Practical No. 1

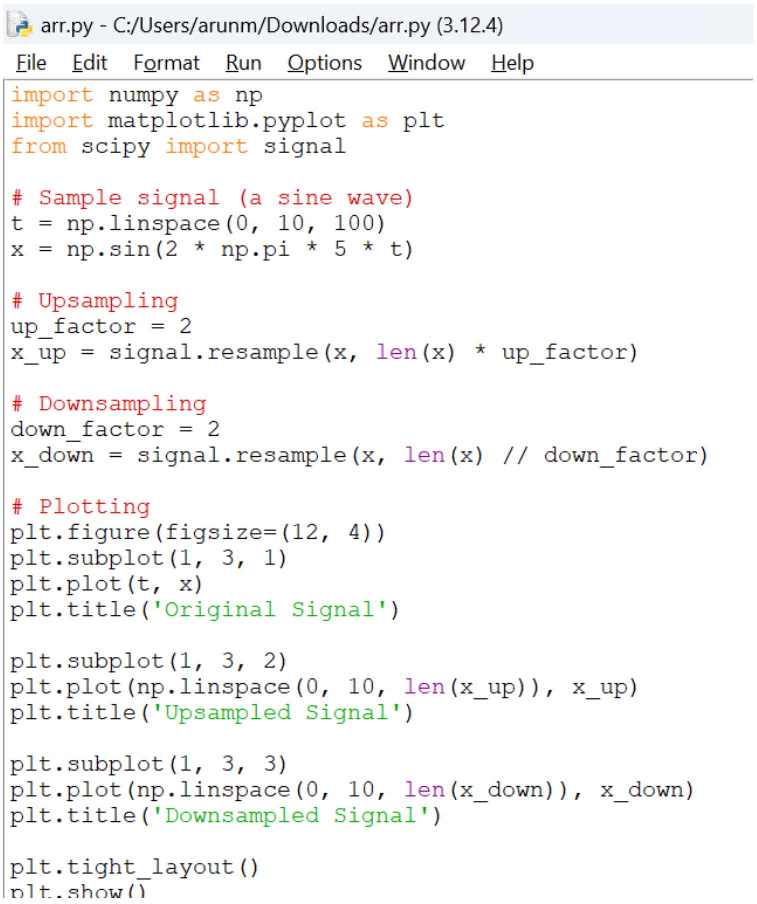
Write program to demonstrate the following aspects of signal processing on suitable data
1. Upsampling and Downsampling on Image/speech signal
2. Fast Fourier Transform to compute DFT

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.

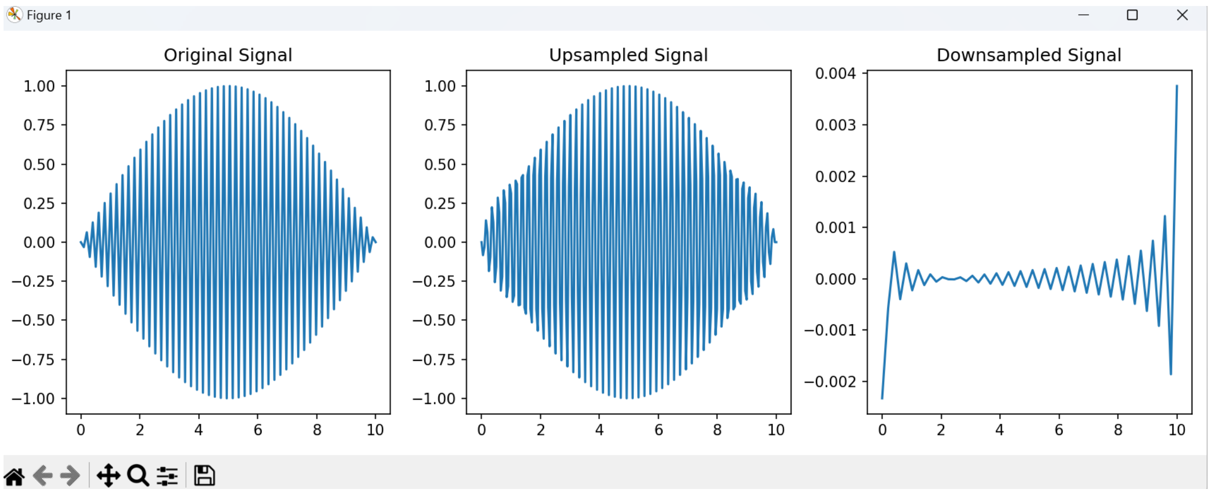
Implementing in Python with Libraries
We'll use Python with libraries like NumPy, SciPy, and Matplotlib for efficient signal processing and visualization.

1. Upsampling and Downsampling on Image/speech signal

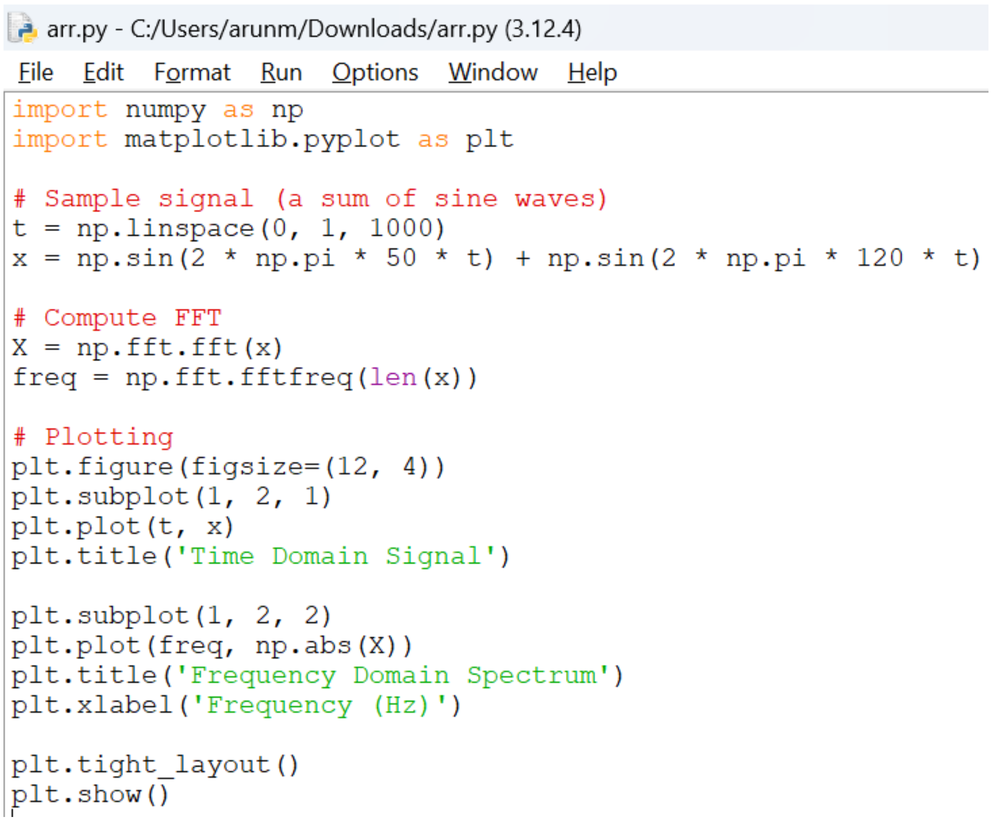

arr.py - C:/Users/arunm/Downloads/arr.py (3.12.4)
File Edit Format Run Options Window Help

```
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

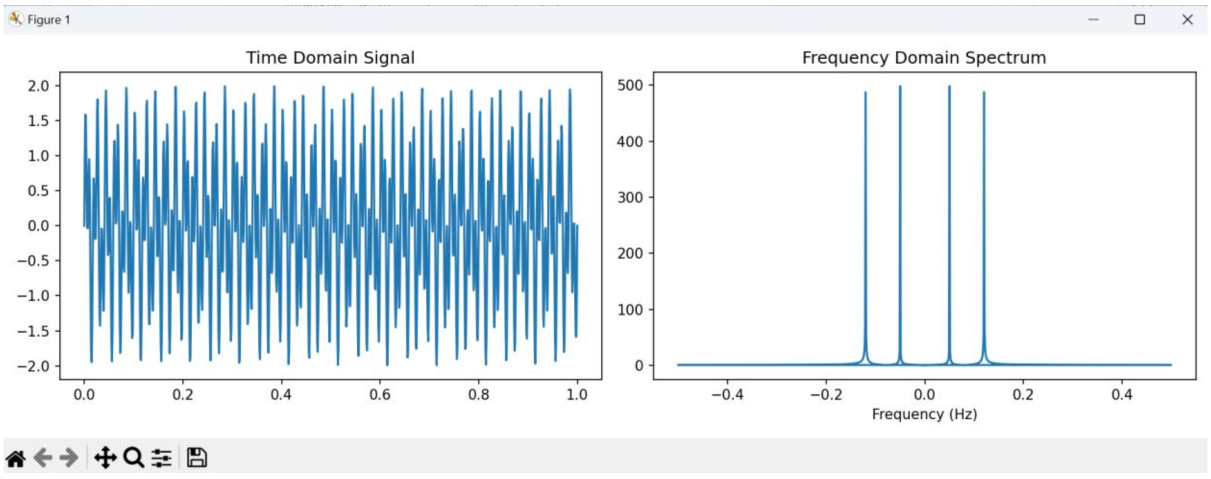


2. Fast Fourier Transform to compute DFT.


arr.py - C:/Users/arunm/Downloads/arr.py (3.12.4)
File Edit Format Run Options Window Help

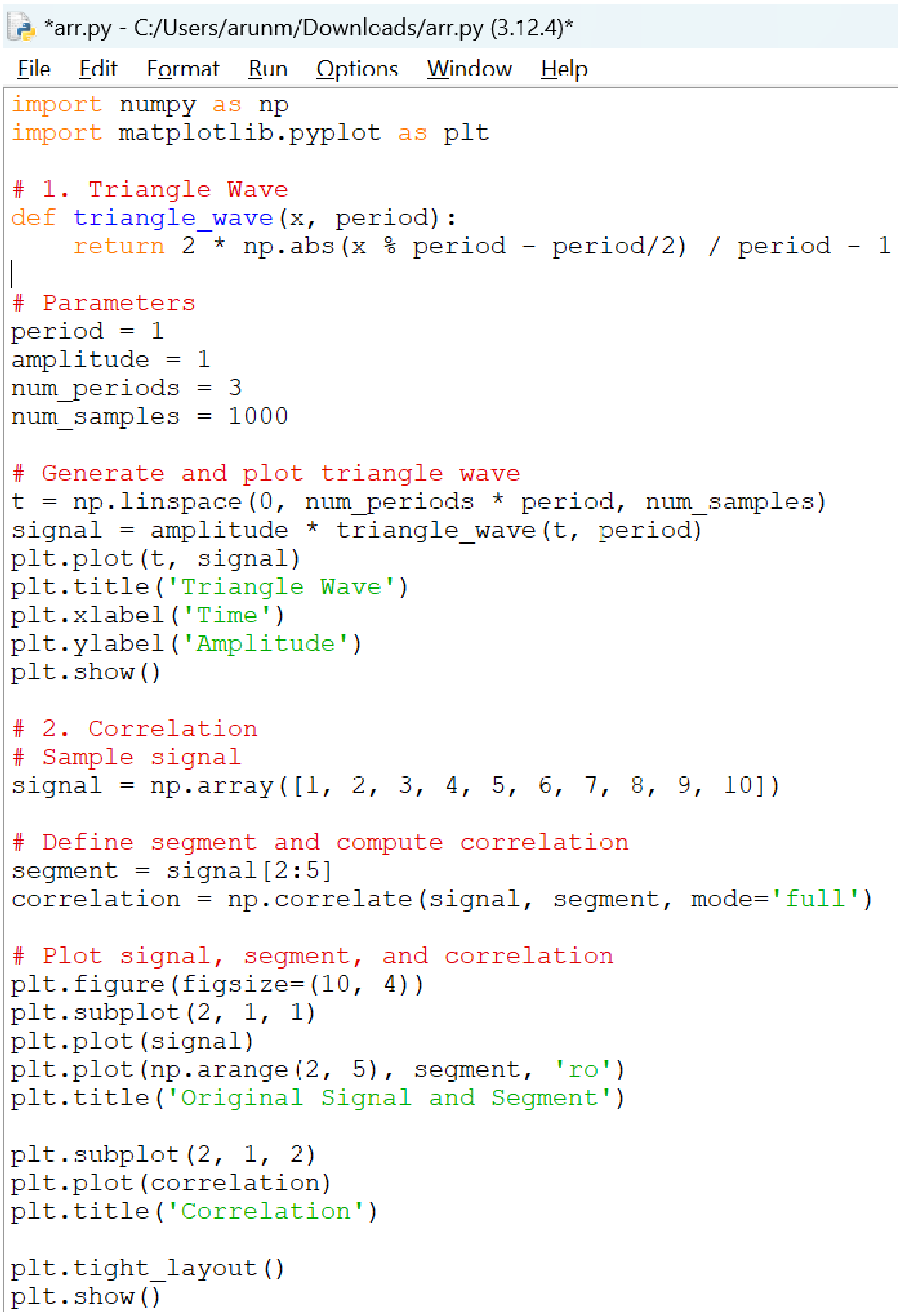
```
import numpy as np
import matplotlib.pyplot as plt

# 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



Practical No. 2

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.


arr.py - C:/Users/arunm/Downloads/arr.py (3.12.4)
File Edit Format Run Options Window Help

```python
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

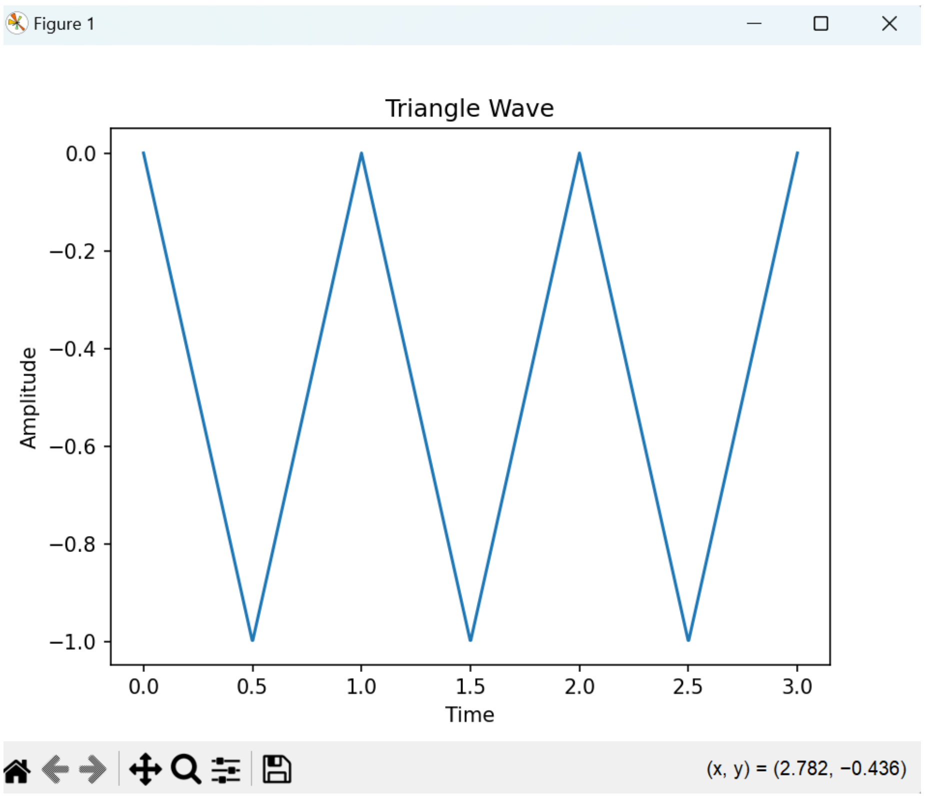


Figure 1
Triangle Wave
Amplitude
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Time
(x, y) = (2.782, −0.436)

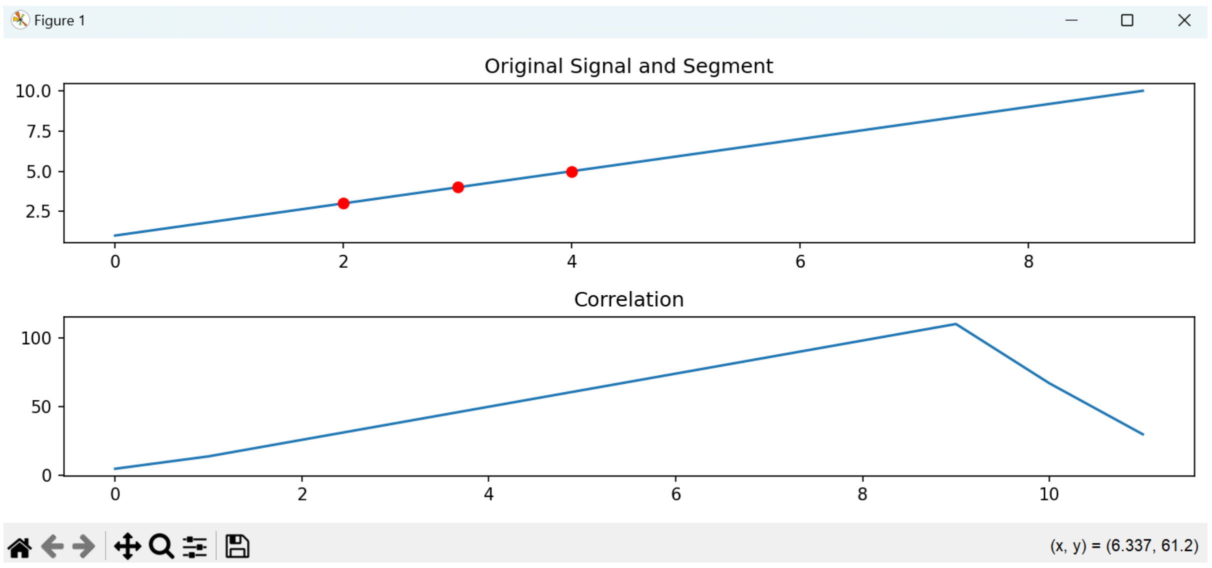


Figure 1
Original Signal and Segment
10.0
7.5
5.0
2.5
0
2
4
6
8
Correlation
100
50
0
0
2
4
6
8
10
(x, y) = (6.337, 61.2)

Practical No. 3

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

Explanation of the Code:

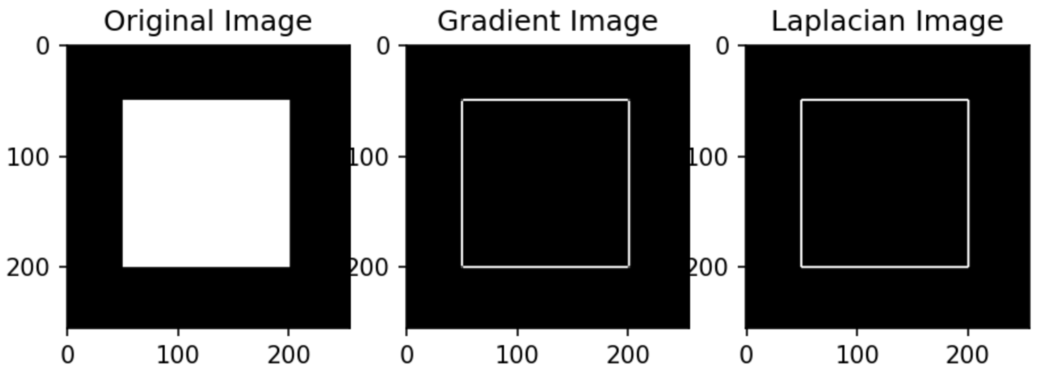
1. Synthetic Image Creation:
    - A 256x256 pixel black image is created using np.zeros().
    - A white square is drawn on this black image using cv2.rectangle(). This square has pixel intensity 255 (white), and its position is from (50, 50) to (200, 200).
2. Gradient Calculation (Sobel Operator):
    - 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.

```
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

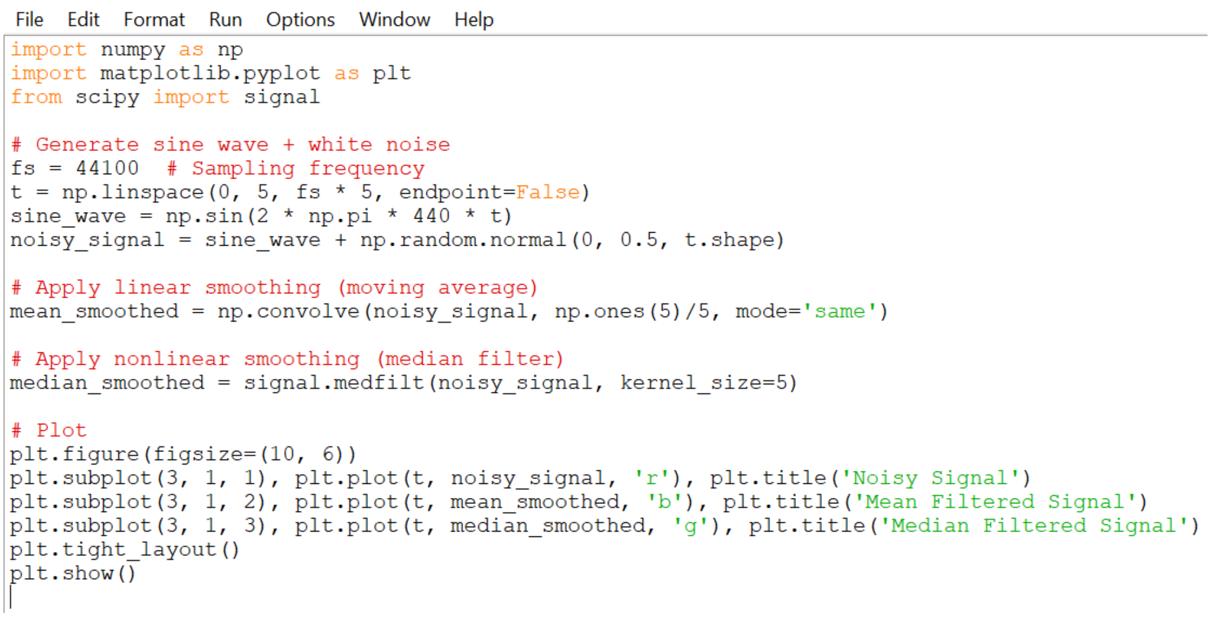

Original Image
Gradient Image
Laplacian Image
0
100
200
0
100
200

Practical No. 4

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

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)



```
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)
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

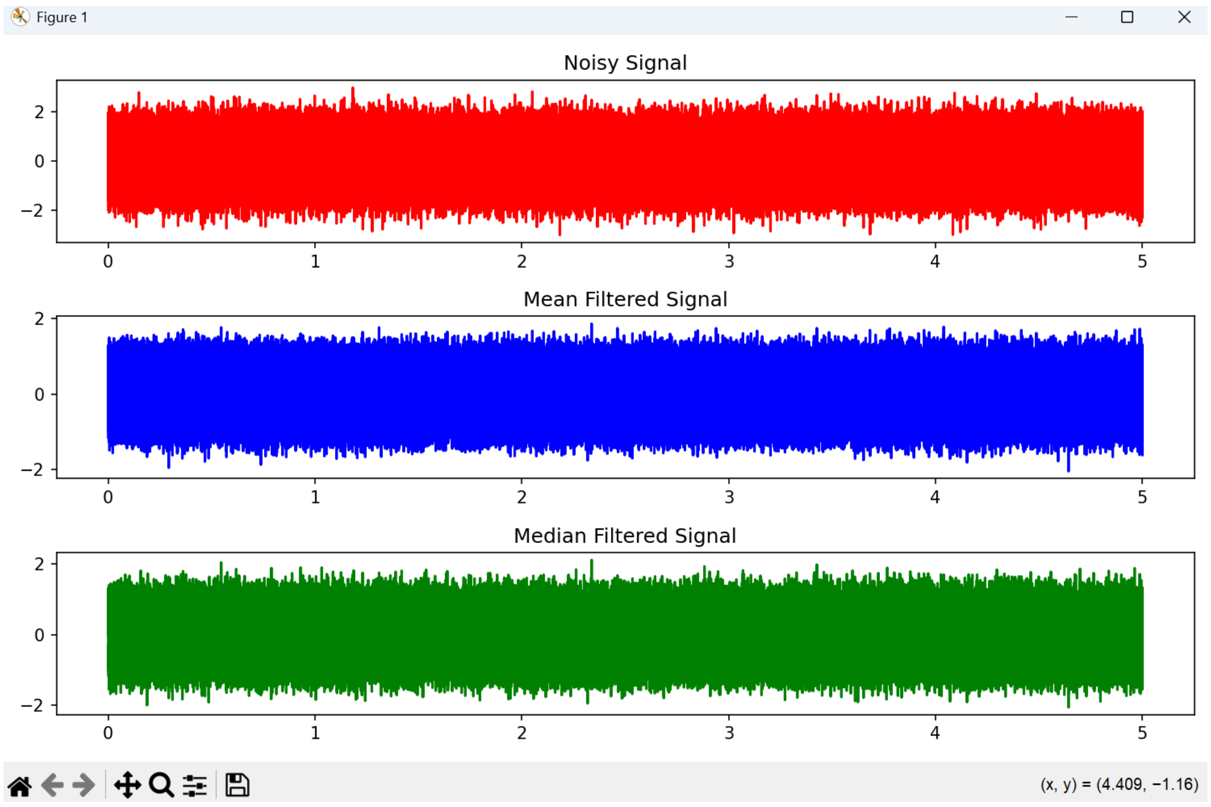


Figure 1
Noisy Signal
Mean Filtered Signal
Median Filtered Signal
(x, y) = (4.409, −1.16)

Practical No. 5

Write the program to implement various morphological image processing techniques.

- 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

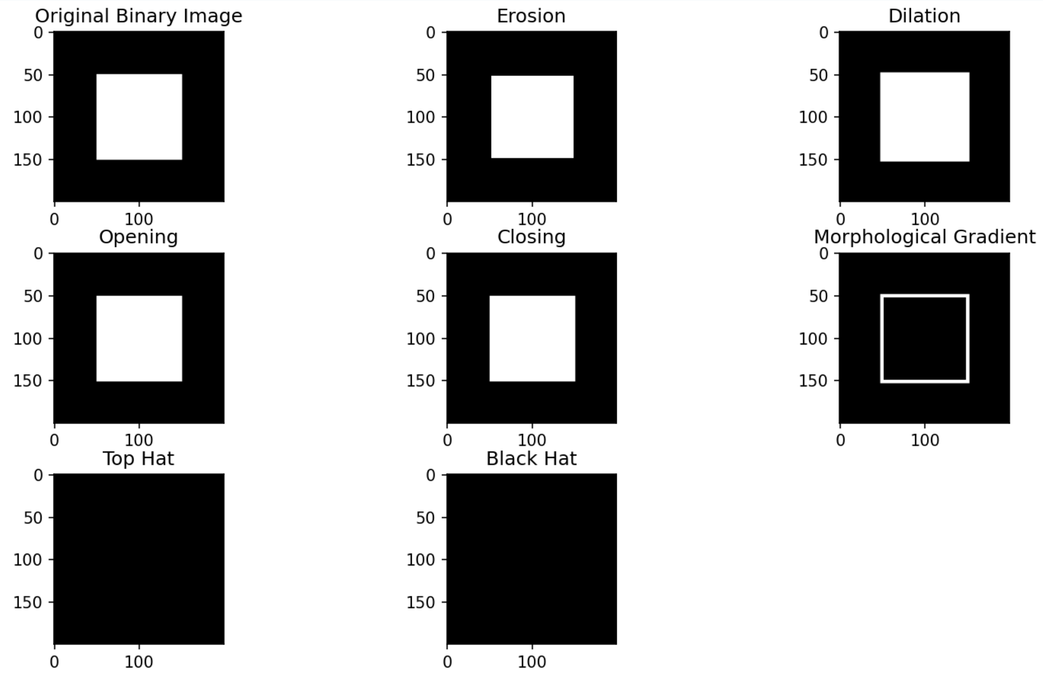
This should work without any issues since it generates an image directly.

```
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)
```

```
# 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


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