

## **Assignment 2**



**Spring 2025**

**CSE-408 Digital Image Processing**

Submitted by: **Suleman Shah**

Registration No.: **21PWCSE1983**

Class Section: **C**

Submitted to:

**Engr. Mehran Ahmad**

Date:

**21<sup>st</sup> 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:**

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  image = cv2.imread('Lenna_(test_image)LC.png', cv2.IMREAD_GRAYSCALE)
6  r1, r2 = 100, 170
7  s1, s2 = 0, 255
8
9  def contrast_stretch(img, r1, r2, s1, s2):
10     result = np.zeros_like(img)
11
12     for i in range(img.shape[0]):
13         for j in range(img.shape[1]):
14             r = img[i, j]
15             if r < r1:
16                 result[i, j] = int((s1 / r1) * r)
17             elif r <= r2:
18                 result[i, j] = int(((s2 - s1) / (r2 - r1)) * (r - r1) + s1)
19             else:
20                 result[i, j] = int(((255 - s2) / (255 - r2)) * (r - r2) + s2)
21
22     return result
23
24  stretched = contrast_stretch(image, r1, r2, s1, s2)
25  plt.figure(figsize=(10, 4))
26  plt.subplot(1, 2, 1)
27  plt.imshow(image, cmap='gray')
28  plt.title('Original Image')
29  plt.axis('off')
30
31  plt.subplot(1, 2, 2)
32  plt.imshow(stretched, cmap='gray')
33  plt.title('Contrast-Stretched Image')
34  plt.axis('off')
35
36  plt.tight_layout()
37  plt.show()
```

**Output:**

Original Image



Contrast-Stretched Image



### 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.

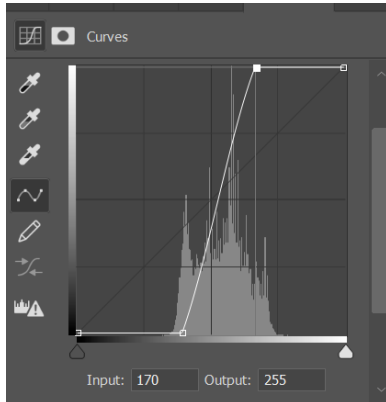


Figure 1 Setting the constants  $r1$  and  $r2$  values on the histogram



Figure 2 Histogram After Contrast-Stretching

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

**Code:**

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4  image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)
5
6  lower = 100
7  upper = 150
8
9  sliced_retained = np.where((image >= lower) & (image <= upper), 255, image)
10 sliced_suppressed = np.where((image >= lower) & (image <= upper), 255, 0)
11
12 plt.figure(figsize=(15, 6))
13
14 plt.subplot(2, 3, 1)
15 plt.imshow(image, cmap='gray')
16 plt.title('Original Image')
17 plt.axis('off')
18
19 plt.subplot(2, 3, 2)
20 plt.imshow(sliced_retained, cmap='gray')
21 plt.title('Sliced (Background Retained)')
22 plt.axis('off')
23
24 plt.subplot(2, 3, 3)
25 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.

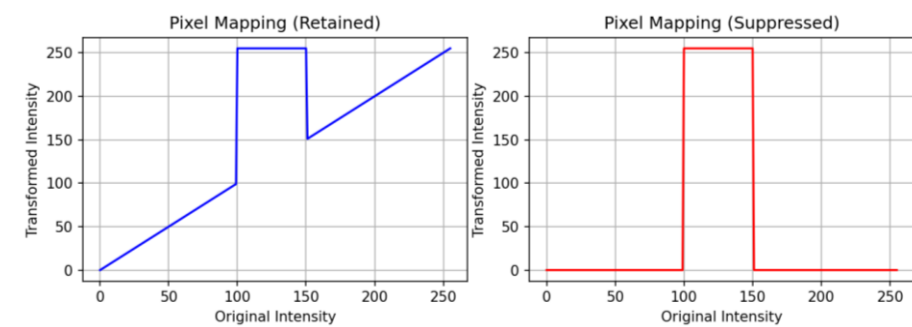


Figure 3 Pixel Mapping Functions for Gray Level Slicing

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.

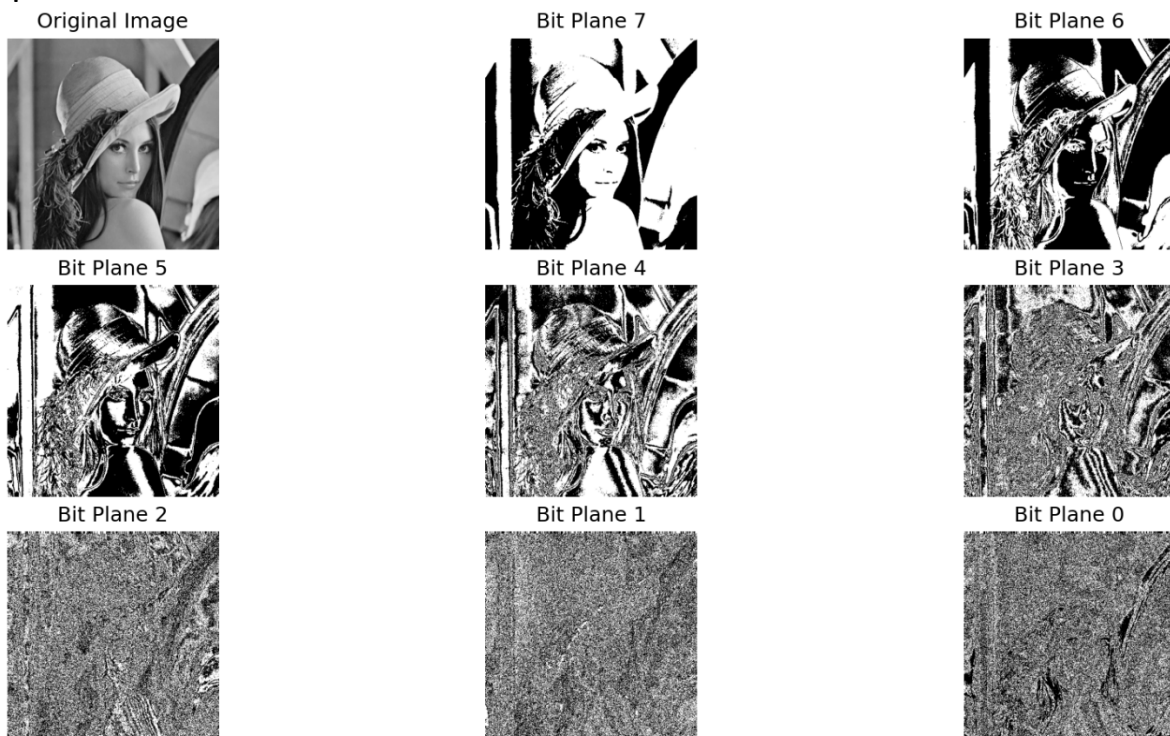
### Activity 3:

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:

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)
6  bit_planes = []
7
8  for i in range(8):
9      plane = (image >> i) & 1      # Get the i-th bit
10     plane_img = (plane * 255).astype(np.uint8) # Scale to 0-255 for display
11     bit_planes.append(plane_img)
12
13     plt.figure(figsize=(12, 8))
14     plt.subplot(3, 3, 1)
15     plt.imshow(image, cmap='gray')
16     plt.title('Original Image')
17     plt.axis('off')
18
19     for i in range(8):
20         plt.subplot(3, 3, i + 2)
21         plt.imshow(bit_planes[7 - i], cmap='gray')
22         plt.title(f'Bit Plane {7 - i}')
23         plt.axis('off')
24
25     plt.tight_layout()
26     plt.show()
27
```

#### 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:

```
1  import cv2
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Load grayscale image
6  image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)
7
8  # Add salt-and-pepper noise
9  noisy = image.copy()
10 noise_density = 0.05
11 num_salt = int(noise_density * image.size * 0.5)
12 num_pepper = int(noise_density * image.size * 0.5)
13
14 # Add salt (white pixels)
15 coords = [np.random.randint(0, i - 1, num_salt) for i in image.shape]
16 noisy[tuple(coords)] = 255
17
18 # Add pepper (black pixels)
19 coords = [np.random.randint(0, i - 1, num_pepper) for i in image.shape]
20 noisy[tuple(coords)] = 0
21
22 # Apply filters on the NOISY image
23 mean_filtered = cv2.blur(noisy, (3, 3))
24 median_filtered = cv2.medianBlur(noisy, 3)
25 min_filtered = cv2.erode(noisy, np.ones((3, 3), np.uint8))
```

```

25 min_filtered = cv2.erode(noisy, np.ones((3, 3), np.uint8))
26 max_filtered = cv2.dilate(min_filtered, np.ones((3, 3), np.uint8))
27
28 # Display results
29 plt.figure(figsize=(12, 8))
30
31 plt.subplot(2, 3, 1)
32 plt.imshow(noisy, cmap='gray')
33 plt.title('Noisy Image (Salt & Pepper)')
34 plt.axis('off')
35
36 plt.subplot(2, 3, 2)
37 plt.imshow(mean_filtered, cmap='gray')
38 plt.title('3x3 Mean Filter')
39 plt.axis('off')
40
41 plt.subplot(2, 3, 3)
42 plt.imshow(median_filtered, cmap='gray')
43 plt.title('Median Filter')
44 plt.axis('off')
45
46 plt.subplot(2, 3, 5)
47 plt.imshow(min_filtered, cmap='gray')
48 plt.title('Min Filter (Erosion)')
49 plt.axis('off')
50
51 plt.subplot(2, 3, 6)
52 plt.imshow(max_filtered, cmap='gray')
53 plt.title('Max Filter (Dilation)')
54 plt.axis('off')
55
56 plt.tight_layout()
57 plt.show()

```

## Output:

Noisy Image (Salt & Pepper)



3x3 Mean Filter



Median Filter



Min Filter (Erosion)



Max Filter (Dilation)



**Analysis:**

By 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.

**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:**

```
1  # sharpening_filter.py
2
3  import cv2
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  image = cv2.imread('Lenna_(test_image).png', cv2.IMREAD_GRAYSCALE)
8
9  # Apply Laplacian Filter
10 laplacian = cv2.Laplacian(image, cv2.CV_64F, ksize=3)
11 laplacian_abs = cv2.convertScaleAbs(laplacian)
12 sharpened = cv2.add(image, laplacian_abs)
13
14 # Display results
15 plt.figure(figsize=(10, 4))
16
17 plt.subplot(1, 3, 1)
18 plt.imshow(image, cmap='gray')
19 plt.title('Original Image')
20 plt.axis('off')
21
22 plt.subplot(1, 3, 2)
23 plt.imshow(laplacian_abs, cmap='gray')
24 plt.title('Laplacian (Edges)')
25 plt.axis('off')
```

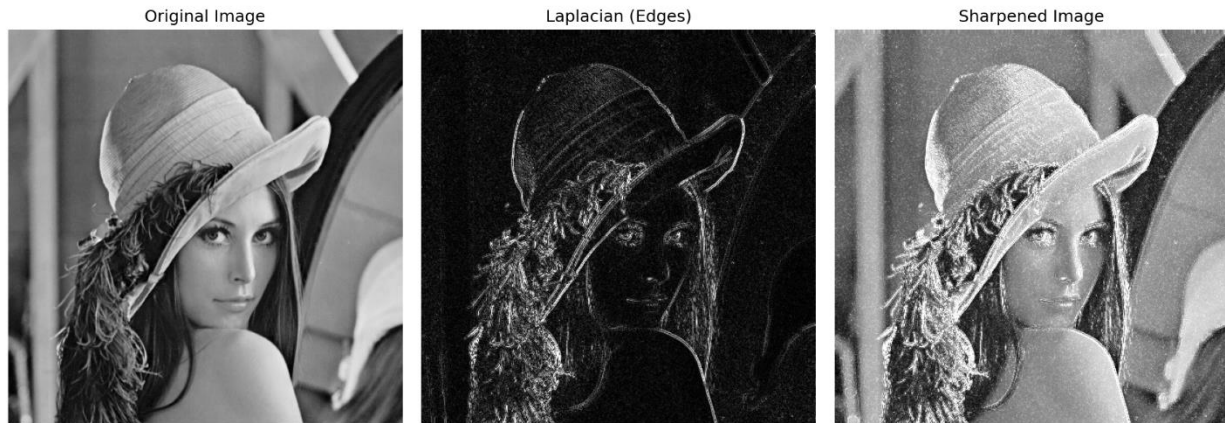


```

26
27 plt.subplot(1, 3, 3)
28 plt.imshow(sharpened, cmap='gray')
29 plt.title('Sharpened Image')
30 plt.axis('off')
31
32 plt.tight_layout()
33 plt.show()
34

```

### 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.