Wroclaw University Of Science And Technology

CANER OLCAY 276715

Artificial Intelligence and Computer Vision

13.11.2024

List 4

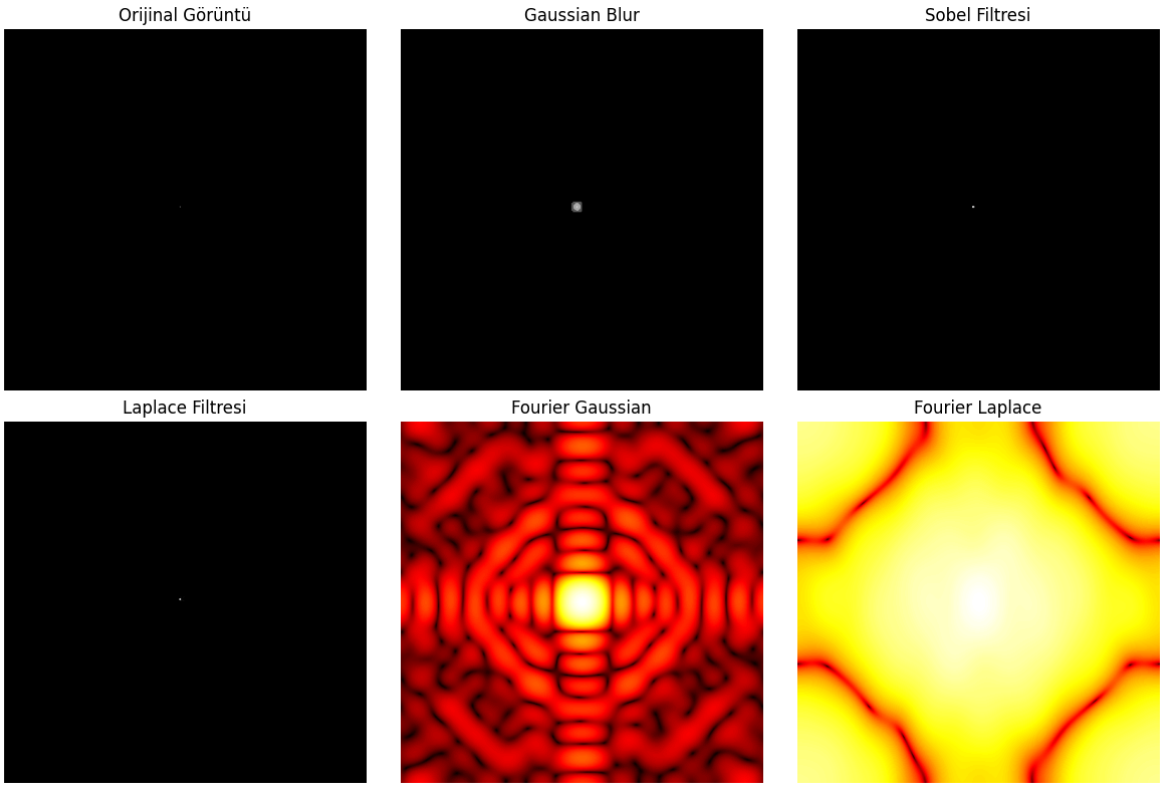
Task 1

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

# Load the image
image_path = r"C:\Users\caner\OneDrive\Desktop\Python\Lab6\dark.jpg"
step_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

if step_image is None:
    print("Image could not be loaded. Check the file path!")
    exit()

# Apply Gaussian Blur
gaussian_blur = cv2.GaussianBlur(step_image, (15, 15), 5)

# Apply Sobel Filter
sobel_x = cv2.Sobel(step_image, cv2.CV_64F, 1, 0, ksize=3)  #
Horizontal edges
sobel_y = cv2.Sobel(step_image, cv2.CV_64F, 0, 1, ksize=3)  # Vertical
edges
sobel_combined = cv2.magnitude(sobel_x, sobel_y)

# Apply Laplace Filter
laplace_filter = cv2.Laplacian(step_image, cv2.CV_64F)
laplace_filter = np.uint8(np.absolute(laplace_filter))

# Function to compute the Fourier spectrum
def compute_fourier_spectrum(img):
    dft = np.fft.fft2(img)
    dft_shift = np.fft.fftshift(dft)
    magnitude_spectrum = 20 * np.log(np.abs(dft_shift) + 1)
    return magnitude_spectrum

# Compute the Fourier spectra
fourier_gaussian = compute_fourier_spectrum(gaussian_blur)
fourier_laplace = compute_fourier_spectrum(laplace_filter)

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

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

# Gaussian Blur
plt.subplot(2, 3, 2)
plt.title("Gaussian Blur")
plt.imshow(gaussian_blur, cmap='gray')
plt.axis('off')

# Sobel Filter
plt.subplot(2, 3, 3)
plt.title("Sobel Filter")
```

```
plt.imshow(sobel_combined, cmap='gray')
plt.axis('off')

# Laplace Filter
plt.subplot(2, 3, 4)
plt.title("Laplace Filter")
plt.imshow(laplace_filter, cmap='gray')
plt.axis('off')

# Fourier Spectrum of Gaussian Blur
plt.subplot(2, 3, 5)
plt.title("Fourier Gaussian")
plt.imshow(fourier_gaussian, cmap='hot')
plt.axis('off')

# Fourier Spectrum of Laplace Filter
plt.subplot(2, 3, 6)
plt.title("Fourier Laplace")
plt.imshow(fourier_laplace, cmap='hot')
plt.axis('off')

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

In this task, Gaussian Blur, Sobel, and Laplace filters were applied to the input image to analyze their effects. The Gaussian Blur smoothed the image, reducing noise and sharp transitions, demonstrating its low-pass filtering property. The Sobel filter highlighted the edges by detecting intensity gradients in both horizontal and vertical directions. The Laplace filter further emphasized areas with rapid intensity changes, showing its high-pass filtering nature. Fourier spectra of the Gaussian Blur and Laplace outputs were computed, revealing the dominance of low frequencies in the blurred image and the prominence of high frequencies in the Laplace-filtered image. This task effectively illustrates how these filters process spatial and frequency components of an image.

Outputs:



Task 2

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


# Gaussian Noise Function
def add_gaussian_noise(image, mean=0, stddev=25):
    """Add Gaussian noise to an image."""
    noise = np.random.normal(mean, stddev,
image.shape).astype(np.float32)
    noisy_image = cv2.add(image.astype(np.float32), noise)
    return np.clip(noisy_image, 0, 255).astype(np.uint8)


# Median Filter Function
def apply_median_filter(image, kernel_size=5):
    """Apply median filtering."""
    return cv2.medianBlur(image, kernel_size)


# Sharpening Function
def sharpen_image(image):
    """Apply sharpening filter."""
```

```
    kernel = np.array([[0, -1, 0],
                       [-1, 5, -1],
                       [0, -1, 0]])
    return cv2.filter2D(image, -1, kernel)


# Visualization Function
def visualize(images, titles):
    """Display images with titles."""
    fig, axes = plt.subplots(2, 4, figsize=(16, 8))
    for i, ax in enumerate(axes.flat[:len(images)]):
        ax.imshow(images[i], cmap='gray')
        ax.set_title(titles[i], fontsize=10)
        ax.axis('off')
    plt.tight_layout()
    plt.show()


# Main Code
if __name__ == "__main__":
    # Load Image
    image_path =
"C:/Users/caner/OneDrive/Desktop/Python/Lab5/lenna.png"
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Ensure the image is loaded properly
    if image is None:
        raise FileNotFoundError(f"Unable to load image at {image_path}.
Please check the file path and ensure the file exists.")

    # Add Gaussian Noise
    noise1 = add_gaussian_noise(image, stddev=10)  # Low noise
    noise2 = add_gaussian_noise(image, stddev=30)  # High noise

    # Apply Median Filtering
    filtered1 = apply_median_filter(noise1)
    filtered2 = apply_median_filter(noise2)

    # Apply Sharpening
    sharpened1 = sharpen_image(filtered1)
    sharpened2 = sharpen_image(filtered2)

    # Titles for Visualization
    titles = [
        "Original Image", "Noise σ=10", "Noise σ=30",
        "Filtered σ=10", "Filtered σ=30",
        "Sharpened σ=10", "Sharpened σ=30"
    ]

    # Images to Visualize
    images = [image, noise1, noise2, filtered1, filtered2, sharpened1,
sharpened2]

    # Visualize Results
    visualize(images, titles)
```

In this task, two types of images were created: a low-frequency image with a horizontal gradient and a high-frequency image with a checkerboard pattern. Gaussian noise was added to both images using low and high standard deviation (σ) values to simulate varying levels of noise. Afterward, the noisy images were processed using a Gaussian blur filter to reduce the noise, followed by a sharpening filter to enhance the edges. The results were visualized to show the original, noisy, filtered, and sharpened versions of the images. This process allowed me to analyze the effects of noise, filtering, and sharpening on images with different frequency components.

Output:

Task 3

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

# Full file path to the image
```

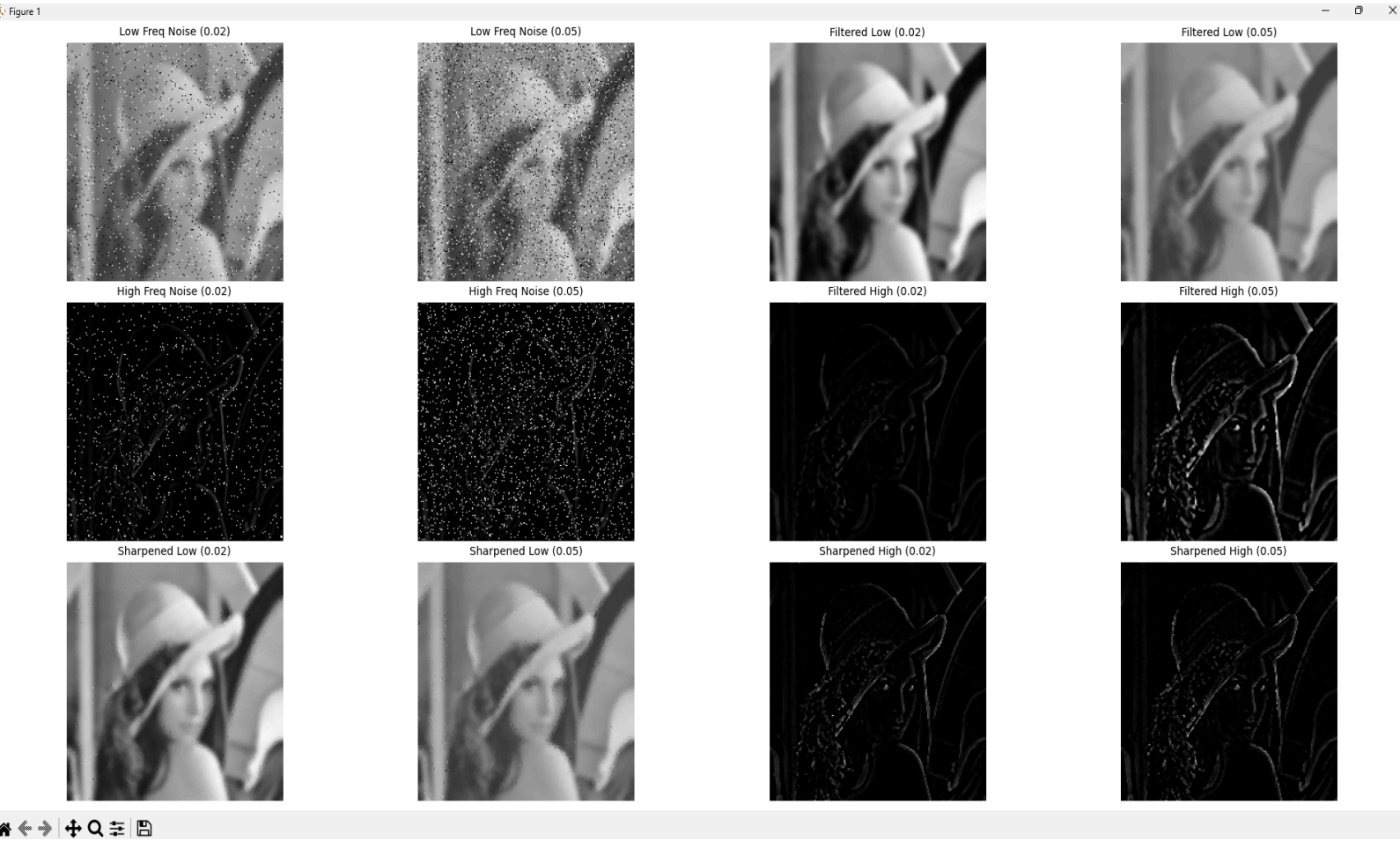
```
image_path = r'C:\Users\caner\OneDrive\Desktop\Python\Lab5\lenna.png'

# Load the image in grayscale
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
if image is None:
    print("Error: Image not found or unable to load. Check the file
path.")
    exit()

# Generate Low-Frequency and High-Frequency Images
low_freq_image = cv2.GaussianBlur(image, (15, 15), 0)
high_freq_image = cv2.subtract(image, low_freq_image)

# Add Salt and Pepper Noise
def add_salt_pepper_noise(image, salt_prob=0.02, pepper_prob=0.02):
    noisy_image = image.copy()
    total_pixels = image.size

    # Add salt (white pixels)
    salt_pixels = int(total_pixels * salt_prob)
    salt_coords = [np.random.randint(0, i - 1, salt_pixels) for i in
image.shape]
    noisy_image[salt_coords[0], salt_coords[1]] = 255

    # Add pepper (black pixels)
    pepper_pixels = int(total_pixels * pepper_prob)
    pepper_coords = [np.random.randint(0, i - 1, pepper_pixels) for i
in image.shape]
    noisy_image[pepper_coords[0], pepper_coords[1]] = 0

    return noisy_image

# Apply Median Filter
def apply_median_filter(image):
    return cv2.medianBlur(image, 3)

# Apply Sharpening
def sharpen_image(image):
    kernel = np.array([[0, -1, 0],
                       [-1, 5, -1],
                       [0, -1, 0]])
    return cv2.filter2D(image, -1, kernel)

# Adding noise to low-frequency and high-frequency images
low_freq_noise1 = add_salt_pepper_noise(low_freq_image, 0.02, 0.02)
low_freq_noise2 = add_salt_pepper_noise(low_freq_image, 0.05, 0.05)
high_freq_noise1 = add_salt_pepper_noise(high_freq_image, 0.02, 0.02)
high_freq_noise2 = add_salt_pepper_noise(high_freq_image, 0.05, 0.05)

# Filtering noisy images
filtered_low1 = apply_median_filter(low_freq_noise1)
filtered_low2 = apply_median_filter(low_freq_noise2)
filtered_high1 = apply_median_filter(high_freq_noise1)
filtered_high2 = apply_median_filter(high_freq_noise2)

# Sharpening filtered images
sharpened_low1 = sharpen_image(filtered_low1)
sharpened_low2 = sharpen_image(filtered_low2)
```

```
sharpened_high1 = sharpen_image(filtered_high1)
sharpened_high2 = sharpen_image(filtered_high2)

# Visualizing Results
fig, axes = plt.subplots(3, 4, figsize=(16, 12))
titles = [
    "Low Freq Noise (0.02)", "Low Freq Noise (0.05)",
    "Filtered Low (0.02)", "Filtered Low (0.05)",
    "High Freq Noise (0.02)", "High Freq Noise (0.05)",
    "Filtered High (0.02)", "Filtered High (0.05)",
    "Sharpened Low (0.02)", "Sharpened Low (0.05)",
    "Sharpened High (0.02)", "Sharpened High (0.05)"
]
images = [
    low_freq_noise1, low_freq_noise2,
    filtered_low1, filtered_low2,
    high_freq_noise1, high_freq_noise2,
    filtered_high1, filtered_high2,
    sharpened_low1, sharpened_low2,
    sharpened_high1, sharpened_high2
]

# Plot the images
for i, ax in enumerate(axes.flat[:len(images)]):
    ax.imshow(images[i], cmap='gray')
    ax.set_title(titles[i], fontsize=10)
    ax.axis('off')

# Hide any remaining empty plots
for ax in axes.flat[len(images):]:
    ax.axis('off')

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

In this task, salt and pepper noise was applied to the low-frequency and high-frequency images. This type of noise adds random white (salt) and black (pepper) pixels to simulate real-world distortions. After adding noise, the images were processed with a median filter, which is effective at removing salt and pepper noise while preserving edges. Subsequently, a sharpening filter was applied to the filtered images to enhance their details. The results were visualized to showcase the original, noisy, filtered, and sharpened versions of the images for both low and high noise levels. This task demonstrated the effectiveness of median filtering against salt and pepper noise

and highlighted how sharpening can improve the visual quality of filtered images.

Output :



Task 4:

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

# Load the image
image_path = r"C:\Users\caner\OneDrive\Desktop\Python\Lab6\Lenna.png"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

if image is None:
    print("Image could not be loaded. Check the file path!")
    exit()

# Create low-frequency and high-frequency versions of the image
low_frequency_image = cv2.GaussianBlur(image, (15, 15), 5)  # Blur
(low-frequency)
high_frequency_image = cv2.addWeighted(image, 1.5, low_frequency_image,
-0.5, 0)  # Sharpen (high-frequency)

# Sobel edge detection function
def sobel_edge_detection(img):
    sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # Sobel X
    sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # Sobel Y
    sobel_combined = cv2.magnitude(sobelx, sobely)  # Combine gradients
    return np.uint8(np.absolute(sobel_combined))

# Perform Sobel edge detection
sobel_edges_low = sobel_edge_detection(low_frequency_image)
sobel_edges_high = sobel_edge_detection(high_frequency_image)

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

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

# Low-frequency image
plt.subplot(2, 3, 2)
plt.title("Low-Frequency Image")
plt.imshow(low_frequency_image, cmap='gray')
plt.axis('off')

# High-frequency image
plt.subplot(2, 3, 3)
plt.title("High-Frequency Image")
plt.imshow(high_frequency_image, cmap='gray')
plt.axis('off')

# Sobel edges (low-frequency)
plt.subplot(2, 3, 4)
plt.title("Edges (Low-Frequency)")
plt.imshow(sobel_edges_low, cmap='gray')
plt.axis('off')
```

```
# Sobel edges (high-frequency)
plt.subplot(2, 3, 5)
plt.title("Edges (High-Frequency)")
plt.imshow(sobel_edges_high, cmap='gray')
plt.axis('off')

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

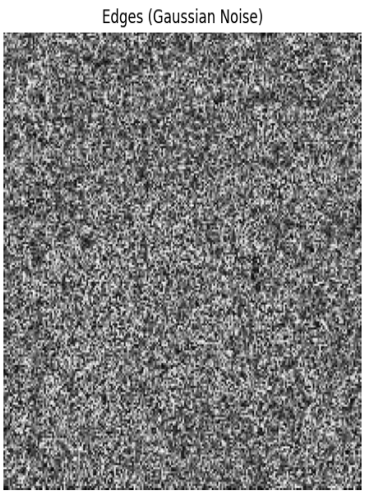
In this task, Sobel filters were used for edge detection on low- and high-frequency images. The low-frequency image, created using Gaussian blur, resulted in smoother, less distinct edges, while the high-frequency image, sharpened for more detail, produced sharper and clearer edges. This demonstrates that high-frequency images are better suited for detecting fine edges, while low-frequency images smooth out details. The task highlights the role of image frequency in edge detection effectiveness.

Task5:

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

# Load the image
image_path = r"C:\Users\caner\OneDrive\Desktop\Python\Lab6\Lenna.png"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

if image is None:
    print("Image could not be loaded. Check the file path!")
    exit()

# Add Gaussian noise function
def add_gaussian_noise(img, mean=0, std=25):
    noise = np.random.normal(mean, std, img.shape).astype(np.float32)
    noisy_img = cv2.add(img.astype(np.float32), noise)
    noisy_img = np.clip(noisy_img, 0, 255).astype(np.uint8)
    return noisy_img

# Sobel edge detection function
def sobel_edge_detection(img):
    sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # Sobel X
    sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # Sobel Y
    sobel_combined = cv2.magnitude(sobelx, sobely)  # Combine gradients
    return np.uint8(np.absolute(sobel_combined))

# Add Gaussian noise to the image
noisy_image = add_gaussian_noise(image, std=30)

# Perform Sobel edge detection
sobel_edges_original = sobel_edge_detection(image)
sobel_edges_noisy = sobel_edge_detection(noisy_image)

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

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

# Gaussian noisy image
plt.subplot(2, 3, 2)
plt.title("Gaussian Noisy Image")
plt.imshow(noisy_image, cmap='gray')
plt.axis('off')

# Sobel edges (original)
plt.subplot(2, 3, 4)
plt.title("Edges (Original)")
plt.imshow(sobel_edges_original, cmap='gray')
plt.axis('off')

# Sobel edges (noisy)
```

```
plt.subplot(2, 3, 5)
plt.title("Edges (Gaussian Noise)")
plt.imshow(sobel_edges_noisy, cmap='gray')
plt.axis('off')

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

In this task, Gaussian noise was added to a high-frequency image to observe its impact on edge detection using Sobel filters. The original image produced clear and distinct edges, while the noisy image resulted in scattered and less accurate edges due to the randomness introduced by the noise. This demonstrates that Gaussian noise significantly disrupts edge detection, reducing the clarity and accuracy of detected edges.



Task6:

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

# Load the image
image_path = r"C:\Users\caner\OneDrive\Desktop\Python\Lab6\Lenna.png"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

if image is None:
    print("Image could not be loaded. Check the file path!")
    exit()

# Create a high-frequency image (Sharpened version)
low_frequency_image = cv2.GaussianBlur(image, (15, 15), 5)  # Blurred
version
high_frequency_image = cv2.addWeighted(image, 1.5, low_frequency_image,
-0.5, 0)

# Custom Laplace filter implementation
def custom_laplace_filter(img):
    # Define a 3x3 Laplacian kernel
    kernel = np.array([[0, -1, 0],
                       [-1, 4, -1],
                       [0, -1, 0]])
    # Apply the kernel using OpenCV's filter2D function
    filtered_image = cv2.filter2D(img, ddepth=cv2.CV_64F,
kernel=kernel)
    return np.uint8(np.absolute(filtered_image))

# Apply the custom Laplace filter
custom_laplace_result = custom_laplace_filter(high_frequency_image)

# Apply OpenCV's built-in Laplace function
built_in_laplace_result = cv2.Laplacian(high_frequency_image,
cv2.CV_64F)
built_in_laplace_result =
np.uint8(np.absolute(built_in_laplace_result))

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

# Original high-frequency image
plt.subplot(2, 3, 1)
plt.title("High-Frequency Image")
plt.imshow(high_frequency_image, cmap='gray')
plt.axis('off')

# Custom Laplace filter result
plt.subplot(2, 3, 2)
plt.title("Custom Laplace Filter")
plt.imshow(custom_laplace_result, cmap='gray')
plt.axis('off')

# OpenCV built-in Laplace result
plt.subplot(2, 3, 3)
```

```
plt.title("Built-in Laplace Filter")
plt.imshow(built_in_laplace_result, cmap='gray')
plt.axis('off')

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

In this task, a custom Laplace filter was implemented using a predefined kernel and applied to a high-frequency image. The results were compared with OpenCV's built-in Laplace filter. Both filters effectively detected edges, with slight differences in intensity and clarity due to implementation details. The custom filter successfully replicated the functionality of the built-in method, demonstrating the accuracy of the manual implementation.