SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203

DEPARTMENT OF CYBER SECURITY

Lab Manual

BIOMETRIC IMAGE PROCESSING LABORATORY

(REGULATION 2023)

V SEMESTER-THIRD YEAR

CY3565 – BIOMETRIC IMAGE PROCESSING LABORATORY

Regulation – 2023

Academic Year: 2025 – 2026 (ODD)

Prepared by

Ms.M.Raghavi, Assistant Professor

PROGRAMME OUTCOMES

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using the first principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions: Design solutions for the complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including the design of experiments, analysis and interpretation of data, and the synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

CY3565 BIOMETRIC IMAGE PROCESSING LABORATORY L T P C
0 0 3 1.5

OBJECTIVES:

The student should be made to:

- To learn about the Image transformation
- To know about the Image enhancement
- To learn about the Image segmentation
- To learn about the Morphological image processing
- To know about the Feature extraction and recognition

LIST OF EXPERIMENTS

1. Write a program to perform Image transformation
2. Write a program to perform Image enhancement
3. Simulate Image segmentation
4. Simulate Morphological image processing
5. Implement the Feature extraction and recognition
6. Simulate Hand Geometry
7. Mini Project

TOTAL: 45 PERIODS

COURSE OUTCOMES:

- ✓ To learn about the Image transformation
- ✓ To know about the Image enhancement
- ✓ To learn about the Image segmentation
- ✓ To learn about the Morphological image processing
- ✓ To know about the Feature extraction and recognition

INDEX

1. Image Transformation

Aim:

To implement the following image transformation such as Image Translation, Reflection, Rotation, Scaling, Cropping, Shearing in x-axis and Shearing in y-axis.

Algorithm:

1. Load the input image.
2. Determine the desired transformation parameters, such as rotation angle, scaling factor, translation values, or perspective transformation points.
3. Create an output image of the desired size to hold the transformed image.
4. Iterate over each pixel in the output image.
5. For each pixel in the output image, calculate the corresponding location in the input image using the inverse transformation.
6. Interpolate the pixel values from the input image using the calculated location to obtain the transformed pixel value.
7. Assign the transformed pixel value to the corresponding pixel in the output image.
8. Repeat steps 4 to 7 for all pixels in the output image.
9. Save or display the transformed image.

Program:

Image Translation

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
M = np.float32([[1, 0, 100], [0, 1, 50]])
dst = cv.warpAffine(img, M, (cols, rows))
cv.imshow('img', dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

Output:

Image Reflection

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
M = np.float32([[1, 0, 0],[0, -1, rows],[0, 0, 1]])
reflected_img = cv.warpPerspective(img, M, (int(cols), int(rows)))
cv.imshow('img', reflected_img)
```

```
cv.imwrite('reflection_out.jpg', reflected_img)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Image Rotation

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
M = np.float32([[1, 0, 0], [0, -1, rows], [0, 0, 1]])
img_rotation = cv.warpAffine(img,
cv.getRotationMatrix2D((cols/2, rows/2),30, 0.6),(cols, rows))
cv.imshow('img', img_rotation)
cv.imwrite('rotation_out.jpg', img_rotation)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Image Scaling

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
img_shrinked = cv.resize(img, (250, 200),
interpolation=cv.INTER_AREA)
cv.imshow('img', img_shrinked)
img_enlarged = cv.resize(img_shrinked, None, fx=1.5, fy=1.5,
interpolation=cv.INTER_CUBIC)
cv.imshow('img', img_enlarged)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Image Cropping

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
cropped_img = img[100:300, 100:300]
cv.imwrite('cropped_out.jpg', cropped_img)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Image Shearing in X-Axis

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
M = np.float32([[1, 0.5, 0], [0, 1, 0], [0, 0, 1]])
sheared_img = cv.warpPerspective(img, M, (int(cols*1.5), int(rows*1.5)))
cv.imshow('img', sheared_img)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Image Shearing in Y-Axis

```
import numpy as np
import cv2 as cv
img = cv.imread('girlImage.jpg', 0)
rows, cols = img.shape
M = np.float32([[1,   0, 0], [0.5, 1, 0], [0,   0, 1]])
sheared_img = cv.warpPerspective(img, M, (int(cols*1.5), int(rows*1.5)))
cv.imshow('sheared_y-axis_out.jpg', sheared_img)
cv.waitKey(0)
cv.destroyAllWindows()
```

OUTPUT:

Result:

Thus, the image transformation program was executed successfully and output was Verified.

2. Image enhancement

Aim:

To Write a program to perform Image enhancement

Algorithm:

1. Load the input image.
2. Convert the image to an appropriate color space if needed (e.g., convert to grayscale or another color space like HSV).
3. Apply preprocessing techniques if necessary, such as noise reduction or smoothing filters, to improve the image quality.
4. Analyze the image characteristics and determine the enhancement techniques to be applied based on the specific requirements or goals (e.g., contrast enhancement, brightness adjustment, sharpening, etc.).
5. Implement the chosen enhancement techniques based on the analysis. Some common enhancement techniques include:
    a. Histogram equalization: Adjust the pixel intensities to improve contrast and enhance details.
    b. Gamma correction: Adjust the image gamma value to modify brightness and contrast.
    c. Adaptive filtering: Apply local filters to enhance specific regions of the image.
    d. Unsharp masking: Create a high-pass filtered version of the image and combine it with the original image to enhance edges and details.
    e. Color adjustment: Modify the color balance, saturation, or hue of the image to improve its appearance.
    f. Noise reduction: Apply denoising algorithms to reduce unwanted noise while preserving image details.
    g. Edge enhancement: Apply filters or algorithms to enhance edges and increase image sharpness.
6. Apply the chosen enhancement techniques to the image, either globally or locally based on the specific requirements.
7. Evaluate the enhanced image quality and make adjustments if necessary.
8. Save or display the enhanced image.

Brightness():

Adjust image brightness. It's accustomed to controlling the brightness of our resulting image. The code for brightness goes like below:

Input:

Program:

```
from PIL import Image
from PIL import ImageEnhance
# Opens the image file
image = Image.open('gfg.png')
# shows image in image viewer
image.show()
# Enhance Brightness
curr_bri = ImageEnhance.Brightness(image)
new_bri = 2.5
# Brightness enhanced by a factor of 2.5
```

```
img_brightened = curr_bri.enhance(new_bri)
# shows updated image in image viewer
img_brightened.show()
```

An enhancement factor of 0.0 results in a full black image and an enhancement factor of 1.0 results the same as the original image.

Output:

Color():

Adjust image color level. It's accustomed to controlling the color level of our resulting image. The code for coloring goes like below:

Input:

Program:

```
from PIL import Image
from PIL import ImageEnhance
# Opens the image file
image = Image.open('gfg.png')
# shows image in image viewer
image.show()
# Enhance Color Level
curr_col = ImageEnhance.Color(image)
new_col = 2.5
# Color level enhanced by a factor of 2.5
img_colored = curr_col.enhance(new_col)
# shows updated image in image viewer
img_colored.show()
```

An enhancement factor of 0.0 results in a full black and white image and an enhancement factor of 1.0 results the same as the original image.

Output:

Contrast():

Adjust image Contrast. It is accustomed to controlling the contrast of our resulting image. The code for changing contrast goes like below:

Input:

Program:

```
from PIL import Image
from PIL import ImageEnhance
# Opens the image file
image = Image.open('gfg.png')
# shows image in image viewer
image.show()
# Enhance Contrast
curr_con = ImageEnhance.Contrast(image)
new_con = 0.3
# Contrast enhanced by a factor of 0.3
img_contrasted = curr_con.enhance(new_con)
# shows updated image in image viewer
img_contrasted.show()
```

An enhancement factor of 0.0 results in a full grey image and an enhancement factor of 1.0 results the same as the original image.

Output:

Sharpness():

Adjust image Sharpness. It's accustomed to controlling the sharpness of our resulting image. The code for changing sharpness goes like below:

Input:

Program:

```
from PIL import Image
from PIL import ImageEnhance
# Opens the image file
image = Image.open('gfg.png')
# shows image in image viewer
image.show()
# Enhance Sharpness
curr_sharp = ImageEnhance.Sharpness(image)
```

```
new_sharp = 8.3
# Sharpness enhanced by a factor of 8.3
img_sharped = curr_sharp.enhance(new_sharp)
# shows updated image in image viewer
img_sharped.show()
```

An enhancement factor of 0.0 results in a blurred image and an enhancement factor of 1.0 results in the same as the original image and a factor > 1.0 results in a sharpened image.

Output:

Result:

Thus, the Image enhancement program was executed successfully and output was verified.

3. Image segmentation

Aim:

To write a program to perform Image segmentation.

Algorithm:

1. Load the input image.
2. Preprocess the image if needed, such as resizing, noise reduction, or color space conversion.
    a. Choose an appropriate segmentation method based on the characteristics of the image and the desired outcome. Some common segmentation methods include:
    b. Thresholding: Convert the image to binary based on a threshold value to separate foreground and background regions.
    c. Region-based segmentation: Divide the image into regions based on similarities in color, texture, or other features.
    d. Edge-based segmentation: Detect and link edges or contours to segment the image.
    e. Clustering: Apply clustering algorithms (e.g., k-means, mean shift) to group similar pixels together.
    f. Watershed segmentation: Use the concept of watershed transformation to partition the image based on gradient or region markers.
    g. Graph-based segmentation: Convert the image into a graph and use graph algorithms to identify meaningful regions.
3. Implement the chosen segmentation method. This may involve applying filters, calculating gradients, performing clustering, or using other techniques specific to the chosen method.
4. Post-process the segmented regions to refine the results. This can include techniques such as morphological operations (e.g., dilation, erosion), noise removal, or region merging/splitting based on certain criteria.
5. Evaluate the segmentation results using metrics like accuracy, precision, recall, or visual inspection.
6. Iterate and fine-tune the segmentation process if needed by adjusting parameters, using different algorithms, or incorporating additional steps.
7. Save or display the segmented image and/or region masks.

RGB to Grayscale

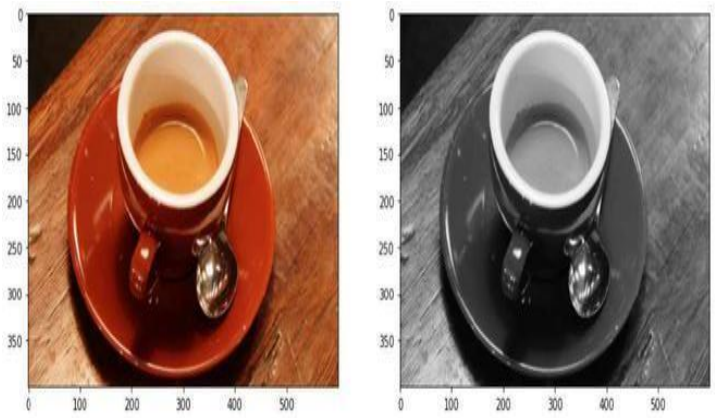
rgb2gray module of skimage package is used to convert a 3-channel RGB Image to one channel monochrome image. In order to apply filters and other processing techniques, the expected input is a two-dimensional vector i.e. a monochrome image.

Program:

```
# Importing Necessary Libraries
from skimage import data
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
# 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)
# 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")
```

skimage.color.rgb2gray() function is used to convert an RGB image to Grayscale format

Output:



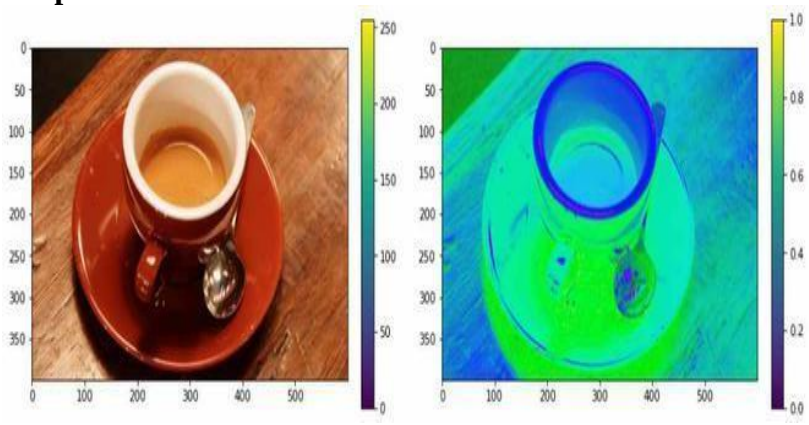
RGB to HSV

The HSV (Hue, Saturation, Value) color model remaps the RGB basic colors into dimensions that are simpler to comprehend for humans. The RGB color space describes the proportions of red, green, and blue in a colour. In the HSV color system, colors are defined in terms of Hue, Saturation, and Value.

Program:

```
# Importing Necessary Libraries
from skimage import data
from skimage.color import rgb2hsv
import matplotlib.pyplot as plt
# 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)
# 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)
# Adjusting colorbar to fit the size of the image
plt.colorbar(hsv_coffee_colorbar, fraction=0.046, pad=0.04)
```

Output:



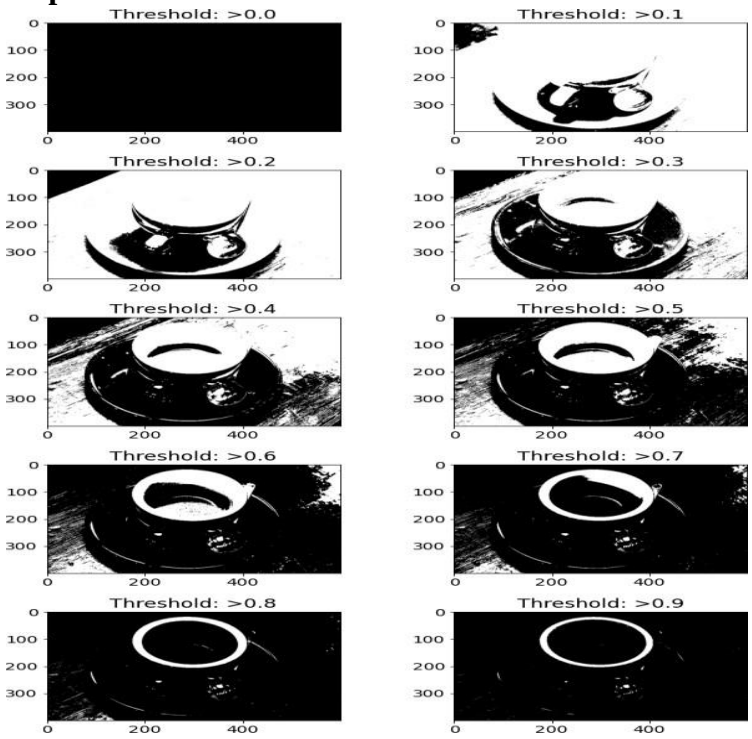
Supervised Segmentation

For this type of segmentation to proceed, it requires external input. This includes things like setting a threshold, converting formats, and correcting external biases.

Program:

```
# Importing Necessary Libraries
# Displaying the sample image - Monochrome Format
from skimage import data
from skimage import filters
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
# 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)))
# Displaying the binarized image
# of various thresholds
plt.imshow(binarized_gray, cmap = 'gray')
plt.tight_layout()
```

Output:



Result:

Thus, the image segmentation program was executed successfully and output was verified.

4. Morphological image processing

Aim:

To Simulate Morphological image processing

Algorithm:

1. Load the input image.
2. Convert the image to a suitable binary format, typically using thresholding or other segmentation techniques to obtain a binary image.
3. Choose the desired morphological operation based on the specific requirements. Some common morphological operations include:
    a. Dilation: Expands the boundaries of objects by adding pixels to their perimeter.
    b. Erosion: Shrinks the boundaries of objects by removing pixels from their perimeter.
    c. Opening: Erosion followed by dilation, used to remove noise or small objects.
    d. Closing: Dilation followed by erosion, used to fill gaps or close small holes.
    e. Boundary extraction: Difference between the input image and its erosion.
    f. Morphological gradient: Difference between dilation and erosion of the input image.
4. Define the structuring element, which is a small shape or template that defines the neighborhood used by the morphological operation. The structuring element can be a simple shape like a square, circle, or line, or more complex shapes depending on the desired effect.
    - ✓ Apply the chosen morphological operation using the structuring element to the binary image. This involves iterating over each pixel in the image and performing the operation within the defined neighborhood.
    - ✓ Repeat steps 4 and 5 if multiple iterations of the operation are required.
    - ✓ Optionally, perform additional post-processing steps like noise removal, region filtering, or object labeling to refine the results.
    - ✓ Save or display the processed image.

Program:

```
# Python program to illustrate
# Opening morphological operation
# on an image
# organizing imports
import cv2
import numpy as np
# return video from the first webcam on your computer.
screenRead = cv2.VideoCapture(0)
# loop runs if capturing has been initialized.
while(1):
        # reads frames from a camera
        _, image = screenRead.read()

        # Converts to HSV color space, OCV reads colors as BGR
        # frame is converted to hsv
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        # defining the range of masking
        blue1 = np.array([110, 50, 50])
        blue2 = np.array([130, 255, 255])

        # initializing the mask to be
        # convoluted over input image
        mask = cv2.inRange(hsv, blue1, blue2)

        # passing the bitwise_and over
```

```
        # each pixel convoluted
        res = cv2.bitwise_and(image, image, mask = mask)

        # defining the kernel i.e. Structuring element
        kernel = np.ones((5, 5), np.uint8)

        # defining the opening function
        # over the image and structuring element
        opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

        # The mask and opening operation
        # is shown in the window
        cv2.imshow('Mask', mask)
        cv2.imshow('Opening', opening)

        # Wait for 'a' key to stop the program
        if cv2.waitKey(1) & 0xFF == ord('a'):
                break
# De-allocate any associated memory usage
cv2.destroyAllWindows()
# Close the window / Release webcam
screenRead.release()
```

Output:

Input Frame:

Mask:

Output Frame:

Result:

Thus, the morphological image processing program was executed successfully and output was verified.

5. Feature Extraction and Recognition

Aim:
To implement the Feature Extraction and Recognition

Algorithm:

1. Loading and Visualizing an Image
2. Analyze image properties
3. Image Feature Extraction using Scikit-Image

a. Importing the required libraries

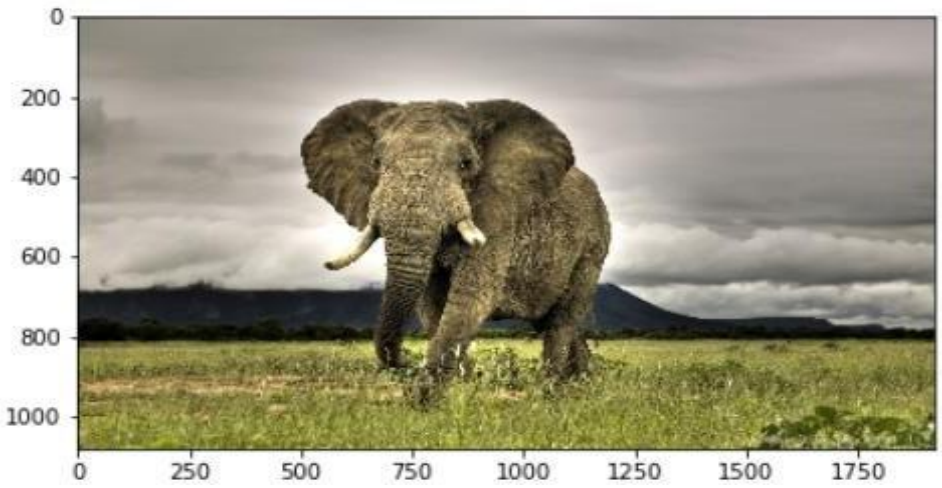
We will look at all the aspects of the image so we need to import different libraries including NumPy, pandas, etc.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from skimage.io import imread, imshow
```

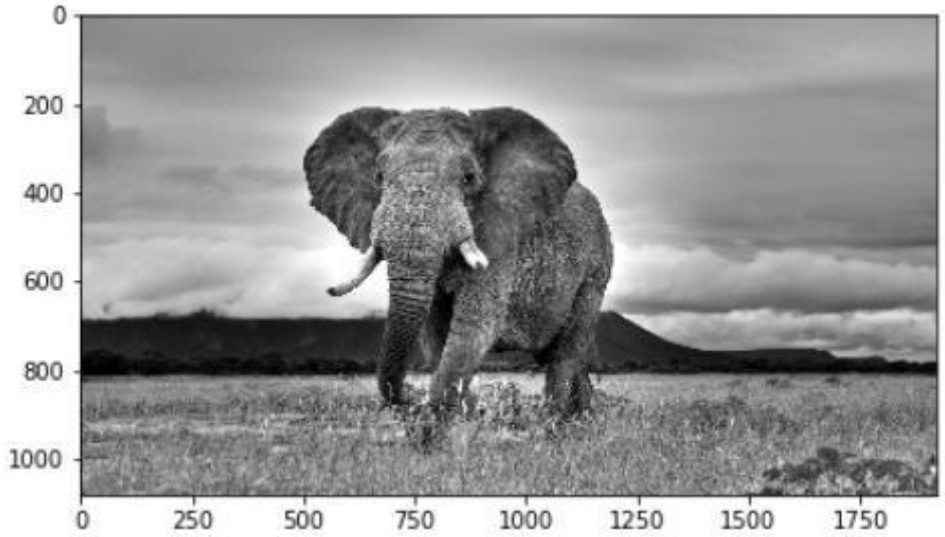
b. Loading the image

We can use any image from your system. I have an image named'elephant.jpg' for which I will be performing feature extraction.

```
image1 = imread('C:/Users/Lenovo/Downloads/elephant.jpg')
imshow(image1);
```




```
image2 = imread('elephant.jpg', as_gray=True)
imshow(image2);
```



c. Analyzing both the images

```
#Shape of images
print(image1.shape)
print(image2.shape)
```

```
(1080, 1920, 3)
(1080, 1920)
```

Here we can see that the colored image contains rows, columns, and channels as it is a colored image there are three channels RGB while grayscale pictures have only one channel. So we can clearly identify the colored and grayscale images by their shapes.

Let's look at the size of the images.

```
print(image1.size)
print(image2.size)
```

```
6220800
2073600
```

Image size is the product of the rows, columns, and channels.

d. Feature Extraction

i. Pixel Features

The number of pixels in an image is the same as the size of the image for grayscale images we can find the pixel features by reshaping the shape of the image and returning the array form of the image.

```
pixel_feat1 = np.reshape(image2, (1080 * 1920)
pixel_feat1
```

```
array([0.52571333, 0.52571333, 0.52571333, ..., 0.36006431, 0.42112784,
       0.45250039])
```

Similarly, we can find the pixel feature for the colored image.

```
pixel_feat2 = np.reshape(image1, (1080 * 1920 * 3)
pixel_feat2
```

```
array([138, 134, 123, ..., 118, 121,  52], dtype=uint8)
```

ii. Edge Features

Edges in an image are the corners where the pixel change drastically, as the images are stored in array form we can visualize different values and see where the change in pixel value is higher but doing it manually takes time, Scikit-Image provides functions for image edge features extraction namely:

- **Prewitt Kernel:**

It is an edge detection kernel that works separately for both horizontal and vertical axis.

-1	0	1
-1	0	1
-1	0	1

Horizontal (f_x)

-1	-1	-1
0	0	0
1	1	1

Vertical (f_y)

- **Sobel Kernel:**

It works on creating images with emphasis on edges

- **Canny Algorithm**

-1	0	+1
-2	0	+2
-1	0	+1

x filter

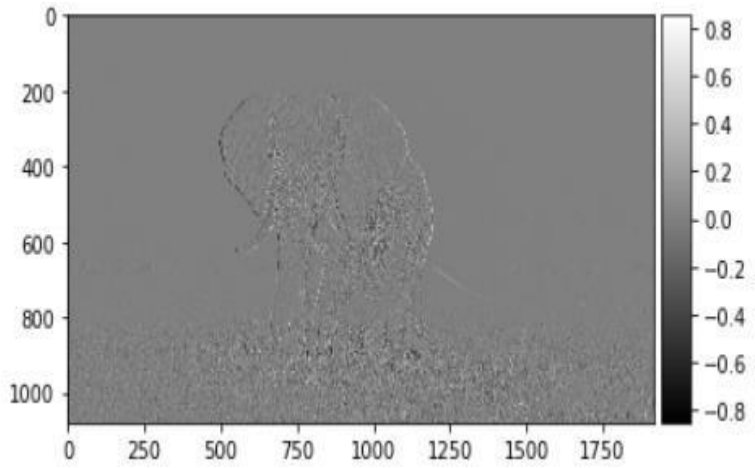
+1	+2	+1
0	0	0
-1	-2	-1

y filter

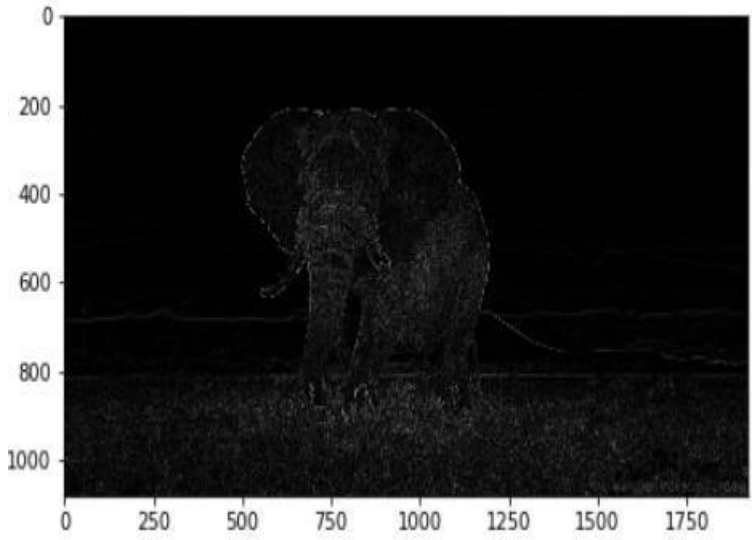
Canny Algorithm is an edge detection technique that uses a multi-stage algorithm to detect a wide range of edges in images.

Using these three algorithms for edge feature extraction.

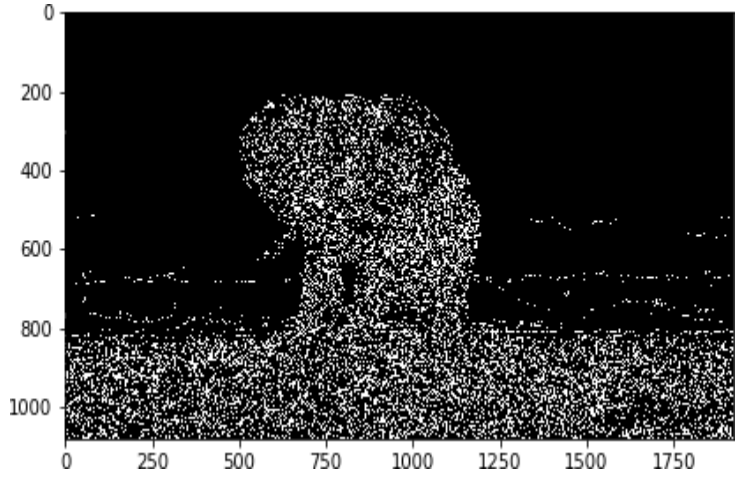
```
from skimage import filters

# prewitt kernel
pre_hor = prewitt_h(image2)
pre_ver = prewitt_v(image2)

# Sobel Kernel
ed_sobel = filters.sobel(image2)

#canny algorithm
can = feature.canny(image2)

imshow(pre_ver, cmap='gray');
```




```
imshow(ed_sobel, cmap='gray');
```



imshow(can, cmap='gray);

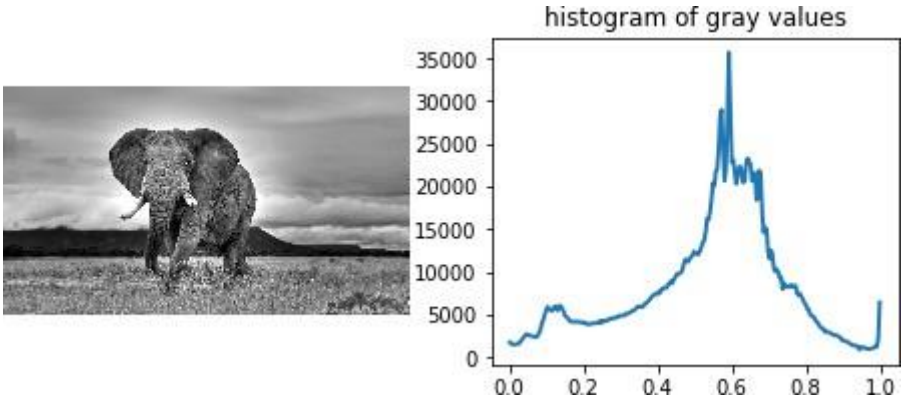


There are many other kernels for edge feature extraction but these three are the most used ones.

- **Region-Based Segmentation**

We use this segment object from a background. We will use our image and try region-based segmentation on it.

```
from skimage.exposure import histogram
hist, hist_centers = histogram(image2)

#Plotting the Image and the Histogram of gray values
fig, axes = plt.subplots(1, 2, figsize=(8, 3))
axes[0].imshow(image2, cmap=plt.cm.gray)
axes[0].axis('off')
axes[1].plot(hist_centers, hist, lw=2)
axes[1].set_title('histogram of gray values')
```



This image clearly depicts the feature we have tried to extract segmentation of object from the background.

Result:
Thus, the feature extraction and recognition program was executed successfully and output was verified.

6. Hand Geometry

Aim:
To Simulate Hand Geometry using Python

Algorithm:

1. Acquire an image or a series of images of the hand using a suitable imaging device, such as a camera or a depth sensor.
2. Pre-process the hand image(s) to enhance the quality and prepare it for feature extraction. This can include steps like noise reduction, background removal, or hand segmentation.
3. Extract hand region of interest (ROI) from the pre-processed image(s). This step involves locating and isolating the hand from the background or other objects present in the image.
4. Apply hand shape normalization to ensure consistent hand posture across different images. This step can involve techniques like scaling, rotation, or translation to align the hand shapes.
5. Identify and extract relevant hand geometry features from the normalized hand shape. Some common hand geometry features include:
    - Finger length: Measure the lengths of fingers from specific landmarks like finger tips or knuckles.

- Palm width and length: Measure the width and length of the palm from certain reference points.
- Finger width: Measure the width of individual fingers or the average finger width.
- Hand perimeter: Calculate the length of the hand perimeter.
- Hand area: Measure the area enclosed by the hand shape.
- Curvature: Analyze the curvature of fingers or the overall hand shape.
- Proportions: Compute the ratios or proportions between different hand measurements.

6. Create a feature vector or descriptor by combining or representing the extracted hand geometry features.
7. Train a classifier or recognition model using the feature vectors and corresponding labels or identities from a training dataset. Common classifiers include:
    - Support Vector Machines (SVM)
    - k-Nearest Neighbors (k-NN)
    - Random Forests
    - Artificial Neural Networks (ANN)
8. Split the dataset into training and testing subsets for evaluation and validation.
9. Train the classifier using the training subset and evaluate its performance on the testing subset using appropriate metrics (e.g., accuracy, precision, recall, F1-score).
10. Fine-tune the feature extraction, normalization, or classification steps based on the evaluation results, if necessary.
11. Once the model is trained and validated, you can use it for hand geometry recognition on new hand images by following these steps:
    - Pre-process the new hand image(s) similar to step 2.
    - Extract the hand region of interest (ROI) using hand segmentation techniques, if needed.
    - Normalize the hand shape, if required, for consistent posture.
    - Extract hand geometry features from the normalized hand shape.
    - Create a feature vector using the extracted features.
    - Feed the feature vector into the trained classifier or recognition model.
    - Obtain the predicted identity or match score for the hand.

Program:

Step 1: Import all required libraries

```
# Importing Libraries
import cv2
import mediapipe as mp

# Used to convert protobuf message
# to a dictionary.
from google.protobuf.json_format import MessageToDict
```

Step 2: Initializing Hands model

```
# Initializing the Model
mpHands = mp.solutions.hands
hands = mpHands.Hands(
        static_image_mode=False,
        model_complexity=1
```

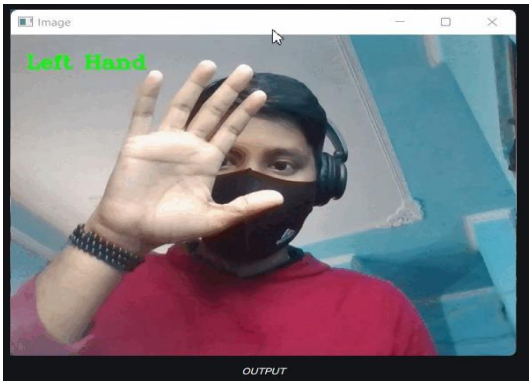
```
        min_detection_confidence=0.75,
        min_tracking_confidence=0.75,
        max_num_hands=2)
```

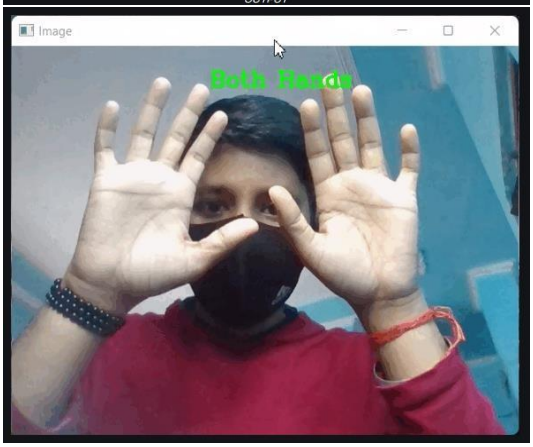
Step 3: Hands model process the image and detect hands

```
# Start capturing video from webcam
cap = cv2.VideoCapture(0)

while True:

        # Read video frame by frame
        success, img = cap.read()

        # Flip the image(frame)
        img = cv2.flip(img, 1)

        # Convert BGR image to RGB image
        imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Process the RGB image
        results = hands.process(imgRGB)

        # If hands are present in image(frame)
        if results.multi_hand_landmarks:

                # Both Hands are present in image(frame)
                if len(results.multi_handedness) == 2:
                                # Display 'Both Hands' on the image
                        cv2.putText(img, 'Both Hands', (250, 50),
                                                cv2.FONT_HERSHEY_COMPLEX, 0.9,
                                                (0, 255, 0), 2)

                # If any hand present
                else:
                        for i in results.multi_handedness:

                                # Return whether it is Right or Left Hand
                                label = MessageToDict(i)[
                                        'classification'][0]['label']

                                if label == 'Left':
                                                # Display 'Left Hand' on left side of window
                                        cv2.putText(img, label+' Hand', (20, 50),
                                                        cv2.FONT_HERSHEY_COMPLEX, 0.9,
                                                                (0, 255, 0), 2)

                                if label == 'Right':
                                                # Display 'Left Hand' on left side of window
                                        cv2.putText(img, label+' Hand', (460, 50),
                                                                cv2.FONT_HERSHEY_COMPLEX,
                                                                0.9, (0, 255, 0), 2)
        # Display Video and when 'q' is entered, destroy the window
        cv2.imshow('Image', img)
        if cv2.waitKey(1) & 0xff == ord('q'):
```

break

Output:





Result:

Thus, the hand geometry program was executed successfully and output was verified.