Padmashri Annasaheb Jadhav Bharatiya Samaj Unnati Mandal's

B. N. N. College of Arts, Science & Commerce, Bhiwandi .

(Self Funded Courses)

(Department of Computer Science)

CERTIFICATE

This is to certify that Mr. / Miss. ANSARI MOHAMMAD ISA

Roll No. _____ Exam Seat No . _____

has satisfactorily completed the Practical in APPLIED SIGNAL & IMAGE PROCESSING

As laid down in the regulation of University of Mumbai for the purpose of _____

Class: M. Sc. C. S. - I

Examination 20 25 - 20 26

Date: /12/2025

Place: BHIWANDI

_____ _____

Professor In-charge Signature of HOD

Signature of External Examiners

1)

2)

INDEX

Sr. No.	Date	Practical / Assignment	Remark
1	19/11/25	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	
2	19/11/25	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.	
3	21/11/25	Write a program to apply various enhancements on images	
		using image derivatives by implementing Gradient and	
		Laplacian operations.	
4	21/11/25	Write a program to implement linear and nonlinear noise	
		smoothing on suitable image or sound signal.	
5	21/11/25	Write the program to implement various morphological	
		image processing techniques.	

B. N. N. COLLEGE, BHIWANDI Page No. :

Practical No. : 01 **Roll No. :**

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

Introduction:

Understanding the Task

Before we dive into the code, let's break down the two tasks:

1. Upsampling and Downsampling:

- Upsampling: Increasing the sampling rate of a signal, effectively adding more samples between existing ones.
- Downsampling: Decreasing the sampling rate, removing samples from the original signal.

2. Fast Fourier Transform (FFT):

- An efficient algorithm to compute the Discrete Fourier Transform (DFT).
- DFT transforms a signal from the time domain to the frequency domain.

Implementing in Python with Libraries

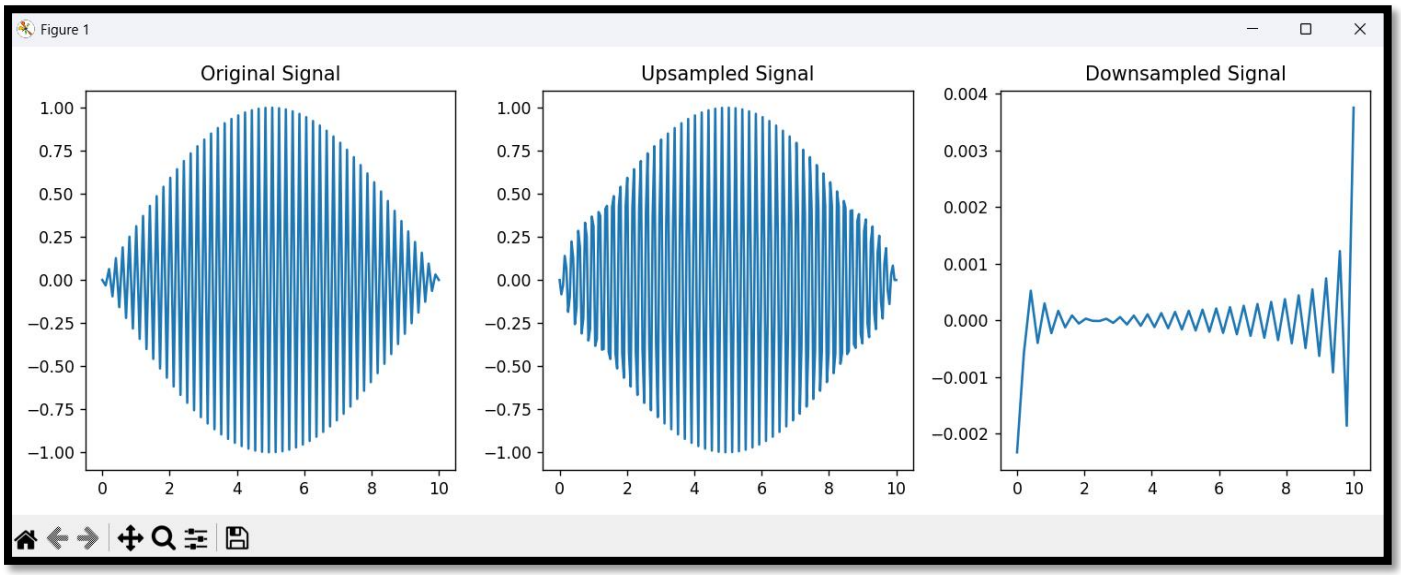
We'll use Python with libraries like NumPy, SciPy, and Matplotlib for efficient signal processing and visualization.

1. Upsampling and Downsampling on Image/speech signal

CODE:

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

# Sample signal (a sine wave)
t = np.linspace(0, 10, 100)
x = np.sin(2 * np.pi * 5 * t)

# Upsampling
up_factor = 2
x_up = signal.resample(x, len(x) * up_factor)

# Downsampling
down_factor = 2
x_down = signal.resample(x, len(x) // down_factor)

# Plotting
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.plot(t, x)
plt.title('Original Signal')

plt.subplot(1, 3, 2)
plt.plot(np.linspace(0, 10, len(x_up)), x_up)
plt.title('Upsampled Signal')

plt.subplot(1, 3, 3)
plt.plot(np.linspace(0, 10, len(x_down)), x_down)
plt.title('Downsampled Signal')

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

OUTPUT:

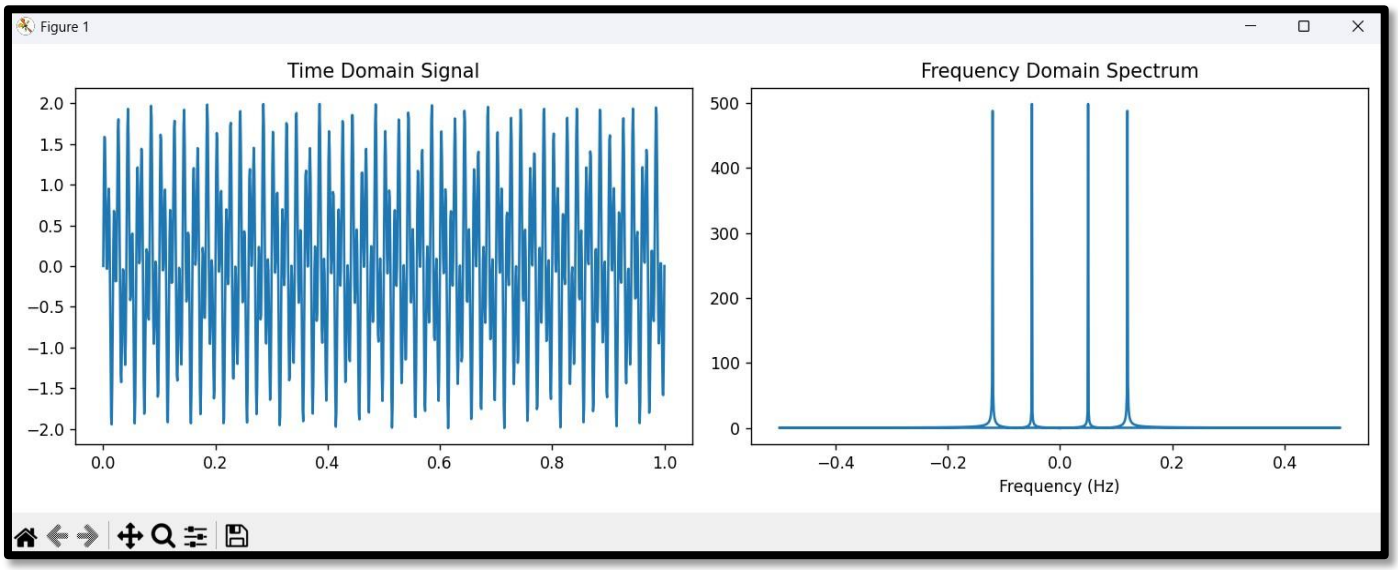


2. Fast Fourier Transform to compute DFT.

CODE:

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

# Sample signal (a sum of sine waves)
t = np.linspace(0, 1, 1000)
x = np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)

# Compute FFT
X = np.fft.fft(x)
freq = np.fft.fftfreq(len(x))

# Plotting
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(t, x)
plt.title('Time Domain Signal')

plt.subplot(1, 2, 2)
plt.plot(freq, np.abs(X))
plt.title('Frequency Domain Spectrum')
plt.xlabel('Frequency (Hz)')

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

OUTPUT:



B. N. N. COLLEGE, BHIWANDI **Page No. :**

Practical No. : 02 **Roll No. :**

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

Introduction:

1. Creating a Triangle Signal

A triangle signal (or triangular wave) is a non-sinusoidal waveform named for its triangular shape. It is a periodic, piecewise linear, and continuous function.

Key Characteristics:

- **Periodicity:** The signal repeats at regular intervals.
- **Amplitude:** The maximum absolute value of the signal's magnitude.
- **Symmetry:** A perfect triangle wave typically rises linearly from a minimum to a maximum and falls linearly back to the minimum over one period.

Plotting a 3-Period Segment:

To plot a specific segment, you define the time axis t over a range that spans at least three times the period T, ensuring sufficient sampling points to accurately represent the sharp changes in slope.

2. Correlation between Signals

Correlation, specifically **cross-correlation**, measures the similarity between two signals as a function of the time difference (lag) applied to one of them. It is a fundamental tool for detecting patterns, identifying features, and determining time delays between signals.

Interpretation:

The result of the cross-correlation is an array (or function) of correlation coefficients for different lags.

- A high value at a specific lag indicates a strong similarity between the signals when one is shifted by that lag.
- The maximum value of the correlation often reveals the time delay between the two signals.

Normalization:

Often, the correlation is normalized (resulting in a coefficient between -1 and 1) to provide a standard measure of similarity, independent of signal power or amplitude. A value of 1 means perfect positive correlation, -1 means perfect negative correlation (inversion), and 0 means no linear correlation.

CODE:

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

# 1. Triangle Wave
def triangle_wave(x, period):
    return 2 * np.abs(x % period - period/2) / period - 1

# Parameters
period = 1
amplitude = 1
num_periods = 3
num_samples = 1000

# Generate and plot triangle wave
t = np.linspace(0, num_periods * period, num_samples)
signal = amplitude * triangle_wave(t, period)
plt.plot(t, signal)
plt.title('Triangle Wave')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.show()

# 2. Correlation
# Sample signal
signal = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Define segment and compute correlation
segment = signal[2:5]
correlation = np.correlate(signal, segment, mode='full')

# Plot signal, segment, and correlation
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
plt.plot(signal)
plt.plot(np.arange(2, 5), segment, 'ro')
plt.title('Original Signal and Segment')

plt.subplot(2, 1, 2)
plt.plot(correlation)
plt.title('Correlation')

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

OUTPUT:

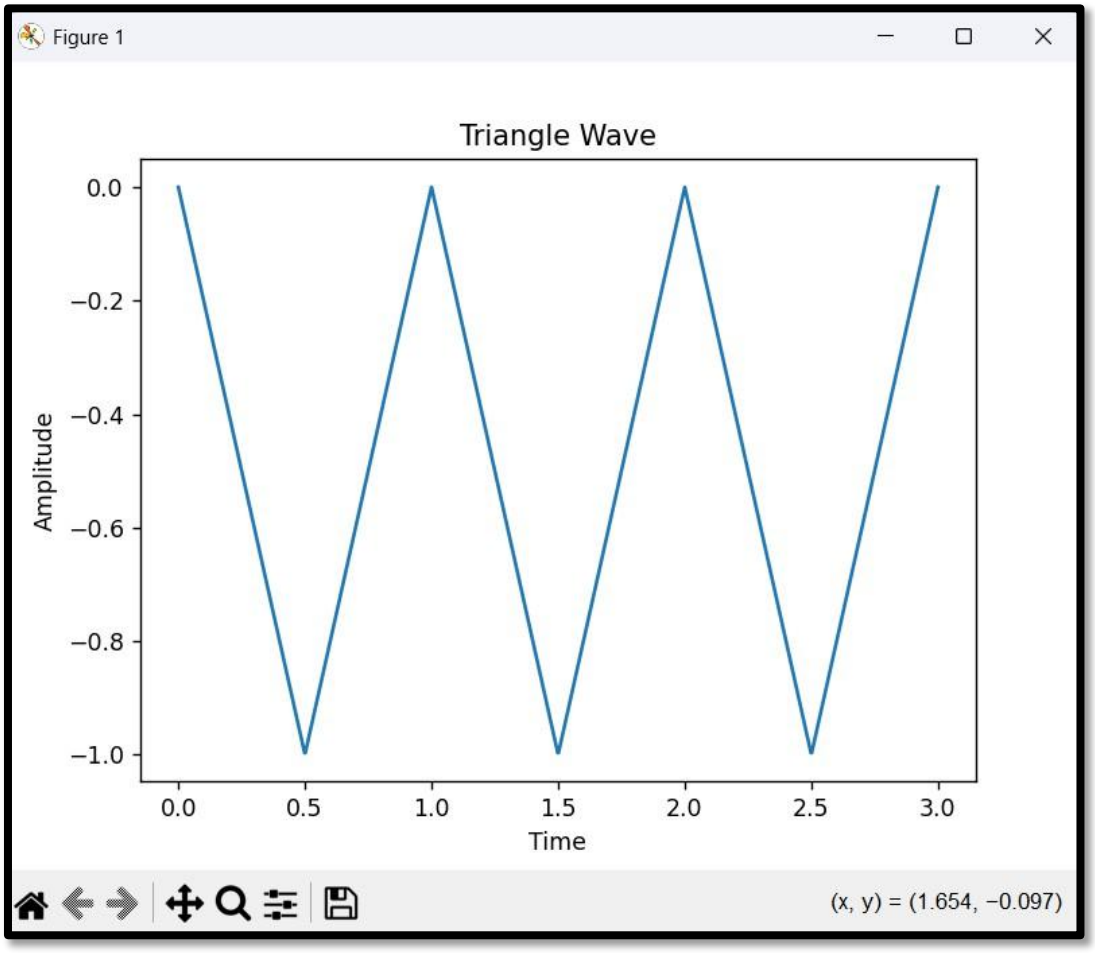


Figure 1
Triangle Wave
Amplitude
Time
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
(x, y) = (1.654, −0.097)

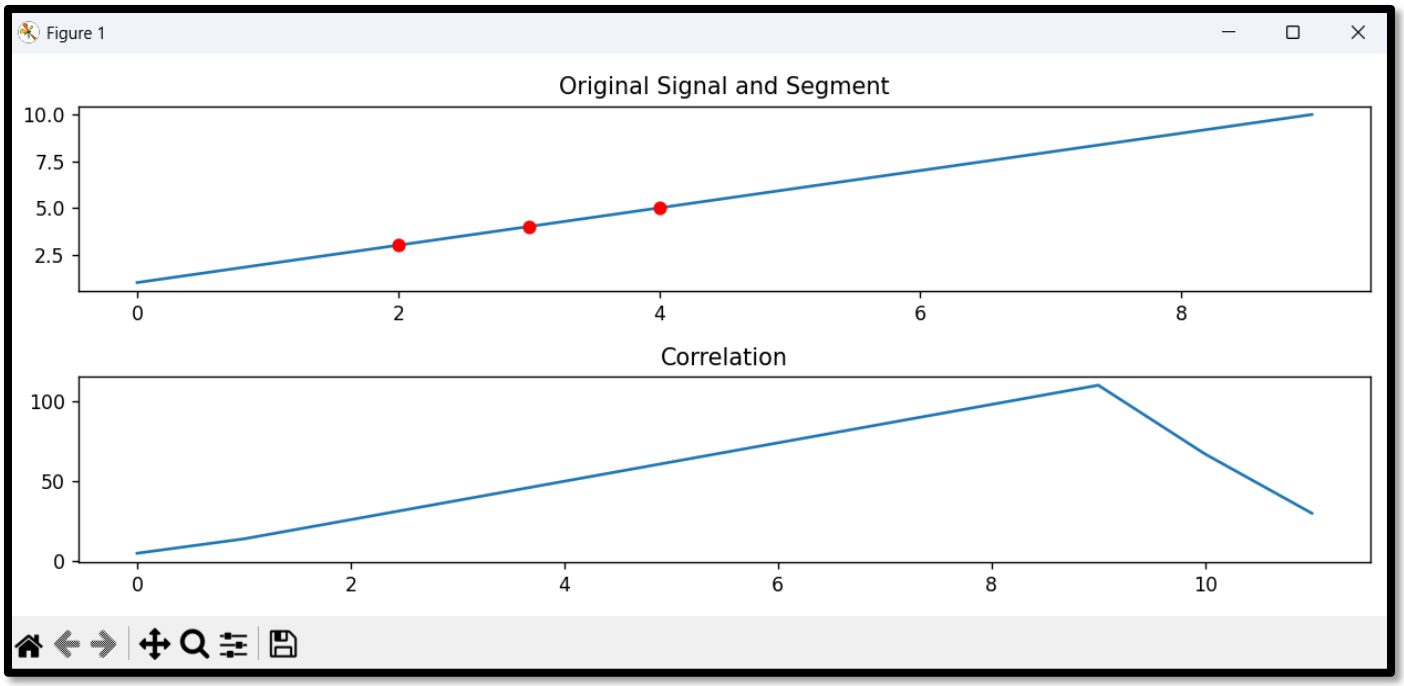


Figure 1
Original Signal and Segment
10.0
7.5
5.0
2.5
0
2
4
6
8
Correlation
100
50
0
0
2
4
6
8
10

B. N. N. COLLEGE, BHIWANDI Page No. :

Practical No. : 03 **Roll No. :**

Aim: Write a program to apply various enhancements on images using image derivatives by implementing Gradient and Laplacian operations.

Introduction:

Explanation of the Code:

1. Synthetic Image Creation:

- A 256x256 pixel black image is created using np.zeros().
- A white square is drawn on this black image using cv2.rectangle(). This square has pixel intensity 255 (white), and its position is from (50, 50) to (200, 200).

2. Gradient Calculation (Sobel Operator):

- The Sobel operator is applied in both the X and Y directions to detect edges
- The gradient magnitude is calculated by combining the Sobel outputs in both directions.
- The result is normalized to fit within the 8-bit range (0–255) and converted to an unsigned 8-bit image.

3. Laplacian Calculation:

- The Laplacian operator is applied to the image to compute the second derivative, which highlights rapid intensity changes (edges).
- Similar to the gradient image, the Laplacian result is normalized and converted to an 8-bit image.

4. Display the Results:

- The original image, gradient image, and Laplacian image are displayed using matplotlib.pyplot

Output:

- The original image (a white square on a black background).
- The gradient image which highlights the edges of the square.
- The Laplacian image which also highlights the edges but emphasizes more prominent changes.

CODE:

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

# Create a simple synthetic image (a white square on a black background)
image = np.zeros((256, 256), dtype=np.uint8)  # Create black image (background)
cv2.rectangle(image, (50, 50), (200, 200), 255, -1)  # Draw a white square

def apply_gradient(image):
    # Applying Sobel filter in both horizontal and vertical directions
    sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)  # Gradient in X direction
    sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)  # Gradient in Y direction

    # Magnitude of the gradient
    gradient_magnitude = cv2.magnitude(sobel_x, sobel_y)

    # Normalize and convert to 8-bit image
    gradient_magnitude = np.uint8(np.absolute(gradient_magnitude))

    return gradient_magnitude

def apply_laplacian(image):
    # Apply Laplacian operator
    laplacian = cv2.Laplacian(image, cv2.CV_64F)

    # Normalize and convert to 8-bit image
    laplacian = np.uint8(np.absolute(laplacian))

    return laplacian

# Apply Gradient and Laplacian
gradient_image = apply_gradient(image)
laplacian_image = apply_laplacian(image)

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

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

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

plt.subplot(1, 3, 3)
plt.imshow(laplacian_image, cmap='gray')
plt.title('Laplacian Image')

plt.show()
```

OUTPUT:

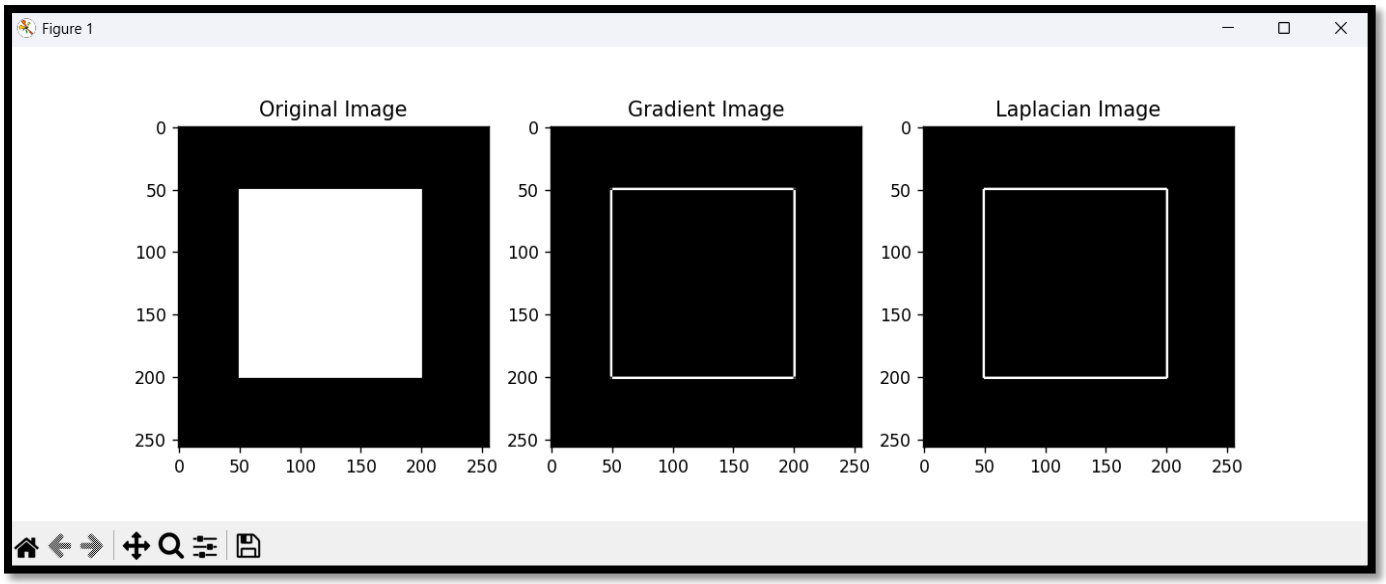


Figure 1
Original Image
Gradient Image
Laplacian Image
0
50
100
150
200
250

B. N. N. COLLEGE, BHIWANDI Page No. :

Practical No. : 04 **Roll No. :**

Aim: Write a program to implement linear and nonlinear noise smoothing on suitable image or sound signal

Introduction:

Here's a more minimal version of the code that uses a sine wave with added white noise, applies linear smoothing (moving average) and nonlinear smoothing (median filter), and plots the results. I've simplified the code and removed unnecessary comments and variables.

1. Signal Generation:

- A sine wave of frequency 440 Hz (A4 note) is generated using np.sin()
- White noise is added using np.random.normal() with a mean of 0 and standard deviation of 0.5

2. Smoothing:

- Linear smoothing is done using a moving average filter (np.convolve()).
- Nonlinear smoothing is done using a median filter (signal.medfilt()).

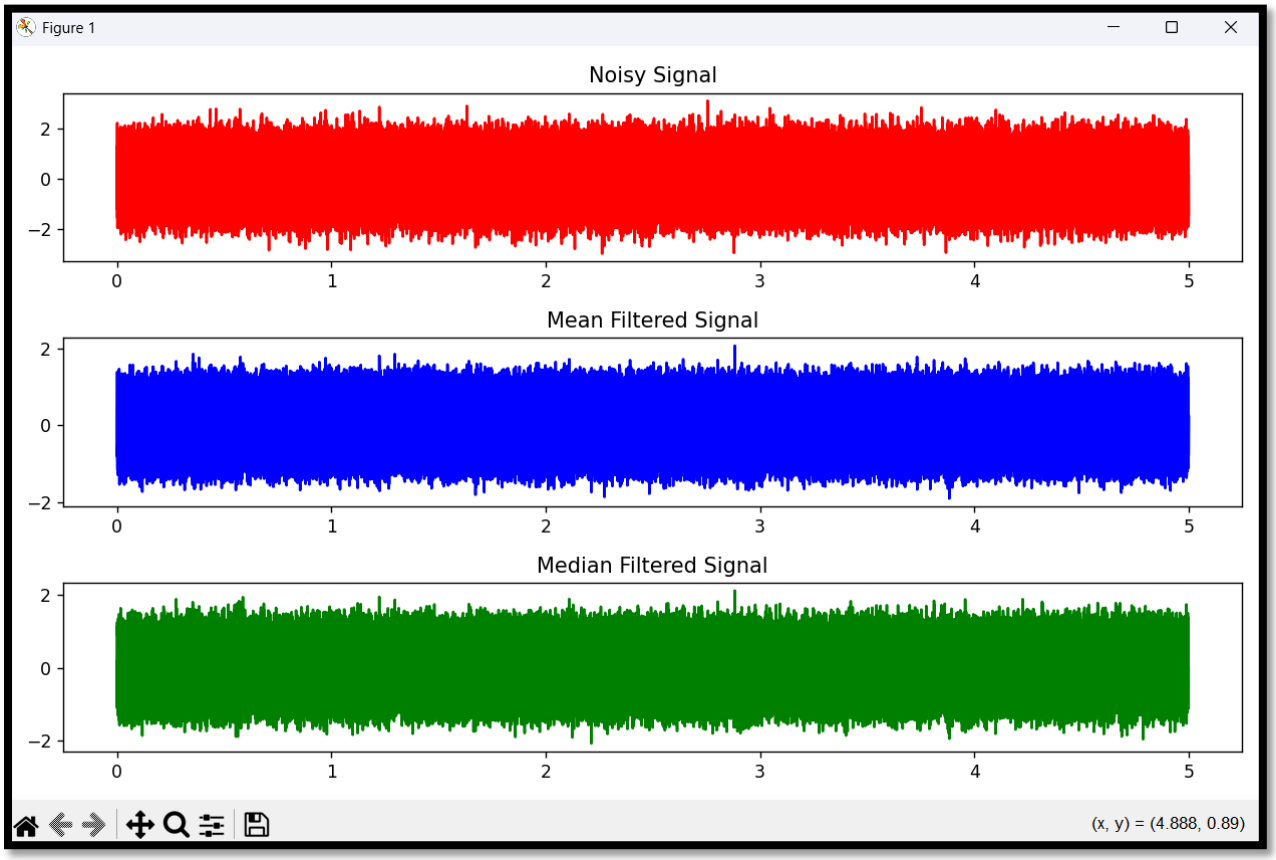
3. Plotting:

- Three subplots are generated:
    - Noisy signal (red)
    - Mean filtered signal (blue)
    - Median filtered signal (green)

CODE:

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

# Generate sine wave + white noise
fs = 44100  # Sampling frequency
t = np.linspace(0, 5, fs * 5, endpoint=False)
sine_wave = np.sin(2 * np.pi * 440 * t)
noisy_signal = sine_wave + np.random.normal(0, 0.5, t.shape)

# Apply linear smoothing (moving average)
mean_smoothed = np.convolve(noisy_signal, np.ones(5)/5, mode='same')

# Apply nonlinear smoothing (median filter)
kernel_size=5
median_smoothed = signal.medfilt(noisy_signal, kernel_size=5)

# Plot
plt.figure(figsize=(10, 6))
plt.subplot(3, 1, 1), plt.plot(t, noisy_signal, 'r')
plt.title('Noisy Signal')
plt.subplot(3, 1, 2), plt.plot(t, mean_smoothed, 'b')
plt.title('Mean Filtered Signal')
plt.subplot(3, 1, 3), plt.plot(t, median_smoothed, 'g')
plt.title('Median Filtered Signal')
plt.tight_layout()
plt.show()
```

OUTPUT:



B. N. N. COLLEGE, BHIWANDI Page No. :

Practical No. : 05 **Roll No. :**

Aim: Write the program to implement various morphological image processing techniques.

Introduction:

- Image Creation: This code creates a simple binary image where a white rectangle is drawn on a black background. This is done using cv2.rectangle().
- Morphological Operations: The same morphological operations are applied on this generated image.
- Result: You will see the results of the morphological operations on the synthetic Image (white rectangle on a black background).

Output:

The code will display a 3x3 grid of images with the following operations:

- Original Binary Image
- Erosion
- Dilation
- Opening
- Closing
- Morphological Gradient
- Top Hat
- Black Hat

This should work without any issues since it generates an image directly.

CODE:

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

# Generate a simple binary image (white rectangle on a black background)
image = np.zeros((200, 200), dtype=np.uint8)
cv2.rectangle(image, (50, 50), (150, 150), 255, -1)  # White rectangle in the middle

# Thresholding is not required since it's already a binary image
binary_image = image

# Define a kernel (structuring element) for morphological operations
kernel = np.ones((5, 5), np.uint8)

# 1. Erosion
erosion = cv2.erode(binary_image, kernel, iterations=1)

# 2. Dilation
dilation = cv2.dilate(binary_image, kernel, iterations=1)

# 3. Opening (erosion followed by dilation)
opening = cv2.morphologyEx(binary_image, cv2.MORPH_OPEN, kernel)

# 4. Closing (dilation followed by erosion)
closing = cv2.morphologyEx(binary_image, cv2.MORPH_CLOSE, kernel)

# 5. Morphological Gradient (difference between dilation and erosion)
gradient = cv2.morphologyEx(binary_image, cv2.MORPH_GRADIENT, kernel)

# 6. Top Hat (difference between the original image and its opening)
tophat = cv2.morphologyEx(binary_image, cv2.MORPH_TOPHAT, kernel)

# 7. Black Hat (difference between the closing and the original image)
blackhat = cv2.morphologyEx(binary_image, cv2.MORPH_BLACKHAT, kernel)

# Display the results
plt.figure(figsize=(12, 8))

plt.subplot(3, 3, 1), plt.imshow(binary_image, cmap='gray')
plt.title('Original Binary Image')
plt.subplot(3, 3, 2), plt.imshow(erosion, cmap='gray'), plt.title('Erosion')
plt.subplot(3, 3, 3), plt.imshow(dilation, cmap='gray'), plt.title('Dilation')
plt.subplot(3, 3, 4), plt.imshow(opening, cmap='gray'), plt.title('Opening')
plt.subplot(3, 3, 5), plt.imshow(closing, cmap='gray'), plt.title('Closing')
plt.subplot(3, 3, 6), plt.imshow(gradient, cmap='gray')
plt.title('Morphological Gradient')
plt.subplot(3, 3, 7), plt.imshow(tophat, cmap='gray'), plt.title('Top Hat')
plt.subplot(3, 3, 8), plt.imshow(blackhat, cmap='gray'), plt.title('Black Hat')

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

OUTPUT:

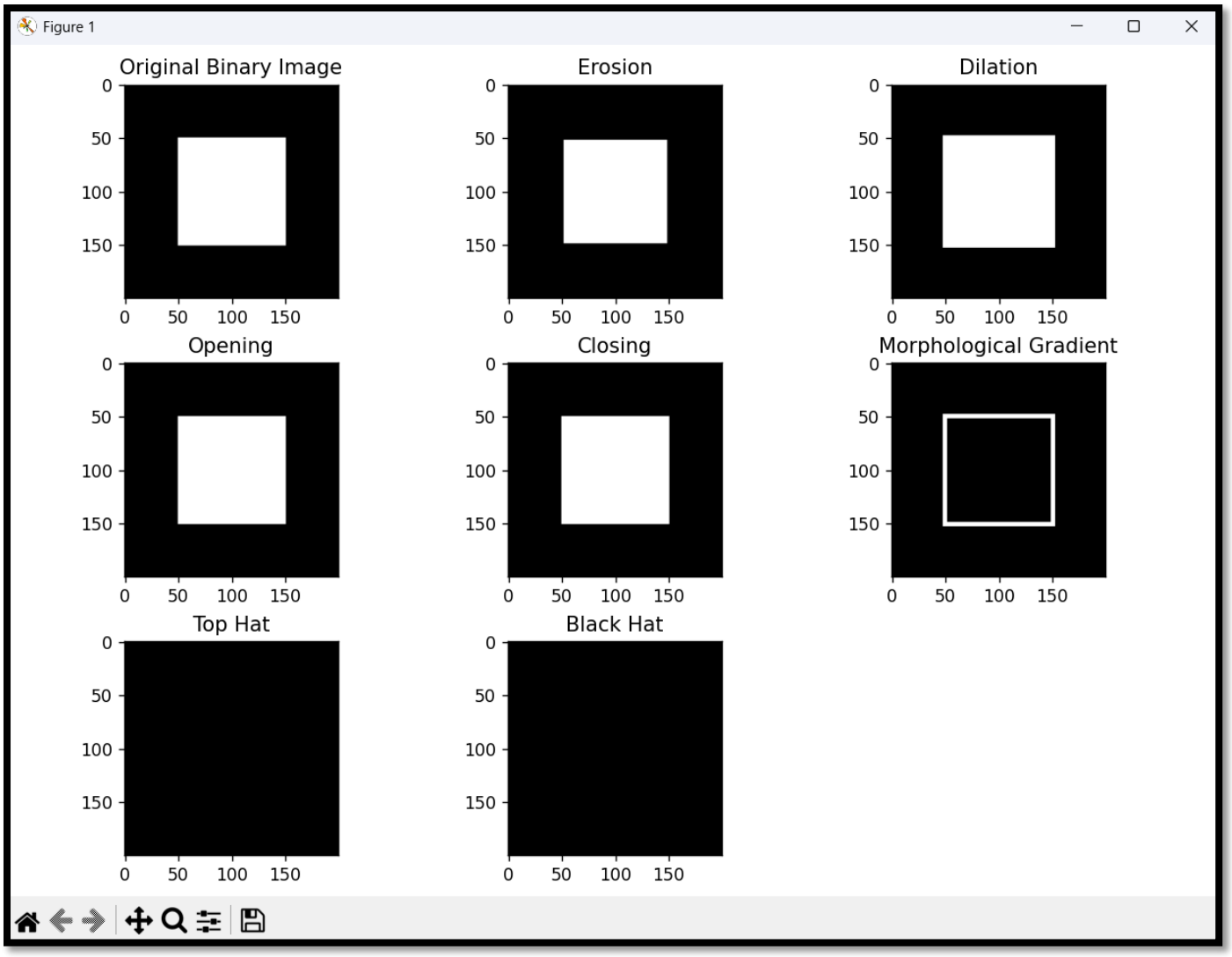


Figure 1
Original Binary Image
Erosion
Dilation
Opening
Closing
Morphological Gradient
Top Hat
Black Hat
0
50
100
150

Practical No. : **06**

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

1. Convolution operation
2. Template Matching

Introduction:

Convolution is a fundamental mathematical operation used in signal and image processing to modify or extract features from data. In image processing, convolution involves overlaying a small matrix called a kernel (filter) on an image to compute the weighted sum of pixel values within a neighborhood. The output is a filtered image where specific features such as edges, blurs, or sharpened details become visible depending on the chosen kernel.

The convolution process slides the kernel across the image pixel by pixel, multiplying overlapping pixel values and summing them up. This operation helps in performing smoothing, sharpening, and edge detection. Common convolution kernels include the averaging filter, Gaussian blur, and Laplacian filter.

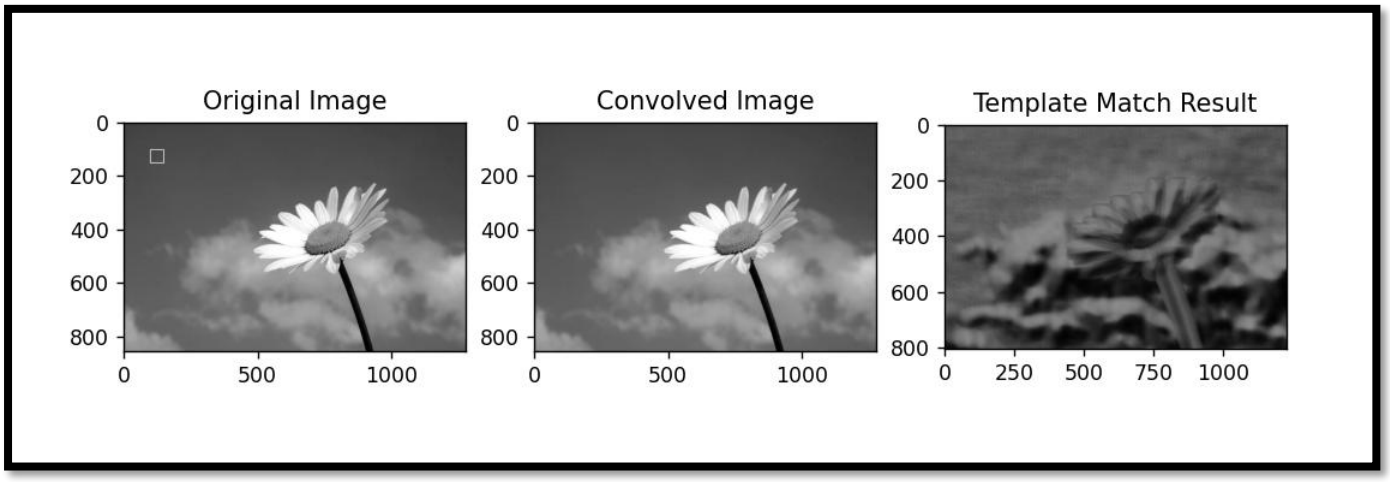
Template Matching is a technique used to find parts of an image that match a given small sample image, called the template. It is often used in object detection and pattern recognition. The method involves sliding the template over the main image and comparing pixel values using similarity measures such as correlation or normalized cross-correlation.

CODE:

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

img = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
kernel = np.ones((3, 3), np.float32) / 9
convolved = cv2.filter2D(img, -1, kernel)

template = img[100:150, 100:150]
result = cv2.matchTemplate(img, template, cv2.TM_CCOEFF_NORMED)

min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
h, w = template.shape
cv2.rectangle(img, max_loc, (max_loc[0] + w, max_loc[1] + h), 255, 2)

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

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

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

plt.subplot(1,3,3)
plt.imshow(result, cmap='gray')
plt.title('Template Match Result')

plt.show()
```

image.jpg:

OUTPUT:


Original Image
Convolved Image
Template Match Result
0
200
400
600
800
0
500
1000
0
250
500
750
1000

Practical No. : **07**

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

Introduction:

Intensity transformations are operations that modify the pixel values of an image to improve visual appearance or highlight important details. These are point-based operations, meaning each output pixel value depends only on the corresponding input pixel value. The transformations are used for contrast improvement, brightness adjustment, and image enhancement.

1. Log Transformation:
   Applies a logarithmic scale to pixel values to expand dark pixels and compress bright ones. It is useful for images with large dynamic ranges.
2. Power-Law (Gamma) Transformation:
   Uses a power function to enhance contrast. For gamma < 1, darker regions are emphasized; for gamma > 1, brighter regions are emphasized.
3. Contrast Adjustment:
   Linearly modifies pixel values using the formula $g(x, y) = \alpha * f(x, y) + \beta$, where α controls contrast and β controls brightness.
4. Histogram Equalization:
   Redistributes pixel intensities to enhance contrast, particularly in low-contrast images.
5. Thresholding:
   Converts a grayscale image to a binary image by setting all pixels above a threshold to white and the rest to black.

CODE:

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

img = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# 1. Log Transformation
c = 255 / np.log(1 + np.max(img))
log_transformed = c * (np.log(1 + img))
log_transformed = np.array(log_transformed, dtype=np.uint8)

# 2. Power-law (Gamma) Transformation
gamma = 0.5   # change gamma value (0.4–1.5)
power_law = np.array(255 * (img / 255) ** gamma, dtype='uint8')

# 3. Contrast Adjustment
contrast = cv2.convertScaleAbs(img, alpha=1.5, beta=0)  # alpha >1 increases
contrast

# 4. Histogram Equalization
hist_eq = cv2.equalizeHist(img)

# 5. Thresholding
_, thresh = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY)

# Display all
titles = ['Original', 'Log', 'Power-Law', 'Contrast', 'Hist Eq', 'Threshold']
images = [img, log_transformed, power_law, contrast, hist_eq, thresh]

plt.figure(figsize=(12,6))
for i in range(6):
    plt.subplot(2,3,i+1), plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')

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

image.jpg:

OUTPUT:

Practical No. : **08**

Aim: Write a program to Apply edge detection techniques such as Sobel and Canny to extract meaningful information from the given image samples

Introduction:

Edge detection is one of the most important tasks in image processing and computer vision. It identifies the boundaries between regions with different intensity levels, enabling object recognition and image segmentation. An edge represents a sudden change in pixel intensity, which can be detected by computing image gradients.

1. Sobel Operator:

   The Sobel operator uses two 3×3 convolution masks (horizontal and vertical) to compute the gradient magnitude in both x and y directions. It emphasizes edges in a particular orientation and is simple to compute. The gradient magnitude is calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

where G_x and G_y are gradients in horizontal and vertical directions respectively.

2. Canny Edge Detector:

   The Canny operator is a multi-stage algorithm that includes:

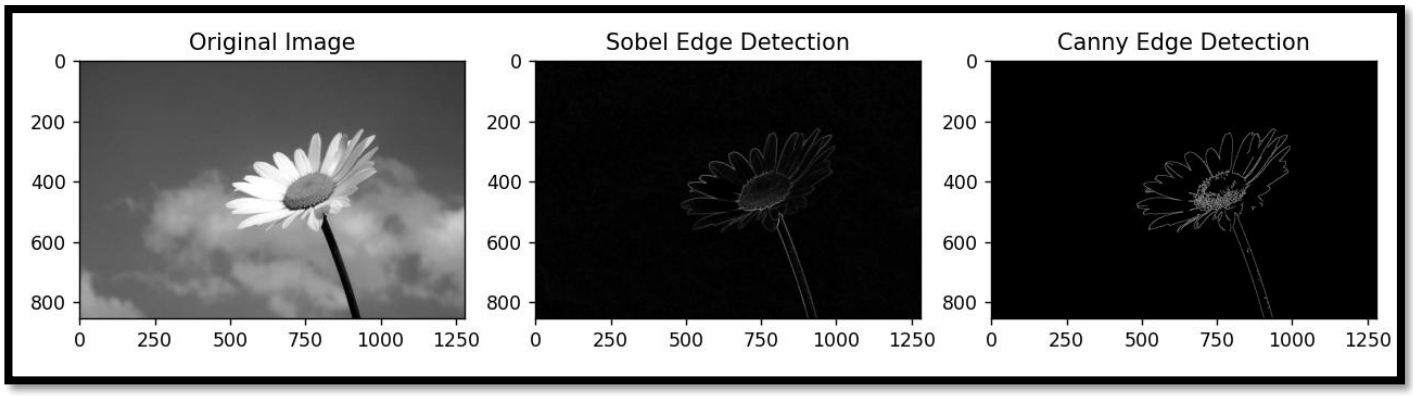
   - Noise reduction using Gaussian blur
   - Gradient computation
   - Non-maximum suppression to thin edges
   - Double thresholding and edge tracking by hysteresis

CODE:

```
import cv2
from matplotlib import pyplot as plt

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

# Sobel Edge Detection
sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # x direction
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # y direction
sobel_combined = cv2.magnitude(sobelx, sobely)

# Canny Edge Detection
edges_canny = cv2.Canny(img, 100, 200)

# Display results
plt.figure(figsize=(10,5))

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

plt.subplot(1,3,2)
plt.imshow(sobel_combined, cmap='gray')
plt.title('Sobel Edge Detection')

plt.subplot(1,3,3)
plt.imshow(edges_canny, cmap='gray')
plt.title('Canny Edge Detection')

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

image.jpg:

OUTPUT:


Original Image
Sobel Edge Detection
Canny Edge Detection
0
200
400
600
800
0
250
500
750
1000
1250