JOURNAL

M. Sc. (Computer Science) (NEP) Semester-I

2025-2026

Applied Signal and Image Processing

INDEX

PRACTICAL NO. 1

Aim: Write program to demonstrate the following aspects of signal processing on suitable data.

1. Upsampling and downsampling on Image/speech signal
2. Fast Fourier Transform to compute DFT

Description:

Upsampling involves increasing the sampling rate of a signal by inserting new samples between existing ones. Downsampling decreases the sampling rate by removing samples from the original signal. The Fourier transform decomposes a signal into its constituent frequencies. We will apply these techniques to sample image and audio data.

Equations:

- Upsampling Factor:

  $L = I/O$

  Where L is upsampling factor, I is input samples, O is output samples

- Downsampling Factor:

  $D = O/I$

  Where D is downsampling factor, I is input samples, O is output samples

- Discrete Fourier Transform:

  $Xk = \sum_{n=0}^{N-1} xn * e^{(-j2\pi kn/N)}$

  Where xn is the input signal, N is the signal length, and Xk is the Fourier transform

Code:

1) Upsampling and downsampling on Image/speech signal.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2
```

- Function to upsample an image

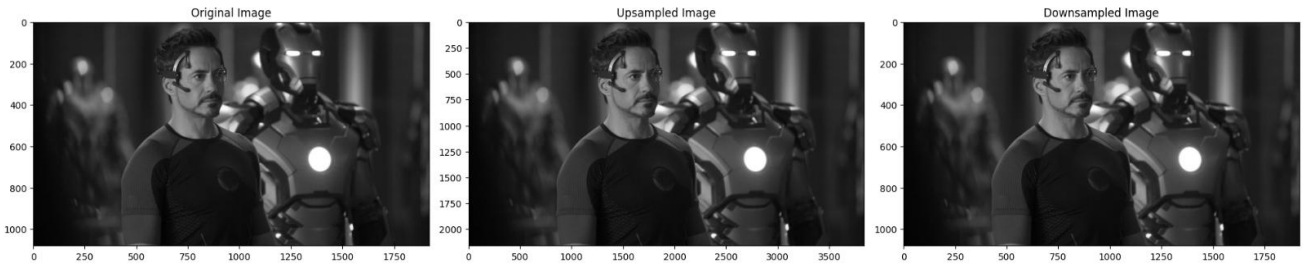
```
def upsample_image(image, factor):
    return ndimage.zoom(image, factor)
```

- Function to downsample an image

```
def downsample_image(image, factor):
    return image[::factor, ::factor]
```

```
img_path = "image.png"
original_imge = cv2.imread(img_path,cv2.IMREAD_GRAYSCALE)
upsampled_imge = upsample_image(original_imge, 2)

downsampled_imge = downsample_image(upsampled_imge, 2)
plt.figure(figsize=(20, 10))
plt.subplot(1, 3, 1)
plt.imshow(original_imge, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(upsampled_imge, cmap='gray')
plt.title('Upsampled Image')

plt.subplot(1, 3, 3)
plt.imshow(downsampled_imge, cmap='gray')
plt.title('Downsampled Image')

plt.tight_layout()
plt.show()
```

Output:



2) Fast Fourier Transform to compute DFT.

```
image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
```

- **Compute DFT**

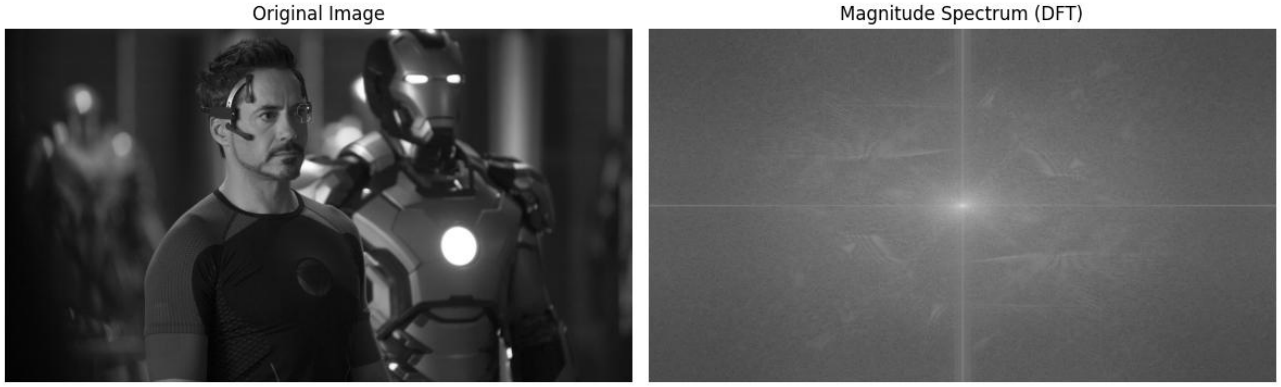
```
dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
```

- **Compute magnitude spectrum**

```
magnitude_spectrum = 20 * np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(magnitude_spectrum, cmap='gray')
```

```
plt.title('Magnitude Spectrum (DFT)')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO. 2

Aim: Write program to perform the following on signal

1. Create a triangle signal and plot a 3-period segment.
2. For a given signal, plot the segment and compute the correlation between them.

Description: Audio signals will be processed using convolution with an impulse response, correlation between signals, and filtering to remove frequencies. This demonstrates core concepts in digital signal processing.

Mathematical Equations:

1. Convolution:
   $y[n] = x[n] * h[n] = \sum_k x[k]h[n-k]$

2. Correlation:
   $R[m] = \sum_n x[n+m]y[n]$

Code:

```
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2
import numpy as np
from scipy.signal import correlate
```

Function to create a triangle wave

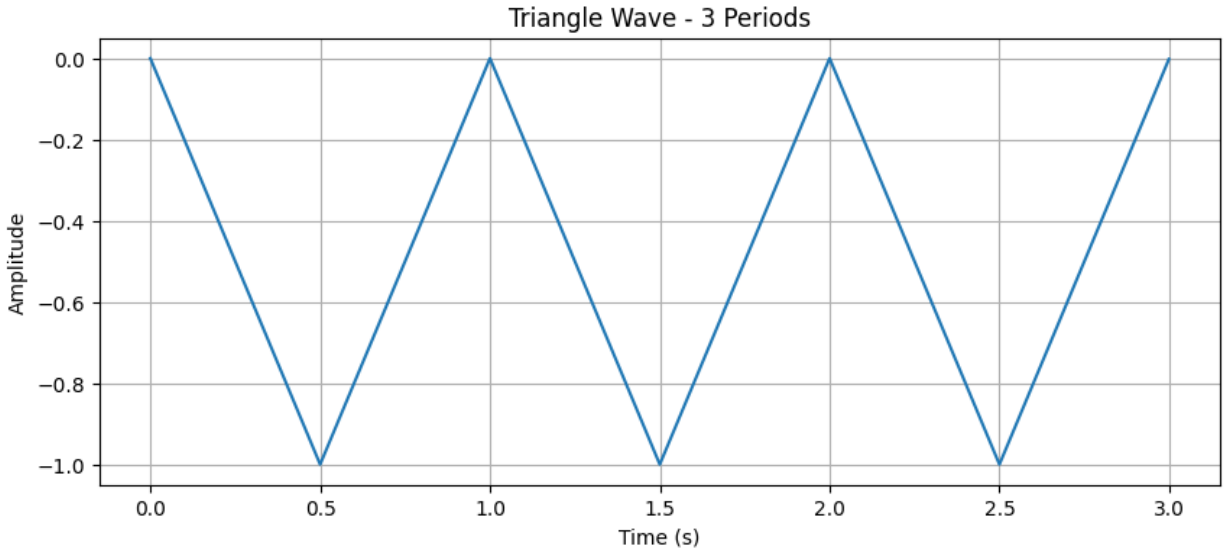
```
def triangle_wave(periods, sampling_rate):
    t = np.linspace(0, periods, int(periods * sampling_rate), endpoint=False)
    return 2 * np.abs((t % 1) - 0.5) - 1

Sampling rate in Hz Number of periods to plot Generate triangle wave signal
sampling_rate = 1000
periods = 3

triangle_signal = triangle_wave(periods, sampling_rate)
```

Plotling the triangle wave

```
plt.figure(figsize=(10, 4))
plt.plot(np.arange(0, periods, 1/sampling_rate), triangle_signal[:int(periods *
sampling_rate)])
plt.title('Triangle Wave - 3 Periods')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
```

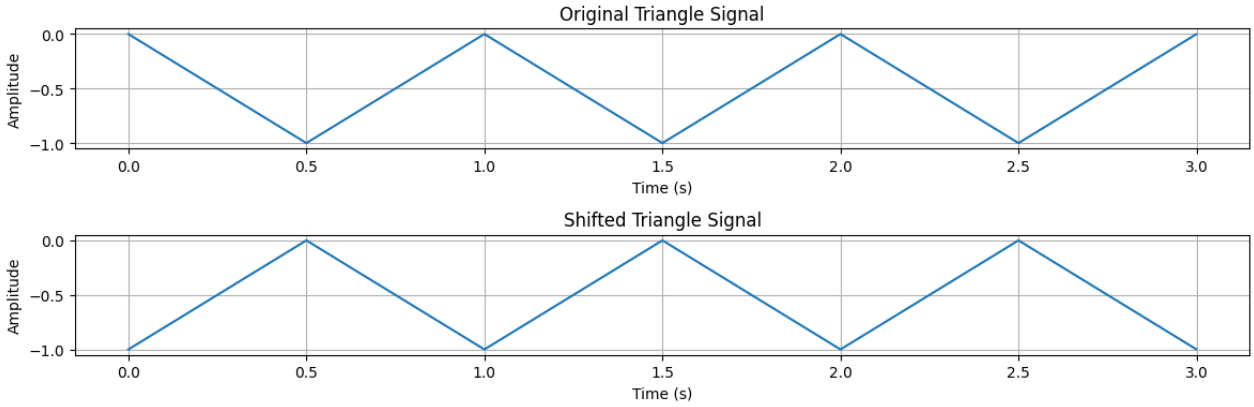
```
plt.show()
```

Output:



Create another signal (e.g., a shifted version of the original signal)

```
shifted_triangle_signal = np.roll(triangle_signal, int(sampling_rate / 2))
plt.figure(figsize=(12, 4))

plt.subplot(2, 1, 1)
plt.plot(np.arange(0, periods, 1/sampling_rate), triangle_signal[:int(periods *
sampling_rate)])
plt.title('Original Triangle Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(np.arange(0, periods, 1/sampling_rate), shifted_triangle_signal[:int(periods *
sampling_rate)])
plt.title('Shifted Triangle Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.tight_layout()
plt.show()
```



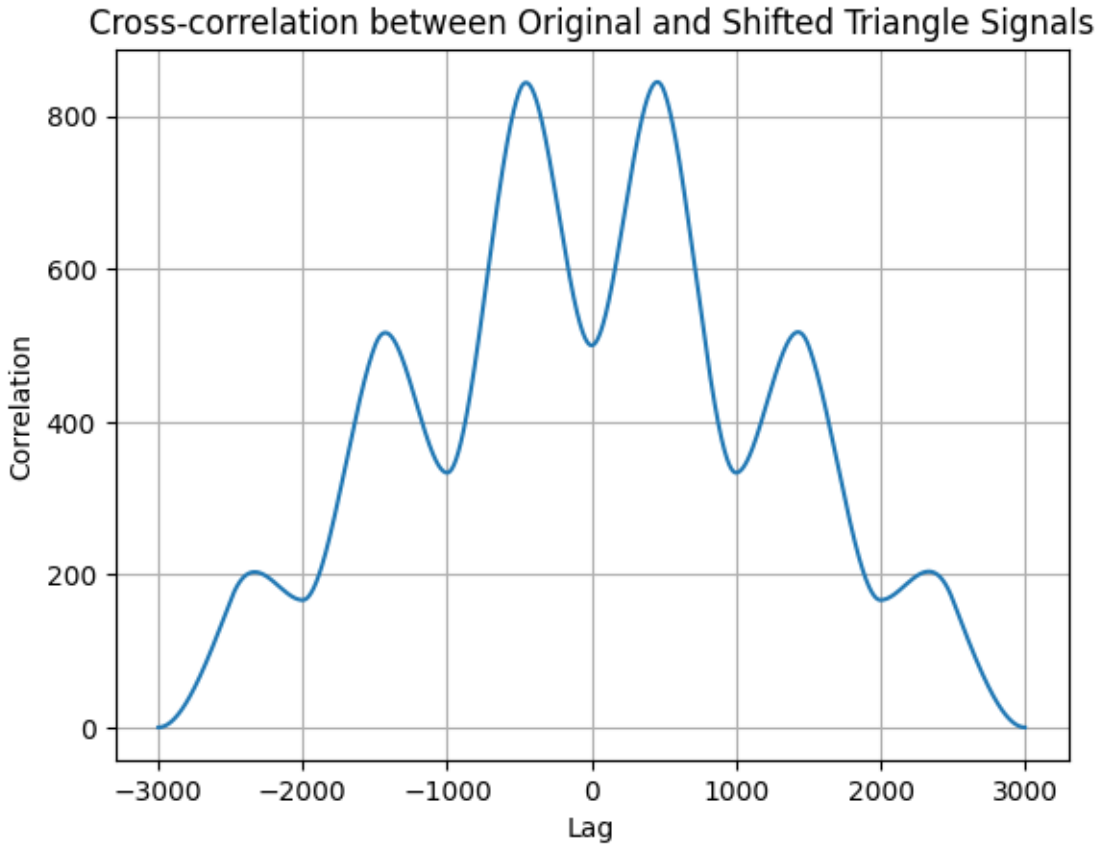
Compute the correlation between the original and shifted signals

```
correlation_result = correlate(triangle_signal, shifted_triangle_signal, mode='full')
```

Plotting the correlation result plt.figure(figsize=(12, 4))

```
plt.plot(np.arange(-len(triangle_signal)+1, len(triangle_signal)), correlation_result)
plt.title('Cross-correlation between Original and Shifted Triangle Signals')
plt.xlabel('Lag')
plt.ylabel('Correlation')
plt.grid(True)
plt.show()
```

Output:



PRACTICAL NO. 3

Aim: Write program to demonstrate the following aspects of signal on sound/image data

1. Convolution operation
2. Template Matching

Description:

Convolution is an important operation in signal and image processing. It involves multiplying a pixel and its neighbors by a small matrix called kernel. This allows effects like blurring, sharpening, edge detection etc.

Template matching is a technique to find parts of an image that match a template image. It slides the template image over the input image and calculates correlation scores to find the best match.

Mathematical Equation:

1. Convolution:

   $g(x, y) = \sum\sum f(m, n) \cdot h(x-m, y-n)$

   Where f is the input image, h is the kernel and g is the output image.

2. Template Matching:

   $R(x,y) = \sum\sum I(x+i, y+j) \cdot T(i, j)$

   Where I is the input image, T is the template and R is the correlation output.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
```

Function to perform convolution operation

```
def convolution(image, kernel):
    return cv2.filter2D(image, -1, kernel)
```

Function to perform template matching

```
def template_matching(image, template):
    result = cv2.matchTemplate(image, template, cv2.TM_CCOEFF_NORMED)
    return result
```

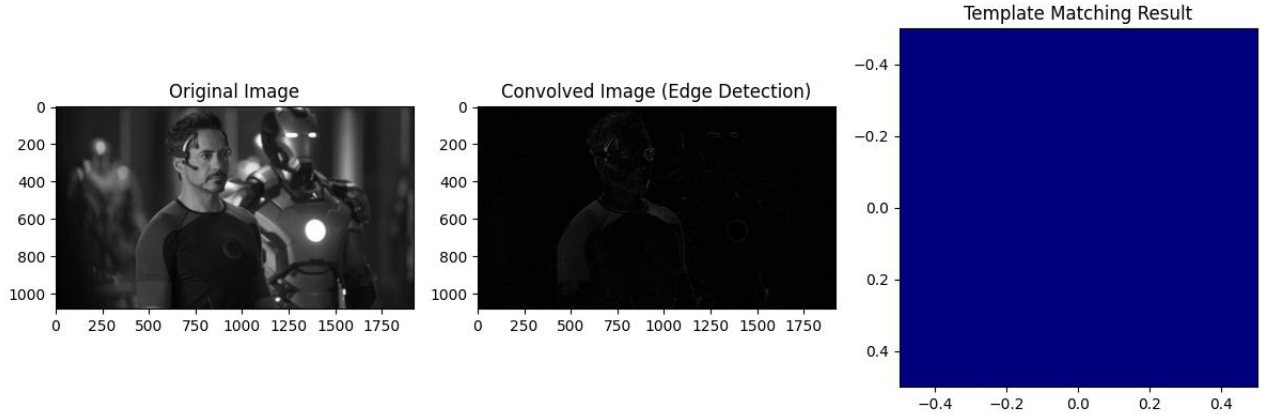
Load the image and template

```
image = cv2.imread('image.png', cv2.IMREAD_GRAYSCALE)
template = cv2.imread('image.png', cv2.IMREAD_GRAYSCALE)
```

Define a simple edge detection kernel for convolution

```
edge_detection_kernel = np.array([[-1, -1, -1],
[-1, 8, -1],
[-1, -1, -1]], dtype=np.float32)

# Convolution operation with the edge detection kernel
convolved_image = convolution(image, edge_detection_kernel)

# Template Matching
matched_result = template_matching(image, template)

# Display the original image, convolved image, and template matching result
plt.figure(figsize=(12, 4))

plt.subplot(131)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot(132)
plt.imshow(convolved_image, cmap='gray')
plt.title('Convolved Image (Edge Detection)')

plt.subplot(133)
plt.imshow(matched_result, cmap='jet') # Use 'jet' colormap for better visibility
plt.title('Template Matching Result')

plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO. 4

Aim: Write program to implement point/pixel intensity transformations such as:

1. Log and Power-law transformations
2. Contrast adjustments
3. Histogram equalization
4. Thresholding, and halftoning operations.

Description:

1. Log Transformation:
   - Transforms pixel values using logarithmic function
   - Compresses high pixel values more than lower values
   - Helps improve contrast
2. Power Law Transformation:
   - Transforms pixel values using power law equation
   - Gamma>1 increases contrast, gamma<1 decreases contrast
3. Contrast Adjustment:
   - Simplest method to enhance contrast
   - Adjusts intensity values by multiplying with scale factor and adding offset
4. Histogram Equalization:
   - Improves global contrast based on image histogram
   - Changes pixel values so that histogram is uniform
5. Thresholding:
   - Converts a grayscale image to a binary image
   - Values above threshold set to white, below set to black
6. Halftoning:
   - Used to represent a continuous-tone image with two levels only
   - Adds visually pleasing pattern using dithering

Mathematical Equations:

1. Log Transformation: s = c * log(1 + r)

   Where, s - output pixel value c - constant r - input pixel value

2. Power Law Transformation: s = c * r^γ

   Where, γ - gamma constant

3. Contrast Adjustment: $\sum j=0 kp(rj)\cdot(L-1)$

4. Histogram Equalization: $sk = \sum ni=0\ P(i)$

5. Thresholding: s = {0 if r < T {255 if r ≥ T

Code:

```
import cv2
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
```

1. Log and Power-law (Gamma) transformations

```
def Log_tranform(img_input,C ):
    Log_image = C * np.log((1 + img_input))
    return Log_image

def power_law_T(img_input,gamma):
    power_law_img = 1 * (img_input**gamma)
    return power_law_img
```

2. Contrast adjustment (contrast stretching)

```
def contrast_st(img_input):
    min_val = np.min(img_input)
    max_val = np.max(img_input)
    for i in range(img_input.shape[0]):
        for pix in range(img_input.shape[1]):
            img_input[i][pix] = ((img_input[i][pix] - min_val) / (max_val - min_val)) * 255

    stretched_image = img_input
    return stretched_image
```

3. Histogram computation and Histogram equalization

```
def calculate_histrogramgram(image):
    histrogramgram = np.zeros(256, dtype=int)
    for row in range(image.shape[0]):
        for pixel_value in range(image.shape[1]):
            histrogramgram[ image[row][pixel_value] ] += 1
    return histrogramgram

def histrogramgram_equalization(image):
    # Calculate the histrogramgram
    row, col = image.shape
    histrogramgram = calculate_histrogramgram(image)
    # Calculate the cumulative distribution function (CDF)
    pdf = histrogramgram / (row * col)
    cdf = np.cumsum(pdf)

    # Normalize CDF to 0-255
    h_eq = np.round(cdf * 255).astype(np.uint8)
    return h_eq
```

4.Thresholding and Halftoning operation

```
def thresolding(image,t : int):
    for i in range(image.shape[0]):
```

```
        for pix in range(image.shape[1]):
            if image[i][pix] <= t:
                image[i][pix] = 0
            else:
                image[i][pix] = 255
    return image

def halftone(image):
    # Define a dithering matrix
    dither_matrix = np.array([[0, 8, 2, 10],
                              [12, 4, 14, 6],
                              [3, 11, 1, 9],
                              [15, 7, 13, 5]])
    # Load the image in grayscale
    image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Initialize halftoned image
    halftoned_image = np.zeros_like(image_gray)
    # Apply halftoning
    for i in range(image_gray.shape[0]):
        for j in range(image_gray.shape[1]):
            if image_gray[i, j] > dither_matrix[i % 4, j % 4]:
                halftoned_image[i, j] = 255
    return halftoned_image
```

Display images

```
def display_histograms(hist_org, hist_eq):
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.bar(range(256), hist_org)
    plt.title('Histogram of Original Image')
    plt.subplot(1, 2, 2)
    plt.bar(range(256), hist_eq)
    plt.title('Histogram of Histogram Equalized Image')
    plt.show()

def display_images_only(g_img, log_image, law_img, threshold_img, c_st,
halftone_img):
    plt.figure(figsize=(15, 10))
    plt.subplot(3, 4, 1)
    plt.imshow(g_img, cmap='gray')
    plt.title('Gray org Image')
    plt.axis('off')
    plt.subplot(3, 4, 2)
    plt.imshow(log_image, cmap='gray')
    plt.title('Log Image')
    plt.axis('off')
    plt.subplot(3, 4, 3)
    plt.imshow(law_img, cmap='gray')
    plt.title('Law Image')
    plt.axis('off')
```

```
    plt.subplot(3, 4, 6)
    plt.imshow(threshold_img, cmap='gray')
    plt.title('Thresholding Image')
    plt.axis('off')
    plt.subplot(3, 4, 7)
    plt.imshow(c_st, cmap='gray')
    plt.title('Contrast Stretched Image')
    plt.axis('off')
    plt.subplot(3, 4, 8)
    plt.imshow(halftone_img, cmap='gray')
    plt.title('Half-toned Image')
    plt.axis('off')
    plt.show()
```

Display utility functions

```
img_path = "image.png"
Log_image = Log_tranform(np.array(cv2.imread(img_path,
cv2.IMREAD_GRAYSCALE)), 1)
power_law_img = power_law_T(np.array(cv2.imread(img_path,
cv2.IMREAD_GRAYSCALE)), 1.2)
contrast_img = contrast_st(np.array(cv2.imread(img_path,
cv2.IMREAD_GRAYSCALE)))
histrogram = calculate_histrogramgram(
np.array(cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)))
histrogram_eq = histrogramgram_equalization(
np.array(cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)))
t_image = thresolding(
np.array(cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)), 120)
halftone_img = halftone(
np.array(cv2.imread(img_path, cv2.IMREAD_COLOR)))
org_img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

display_images_only(
org_img,
Log_image,
power_law_img,
t_image,
contrast_img,
halftone_img)
display_histograms(
histrogram,
histrogram_eq
)
```

Output:


Gray org Image

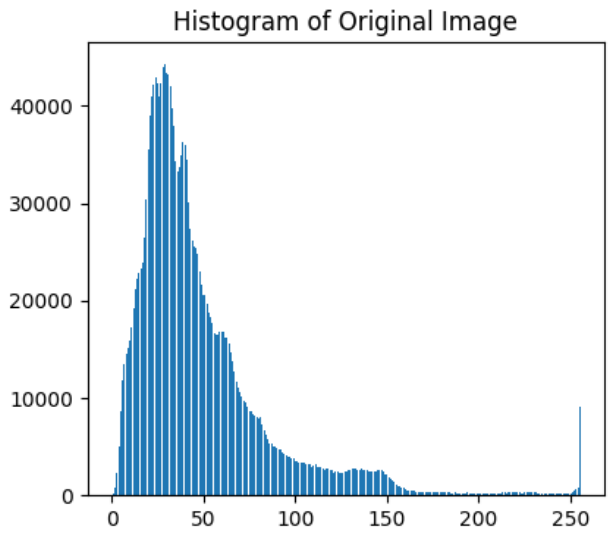

Log Image

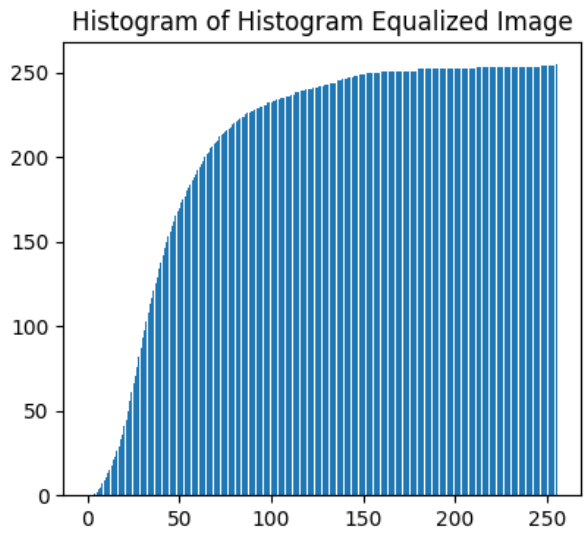

Law Image


Thresholding Image


Contrast Stretched Image


Half-toned Image


Histogram of Original Image
40000
30000
20000
10000
0
0
50
100
150
200
250


Histogram of Histogram Equalized Image
250
200
150
100
50
0
0
50
100
150
200
250

PRACTICAL NO. 5

Aim: To apply gradient and Laplacian for image enhancement

Description:

1. Gradient
   - Computes directional change in brightness/color in image
   - Useful to identify edges and boundary regions
   - We apply Sobel operator to find horizontal and vertical gradients
2. Laplacian
   - Computes second order derivative to find rapid change in values
   - Highlights regions of rapid changes like edges and textures
3. Enhanced Image
   - Blend the gradient and Laplacian images
   - Gradient shows edges and boundaries
   - Laplacian highlights internal texture
   - Blending accentuates edges, boundaries and internal details

Mathematical Equations:

1. Gradient Magnitude: $G = \sqrt{(Gx^2 + Gy^2)}$
2. Laplacian:
   $\nabla^2 f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$

Code:

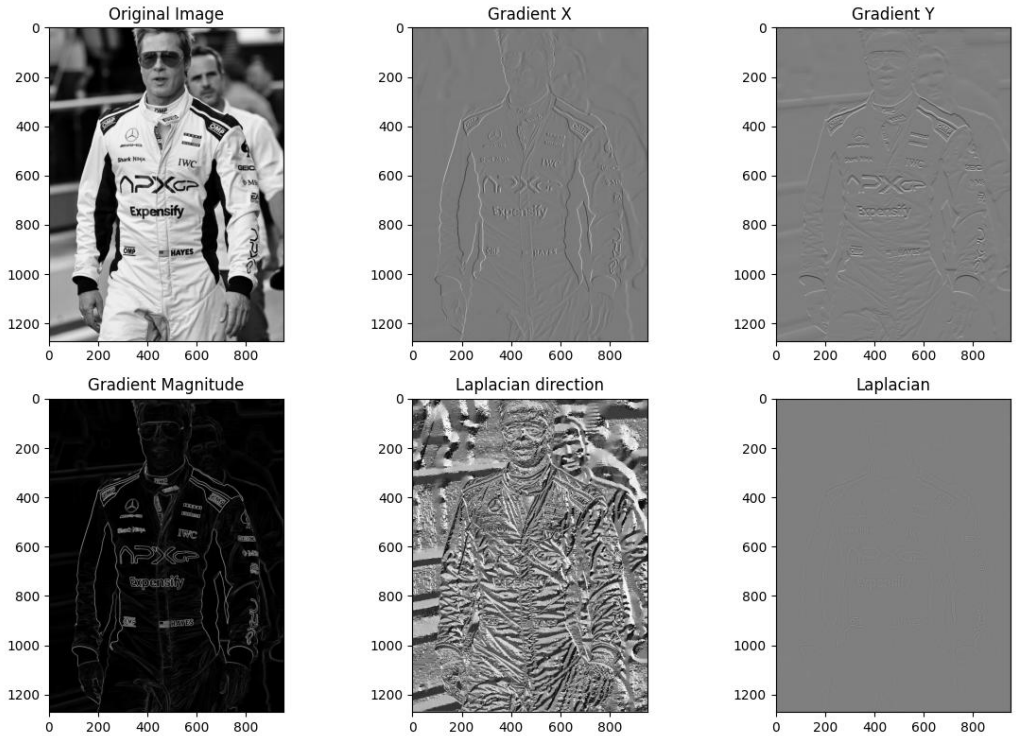
1. Laplacian

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage

# Load an image
image_path = "image2.png"
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Apply Gradient (Sobel) operation
gradient_x = cv2.Sobel(original_imge, cv2.CV_64F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(original_imge, cv2.CV_64F, 0, 1, ksize=3)

# Combine the gradients to get the magnitude and direction
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_direction = np.arctan2(gradient_y,gradient_x)

# Laplacian
laplacian_image = cv2.Laplacian(original_imge, cv2.CV_64F)
```

```
# Display the results
plt.figure(figsize=(12, 8))
plt.subplot(2, 3, 1)
plt.imshow(original_imge, cmap='gray')
plt.title('Original Image')
plt.subplot(2, 3, 2)
plt.imshow(gradient_x, cmap='gray')
plt.title('Gradient X')
plt.subplot(2, 3, 3)
plt.imshow(gradient_y, cmap='gray')
plt.title('Gradient Y')
plt.subplot(2, 3, 4)
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Gradient Magnitude')
plt.subplot(2, 3, 5)
plt.imshow(gradient_direction, cmap='gray')
plt.title('Laplacian direction')
plt.subplot(2, 3, 6)
plt.imshow(laplacian_image, cmap='gray')
plt.title('Laplacian')
plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO. 6

Aim: The aim of this program is to demonstrate two common techniques used for noise reduction in images: linear smoothing and nonlinear smoothing.

Description:

1. Linear Smoothing: Linear smoothing techniques like Gaussian blur work by convolving the image with a kernel that gives more weight to nearby pixels and less weight to distant pixels. This effectively blurs the image, reducing high-frequency noise.
2. Nonlinear Smoothing: Nonlinear smoothing techniques, such as median filtering, replace each pixel value with the median value of its neighborhood. This method is effective at removing salt- and-pepper noise, where random pixels are either very bright or very dark.

Mathematical Equation:

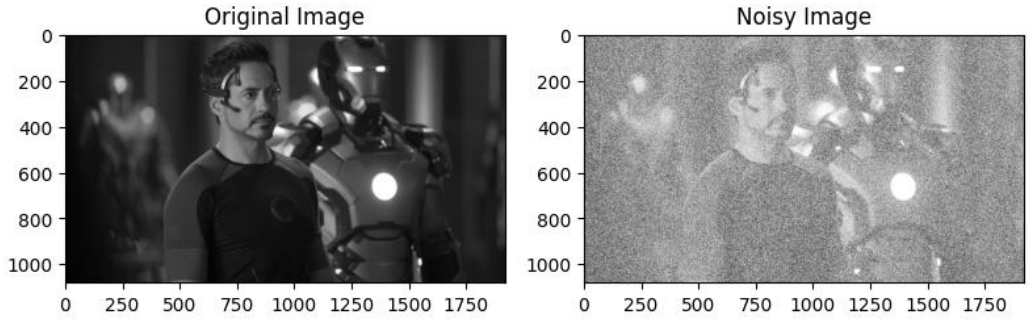
1. Linear Smoothing using Gaussian Blur:
   Smoothed Image=GaussianBlur(Image,Kernel Size)Smoothed Image=GaussianBlur(Image,Kernel Size) Here, the GaussianBlur function applies Gaussian smoothing to the image using a specified kernel size.

2. Nonlinear Smoothing using Median Filter:
   Smoothed Image=medianBlur(Image,Kernel Size)Smoothed Image=medianBlur(Image,Kernel Siz e) The medianBlur function replaces each pixel value with the median value of its neighborhood, specified by the kernel size.

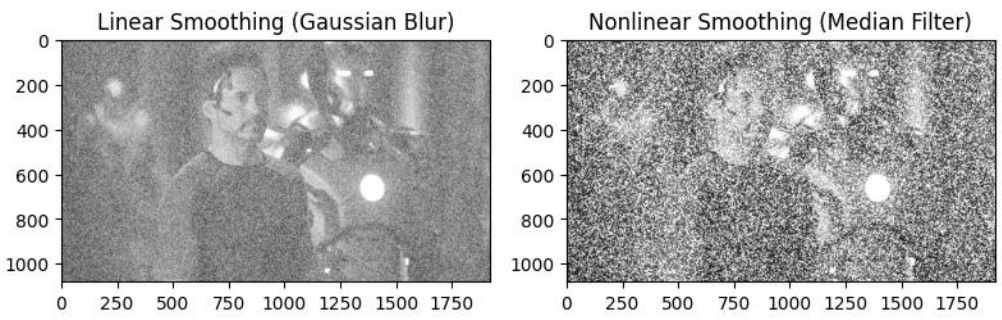
Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage

image_path = 'image.png'
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Check if the image is loaded successfully
if original_imge is None:
    print("Error: Image not found or could not be loaded.")
    exit()

# Add Gaussian noise to the image
noise = np.random.normal(0, 25, original_imge.shape).astype('uint8')
noisy_image = cv2.add(original_imge, noise)

# Function to apply linear (Gaussian blur) smoothing
def linear_smoothing(image, kernel_size):
    return cv2.GaussianBlur(image, (kernel_size, kernel_size), 0)
# Function to apply nonlinear (median filter) smoothing
def nonlinear_smoothing(image, kernel_size):
```

```
    return cv2.medianBlur(image, kernel_size)

# Apply linear smoothing (Gaussian blur)
linear_smoothed_image = linear_smoothing(noisy_image, kernel_size=5)

# Apply nonlinear smoothing (median filter)
nonlinear_smoothed_image = nonlinear_smoothing(noisy_image, kernel_size=5)

# Display the results
plt.figure(figsize=(12, 6))

plt.subplot(2, 3, 1)
plt.imshow(original_imge, cmap='gray')
plt.title('Original Image')

plt.subplot(2, 3, 2)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

plt.tight_layout()
plt.show()
```




```
# Display the results
plt.figure(figsize=(12, 6))
plt.subplot(2, 3, 1)
plt.imshow(linear_smoothed_image, cmap='gray')
plt.title('Linear Smoothing (Gaussian Blur)')
plt.subplot(2, 3, 2)
plt.imshow(nonlinear_smoothed_image, cmap='gray')
plt.title('Nonlinear Smoothing (Median Filter)')
plt.tight_layout()
plt.show()
```



PRACTICAL NO. 7

Aim: The aim of this program is to demonstrate various image enhancement techniques using image derivatives, including smoothing, sharpening, and unsharp masking, to generate images suitable for specific application requirements.

Description:

1. Smoothing: Smoothing techniques, such as Gaussian blur, help to reduce noise and make the image appear more uniform by averaging pixel values over a defined neighborhood. This is useful for tasks like preprocessing before feature extraction or reducing the impact of noise in the image.

2. Sharpening: Sharpening enhances edges and fine details in the image, making it appear clearer and more defined. This is achieved by increasing the contrast around edges. Sharpening is commonly used to improve the visual quality of images for visualization or printing purposes.

3. Unsharp Masking: Unsharp masking is a technique used to enhance edges by subtracting a blurred version of the image from the original and then adding the result back to the original image. This emphasizes edges and enhances image details while reducing noise. Unsharp masking is particularly effective for improving the visual appearance of images for specific applications such as medical imaging or microscopy.

Mathematical Equations:

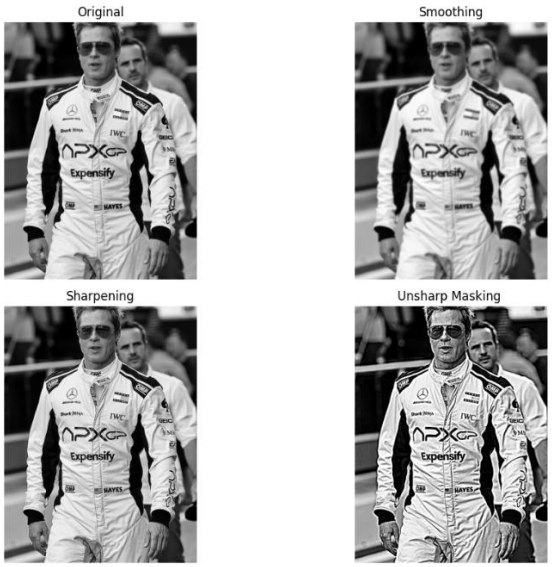
- Smoothing using Gaussian Blur:
  Smoothed Image=GaussianBlur(Image,Kernel Size)Smoothed Image=GaussianBlur(Image,Kernel Size)
- Sharpening using a Custom Kernel:
  Sharpened Image=Image∗KernelSharpened Image=Image∗Kernel
- Unsharp Masking:
  Unsharp Masked Image=Image+Strength×(Image−Smoothed Image)Unsharp Masked Image=Ima ge+Strength×(Image−Smoothed Image)

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# load image in grayscale
image_path = 'image3.jpg'
original_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# -------- FILTER FUNCTIONS --------
def apply_smoothing(image, kernel_size):
    return cv2.GaussianBlur(image, (kernel_size, kernel_size), 0)

def apply_sharpening(image):
    kernel = np.array([
        [-1, -1, -1],
        [-1,  9, -1],
        [-1, -1, -1]
```

```
    ])
    return cv2.filter2D(image, -1, kernel)

def apply_unsharp_masking(image, sigma=4.0, strength=5.0):
    blurred = cv2.GaussianBlur(image, (0, 0), sigma)
    sharpened = cv2.addWeighted(image, 1.0 + strength, blurred, -strength, 0)
    return sharpened
# -------- APPLY FILTERS --------
smoothed_image = apply_smoothing(original_image, kernel_size=17)
sharpened_image = apply_sharpening(original_image)
unsharp_masked_image = apply_unsharp_masking(original_image)
# -------- DISPLAY OUTPUT --------
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(smoothed_image, cmap='gray')
plt.title('Smoothing')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(sharpened_image, cmap='gray')
plt.title('Sharpening')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(unsharp_masked_image, cmap='gray')
plt.title('Unsharp Masking ')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO. 8

Aim: The aim of this program is to extract meaningful information from given image samples using edge detection techniques such as Sobel and Canny.

Description:

- Edge detection is a fundamental image processing technique used to identify boundaries within images. It plays a crucial role in various computer vision tasks, including object detection, image segmentation, and feature extraction.

- Sobel and Canny are two popular edge detection algorithms. Sobel operator computes an approximation of the gradient of the image intensity function, highlighting areas of rapid intensity change. Canny edge detection is a multi-stage algorithm that identifies edges while suppressing noise and detecting the true edges more accurately.

Mathematical Equations:

1. Sobel Edge Detection: Sobel_x = Sobel(Image, CV_64F, 1, 0, ksize=3) Sobel_y = Sobel(Image, CV_64F, 0, 1, ksize=3) Gradient Magnitude = sqrt(Sobel_x^2 + Sobel_y^2) Gradient Direction = arctan(Sobel_y / Sobel_x)

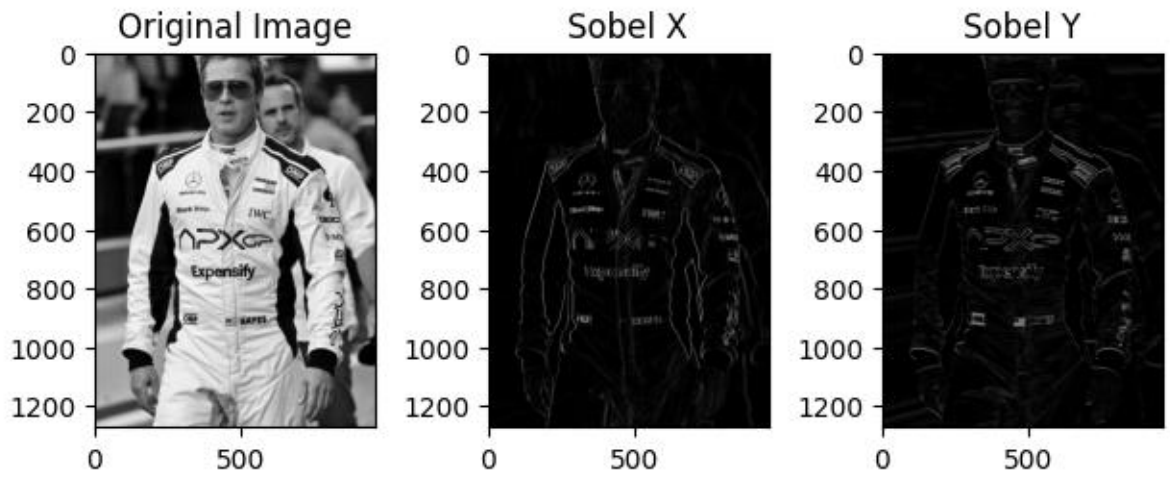
2. Canny Edge Detection: Edges = Canny(Image, Threshold1, Threshold2)

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2

image_path = 'image3.jpg'
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Apply Sobel edge detection
sobel_x = cv2.Sobel(original_imge, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(original_imge, cv2.CV_64F, 0, 1, ksize=3)

sobel_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)

sobel_direction = np.arctan2(sobel_y,sobel_x)
# Apply Canny edge detection
canny_edges = cv2.Canny(original_imge, 50, 150)

# Display the results
plt.figure(figsize=(12, 6))
```

```
plt.subplot(2, 3, 1)
plt.imshow(original_imge, cmap='gray')
plt.title('Original Image')

plt.subplot(2, 3, 2)
plt.imshow(np.abs(sobel_x), cmap='gray')
plt.title('Sobel X')

plt.subplot(2, 3, 3)
plt.imshow(np.abs(sobel_y), cmap='gray')
plt.title('Sobel Y')

plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO. 9

Aim: The aim of this program is to implement various morphological image processing techniques to manipulate the shapes and structures within an image.

Description:

1. Morphological image processing is a set of operations that modify the geometric structure of an image. These operations are based on the shape of the structuring element (kernel) and include erosion, dilation, opening, closing.

2. Erosion shrinks the shapes in the image, while dilation expands them.

3. Opening is erosion followed by dilation, which helps in removing noise and small objects.

4. Closing is dilation followed by erosion, which helps in closing small gaps and joining nearby objects.

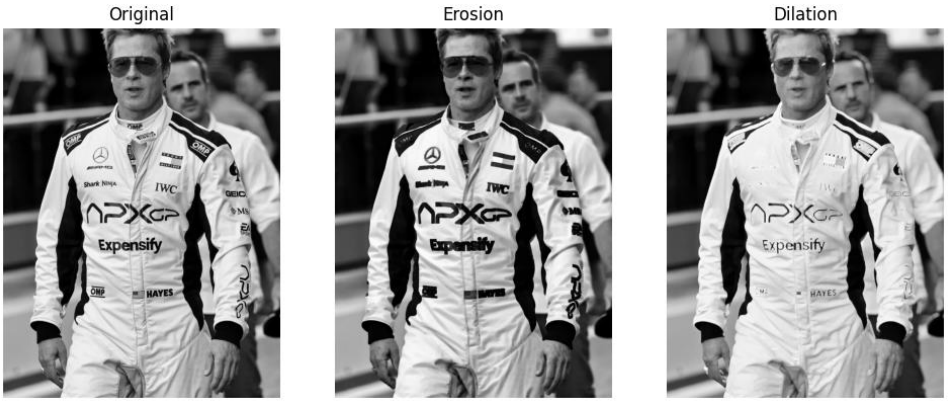
Mathematical Equations:

1. Erosion: erosion(x, y) = min_{(s, t) in SE} {image(x + s, y + t)}
2. Dilation: dilation(x, y) = max_{(s, t) in SE} {image(x + s, y + t)}
3. Opening: opening = dilation(erosion(image))
4. Closing: closing = erosion(dilation(image))

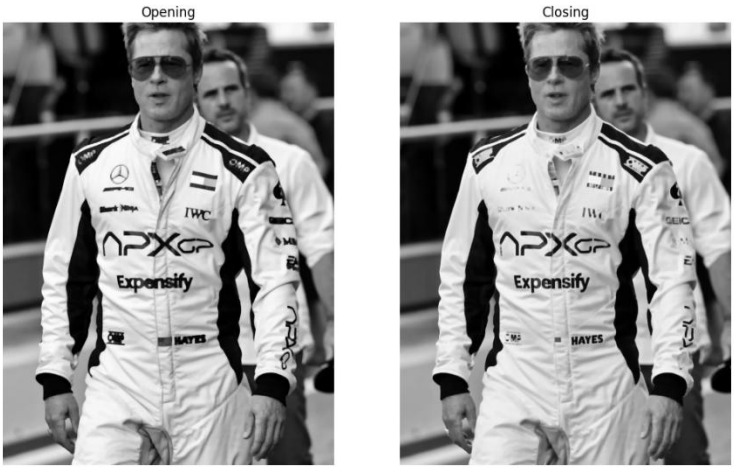
Code:

```
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2
import os

image_path = 'image3.jpg'
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

image_path = 'image3.jpg'
if not os.path.exists(image_path):
  raise FileNotFoundError(f"File not found: {image_path}")

original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
if original_imge is None:
  raise ValueError("Image failed to load")

def display_images(images, titles):
  plt.figure(figsize=(12, 8))
```

```
  for i in range(len(images)):
    plt.subplot(1, len(images), i + 1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')
  plt.show()

def apply_morphological_operations(image):
  kernel = np.ones((5, 5), np.uint8)
  erosion = cv2.erode(image, kernel, iterations=1)
  dilation = cv2.dilate(image, kernel, iterations=1)
  opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
  closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
  return [image, erosion, dilation, opening, closing]

morph_images = apply_morphological_operations(original_imge)
titles = ["Original", "Erosion", "Dilation", "Opening", "Closing"]
display_images(morph_images[:3], titles[:3])
```




```
display_images(morph_images[3:], titles[3:])
```



PRACTICAL NO. 10

Aim: Implement feature detectors and descriptors for image analysis.

Description:

Feature extraction simplifies image analysis by extracting useful information from images.

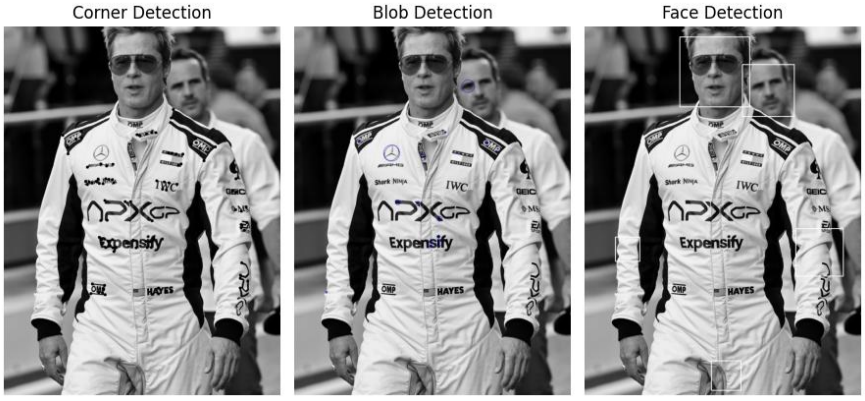
1. Corner detection identifies interest points using Harris corner detector.
2. Blob detection uses Difference of Gaussian (DoG) to find bright regions on dark backgrounds.
3. HoG extracts shape and texture information.
4. Haar features encode contrast patterns in image.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2
import os

image_path = 'image3.jpg'
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Corner Detection
corners = cv2.goodFeaturesToTrack(original_imge, 100, 0.01, 10)
corners = corners.reshape(-1, 2)
corner_image = original_imge.copy()
for corner in corners:
  x, y = corner
  cv2.circle(corner_image, (int(x), int(y)), 5, (0, 255, 0),-1)

# Blob Detection
detector = cv2.SimpleBlobDetector_create()
keypoints = detector.detect(original_imge)
blob_image = cv2.drawKeypoints(original_imge, keypoints, None, (0, 0, 255),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Haar Features (Face Detection)
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
faces = face_cascade.detectMultiScale(original_imge, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))
face_image = original_imge.copy()
for (x, y, w, h) in faces:
  cv2.rectangle(face_image, (x, y), (x + w, y + h), (255, 0, 0), 2)

# Display results using matplotlib
```

```
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 4, 1)
plt.imshow(cv2.cvtColor(original_imge, cv2.COLOR_GRAY2BGR))
plt.title('Original Image')
plt.axis('off')
```


```
plt.figure(figsize=(12, 6))
plt.subplot(1, 4, 2)
plt.imshow(cv2.cvtColor(corner_image, cv2.IMREAD_GRAYSCALE))
plt.title('Corner Detection')
plt.axis('off')
# Blob Detection Image
plt.subplot(1, 4, 3)
plt.imshow(cv2.cvtColor(blob_image, cv2.IMREAD_GRAYSCALE))
plt.title('Blob Detection')
plt.axis('off')
# Face Detection Image
plt.subplot(1, 4, 4)
plt.imshow(cv2.cvtColor(face_image, cv2.IMREAD_GRAYSCALE))
plt.title('Face Detection')
plt.axis('off')
plt.tight_layout()
plt.show()
```



PRACTICAL NO. 11

Aim: The aim of this program is to apply segmentation techniques to detect lines, circles, and other shapes/objects in an image using both edge-based and region-based segmentation methods.

Description:

Segmentation is a process of partitioning an image into multiple regions or segments to simplify its representation and make it easier to analyze. In this program, we use two segmentation techniques: edge- based segmentation and region-based segmentation.

1. Edge-based Segmentation: Edge-based segmentation relies on detecting edges in an image. We use the Canny edge detection algorithm to identify strong gradients (edges) in the image. These edges are then used to separate different objects in the image.

2. Region-based Segmentation: Region-based segmentation partitions an image into regions based on properties such as intensity, color, or texture. In this program, we use thresholding to create binary regions in the image. Areas with intensities above a certain threshold are considered one region, while those below the threshold are considered another.

Mathematical Equations:

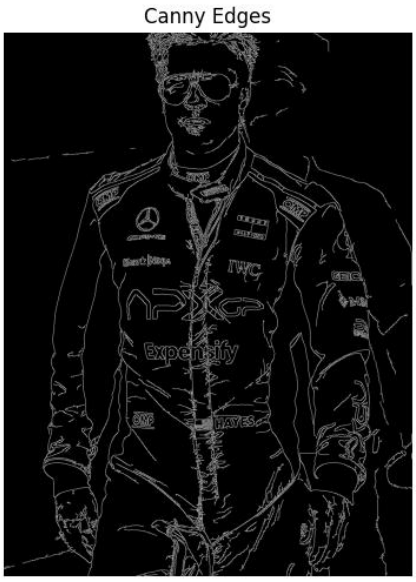
1. Edge-based Segmentation:
   - Canny Edge Detection:
     Edges=Canny(Image,Threshold1,Threshold2)Edges=Canny(Image,Threshol d1,Threshold2
)

2. Region-based Segmentation:
   - Thresholding:
     Thresholded Image=Threshold(Gray Image,Threshold Value,Max Value,Threshold Type)T hresholded Image=Threshold(Gray Image,Threshold Value,Max Value,Threshold Type)

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
import cv2
import os

image_path = 'image3.jpg'
original_imge = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

def display_images(images, titles):
```

```
    plt.figure(figsize=(12, 6))
    for i in range(len(images)):
        plt.subplot(1, len(images), i + 1)
        plt.imshow(images[i], cmap='gray')
        plt.title(titles[i])
        plt.axis('off')
    plt.show()

def edge_based_segmentation(image):
    edges = cv2.Canny(image, 50, 150)
    lines = cv2.HoughLinesP(edges, 1, np.pi/180, threshold=50, minLineLength=50,
maxLineGap=10)
    lines_image = np.copy(image)
    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line[0]
            cv2.line(lines_image, (x1, y1), (x2, y2), 255, 2)
    return [edges, lines_image], ['Canny Edges', 'Detected Lines']

def region_based_segmentation(image):
    # Thresholding to create a binary image
    _, binary_image = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)
    # Find contours of objects
    contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)
    # Draw contours on a black image
    contour_image = np.zeros_like(image)
    cv2.drawContours(contour_image, contours,-1, 255, 2)
    return [binary_image, contour_image], ['Binary Image', 'Detected Contours']

edge_segmentation_result, edge_segmentation_titles =
edge_based_segmentation(original_imge)
# Apply region-based segmentation
region_segmentation_result, region_segmentation_titles =
region_based_segmentation(original_imge)
# Display the results
display_images(edge_segmentation_result, edge_segmentation_titles)
```



display_images(region_segmentation_result, region_segmentation_titles)