ASSIGNMENT - 2

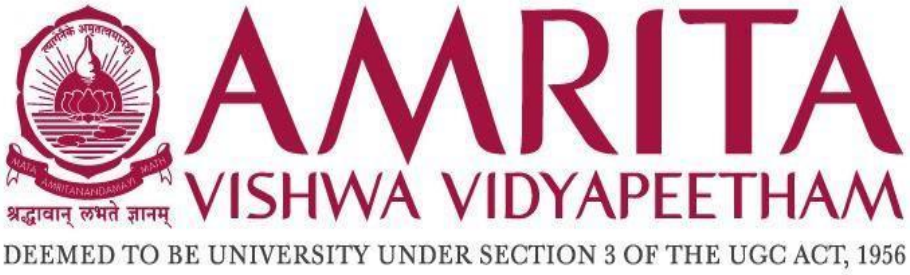
An assignment
Submitted in fulfilment of the
Completion of

the course 19CCE447 Image Processing under the

Faculty of Electronics and Communication

Engineering

By:
SANTOSH [CB.EN.U4CCE20006]



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE CAMPUS

FACULTY IN-CHARGE: SUGUNA G.

ASSIGNMENT 2 – IMAGE ANALYSIS AND OPERATIONS

AIM:

To explore different methods for improving the visual quality of digital images.

SOFTWARE:

Product version: Anaconda3 2021.11
Spyder version: 5.1.5
Python version: 3.9.7 64-bit
Qt version: 5.9.7
PyQt5 version: 5.9.2
Operating System: Windows 10

THEORY AND PYTHON CODES:

I. Edge Detection – Sobel Edge Detection and Canny Edge Detection in Image Processing

Edge detection is a fundamental task in image processing that aims to identify the boundaries between different regions or objects within an image. It is widely used in various applications, such as object recognition, image segmentation, and feature extraction. Two popular methods for edge detection are the Sobel edge detection and the Canny edge detection.

The Sobel edge detection is a gradient-based method that calculates the gradient magnitude of an image to identify edges. It operates by convolving the image with a set of predefined kernels in the horizontal and vertical directions. These kernels highlight the intensity changes in the image along the x and y axes. The gradient magnitude is then computed by combining the horizontal and vertical gradients. Higher gradient magnitudes indicate stronger edges in the image. The Sobel edge detection is relatively simple and efficient but may produce relatively thick edges and be sensitive to noise.

On the other hand, the Canny edge detection is a more advanced and widely used technique that provides better edge localization and noise robustness. It consists of multiple stages. First, the image is smoothed using a Gaussian filter to reduce noise. Then, the gradients of the smoothed image are calculated using Sobel operators. Next, non-maximum suppression is applied to thin out the detected edges by preserving only the local maxima in the gradient direction. Finally, a thresholding and hysteresis process is employed to determine the final set of edges. The thresholding step helps to discard weak edges, while the hysteresis step links strong edges to adjacent weak edges, thereby forming continuous edge contours.

In summary, both Sobel edge detection and Canny edge detection are popular techniques for detecting edges in images. The Sobel method provides a simple and efficient approach, while the Canny method offers more advanced features and better performance in terms of edge localization and noise suppression. The choice of method depends on the specific requirements of the application and the desired trade-off between simplicity and accuracy.

Q. Write a program in Python to implement Sobel Edge Detection and to observe its effect on the image.

This Python program demonstrates the application of Sobel edge detection on an image. It uses the OpenCV library to read the original image in grayscale. The image is then blurred using a Gaussian blur filter for improved edge detection. The Sobel operator is applied to compute the gradients along the x-axis, y-axis, and both axes combined. The resulting Sobel edge detection images are displayed using matplotlib's subplots. Each subplot represents a different Sobel edge detection result. This program helps visualize and identify the edges present in the image using the Sobel technique.

```
import cv2
import matplotlib.pyplot as plt

# Read the original image in grayscale
img_gray = cv2.imread('Lighthouse.jpg', cv2.IMREAD_GRAYSCALE)
if img_gray is None:
print("Error: Could not read image file")
else:
# Display original image
plt.imshow(cv2.cvtColor(img_gray, cv2.COLOR_BGR2RGB))  # Convert BGR to RGB
colorspace for display in matplotlib

plt.title('Original')
plt.axis('off')
plt.show()

# Blur the image for better edge detection
img_blur = cv2.GaussianBlur(img_gray, (3, 3), 0)

# Sobel Edge Detection
sobelx = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5)  #
Sobel Edge Detection on the X axis

sobely = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5)  #
Sobel Edge Detection on the Y axis

sobelxy = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=5)  #
Combined X and Y Sobel Edge Detection



# Display Sobel Edge Detection Images using subplots
fig, axes = plt.subplots(nrows=1, ncols=3)  # Create a 1x3 grid of plots
```

```
titles = ['Sobel X', 'Sobel Y', 'Sobel X Y using Sobel() function']
images = [sobelx, sobely, sobelxy]

for i, ax in enumerate(axes.flat):
ax.imshow(images[i], cmap='gray')  # Display each sobel image on a different
subplot
ax.set_title(titles[i])
ax.axis('off')
plt.tight_layout()  # Adjust the spacing between subplots for better
visibility
plt.show()
```

Figure 1. Original Image

Figure 2. Original Image in Greyscale

Figure 3. Sobel Edge Detection Images

Q. Write a program in Python to implement Canny Edge Detection and to observe its effect on the image.

```
import cv2
import matplotlib.pyplot as plt

# Read the original image in grayscale
img_gray = cv2.imread('Lighthouse.jpg', cv2.IMREAD_GRAYSCALE)

if img_gray is None:
    print("Error: Could not read image file")
else:
    # Display original image
    plt.imshow(cv2.cvtColor(img_gray, cv2.COLOR_BGR2RGB))  # Convert BGR to
RGB colorspace for display in matplotlib
    plt.title('Original')
    plt.axis('off')
    plt.show()

    # Blur the image for better edge detection
    img_blur = cv2.GaussianBlur(img_gray, (3, 3), 0)

# Canny Edge Detection
    edges = cv2.Canny(image=img_blur, threshold1=100, threshold2=200)  #
Canny Edge Detection using img_blur
    # Display Canny Edge Detection Image
    plt.imshow(edges, cmap='gray')
    plt.title('Canny Edge Detection')
    plt.axis('off')
    plt.show()
```

Figure 4. Original Image

Figure 5. Original Image in Greyscale

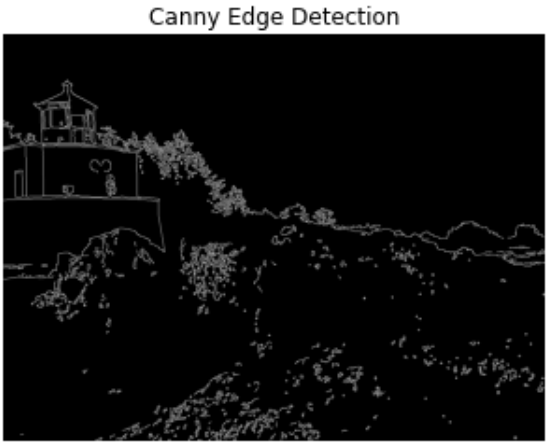


Figure 6. Canny Edge Detection

II. Image Moments

Image moments are mathematical descriptors used in image analysis to extract various features and properties of an image. Moments provide quantitative measures of the shape, intensity distribution, and spatial characteristics of objects within an image. They are widely used in computer vision, pattern recognition, and image processing applications.

The concept of moments is derived from mathematical moments in statistics. In image analysis, moments are calculated by summing the pixel intensities raised to a certain power over the entire image or a specific region of interest. These moments can reveal important information about the image content and can be used to perform tasks such as object recognition, shape analysis, and image matching.

Image moments can be calculated for the entire image or specific regions of interest, such as objects or contours within the image. The moments can then be used to extract features such as area, centroid coordinates, orientation, compactness, and higher-order shape characteristics.

In summary, image moments are mathematical descriptors that provide quantitative information about an image's shape, intensity distribution, and spatial properties. They are widely used in various image analysis applications and can be calculated using raw moments, central moments, and normalized moments. Image moments enable the extraction of valuable features for tasks like object recognition, shape analysis, and image matching.

Q. Write a program in Python to implement Image Moments and to observe its effect on the image.

```
import cv2
import matplotlib.pyplot as plt

# Read the original image in grayscale
img_gray = cv2.imread('Lighthouse.jpg', cv2.IMREAD_GRAYSCALE)

# apply thresholding on gray image
ret, thresh = cv2.threshold(img_gray, 150, 255, 0)

# Find the contours in the image
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# Print the number of detected contours
print("Number of Contours detected:", len(contours))

# Draw the first contour
plt.imshow(img_gray, cmap='gray')
plt.title('Contour: 1') # Set the title of the plot
plt.plot(contours[0][:, 0, 0], contours[0][:, 0, 1], '-g', linewidth=2) #
Plot the first contour
plt.show() # Display the plot

# print the moments of the first contour
cnt = contours[0] # Get the first contour
M = cv2.moments(cnt) # Compute the moments of the first contour
print("Moments of first contour:", M) # Print the moments
```

Figure 6. Original Image

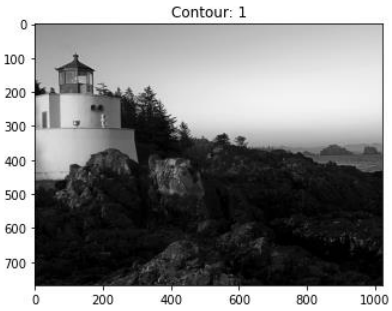


Figure 7. Draw the First Contour

```
Number of Contours detected: 760
Moments of first contour: {'m00': 0.0, 'm10': 0.0, 'm01': 0.0, 'm20': 0.0, 'm11': 0.0, 'm02': 0.0,
'm30': 0.0, 'm21': 0.0, 'm12': 0.0, 'm03': 0.0, 'mu20': 0.0, 'mu11': 0.0, 'mu02': 0.0, 'mu30': 0.0,
'mu21': 0.0, 'mu12': 0.0, 'mu03': 0.0, 'nu20': 0.0, 'nu11': 0.0, 'nu02': 0.0, 'nu30': 0.0, 'nu21':
0.0, 'nu12': 0.0, 'nu03': 0.0}
```

Figure 8. Image Moments Print Statements

III. Image Operations – Erosion, Dilation, Denoising, Histogram, and Borders in Image Processing

Image operations play a vital role in image processing, allowing us to enhance and manipulate images in various ways. Some commonly used image operations: erosion, dilation, denoising, histogram, and borders.

1. ***Erosion:***
Erosion is a morphological operation that shrinks the boundaries of objects in an image. It achieves this by sliding a structuring element over the image and replacing each pixel with the minimum value within the neighbourhood defined by the structuring element. Erosion helps in removing small objects, smoothing boundaries, and separating connected components.

2. ***Dilation:***
Dilation is the opposite of erosion and is also a morphological operation. It expands the boundaries of objects in an image by replacing each pixel with the maximum value within the neighbourhood defined by the structuring element. Dilation helps in filling gaps, joining broken parts of objects, and enlarging structures.

3. ***Denoising:***
Denoising is the process of reducing or removing noise from an image. Various denoising techniques exist, including spatial filters like the averaging filter and median filter mentioned earlier. Denoising methods aim to preserve important image features while reducing unwanted noise, enhancing image quality, and improving subsequent analysis or visualization tasks.

4. ***Histogram:***
A histogram is a graphical representation of the distribution of pixel intensities in an image. It provides valuable insights into the image's contrast, brightness, and dynamic range. Analysing the histogram helps in understanding the image's characteristics and can guide further processing steps, such as contrast adjustment or thresholding.

5. ***Borders:***
Borders, also known as image borders or image edges, represent the transition between different regions or objects within an image. Identifying and extracting borders can be crucial for various image analysis tasks, such as object recognition, segmentation, and feature extraction. Techniques like edge detection algorithms, such as the Canny edge detector or Sobel operator, are commonly used for border detection.

Each of these image operations serves a specific purpose and can be applied individually or in combination with other techniques to achieve the desired image processing goals.

In summary, image operations like erosion, dilation, denoising, histogram analysis, and border detection form the foundation of many image processing tasks. Understanding and applying these operations appropriately can lead to improved image quality, object extraction, and subsequent analysis and interpretation.

Q. Write a program in Python to implement Erosion, Dilation, Denoising, Histogram, and Borders in Image Processing.

```
import numpy as np
from concurrent import futures
from matplotlib import pyplot as plt
from skimage import io
import cv2

# Read the image
def read_image(filename):
    return io.imread(filename)
image = read_image("Lighthouse.jpg")
# Create a kernel
def create_kernel(size):
    return np.ones((size, size), np.uint8)
kernel = create_kernel(5)
# Erode the image with kernel
def erode_image(image, kernel):
    return cv2.erode(image, kernel)
# Dilate the image with kernel
def dilate_image(image, kernel, iterations=1):
    return cv2.dilate(image, kernel, iterations=iterations)
# Open the image with kernel
def open_image(image, kernel):
    return cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
# Denoise the image using fastNlMeansDenoisingColored() method
def denoise_image(image, denoising_parameters):
    return cv2.fastNlMeansDenoisingColored(image, None,
*denoising_parameters)
# Add a border to the image using copyMakeBorder() method
def add_border(image, top, bottom, left, right, border_type, value):
    return cv2.copyMakeBorder(image, top, bottom, left, right, border_type,
None, value=value)
# Convert the image to grayscale and calculate its histogram
```

```
def convert_to_grayscale(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Plot the histogram of the color image
def plot_histogram(image):
    histr = cv2.calcHist([image], [0], None, [256], [0, 256])
    plt.plot(histr)
    plt.xlabel('Pixel Value')
    plt.ylabel('Frequency')
# Plot the individual color histograms
def plot_color_histograms(image):
    color = ('b', 'g', 'r')
    for i, col in enumerate(color):
        hist = cv2.calcHist([image], [i], None, [256], [0, 256])
        plt.plot(hist, color=col)
        plt.xlim([0, 256])
    plt.xlabel('Pixel Value')
    plt.ylabel('Frequency')
# Define parameters for denoising_image function
denoising_parameters = (15, 8, 8, 15)
# Execute functions concurrently
with futures.ThreadPoolExecutor() as executor:
    image_erode = executor.submit(erode_image, image, kernel)
    image_dilation = executor.submit(dilate_image, image, kernel)
    image_erode_dilate = executor.submit(open_image, image, kernel)
    denoised_image = executor.submit(denoise_image, image,
denoising_parameters)
# Plot original image and processed images
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0, 0].imshow(image)
axs[0, 0].set_xlabel('X-axis')
axs[0, 0].set_ylabel('Y-axis')
axs[0, 0].set_title('Original Image')
axs[0, 1].imshow(image_erode.result())
axs[0, 1].set_xlabel('X-axis')
axs[0, 1].set_ylabel('Y-axis')
axs[0, 1].set_title('Eroded Image')
axs[1, 0].imshow(image_dilation.result())
axs[1, 0].set_xlabel('X-axis')
axs[1, 0].set_ylabel('Y-axis')
```

```
axs[1, 0].set_title('Dilated Image')
axs[1, 1].imshow(image_erode_dilate.result())
axs[1, 1].set_xlabel('X-axis')
axs[1, 1].set_ylabel('Y-axis')
axs[1, 1].set_title('Eroded and Dilated Image')
plt.show()
# Plot images with border
image_border1 = add_border(image, 25, 25, 10, 10, cv2.BORDER_CONSTANT, 0)
plt.imshow(image_border1)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Image with Black Border')
plt.show()
image_border2 = add_border(image, 250, 250, 250, 250, cv2.BORDER_REFLECT,
value=None)
plt.imshow(image_border2)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Image with Mirrored Border')
plt.show()
image_border3 = add_border(image, 300, 250, 100, 50, cv2.BORDER_REFLECT,
value=None)
plt.imshow(image_border3)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Image with Modified Mirrored Border')
plt.show()
# Plot denoised image
plt.imshow(denoised_image.result())
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Denoised Image')
plt.show()
# Plot histogram of color image
plt.subplot(1,2,1)
plot_histogram(image)
plt.subplot(1,2,2)
plt.hist(image.ravel(), 256, [0, 256])
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
```

```
plt.suptitle('Histogram of the Color Image')
plt.tight_layout()
plt.show()
# Plot grayscale image and its histogram
grey_image = convert_to_grayscale(image)
histogram = cv2.calcHist([grey_image], [0], None, [256], [0, 256])
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(grey_image, cmap='gray')
axs[0].set_xlabel('X-axis')
axs[0].set_ylabel('Y-axis')
axs[0].set_title('Grayscale Image')
axs[1].plot(histogram, color='k')
axs[1].set_xlabel('Pixel Value')
axs[1].set_ylabel('Frequency')
axs[1].set_title('Histogram of Grayscale Image')
plt.tight_layout()
plt.show()
# Plot individual color histograms
plot_color_histograms(image)
plt.title('Individual Color Histograms')
plt.show()
```

Figure 9. Original Image

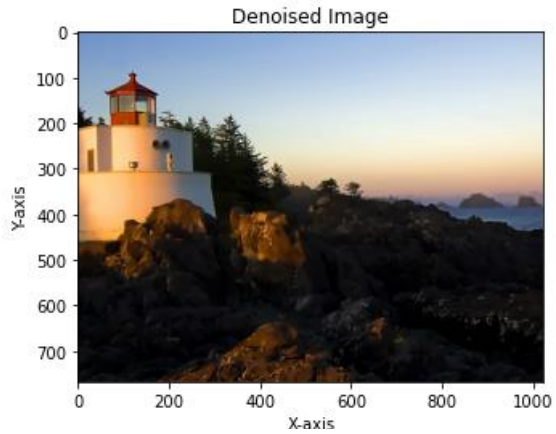


Figure 10. Denoised Image

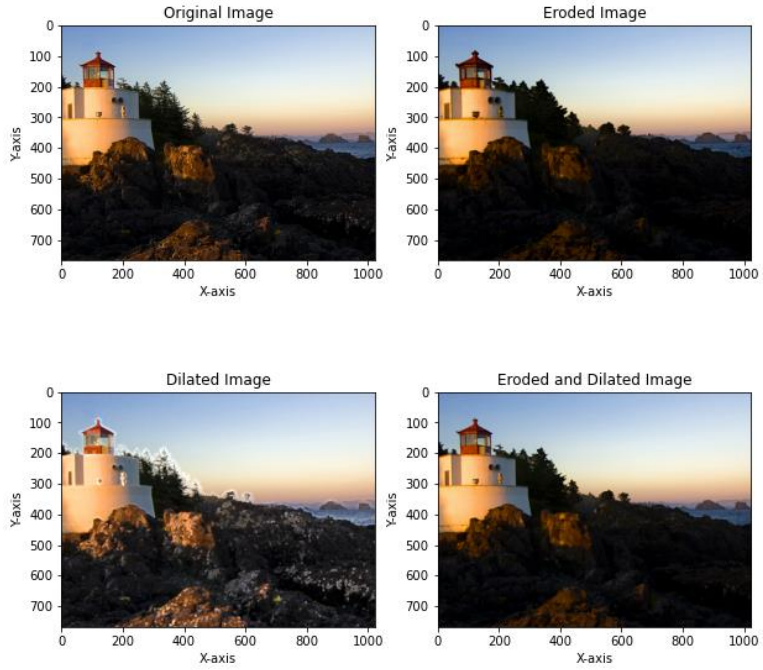


Figure 11. Eroded & Dilated Image

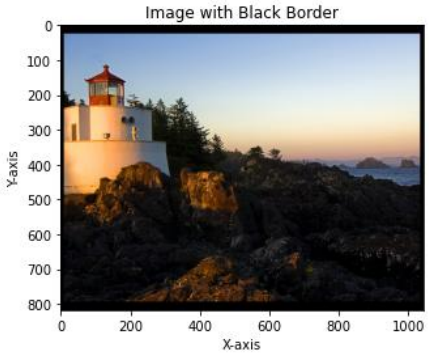


Figure 12. Image with Black Border

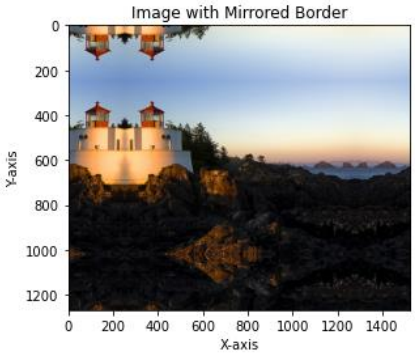


Figure 13. Image with Mirrored Border

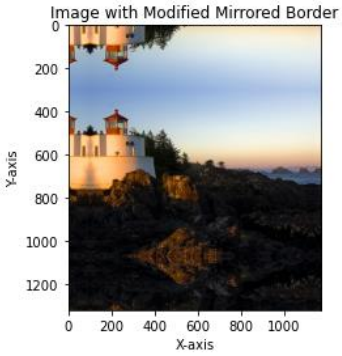


Figure 14. Image with Modified Mirrored Border

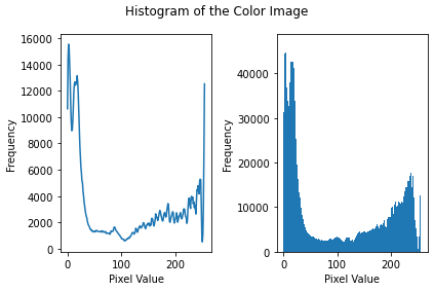


Figure 15. Histogram of The Color Image

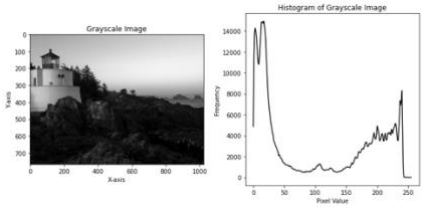


Figure 16. (a) Grayscale Image (b) Histogram of Grayscale Image

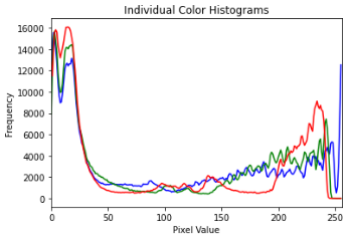


Figure 17. Individual Color Histograms

IV. Segmentation - Active Contour Segmentation, Chan-Vese Segmentation, Felzenswalb Segmentations, SLIC Segmentation in Image Processing

Segmentation is a fundamental task in image processing that involves dividing an image into meaningful regions or segments. Active contour segmentation, Chan-Vese segmentation, Felzenszwalb segmentation, and SLIC (Simple Linear Iterative Clustering) segmentation are popular methods used for image segmentation.

Active contour segmentation, also known as snake-based segmentation, is an iterative technique that utilizes deformable contours to delineate object boundaries in an image. It involves defining an initial contour close to the object of interest and iteratively adjusting the contour based on certain energy criteria, such as minimizing the gradient or edge information along the contour. Active contour segmentation is effective for segmenting objects with well-defined boundaries and can handle complex shapes.

Chan-Vese segmentation is a variational segmentation method that aims to partition an image into regions based on pixel intensity similarities. It utilizes a level set formulation, where a level set function evolves to separate the regions. The energy functional in Chan-Vese segmentation incorporates both region-based terms, which measure the difference in pixel intensities within a region, and boundary-based terms, which control the smoothness of the region boundaries. By minimizing the energy functional, the segmentation algorithm identifies regions with distinct intensities.

Felzenszwalb segmentation is a bottom-up region-based segmentation algorithm that groups pixels into regions based on a similarity measure. It operates by repeatedly merging adjacent regions that satisfy a similarity criterion. The algorithm considers both color similarity and spatial proximity of pixels when merging regions, resulting in segmentations that adhere to object boundaries while considering local variations in intensity. Felzenszwalb segmentation is efficient and capable of producing oversegmentations suitable for subsequent object recognition tasks.

SLIC segmentation is a superpixel-based segmentation method that clusters pixels into compact and uniform groups called superpixels. It combines the benefits of both region-based and boundary-based segmentation. SLIC operates by first initializing superpixel centers on a regular grid and then iteratively adjusting their positions based on color similarity and spatial proximity. By enforcing compactness and uniformity constraints, SLIC generates superpixels that represent meaningful image regions and can be used for subsequent image analysis tasks.

In summary, active contour segmentation, Chan-Vese segmentation, Felzenszwalb segmentation, and SLIC segmentation are widely used techniques for image segmentation. Each method employs different principles and algorithms to partition images into coherent regions, allowing for various applications in image analysis, computer vision, and pattern recognition.

Q. Write a program in Python to implement *Segmentation* - Active Contour Segmentation, Chan-Vese Segmentation, Felzenswalb Segmentations, SLIC Segmentation in Image Processing.

This Python program showcases various image segmentation techniques using the scikit-image library. It includes examples of active contour segmentation, Chan-Vese segmentation, SLIC segmentation, and Felzenszwalb segmentation.

For active contour segmentation, the program applies a Gaussian filter to reduce noise and uses the active contour function to segment the image, emphasizing the boundaries of a detected face.

In Chan-Vese segmentation, the program iteratively segments a grayscale image using the Chan-Vese algorithm and displays the original image, intermediate segmented images, and the final level set.

SLIC segmentation divides an image into superpixels, and the program visualizes the original image along with the marked boundaries of the segmented regions.

Felzenszwalb segmentation segments the image based on the Felzenszwalb algorithm, and the program displays the original image with the marked boundaries of the segmented regions.

These examples provide a concise demonstration of different image segmentation techniques, enabling users to observe and analyze the results.

```
# Importing necessary Libraries
from skimage import data, filters
from skimage.color import rgb2gray, rgb2hsv, label2rgb
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import gaussian
from skimage.segmentation import active_contour, chan_vese, slic,
mark_boundaries, felzenszwalb
from skimage.data import astronaut

# Setting the plot size to 15,15
plt.figure(figsize=(15, 15))

# Sample Image of scikit-image package
coffee = data.coffee()
plt.subplot(1, 2, 1)
```

```
# Displaying the sample image
plt.imshow(coffee)
plt.title('Original RGB Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Converting RGB image to Monochrome
gray_coffee = rgb2gray(coffee)
plt.subplot(1, 2, 2)

# Displaying the sample image - Monochrome Format
plt.imshow(gray_coffee, cmap="gray")
plt.title('Monochrome Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Setting the plot size to 15,15
plt.figure(figsize=(15, 15))

# Sample Image of scikit-image package
coffee = data.coffee()
plt.subplot(1, 2, 1)

# Displaying the sample image
plt.imshow(coffee)
plt.title('Original RGB Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Converting RGB Image to HSV Image
hsv_coffee = rgb2hsv(coffee)
plt.subplot(1, 2, 2)

# Displaying the sample image - HSV Format
hsv_coffee_colorbar = plt.imshow(hsv_coffee)
plt.title('HSV Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Adjusting colorbar to fit the size of the image
```

```
plt.colorbar(hsv_coffee_colorbar, fraction=0.046, pad=0.04)

# Displaying the sample image - Monochrome Format
# Sample Image of scikit-image package
coffee = data.coffee()
gray_coffee = rgb2gray(coffee)

# Setting the plot size to 15,15
plt.figure(figsize=(15, 15))

for i in range(10):

    # Iterating different thresholds
    binarized_gray = (gray_coffee > i*0.1)*1
    plt.subplot(5,2,i+1)

    # Rounding of the threshold value to 1 decimal point
    plt.title("Threshold: >"+str(round(i*0.1,1)))
    plt.xlabel('X Label')
    plt.ylabel('Y Label')

    # Displaying the binarized image
    # of various thresholds
    plt.imshow(binarized_gray, cmap = 'gray')

plt.tight_layout()

# Setting plot size to 15, 15
plt.figure(figsize=(15, 15))

# Sample Image of scikit-image package
coffee = data.coffee()
gray_coffee = rgb2gray(coffee)

# Computing Otsu's thresholding value
threshold = filters.threshold_otsu(gray_coffee)

# Computing binarized values using the obtained threshold
binarized_coffee = (gray_coffee > threshold)*1
plt.subplot(2,2,1)
```

```
plt.title("Threshold: >"+str(threshold))
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Displaying the binarized image
plt.imshow(binarized_coffee, cmap = "gray")

# Computing Ni black's local pixel threshold values for every pixel
threshold = filters.threshold_niblack(gray_coffee)

# Computing binarized values using the obtained threshold
binarized_coffee = (gray_coffee > threshold)*1
plt.subplot(2,2,2)
plt.title("Niblack Thresholding")
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Displaying the binarized image
plt.imshow(binarized_coffee, cmap = "gray")

# Computing Sauvola's local pixel threshold values for every pixel - Not
Binarized
threshold = filters.threshold_sauvola(gray_coffee)
plt.subplot(2,2,3)
plt.title("Sauvola Thresholding")
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Displaying the local threshold values
plt.imshow(threshold, cmap = "gray")

# Computing Sauvola's local pixel threshold values for every pixel -
Binarized
binarized_coffee = (gray_coffee > threshold)*1
plt.subplot(2,2,4)
plt.title("Sauvola Thresholding - Converting to 0's and 1's")
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Displaying the binarized image
```

```
plt.imshow(binarized_coffee, cmap = "gray")

# Sample Image of scikit-image package
astronaut = astronaut()
gray_astronaut = rgb2gray(astronaut)

# Applying Gaussian Filter to remove noise
gray_astronaut_noiseless = gaussian(gray_astronaut, 1)

# Localising the circle's center at 220, 110
x1 = 220 + 100*np.cos(np.linspace(0, 2*np.pi, 500))
x2 = 100 + 100*np.sin(np.linspace(0, 2*np.pi, 500))

# Generating a circle based on x1, x2
snake = np.array([x1, x2]).T

# Computing the Active Contour for the given image
astronaut_snake = active_contour(gray_astronaut_noiseless, snake)

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

# Adding subplots to display the markers
ax = fig.add_subplot(111)

# Plotting sample image
ax.imshow(gray_astronaut_noiseless)
plt.title('Active Contour Segmentation')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Plotting the face boundary marker
ax.plot(astronaut_snake[:, 0], astronaut_snake[:, 1], '-b', lw=5)

# Plotting the circle around face
ax.plot(snake[:, 0], snake[:, 1], '--r', lw=5)

fig, axes = plt.subplots(1, 3, figsize=(10, 10))

# Sample Image of scikit-image package
astronaut = data.astronaut()
```

```
gray_astronaut = rgb2gray(astronaut)

# Computing the Chan VESE segmentation technique
chanvese_gray_astronaut = chan_vese(gray_astronaut, max_num_iter=100,
extended_output=True)

ax = axes.flatten()

# Plotting the original image
ax[0].imshow(gray_astronaut, cmap="gray")
ax[0].set_title("Original Image")
ax[0].set_xlabel('X Label')
ax[0].set_ylabel('Y Label')

# Plotting the segmented - 100 iterations image
ax[1].imshow(chanvese_gray_astronaut[0], cmap="gray")
title = "Chan-Vese Segmentation - {}
iterations".format(len(chanvese_gray_astronaut[2]))

ax[1].set_title(title)
ax[1].set_xlabel('X Label')
ax[1].set_ylabel('Y Label')

# Plotting the final level set
ax[2].imshow(chanvese_gray_astronaut[1], cmap="gray")
ax[2].set_title("Final Level Set")
ax[2].set_xlabel('X Label')
ax[2].set_ylabel('Y Label')
plt.show()

# Setting the plot figure as 15, 15
plt.figure(figsize=(15, 15))

# Sample Image of scikit-image package
astronaut = data.astronaut()

# Applying SLIC segmentation for the edges to be drawn over
astronaut_segments = slic(astronaut, n_segments=100, compactness=1)

plt.subplot(1, 2, 1)
```

```
# Plotting the original image
plt.imshow(astronaut)
plt.title('Original Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Detecting boundaries for labels
plt.subplot(1, 2, 2)

# Plotting the output of marked_boundaries function i.e. the image with
segmented boundaries
plt.imshow(mark_boundaries(astronaut, astronaut_segments))
plt.title('SLIC Segmentation')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Setting the plot size as 15, 15
plt.figure(figsize=(15,15))

# Sample Image of scikit-image package
astronaut = data.astronaut()

# Applying Simple Linear Iterative
# Clustering on the image - 50 segments & compactness = 10
astronaut_segments = slic(astronaut, n_segments=50, compactness=10)
plt.subplot(1,2,1)

# Plotting the original image
plt.imshow(astronaut)
plt.title('Original Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

plt.subplot(1,2,2)

# Converts a label image into an RGB color image for visualizing the labeled
regions
plt.imshow(label2rgb(astronaut_segments, astronaut, kind = 'avg'))
plt.title('SLIC Segmentation with Color Mapping')
```

```
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Setting the figure size as 15, 15
plt.figure(figsize=(15,15))

# Sample Image of scikit-image package
astronaut = data.astronaut()

# computing the Felzenszwalb's Segmentation with sigma = 5 and minimum size
= 100
astronaut_segments = felzenszwalb(astronaut, scale = 2, sigma=5,
min_size=100)

# Plotting the original image
plt.subplot(1,2,1)
plt.imshow(astronaut)
plt.title('Original Image')
plt.xlabel('X Label')
plt.ylabel('Y Label')

# Marking the boundaries of Felzenszwalb's segmentations
plt.subplot(1,2,2)
plt.imshow(mark_boundaries(astronaut, astronaut_segments))
plt.title('Felzenszwalb Segmentations')
plt.xlabel('X Label')
plt.ylabel('Y Label')
```

Figure 18. Original Image

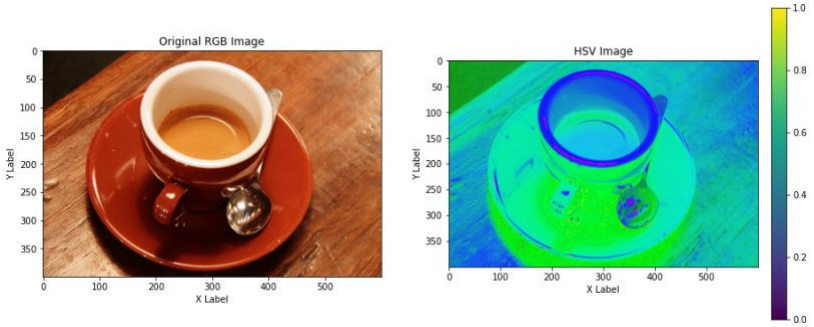


Figure 19. (a) Original RGB Image (b) HSV Image

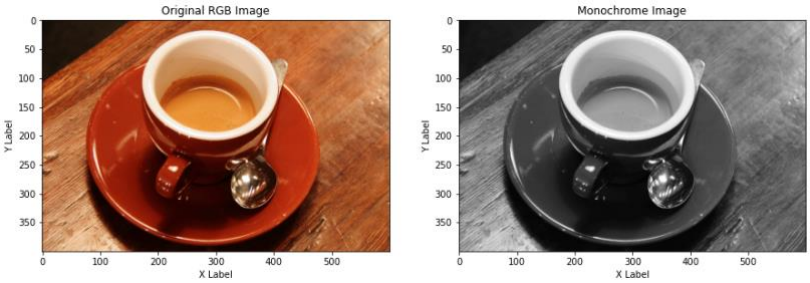


Figure 20. (a) Original RGB Image (b) Monochrome Image

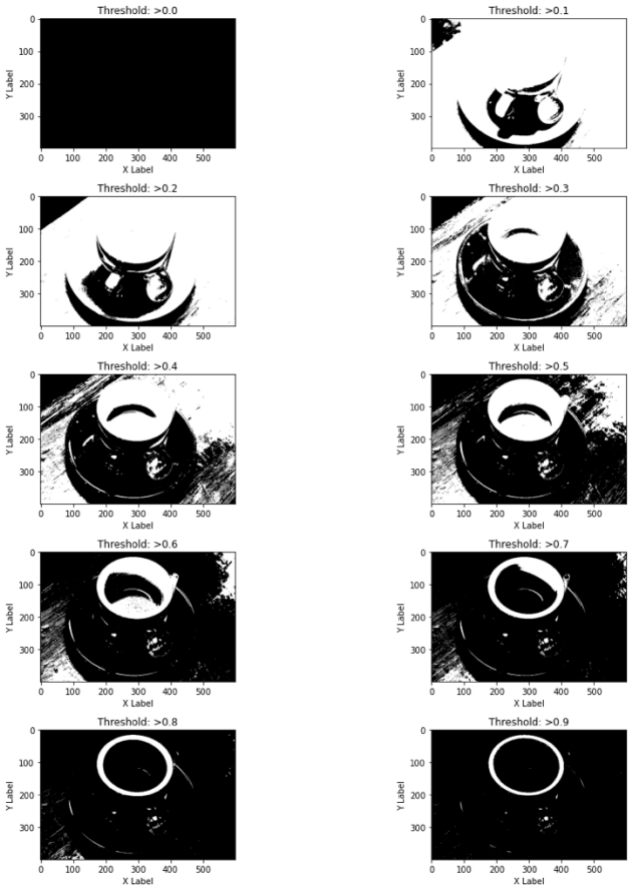


Figure 21. Thresholds

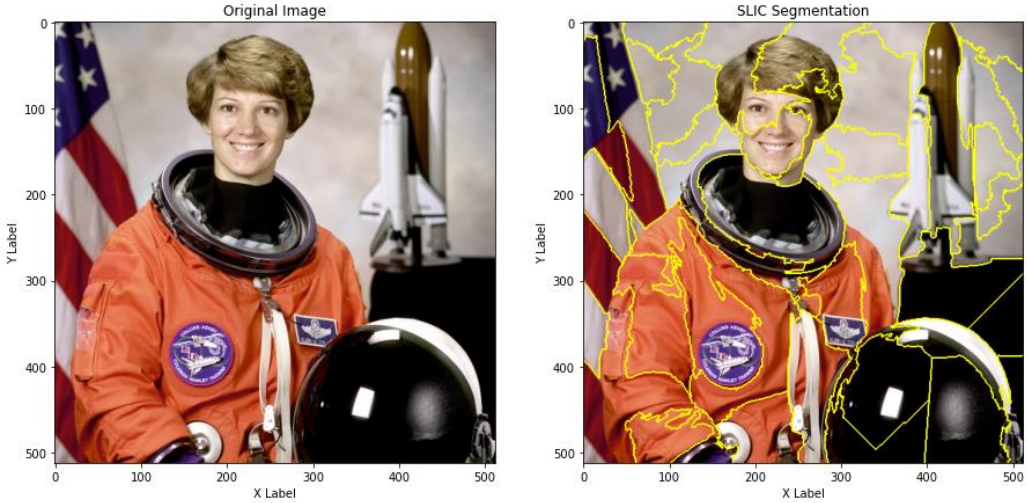


Figure 22. (a) SLIC Segmentation

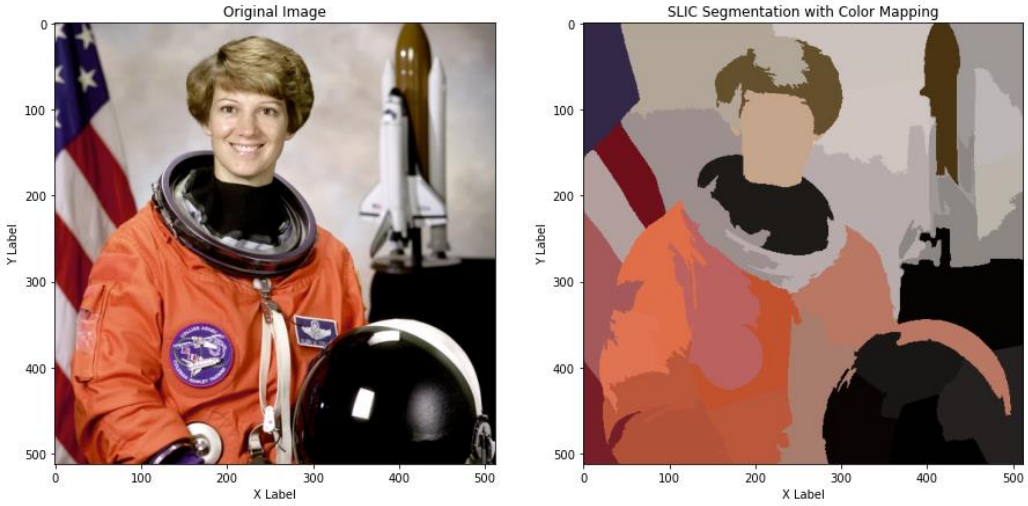


Figure 22. (b) SLIC Segmentation with Color Mapping

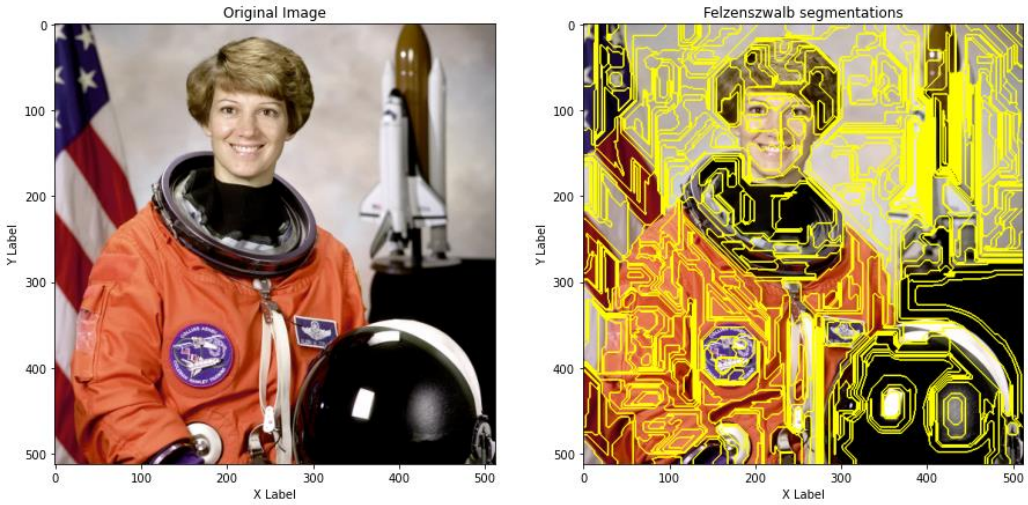


Figure 23. Felzenswalb Segmentation

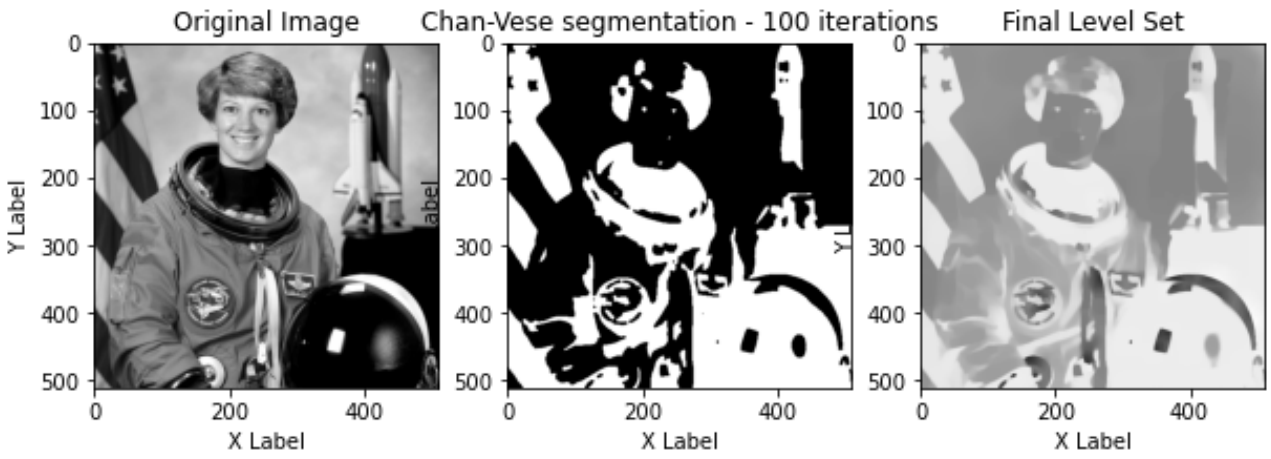


Figure 24. Chan-Vese Segmentation

INFERENCE:

- The image processing assignment on image analysis and operations has been completed, exploring various methods for analysing and performing operations on digital images.
- Theoretical background research and practical implementation of image analysis techniques using Python programming have provided a comprehensive understanding of different methods and their applications.
- The documentation includes labelled images of input and output figures, demonstrating the effectiveness of the methods and the results achieved through image analysis.
- The provided references allow for further exploration of the topic. The assignment has significantly contributed to the development of skills in image processing and the use of Python programming for image analysis and operations.

REFERENCES:

Serial Number	Online Portal	Content	URL
1	Geeks For Geeks	Python – Edge Detection using Pillow	https://www.geeksforgeeks.org/python-edge-detection-using-pillow/
2		Image Segmentation using Python's scikit-image module	https://www.geeksforgeeks.org/image-segmentation-using-pythons-scikit-image-module/
3	LearnOpenCV	Edge Detection Using OpenCV	https://learnopencv.com/edge-detection-using-opencv/

4	TutorialsPoint	How to Compute Image Moments in OpenCV Python?	https://www.tutorialspoint.com/how-to-compute-image-moments-in-opencv-python
5	Analytics Vidhya	Image Operations in Python with OpenCV: Eroding, Dilation, and more!	https://www.analyticsvidhya.com/blog/2021/12/image-operations-in-python-with-opencv/#:~:text=Morphological%20Transformations%20are%20image%20processing,the%20output%20image%20is%20generated

Textbooks:

1. Digital Image Processing, Fourth Edition, Rafael C. Gonzalez, Richard E. Woods, Pearson, 2018
2. Digital Image Processing, S Jayaraman, S Esakkirajan, T Veerakumar, Tata McGraw Hill, 2009
3. Fundamentals of Digital Image Processing, Anil K. Jain, Pearson Education, Prentice Hall

References:

1. Digital Image Processing - PIKS Scientific Inside, Fourth Edition, Willian K Pratt, Wiley-Interscience, 2007
2. Multidimensional Signal, Image, and Video Processing and Coding, Second Edition, John W. Woods, Elsevier, 2012