Assignment 2

Spring 2025

CSE-408 Digital Image Processing

Submitted by: **Ali Asghar**

Registration No.: **21PWCSE2059**

Class Section: **C**

Submitted to:

Engr. Mehran Ahmad

Date:

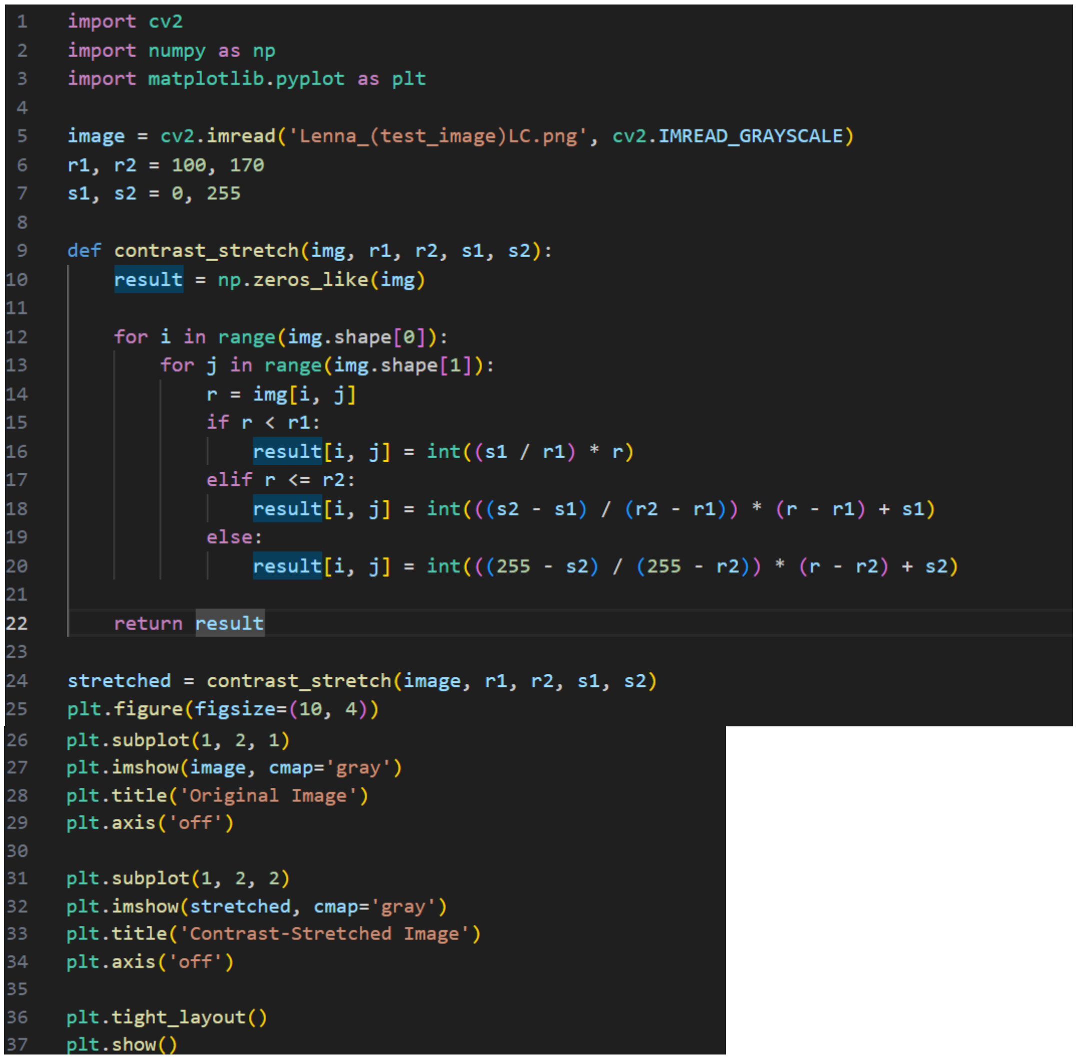
21st May 2025

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

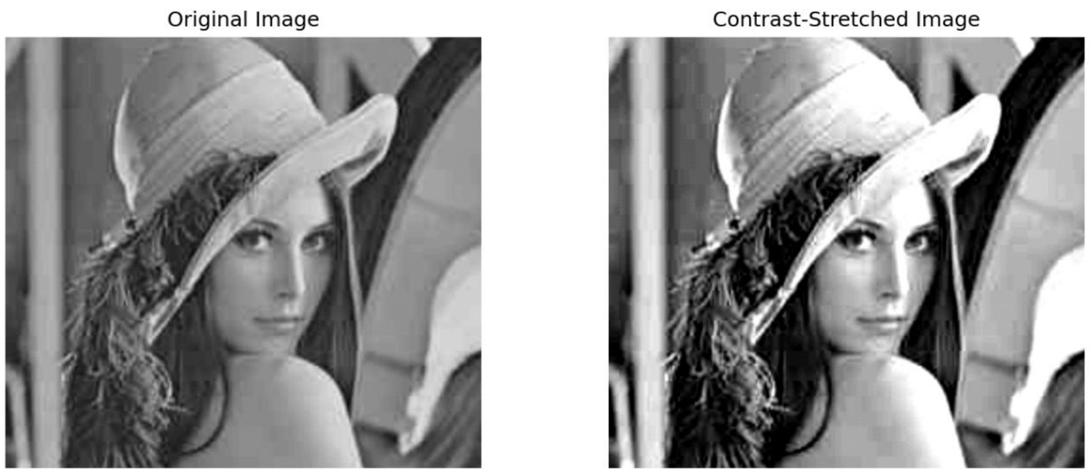
Activity 1: Write a MATLAB/Python script to perform piecewise linear contrast stretching.

Code:



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

image = cv2.imread('Lenna_(test_image)LC.png', cv2.IMREAD_GRAYSCALE)
r1, r2 = 100, 170
s1, s2 = 0, 255

def contrast_stretch(img, r1, r2, s1, s2):
    result = np.zeros_like(img)

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            r = img[i, j]
            if r < r1:
                result[i, j] = int((s1 / r1) * r)
            elif r <= r2:
                result[i, j] = int(((s2 - s1) / (r2 - r1)) * (r - r1) + s1)
            else:
                result[i, j] = int(((255 - s2) / (255 - r2)) * (r - r2) + s2)

    return result

stretched = contrast_stretch(image, r1, r2, s1, s2)
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(stretched, cmap='gray')
plt.title('Contrast-Stretched Image')
plt.axis('off')

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

Output:



Analysis:
Contrast stretching enhances the dynamic range of pixel intensities, making features more visible in low-contrast images. In this case, intensities between r1=100 and r2=170 are mapped to the full output range s1=0 to s2=255, effectively spreading mid-tone values across the entire grayscale spectrum. This significantly improves the visual quality by brightening darker regions and adding more contrast to subtle details. It's especially useful when the original image appears dull or flat due to limited intensity variation.

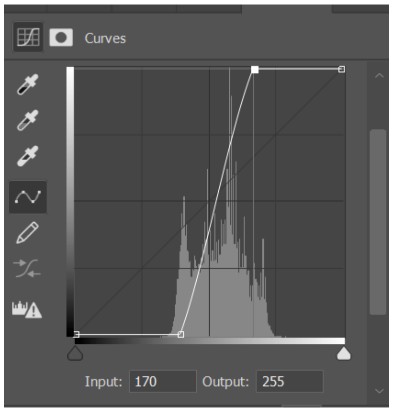


Figure 1 Setting the constants r1 and r2 values on the histogram

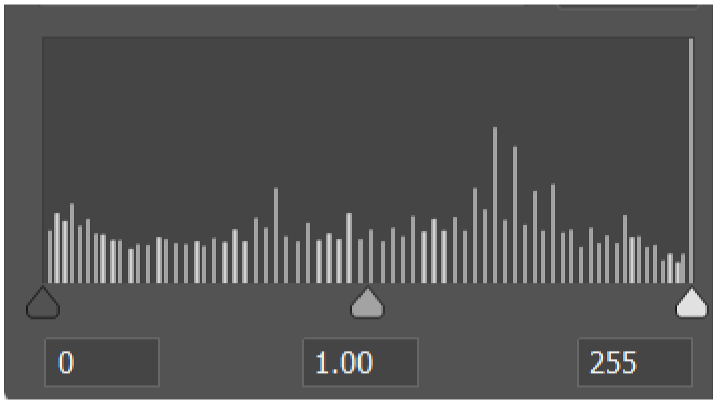
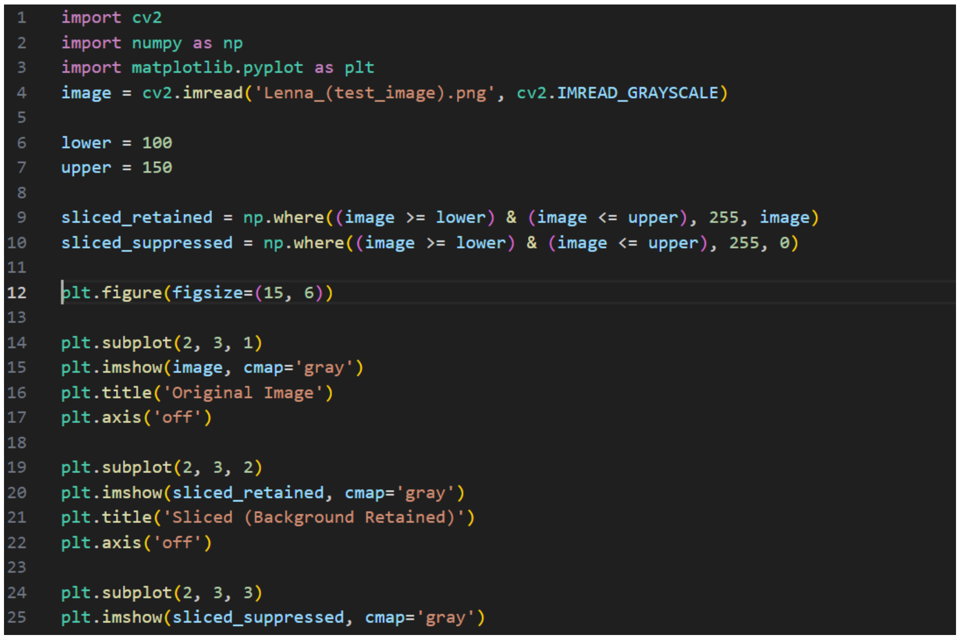


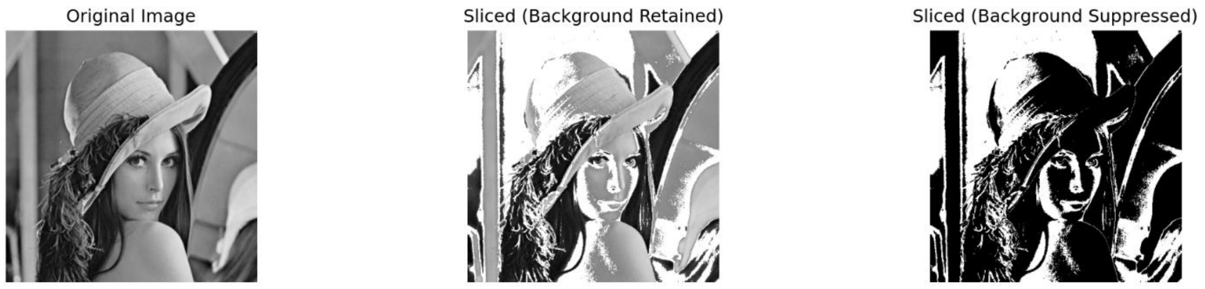
Figure 2 Histogram After Contrast-Stretching

Activity 2: Take an input image and implement gray-level slicing in MATLAB/Python.
Code:



```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)

lower = 100
upper = 150

sliced_retained = np.where((image >= lower) & (image <= upper), 255, image)
sliced_suppressed = np.where((image >= lower) & (image <= upper), 255, 0)

plt.figure(figsize=(15, 6))

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

plt.subplot(2, 3, 2)
plt.imshow(sliced_retained, cmap='gray')
plt.title('Sliced (Background Retained)')
plt.axis('off')

plt.subplot(2, 3, 3)
plt.imshow(sliced_suppressed, cmap='gray')
```

Output:



Analysis:
Gray level slicing enhances specific intensity ranges in an image to emphasize important features:

- **In the background retained version**, pixel values in the range [100, 150] are set to white (255), while others remain unchanged. This highlights the region of interest without losing context, as shown in the function plot where only a slice of the curve spikes to 255.

- **In the background suppressed version**, only pixels in the target range are shown (set to 255), while all others are suppressed to 0 (black). The function plot for this is a flat binary-like response — useful for isolating structures with minimal distraction from the background.

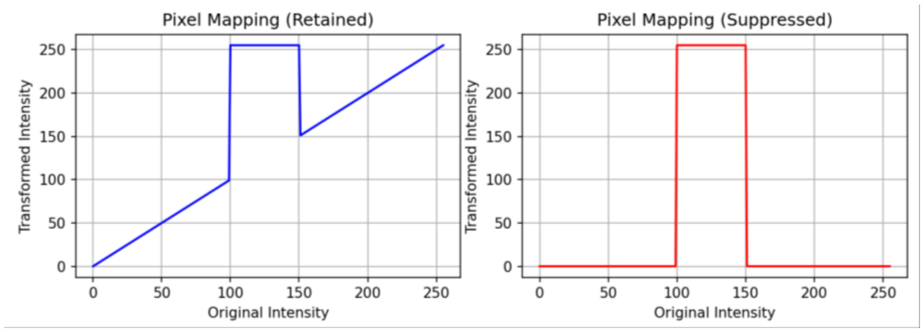


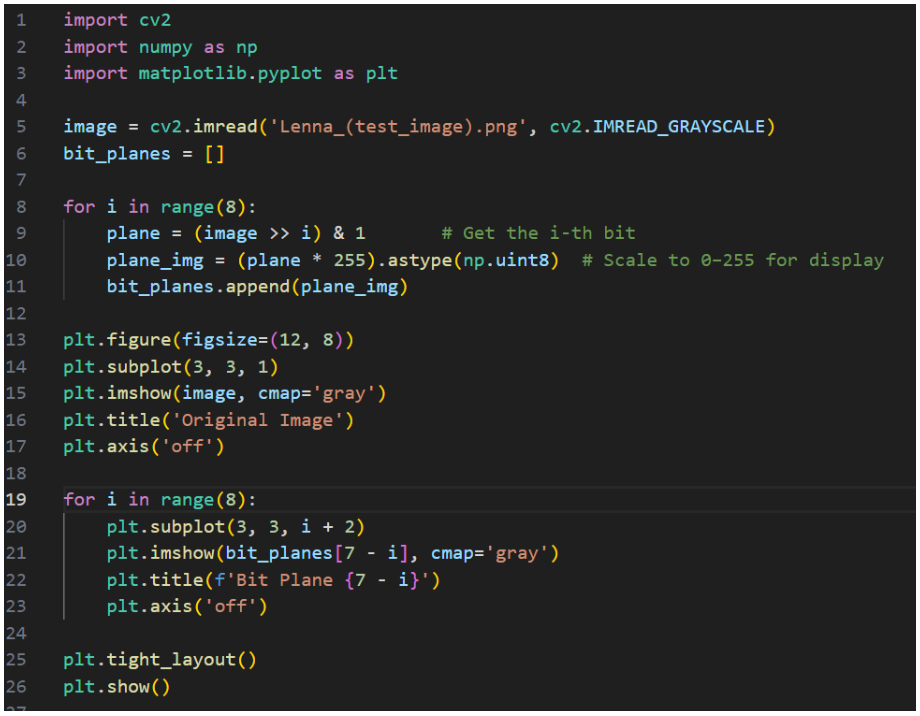
Figure 3 Pixel Mapping Functions for Gray Level Slicing

The left plot shows the retained background mapping, where pixel intensities between 100 and 150 are mapped to white (255), while other intensities remain unchanged.
The right plot illustrates the suppressed background mapping, where only the specified range (100–150) is shown in white and all other intensities are set to black (0), effectively isolating the region of interest.

Activity 3:

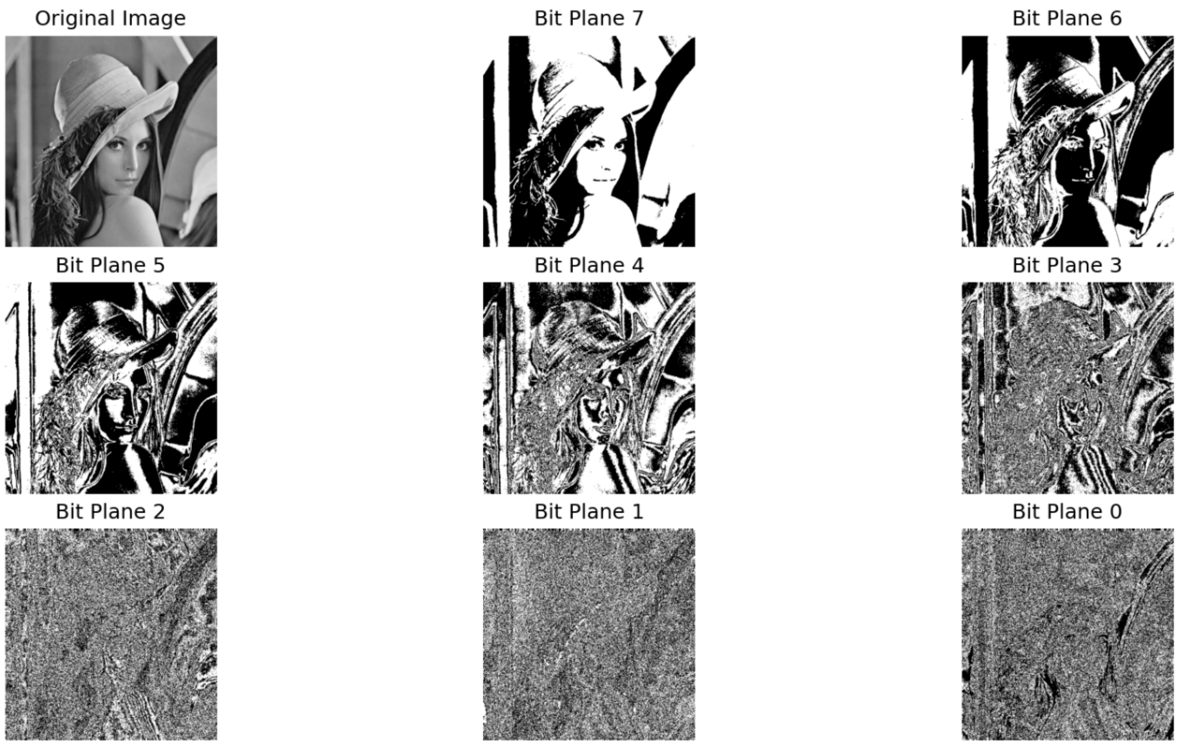
Implement bit-plane slicing in MATLAB and extract all 8 bit planes. **Show the code and the output result** (original image and bit planes from MSB to LSB).

Code:



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

image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)
bit_planes = []

for i in range(8):
    plane = (image >> i) & 1       # Get the i-th bit
    plane_img = (plane * 255).astype(np.uint8)  # Scale to 0-255 for display
    bit_planes.append(plane_img)

plt.figure(figsize=(12, 8))
plt.subplot(3, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

for i in range(8):
    plt.subplot(3, 3, i + 2)
    plt.imshow(bit_planes[7 - i], cmap='gray')
    plt.title(f'Bit Plane {7 - i}')
    plt.axis('off')

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

Output:



Analysis:
Bit-plane slicing breaks an 8-bit grayscale image into its individual binary layers, from the Most Significant Bit (MSB) to the Least Significant Bit (LSB):

- Higher-order bit planes (e.g., Bit 7, Bit 6) capture most of the significant visual information and structure of the image. These bits contribute heavily to the overall intensity and clarity.

- Lower-order bits (e.g., Bit 0, Bit 1) appear as fine noise-like patterns. They carry less visual weight but may encode subtle texture or noise.

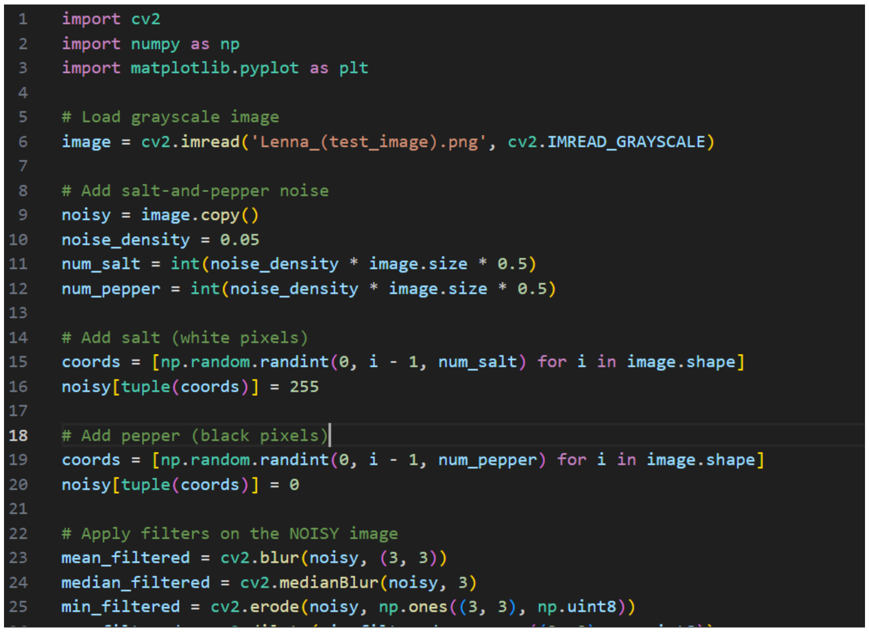
Activity 4:

Part A – Smoothing Filters (Noise Reduction & Background Enhancement)

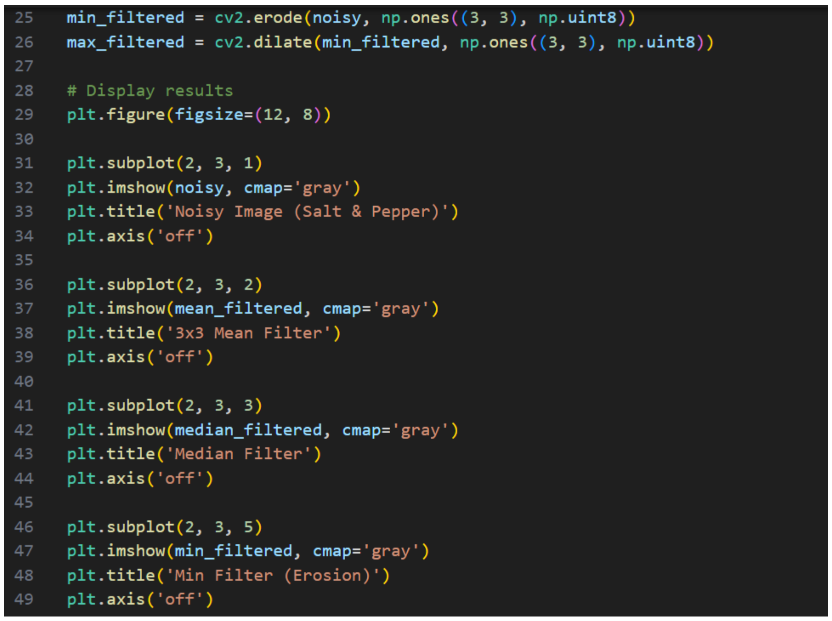
Select an image of your choice and investigate hidden or obscured objects in the background using the following smoothing techniques:

- **Smoothing Spatial Filtering** (e.g., 3×3 Moving Average Filter)
- **Order-Statistic Nonlinear Filters** o Median Filter o Min Filter o Max Filter

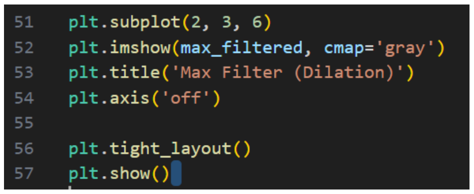
Code:



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

# Load grayscale image
image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)

# Add salt-and-pepper noise
noisy = image.copy()
noise_density = 0.05
num_salt = int(noise_density * image.size * 0.5)
num_pepper = int(noise_density * image.size * 0.5)

# Add salt (white pixels)
coords = [np.random.randint(0, i - 1, num_salt) for i in image.shape]
noisy[tuple(coords)] = 255

# Add pepper (black pixels)
coords = [np.random.randint(0, i - 1, num_pepper) for i in image.shape]
noisy[tuple(coords)] = 0

# Apply filters on the NOISY image
mean_filtered = cv2.blur(noisy, (3, 3))
median_filtered = cv2.medianBlur(noisy, 3)
min_filtered = cv2.erode(noisy, np.ones((3, 3), np.uint8))
```

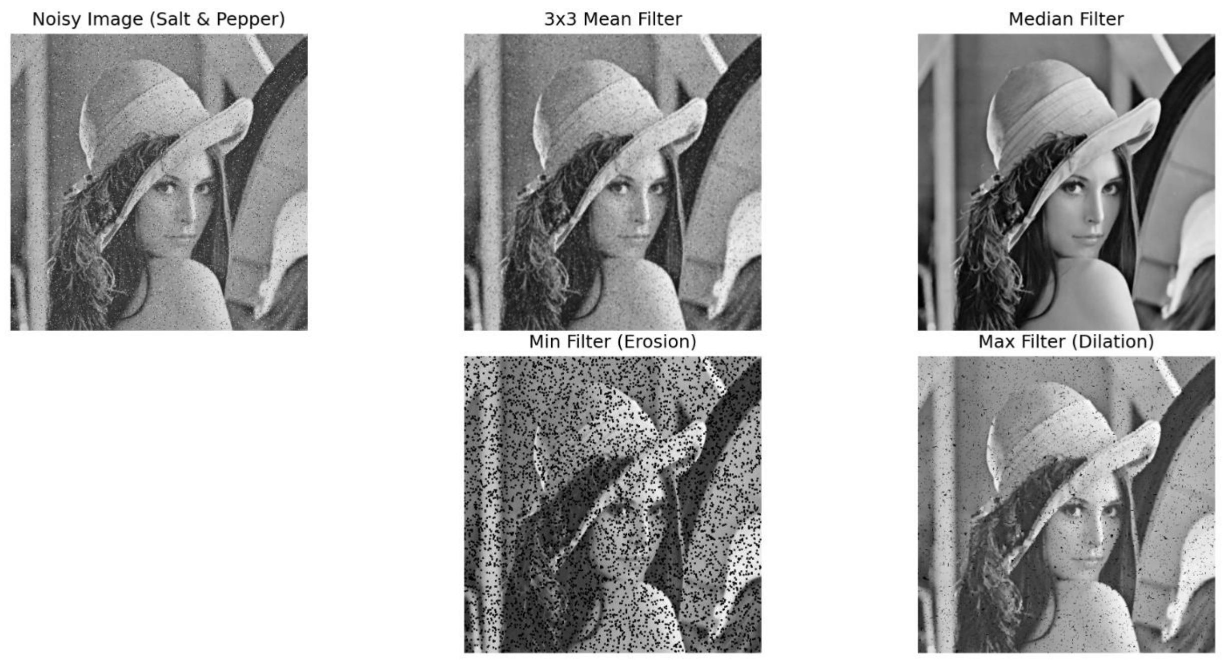


```python
min_filtered = cv2.erode(noisy, np.ones((3, 3), np.uint8))
max_filtered = cv2.dilate(min_filtered, np.ones((3, 3), np.uint8))

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

plt.subplot(2, 3, 1)
plt.imshow(noisy, cmap='gray')
plt.title('Noisy Image (Salt & Pepper)')
plt.axis('off')

plt.subplot(2, 3, 2)
plt.imshow(mean_filtered, cmap='gray')
plt.title('3x3 Mean Filter')
plt.axis('off')

plt.subplot(2, 3, 3)
plt.imshow(median_filtered, cmap='gray')
plt.title('Median Filter')
plt.axis('off')

plt.subplot(2, 3, 5)
plt.imshow(min_filtered, cmap='gray')
plt.title('Min Filter (Erosion)')
plt.axis('off')
```



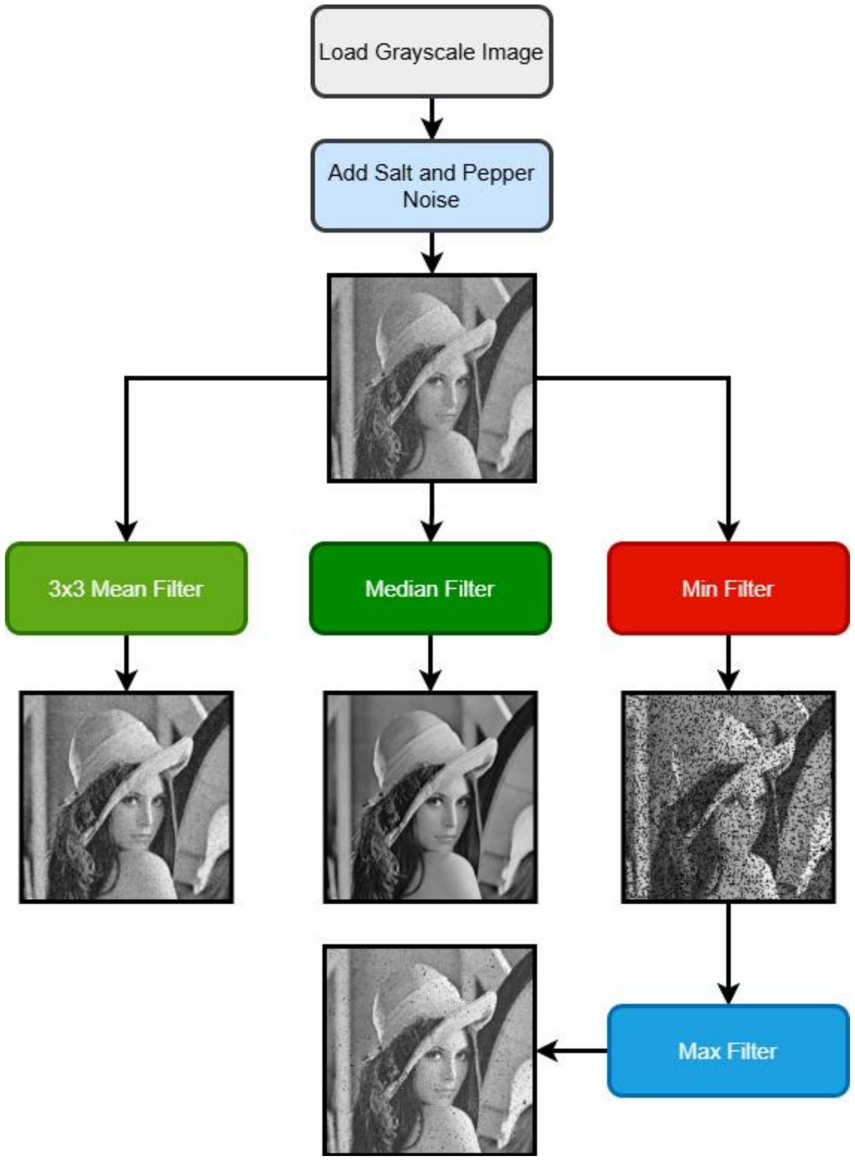
```python
plt.subplot(2, 3, 6)
plt.imshow(max_filtered, cmap='gray')
plt.title('Max Filter (Dilation)')
plt.axis('off')

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

Output:



Analysis:

The figure below demonstrates the process of applying various smoothing filters to a grayscale image affected by salt-and-pepper noise. Initially, the original image is corrupted by randomly adding white (salt) and black (pepper) pixels. This noisy image is then processed through three distinct filtering paths: the 3×3 mean filter, the median filter, and a combination of min and max filters. The mean filter reduces noise by averaging pixel values but tends to blur image details. The median filter, on the other hand, effectively removes salt-and-pepper noise while preserving edges, making it the most effective in this scenario. The min filter eliminates white noise but darkens the image, and the subsequent max filter attempts to restore it by expanding bright regions. Overall, it highlights that while all filters provide some level of noise reduction, the median filter offers the best balance between noise removal and detail retention.

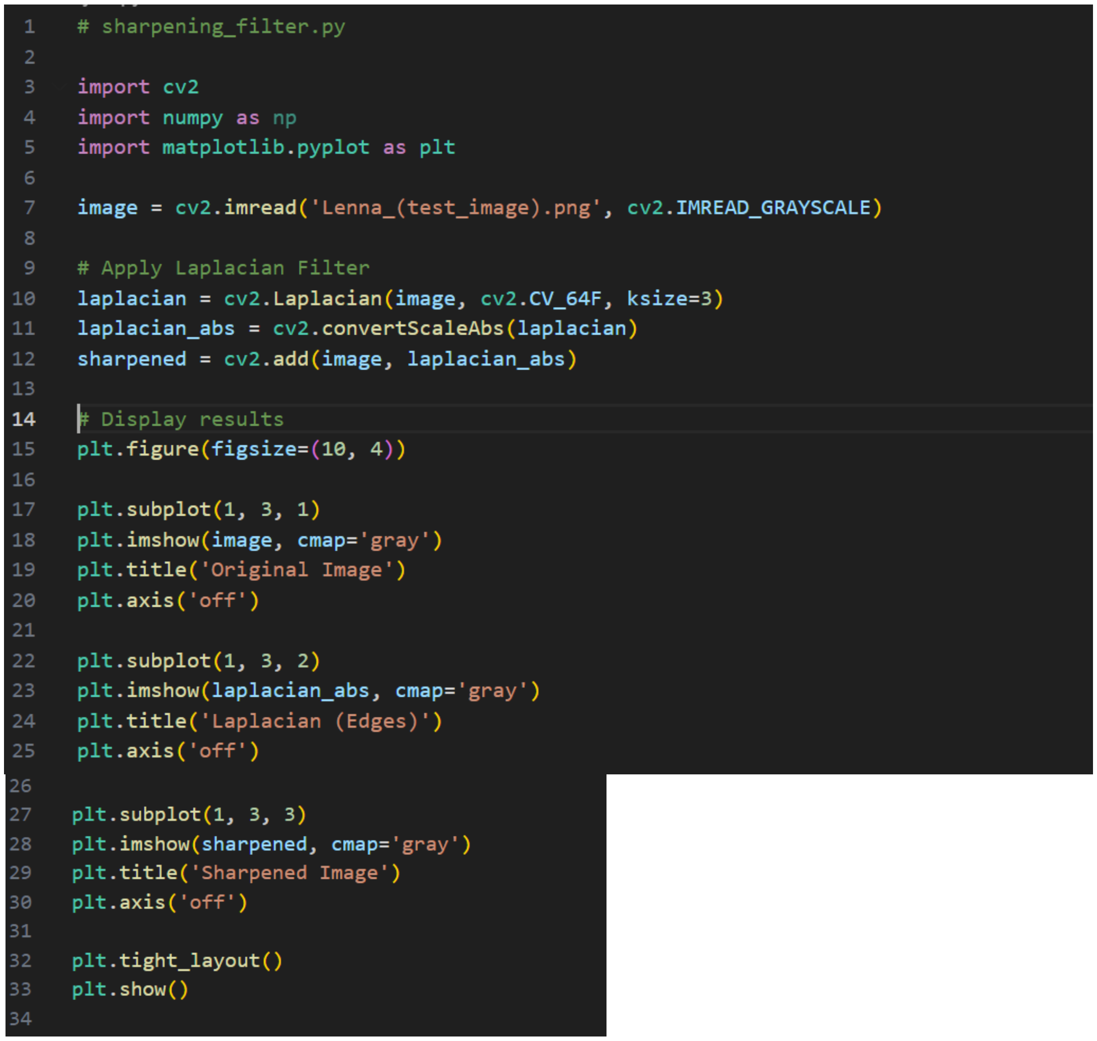


Part B – Sharpening Filter (Detail Enhancement & Edge Detection)

Use the same image to enhance important visual features (e.g., edges, textures, fine details) using the following sharpening technique:

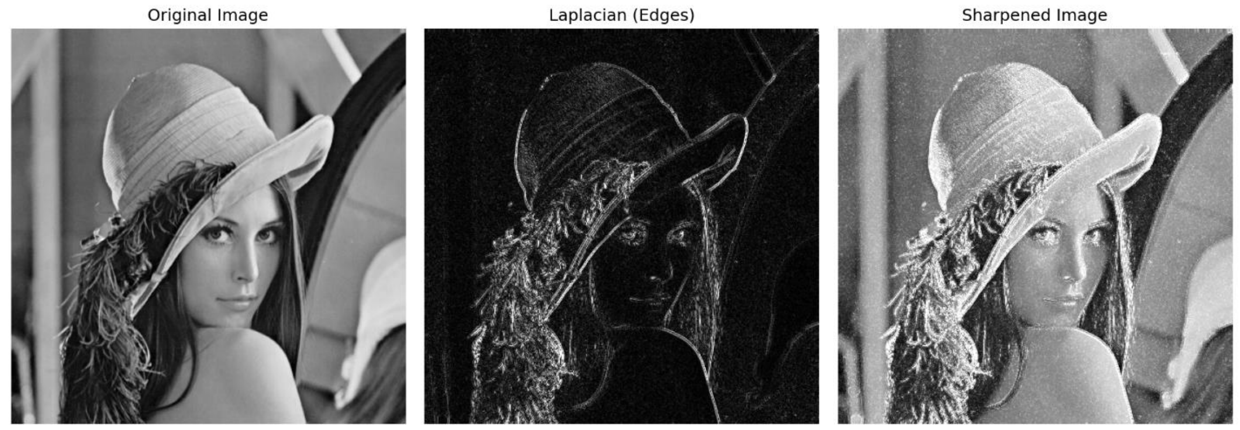
- **Laplacian Filter**

Code:



```python
# sharpening_filter.py

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

image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)

# Apply Laplacian Filter
laplacian = cv2.Laplacian(image, cv2.CV_64F, ksize=3)
laplacian_abs = cv2.convertScaleAbs(laplacian)
sharpened = cv2.add(image, laplacian_abs)

# Display results
plt.figure(figsize=(10, 4))

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

plt.subplot(1, 3, 2)
plt.imshow(laplacian_abs, cmap='gray')
plt.title('Laplacian (Edges)')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(sharpened, cmap='gray')
plt.title('Sharpened Image')
plt.axis('off')

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

Output:



Analysis:
In this task, I demonstrated image sharpening using the Laplacian filter, a second-order derivative operator that highlights regions of rapid intensity change—typically edges. The original grayscale image is first processed with the Laplacian operator to detect edges, producing a high-frequency image that emphasizes fine details and transitions. Since the Laplacian output contains both positive and negative values, it is converted to an absolute scale for visualization. The final sharpened image is obtained by adding the edge-enhanced Laplacian image back to the original, effectively increasing the contrast along edges while preserving the overall structure. This technique enhances visual sharpness and detail, making it useful for applications such as medical imaging, object recognition, or any task requiring enhanced edge definition.