1. Write a Python program to implement the following:
    i) Histogram of a gray image and a color image
    ii) Histogram Equalization *(logic)*
    iii) Histogram Matching
    iv) Adaptive Histogram Equalization

2. Given a noisy image and write the program using Python for image filtering:
    i) Image filtering: (reduction: salt-pepper)
        a) Mean filter
        b) Median filter
        c) Gaussian filter *(logic)*
    ii) Noise:

        a) Gaussian noise
        b) salt-and-pepper noise
        c) speckle noise

3. Given a gray value image and write the program of the following tasks:
    i) Using different masks find the edges of the image:
        a) Robert's operator *(logic)*
        b) Sobel operator *(logic)*
        c) Prewitt operator *(logic)*
        d) Robinson operator
        e) Kirsch operator
        f) Laplacian operator

***Note: Write the programs without using opencv functions if mentioned (logic).*

Assignment 6

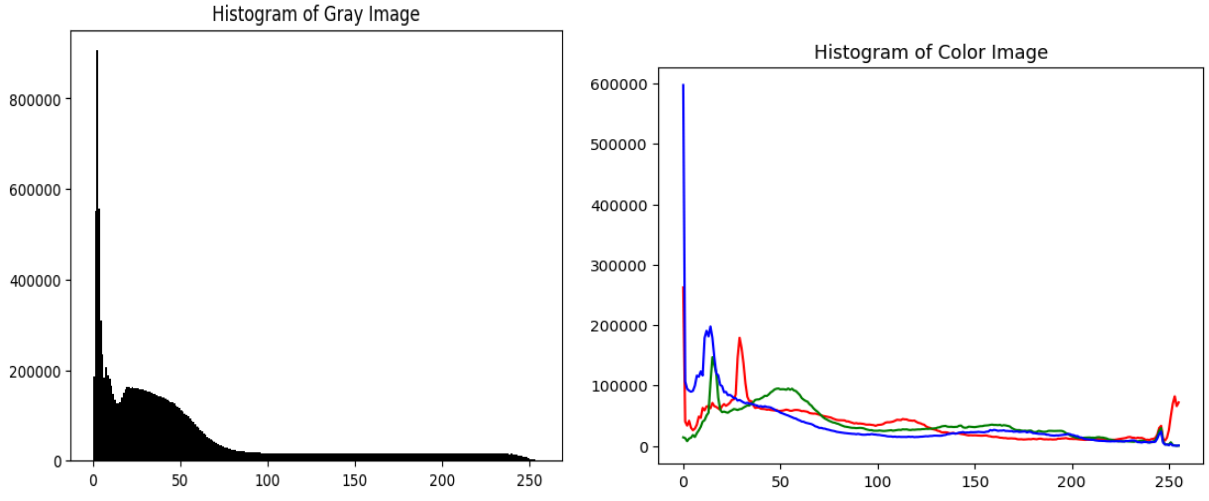
Histogram and Spatial Filtering

1. Write a Python program to implement the following:
Note: Write the programs without using opencv functions if mentioned (logic).
i) Histogram of a gray image and a color image

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Plot histogram for grayscale and color images
def plot_histogram(image, title):
    plt.figure()
    if len(image.shape) == 2:  # Grayscale image
        plt.hist(image.ravel(), bins=256, range=[0, 256], color='black')
    else:  # Color image
        colors = ('r', 'g', 'b')
        for i, col in enumerate(colors):
            hist, _ = np.histogram(image[:, :, i], bins=256, range=[0,
256])
            plt.plot(hist, color=col)
    plt.title(title)
    plt.show()
# Load images
gray_image = np.array(Image.open('gray_images.jpg').convert('L'))
color_image = np.array(Image.open('color_images.jpg'))
# Plot histograms
plot_histogram(gray_image, 'Histogram of Gray Image')
plot_histogram(color_image, 'Histogram of Color Image')
```

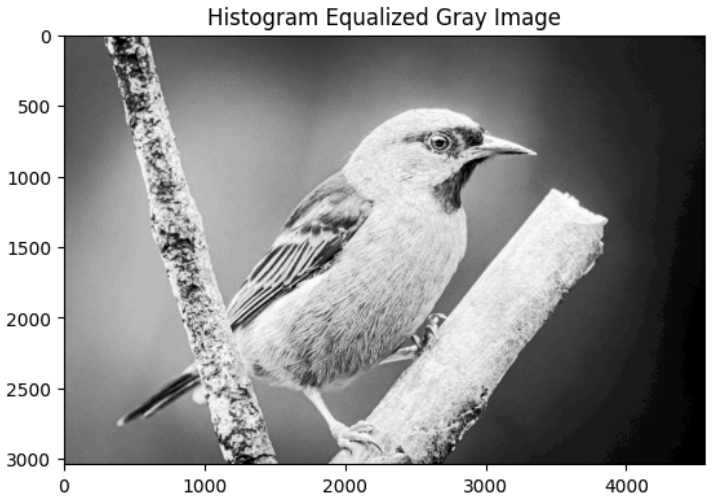
Output:



ii) Histogram Equalization (logic)

```
def histogram_equalization(image):
    if len(image.shape) == 2:  # Grayscale image
        hist, _ = np.histogram(image.flatten(), bins=256, range=[0, 256])
        cdf = hist.cumsum()
        cdf_normalized = cdf / cdf[-1]  # Normalize the CDF
        equalized_image = np.interp(image.flatten(), np.arange(0, 256),
cdf_normalized * 255)
        return equalized_image.reshape(image.shape).astype(np.uint8)
    return image  # For color images, return original for now
# Apply histogram equalization
equalized_gray_image = histogram_equalization(gray_image)
# Display results
plt.imshow(equalized_gray_image, cmap='gray')
plt.title('Histogram Equalized Gray Image')
plt.show()
```

Output:

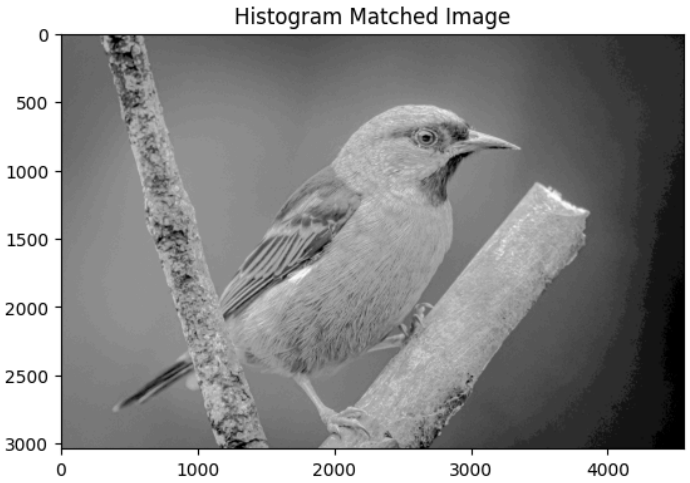


iii) Histogram Matching

```
from skimage import exposure
def histogram_matching(source, template):
    matched = exposure.match_histograms(source, template,
channel_axis=None)
    return matched
# Load source and reference images
reference_image =
np.array(Image.open('reference_images.jpg').convert('L'))
# Apply histogram matching
matched_image = histogram_matching(gray_image, reference_image)
# Display results
```

```
plt.imshow(matched_image, cmap='gray')
plt.title('Histogram Matched Image')
plt.show()
```

Output:



iv) Adaptive Histogram Equalization

```
from skimage import exposure
def adaptive_histogram_equalization(image):
    return exposure.equalize_adapthist(image, clip_limit=0.03)
# Apply adaptive histogram equalization
adaptive_equalized_image = adaptive_histogram_equalization(gray_image)
# Display results
plt.imshow(adaptive_equalized_image, cmap='gray')
plt.title('Adaptive Histogram Equalized Image')
plt.show()
```

Output:

2. Given a noisy image and write the program using Python for image filtering:
i) Image filtering: (reduction: salt-pepper)
a) Mean filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter
from PIL import Image
# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)
# Mean Filter
def mean_filter(image, size=3):
    return uniform_filter(image, size=size)
# Display image
def display_image(original, filtered, title="Filtered Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(filtered, cmap='gray')
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()
# Apply the mean filter to an image
def apply_mean_filter(image_path='image.jpg'):
    original_image = load_image(image_path)
    filtered_image = mean_filter(original_image, size=3)
    display_image(original_image, filtered_image, title="Mean Filtered
Image")
# Run the mean filter example
apply_mean_filter('image.jpg')
```

Output:

b) Median filter

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import median_filter
from PIL import Image
# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)
# Median Filter
def median_filter_custom(image, size=3):
    return median_filter(image, size=size)
# Display image
def display_image(original, filtered, title="Filtered Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(filtered, cmap='gray')
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()
# Apply the median filter to an image
def apply_median_filter(image_path='image.jpg'):
    original_image = load_image(image_path)
    filtered_image = median_filter_custom(original_image, size=3)
    display_image(original_image, filtered_image, title="Median Filtered
Image")
# Run the median filter example
apply_median_filter('image.jpg')
```

Output:

c) Gaussian filter (logic)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter
from PIL import Image
# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)
# Gaussian Filter
def gaussian_filter_custom(image, sigma=1):
    return gaussian_filter(image, sigma=sigma)
# Display image
def display_image(original, filtered, title="Filtered Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(filtered, cmap='gray')
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()
# Apply the Gaussian filter to an image
def apply_gaussian_filter(image_path='image.jpg'):
    original_image = load_image(image_path)
    filtered_image = gaussian_filter_custom(original_image, sigma=1)
    display_image(original_image, filtered_image, title="Gaussian Filtered
Image")
# Run the Gaussian filter example
apply_gaussian_filter('image.jpg')
```

Output:

ii) Noise:

a) Gaussian noise

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)
# Add Gaussian Noise
def add_gaussian_noise(image, mean=0, std=25):
    gaussian_noise = np.random.normal(mean, std, image.shape)
    noisy_image = image + gaussian_noise
    return np.clip(noisy_image, 0, 255).astype(np.uint8)
# Display image
def display_image(original, noisy, title="Noisy Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(noisy, cmap='gray')
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()
# Apply Gaussian noise to an image
def apply_gaussian_noise(image_path='image.jpg'):
    original_image = load_image(image_path)
    noisy_image = add_gaussian_noise(original_image)
    display_image(original_image, noisy_image, title="Gaussian Noisy
Image")
apply_gaussian_noise('image.jpg')
```

Output:

b) salt-and-pepper noise

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)

# Add Salt-and-Pepper Noise
def add_salt_and_pepper_noise(image, prob=0.05):
    noisy_image = np.copy(image)
    num_salt = int(prob * image.size / 2)
    num_pepper = int(prob * image.size / 2)

    # Add salt noise (white pixels)
    salt_coords = (np.random.randint(0, image.shape[0], num_salt),
                   np.random.randint(0, image.shape[1], num_salt))
    noisy_image[salt_coords] = 255
    # Add pepper noise (black pixels)
    pepper_coords = (np.random.randint(0, image.shape[0], num_pepper),
                     np.random.randint(0, image.shape[1], num_pepper))
    noisy_image[pepper_coords] = 0
    return noisy_image

# Display image
def display_image(original, noisy, title="Noisy Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(noisy, cmap='gray')
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()

# Apply Salt-and-Pepper noise to an image
def apply_salt_and_pepper_noise(image_path='image.jpg'):
    original_image = load_image(image_path)
    noisy_image = add_salt_and_pepper_noise(original_image)
```

```
    display_image(original_image, noisy_image, title="Salt-and-Pepper
Noisy Image")

# Run the Salt-and-Pepper noise example
apply_salt_and_pepper_noise('image.jpg')
```

Output:

c) speckle noise

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Load image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)

# Add Speckle Noise
def add_speckle_noise(image, mean=0, std=0.2):
    speckle = np.random.normal(mean, std, image.shape) * image
    noisy_image = image + speckle
    return np.clip(noisy_image, 0, 255).astype(np.uint8)

# Display image
def display_image(original, noisy, title="Noisy Image"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')

    axes[1].imshow(noisy, cmap='gray')
```

```
    axes[1].set_title(title)
    axes[1].axis('off')
    plt.show()

# Apply Speckle noise to an image
def apply_speckle_noise(image_path='image.jpg'):
    original_image = load_image(image_path)
    noisy_image = add_speckle_noise(original_image)
    display_image(original_image, noisy_image, title="Speckle Noisy
Image")

# Run the Speckle noise example
apply_speckle_noise('image.jpg')
```

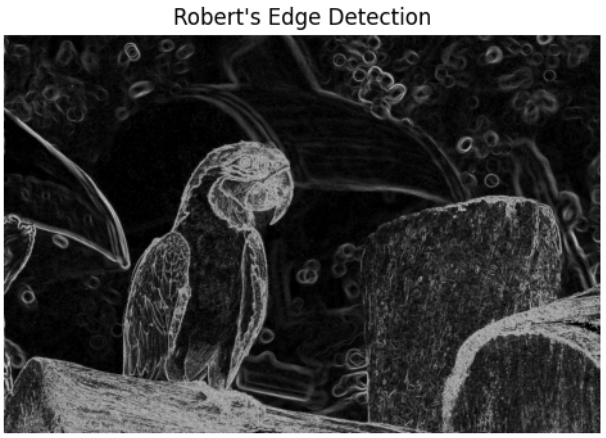
Output:

3. Given a gray value image and write the program of the following tasks:
i) Using different masks find the edges of the image:
a) Robert's operator (logic)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import convolve
from PIL import Image
# Load the image
def load_image(path='image.jpg', grayscale=True):
    image = Image.open(path)
    if grayscale:
        image = image.convert('L')  # Convert to grayscale
    return np.array(image)
# Robert's operator
def roberts_operator(image):
    # Define the Roberts cross kernels
    kernel_x = np.array([[1, 0], [0, -1]])
    kernel_y = np.array([[0, 1], [-1, 0]])
```

```
    # Apply the kernels using convolution
    gx = convolve(image, kernel_x)
    gy = convolve(image, kernel_y)
    # Compute gradient magnitude
    edge_image = np.sqrt(gx**2 + gy**2)
    return np.clip(edge_image, 0, 255).astype(np.uint8)
# Display the result
def display_image(image, title):
    plt.figure(figsize=(6, 6))
    plt.imshow(image, cmap='gray')
    plt.title(title)
    plt.axis('off')
    plt.show()
# Load the image and apply the operator
image = load_image('image.jpg')
roberts_edge_image = roberts_operator(image)
display_image(roberts_edge_image, "Robert's Edge Detection")
```

Output:



b) Sobel operator (logic)

```
# Sobel operator
def sobel_operator(image):
    # Define the Sobel kernels
    kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    kernel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    # Apply the kernels using convolution
    gx = convolve(image, kernel_x)
    gy = convolve(image, kernel_y)
    # Compute gradient magnitude
    edge_image = np.sqrt(gx**2 + gy**2)
    return np.clip(edge_image, 0, 255).astype(np.uint8)
# Display the result
sobel_edge_image = sobel_operator(image)
```

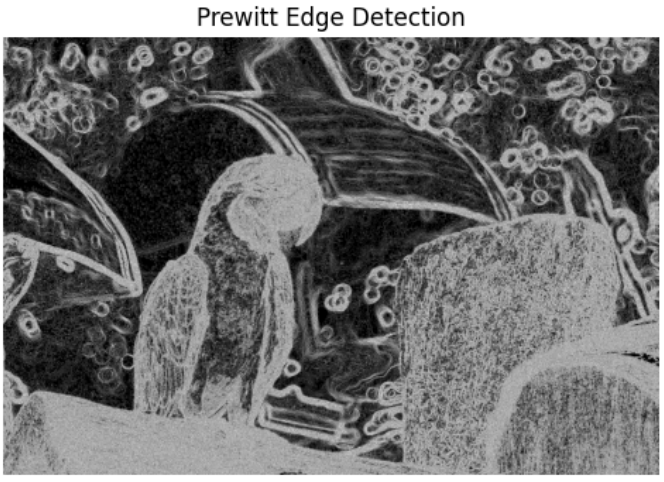
```
display_image(sobel_edge_image, "Sobel Edge Detection")
```

Output:

c) Prewitt operator (logic)

```
# Prewitt operator
def prewitt_operator(image):
    # Define the Prewitt kernels
    kernel_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    kernel_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
    # Apply the kernels using convolution
    gx = convolve(image, kernel_x)
    gy = convolve(image, kernel_y)
    # Compute gradient magnitude
    edge_image = np.sqrt(gx**2 + gy**2)
    return np.clip(edge_image, 0, 255).astype(np.uint8)
# Display the result
prewitt_edge_image = prewitt_operator(image)
display_image(prewitt_edge_image, "Prewitt Edge Detection")
```

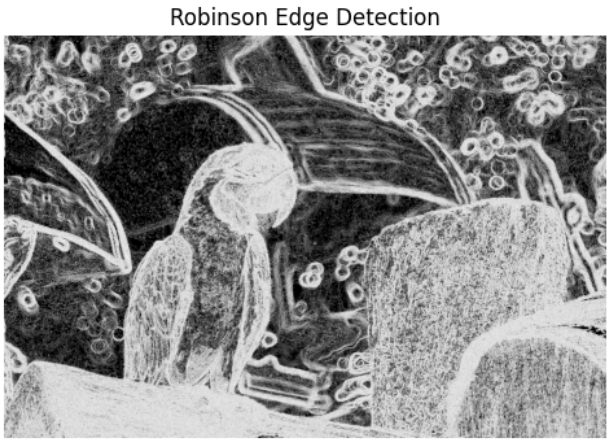
Output:



d) Robinson operator

```
# Robinson operator
def robinson_operator(image):
    # Define the Robinson direction kernels
    kernels = [
        np.array([[ 1,  0, -1], [ 2,  0, -2], [ 1,  0, -1]]),
        np.array([[ 0,  1,  2], [-1,  0,  1], [-2, -1,  0]]),
        np.array([[-1, -2, -1], [ 0,  0,  0], [ 1,  2,  1]]),
        np.array([[-2, -1,  0], [-1,  0,  1], [ 0,  1,  2]]),
        np.array([[-1,  0,  1], [-2,  0,  2], [-1,  0,  1]]),
        np.array([[ 0, -1, -2], [ 1,  0, -1], [ 2,  1,  0]]),
        np.array([[ 1,  2,  1], [ 0,  0,  0], [-1, -2, -1]]),
        np.array([[ 2,  1,  0], [ 1,  0, -1], [ 0, -1, -2]]) ]
    edge_images = [np.sqrt(convolve(image, kernel) ** 2) for kernel in
kernels]
    return np.max(edge_images, axis=0).astype(np.uint8)
# Display the result
robinson_edge_image = robinson_operator(image)
display_image(robinson_edge_image, "Robinson Edge Detection")
```

Output:

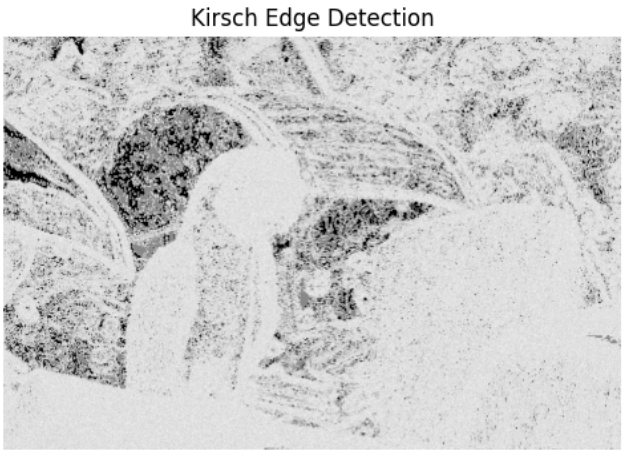


e) Kirsch operator

```
# Kirsch operator
def kirsch_operator(image):
    # Define the Kirsch direction kernels
    kernels = [
        np.array([[ 5,  5,  5], [-3,  0, -3], [-3, -3, -3]]),
        np.array([[ 5,  5, -3], [ 5,  0, -3], [-3, -3, -3]]),
        np.array([[-3,  5,  5], [-3,  0, -3], [-3, -3, -3]]),
        np.array([[-3,  5,  5], [-3,  0,  5], [-3, -3, -3]]),
        np.array([[-3, -3, -3], [ 5,  0,  5], [ 5,  5,  5]]),
        np.array([[-3, -3, -3], [-3,  0,  5], [ 5,  5,  5]]),
```

```
        np.array([[-3, -3, -3], [-3, 0, -3], [ 5, 5, 5]]),
        np.array([[-3, -3, -3], [ 5, 0, 5], [ 5, 5, 5]]) ]
    edge_images = [np.sqrt(convolve(image, kernel) ** 2) for kernel in
kernels]
    return np.max(edge_images, axis=0).astype(np.uint8)
# Display the result
kirsch_edge_image = kirsch_operator(image)
display_image(kirsch_edge_image, "Kirsch Edge Detection")
```

Output:



f) Laplacian operator

```
# Laplacian operator
def laplacian_operator(image):
    kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    laplacian_edge_image = convolve(image, kernel)
    return np.clip(laplacian_edge_image, 0, 255).astype(np.uint8)
# Display the result
laplacian_edge_image = laplacian_operator(image)
display_image(laplacian_edge_image, "Laplacian Edge Detection")
```

Output: