuable tool for adjusting the contrast of an image by emphasizing certain intensity ranges. It finds applications in medical imaging, satellite imaging, and various other fields where enhancing specific details in images is crucial. Experimentation with different gamma values allows for fine-tuning the transformation based on the characteristics of the image and the desired visual outcome.

**LAB 11**

**Title: implementation of weighted average filtering**

**Background:**

Weighted average filtering is a technique used in image processing to smooth or blur an image while preserving important details. In this technique, each pixel in the output image is a weighted average of its neighboring pixels in the input image. The weights assigned to each neighboring pixel determine the impact of that pixel on the final result.

**Tools Used:**

Visual studio code

**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("me.jpg")

# Apply weighted average filtering with a 3x3 kernel

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

# Display the results

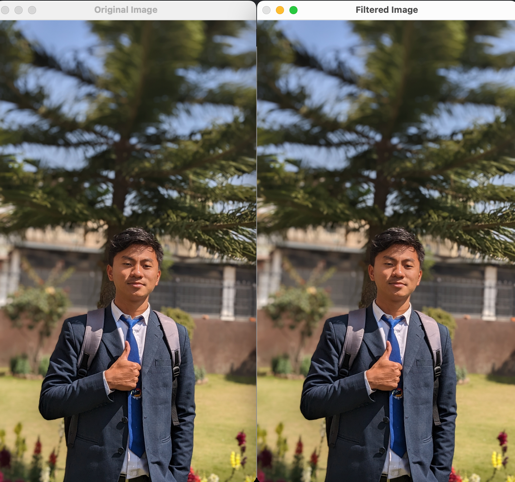
cv2.imshow("Original Image", input\_image)

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

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output:**



**Conclusion:**

Weighted average filtering is a versatile method for image smoothing. By adjusting the kernel size, users can control the level of smoothing applied, striking a balance between noise reduction and preservation of essential details. This technique is widely used in various applications, such as computer vision and image analysis, to improve the quality of images and facilitate subsequent processing tasks.

**LAB 12**

**Title: Implementation of median filtering**

**Background:**

Median filtering is a widely-used technique in image processing for reducing noise while preserving important details. Unlike mean filtering, which uses the average of pixel values in a neighborhood, median filtering replaces each pixel with the median value of its neighboring pixels. This method is particularly effective in mitigating salt-and-pepper noise.

**Tools Used:**

Visual studio code

**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)

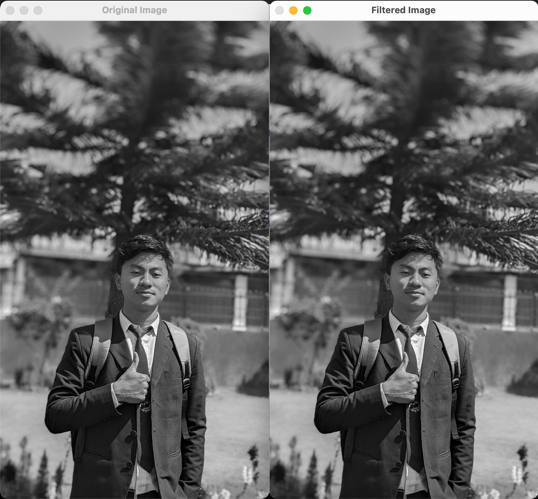
cv2.imshow("Original Image", input\_image)

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

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**

****

**Conclusion:**

Median filtering is a powerful tool for noise reduction in images, especially in scenarios where salt-and-pepper noise is prevalent. This Python implementation demonstrates a straightforward approach to applying median filtering using the NumPy and OpenCV libraries. By replacing each pixel with the median value in its neighborhood, the technique effectively reduces noise while preserving image details. Adjust the kernel size parameter based on the noise characteristics in image to achieve optimal results.

**LAB 13**

**Title: Implementation of minimum filtering**

**Background:**

Minimum filtering, also known as erosion, is a morphological operation in image processing that involves replacing each pixel in an image with the minimum value in its neighborhood. This operation is often used to reduce the size of bright regions or enhance dark regions in an image.

**Tools Used:**

Visual studio code

**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[ipad\_size:i+pad\_size+1,jpad\_size:j+pad\_size+1]

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

return result.astype(np.uint8)

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

# Apply minimum filtering with a 3x3 kernel

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

# Display the results

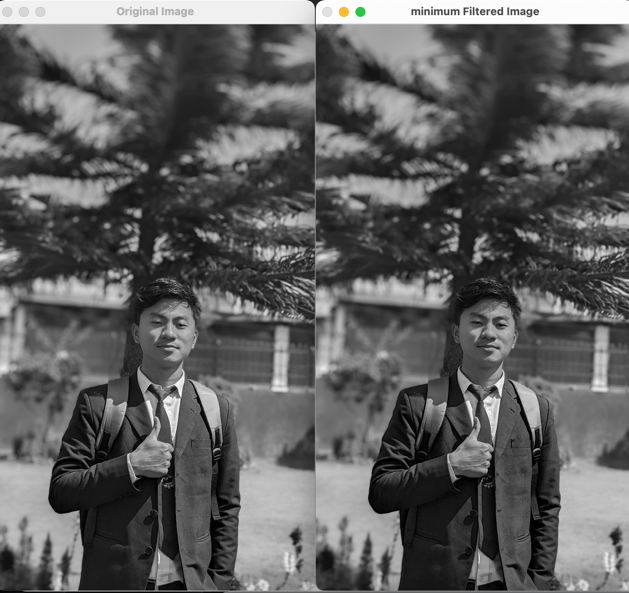
cv2.imshow("Original Image", input\_image)

cv2.imshow("minimum filtered Image", output\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**

****

**Conclusion:**

Minimum filtering is a morphological operation that plays a crucial role in image processing, particularly in size reduction of bright regions or enhancement of dark regions.

**LAB 14**

**Title: Implementation of minimum filtering**

**Background:**

Maximum filtering, also known as dilation, is a fundamental morphological operation in image processing. It involves replacing each pixel in an image with the maximum value within its neighborhood. This operation is commonly used to expand bright regions or emphasize bright features in an image.

**Tools Used:**

Visual studio code

**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:**

Maximum filtering, or dilation, is a vital operation in image processing for expanding bright regions or emphasizing bright features.

**LAB 15**

**Title: Implement Gaussian Blur**

**Background:**

Gaussian Blur is a widely used image processing technique for reducing noise and smoothing an image. It involves convolving the image with a Gaussian kernel, which gives more weight to the central pixels and progressively less weight to the pixels farther away.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

# provide the source file as you want

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

# using cvtcolor we can convert RGB imsge to GRAY

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

#using gaussin blure function we can blure the image

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

# to show the out put image

cv2.imshow( 'output' , img)

# to show the blure image

cv2.imshow('blur image' ,imgblur)

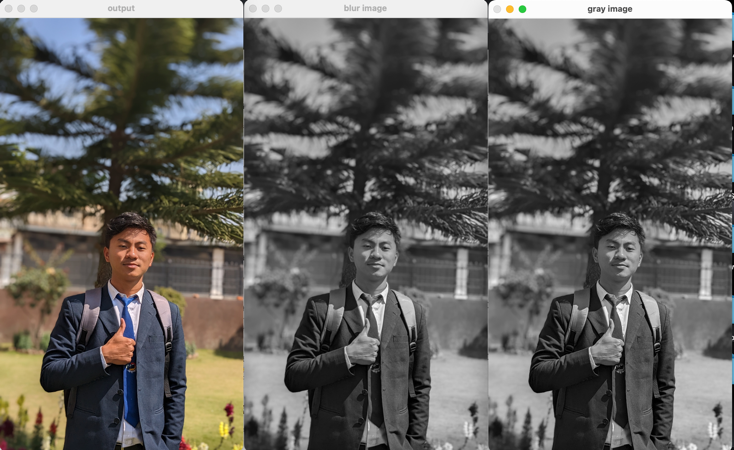
## to show the blur image

cv2.imshow('gray image', imggray)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output**

****

**Conclusion:**

Gaussian Blur is a powerful tool in image processing, commonly used for noise reduction and smoothing.

**LAB 15**

**Title: Implement dilation and erosion of an image**

**Background:**

Dilation and erosion are fundamental morphological operations in image processing. These operations play a crucial role in shape analysis, feature extraction, and noise reduction. Dilation expands the brighter regions in an image, while erosion contracts them. These operations are commonly used in combination to achieve specific image processing goals.

**Tools Used:**

Visual studio code

**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)

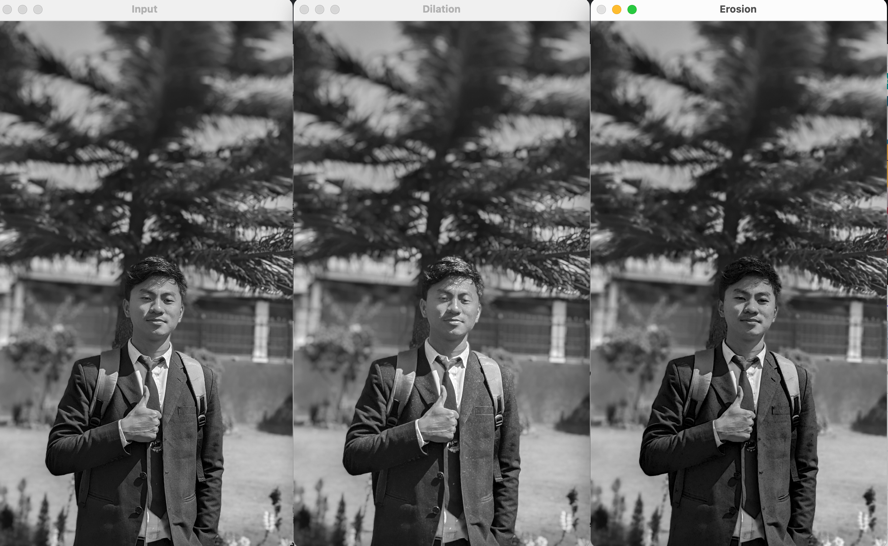
cv2.imshow('Input', img)

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

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

cv2.waitKey(0)

**Output:**

****

**Conclusion:**

Dilation and erosion are essential morphological operations in image processing, commonly used for shape analysis and noise reduction. The choice of kernel size in these operations influences the extent of expansion (dilation) or contraction (erosion).

**LAB 17**

**Title: Implement opening and closing of an image**

**Background:**

Opening and closing are morphological operations in image processing that involve combining erosion and dilation in specific sequences. Opening is an operation that first applies erosion to the image and then dilation. It is useful for removing small objects and brightening dark areas. Closing, on the other hand, performs dilation followed by erosion and is effective in closing small holes or gaps in bright regions.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

import numpy as np

# Read the image in grayscale

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

# Create a rectangular kernel for morphological operations

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)

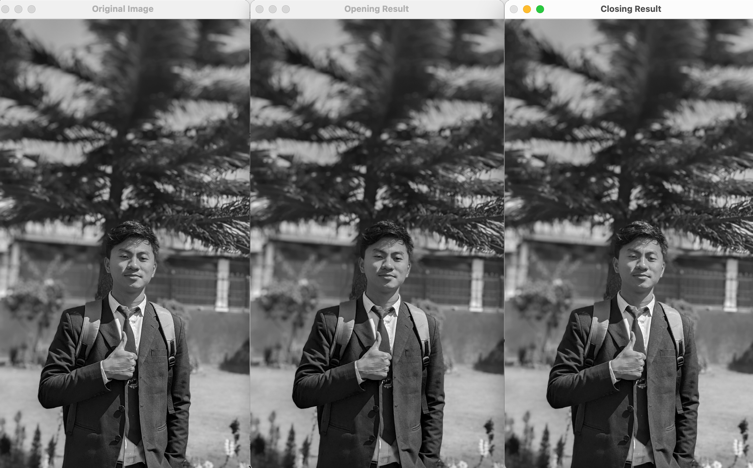
cv2.imshow('Opening Result', opening)

cv2.imshow('Closing Result', closing)

cv2.waitKey(0)

cv2.destroyAllWindows()

**Output:**



**Conclusion:**

Opening and closing operations are valuable in morphological image processing for enhancing or suppressing specific features. Opening is effective in removing small objects and brightening dark areas, while closing is useful for closing small holes or gaps in bright regions.

**LAB 18**

**Title: Run Length coding**

**Background:**

Run-Length Coding (RLC) is a simple data compression technique that represents consecutive identical elements in a sequence with a single element and a count of repetitions. It is effective for compressing data with long runs of the same symbol, making it more space-efficient. RLC is commonly used in scenarios where simplicity is prioritized over achieving maximum compression ratios, such as in image compression and fax transmission.

**Tools Used:**

Visual studio code

**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

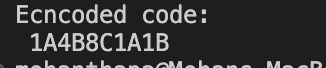
return encoded\_message

#Provide different values for message and test your program

encoded\_message=encode("ABBBBCCCCCCCCAB")

print(“Encoded code:\n”,encoded\_message)

**Output:**

****

**LAB 19**

**Title: Huffman Coding**

**Background:**

Huffman Coding is a widely used data compression technique that assigns shorter codes to more frequent symbols and longer codes to less frequent ones. It builds a binary tree based on symbol frequencies, and the resulting variable-length codes enable efficient compression and decompression. This method is commonly employed in file compression formats like ZIP and JPEG.

**Tools Used:**

Visual studio code

**Source Code:**

import heapq

class node:

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

# frequency of symbol

self.freq = freq

# symbol name (character)

self.symbol = symbol

# node left of current node

self.left = left

# node right of current node

self.right = right

# tree direction (0/1)

self.huff = ''

def \_\_lt\_\_(self, nxt):

return self.freq < nxt.freq

def printNodes(node, val=''):

# huffman code for current node

newVal = val + str(node.huff)

# if node is not an edge node

# then traverse inside it

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]

# list containing unused nodes

nodes = []

# converting characters and frequencies

# into huffman tree nodes

for x in range(len(chars)):

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

while len(nodes) > 1:

# sort all the nodes in ascending order

# based on their frequency

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

# assign directional value to these nodes

left.huff = 0

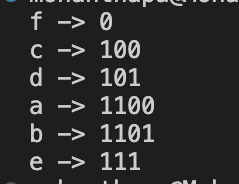
right.huff = 1

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

heapq.heappush(nodes, newNode)

printNodes(nodes[0])

**Output:**

****

**LAB 20**

**Title: Program to extract edge of an Image**

**Background:**

Edge extraction is a fundamental image processing technique used to identify boundaries within an image. It highlights significant changes in intensity or color, which often correspond to object boundaries. One common method for edge detection is the application of convolution masks or filters, such as the Sobel or Canny operators.

**Tools Used:**

Visual studio code

**Source Code:**

import cv2

import numpy as np

from matplotlib import pyplot as plt

# Load the image

image = cv2.imread('your\_image\_path.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:**

Edge extraction plays a crucial role in various computer vision and image processing applications. By highlighting boundaries and distinguishing objects from the background, it enables more advanced analysis and recognition tasks.