Image Processing Lab 1: Spatial and Frequency Domain Filtering

1. Introduction to Spatial Filtering:

1.1 What is Spatial Filtering?

Spatial filtering is a fundamental technique in image processing that modifies or enhances images by applying operations to pixel neighborhoods. The value of each output pixel is determined by applying an algorithm to the values of pixels about the corresponding input pixel.

1.2 Key Concepts:

- **Neighborhood Operation**: Operations performed on a set of pixels surrounding a target pixel.
- **Linear Filtering**: Output pixel values are linear combinations of input neighborhood values.
- **Convolution Kernel**: A matrix of weights used to compute the weighted sum of neighboring pixels.

1.3 Common Applications:

- Image smoothing and noise reduction.
- Edge detection and enhancement.
- Sharpening and detail enhancement.
- Feature extraction.

2. Convolution Operations:

2.1 Basic Convolution

Convolution is a neighborhood operation where each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called the convolution kernel or filter.

Basic Principle: Look for neighborhoods with strong signs of change or specific patterns.

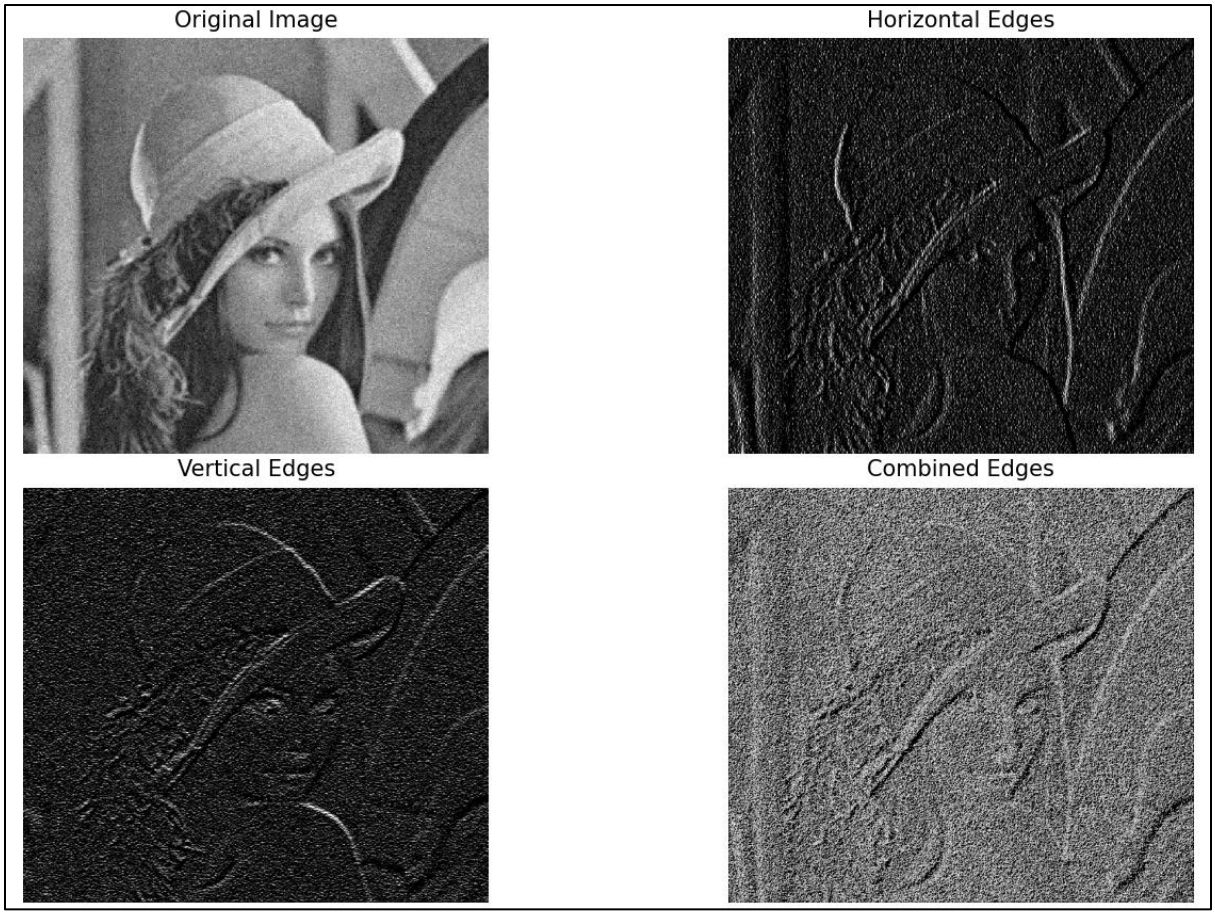
Example 1: Edge Detection with Sobel Operator

- **Code:**

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

# Load an image as grayscale
image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

# Define Sobel kernels for edge detection
# Horizontal edge detection
horizontal_kernel = np.array([[-1, 0, 1],
                              [-2, 0, 2],
                              [-1, 0, 1]], dtype=np.float32)

# Vertical edge detection
vertical_kernel = np.array([[-1, -2, -1],
                            [0,  0,  0],
```

```
                            [1,  2,  1]], dtype=np.float32)

# Apply convolution
horizontal_edges = cv2.filter2D(image, -1, horizontal_kernel)
vertical_edges = cv2.filter2D(image, -1, vertical_kernel)

# Combine edges using magnitude
combined_edges = np.sqrt(horizontal_edges**2 + vertical_edges**2)
combined_edges = np.uint8(combined_edges)

# Display results
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes[0, 0].imshow(image, cmap='gray')
axes[0, 0].set_title('Original Image')
axes[0, 0].axis('off')

axes[0, 1].imshow(horizontal_edges, cmap='gray')
axes[0, 1].set_title('Horizontal Edges')
axes[0, 1].axis('off')

axes[1, 0].imshow(vertical_edges, cmap='gray')
axes[1, 0].set_title('Vertical Edges')
axes[1, 0].axis('off')

axes[1, 1].imshow(combined_edges, cmap='gray')
axes[1, 1].set_title('Combined Edges')
axes[1, 1].axis('off')

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

- **Output:**



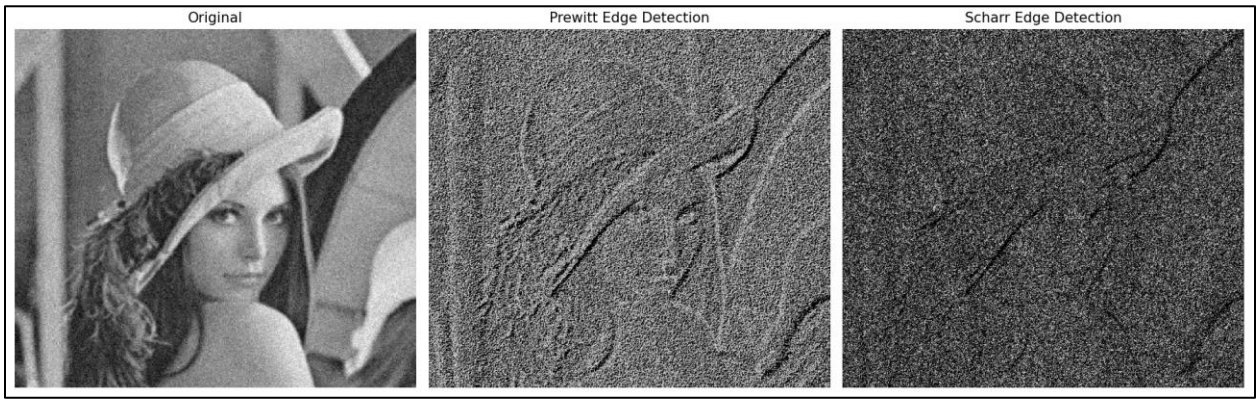
Example 2: Custom Edge Detection Kernels

- **Code:**

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

# Load an image as grayscale
image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

# Prewitt operator for edge detection
prewitt_x = np.array([[-1, 0, 1],
                      [-1, 0, 1],
                      [-1, 0, 1]], dtype=np.float32)

prewitt_y = np.array([[-1, -1, -1],
                      [0,  0,  0],
                      [1,  1,  1]], dtype=np.float32)

# Scharr operator (more accurate than Sobel)
scharr_x = np.array([[-3, 0, 3],
```

```
                     [-10, 0, 10],
                     [-3, 0, 3]], dtype=np.float32)

scharr_y = np.array([[-3, -10, -3],
                     [0,  0,  0],
                     [3,  10,  3]], dtype=np.float32)

# Apply all filters
prewitt_edges_x = cv2.filter2D(image, -1, prewitt_x)
prewitt_edges_y = cv2.filter2D(image, -1, prewitt_y)
scharr_edges_x = cv2.filter2D(image, -1, scharr_x)
scharr_edges_y = cv2.filter2D(image, -1, scharr_y)

# Compute magnitudes
prewitt_magnitude = np.sqrt(prewitt_edges_x**2 + prewitt_edges_y**2)
scharr_magnitude = np.sqrt(scharr_edges_x**2 + scharr_edges_y**2)

# Display comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original')
axes[1].imshow(prewitt_magnitude, cmap='gray')
axes[1].set_title('Prewitt Edge Detection')
axes[2].imshow(scharr_magnitude, cmap='gray')
axes[2].set_title('Scharr Edge Detection')

for ax in axes:
    ax.axis('off')

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

- **Output:**



2.2 Understanding Kernel Effects

In image processing, a kernel (or filter) is a small matrix that is convolved with an image to extract or enhance certain visual features such as edges, textures, or sharpness. By modifying the kernel's values, we can control how the image is transformed — for example, emphasizing boundaries, smoothing noise, or highlighting fine details. The following code demonstrates how different kernels affect an image's appearance by applying several commonly used filters (sharpening, edge enhancement, and embossing) and visualizing their effects side-by-side for comparison.

Example 3: Understanding Kernel Effects

- **Code:**

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

def visualize_kernel_effect(image, kernels, titles):
    """
    Visualize the effect of multiple kernels on an image
    """
    n = len(kernels)
    fig, axes = plt.subplots(1, n+1, figsize=(4*(n+1), 4))

    # Show original
    axes[0].imshow(image, cmap='gray')
    axes[0].set_title('Original')
    axes[0].axis('off')

    # Show filtered results
    for i, (kernel, title) in enumerate(zip(kernels, titles)):
        filtered = cv2.filter2D(image, -1, kernel)
        axes[i+1].imshow(filtered, cmap='gray')
        axes[i+1].set_title(title)
        axes[i+1].axis('off')

    plt.tight_layout()
    plt.show()

# Example usage
image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

kernels = [
    np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]),  # Sharpen
    np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]),  # Edge enhance
    np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]]) / 4  # Emboss
]
```

```
titles = ['Sharpen', 'Edge Enhance', 'Emboss']
visualize_kernel_effect(image, kernels, titles)
```

- **Output:**

3. Correlation:

Correlation measures the **degree of similarity** between two images or between an image and its **transformed version**. It quantifies how closely one image matches another in terms of pixel intensity patterns. A high correlation value indicates that the images are very similar—meaning their features, brightness, and structure align well—while a low correlation suggests significant differences. In image processing, correlation is commonly used for tasks such as **template matching**, **image registration**, and **motion tracking**, where comparing spatial similarity is essential.

Example 4: Computing Correlation Between Two Images

- **Code:**

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

def compute_image_correlation(img1_path, img2_path):
    """
    Compute correlation coefficient between two images
    """
    # Read images
    I = cv2.imread(img1_path, cv2.IMREAD_GRAYSCALE)
    J = cv2.imread(img2_path, cv2.IMREAD_GRAYSCALE)

    if I is None or J is None:
        raise Exception("Failed to load one or both images")

    # Resize second image to match first
    if I.shape != J.shape:
        J = cv2.resize(J, (I.shape[1], I.shape[0]))
```

```
    # Compute correlation coefficient
    correlation = np.corrcoef(I.ravel(), J.ravel())[0, 1]

    # Display images and result
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(I, cmap='gray')
    axes[0].set_title('Image 1')
    axes[0].axis('off')

    axes[1].imshow(J, cmap='gray')
    axes[1].set_title('Image 2')
    axes[1].axis('off')

    plt.suptitle(f'Correlation Coefficient: {correlation:.4f}')
    plt.tight_layout()
    plt.show()

    return correlation

# Example usage
correlation = compute_image_correlation('picture1.png', 'picture2.png')
print(f"Correlation Coefficient: {correlation:.4f}")
```

- **Output:**

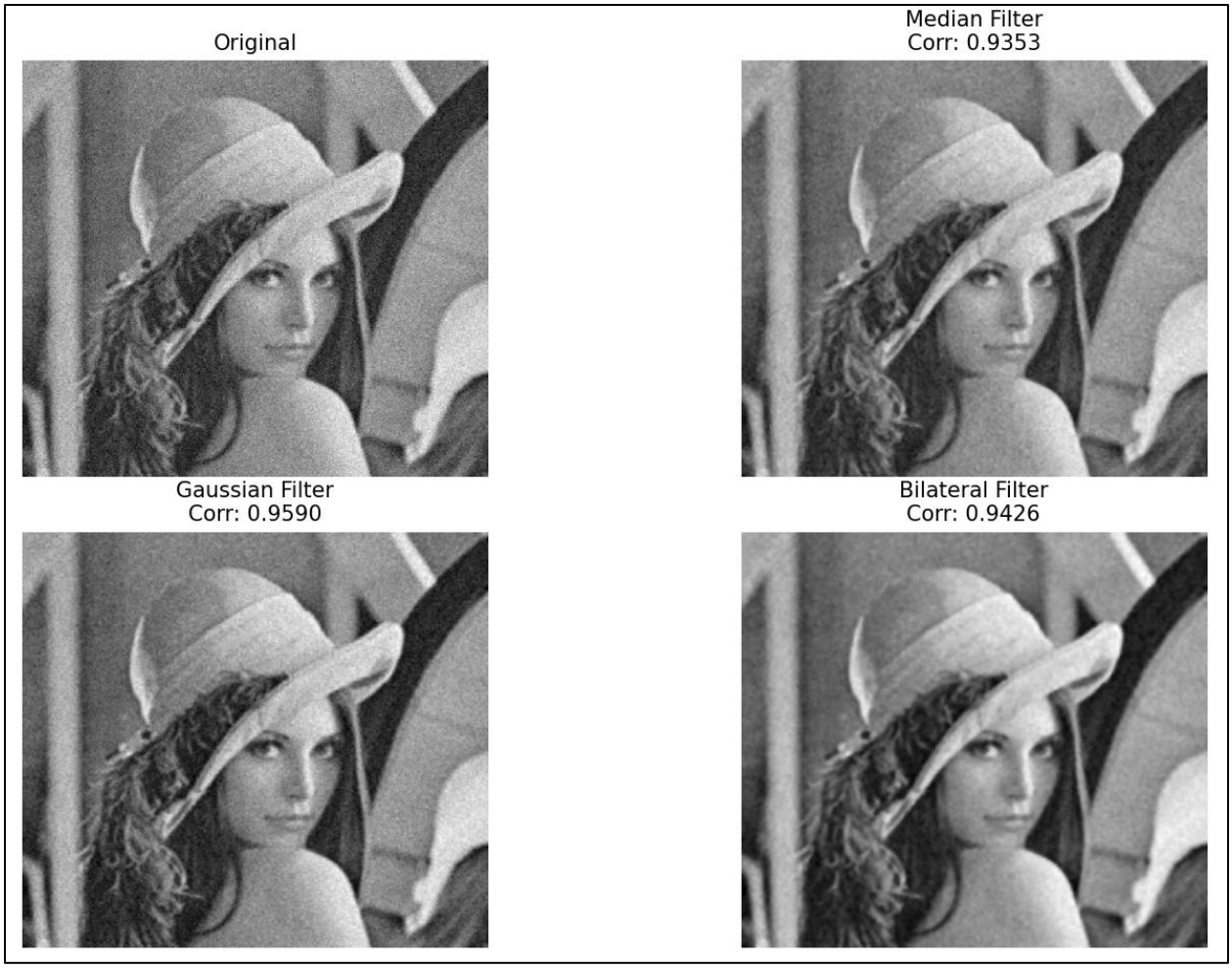
Example 5: Correlation with Filtered Images

- **Code:**

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

def correlation_with_filters(image_path):
    """
    Compute correlation between original image and various filtered
versions
    """
    # Read image
    I = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Apply different filters
```

```
    median_filtered = cv2.medianBlur(I, 5)
    gaussian_filtered = cv2.GaussianBlur(I, (5, 5), 1)
    bilateral_filtered = cv2.bilateralFilter(I, 9, 75, 75)

    # Compute correlations
    corr_median = np.corrcoef(I.ravel(), median_filtered.ravel())[0, 1]
    corr_gaussian = np.corrcoef(I.ravel(), gaussian_filtered.ravel())[0,
1]
    corr_bilateral = np.corrcoef(I.ravel(), bilateral_filtered.ravel())[0,
1]

    # Display results
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    axes[0, 0].imshow(I, cmap='gray')
    axes[0, 0].set_title('Original')
    axes[0, 0].axis('off')

    axes[0, 1].imshow(median_filtered, cmap='gray')
    axes[0, 1].set_title(f'Median Filter\nCorr: {corr_median:.4f}')
    axes[0, 1].axis('off')

    axes[1, 0].imshow(gaussian_filtered, cmap='gray')
    axes[1, 0].set_title(f'Gaussian Filter\nCorr: {corr_gaussian:.4f}')
    axes[1, 0].axis('off')

    axes[1, 1].imshow(bilateral_filtered, cmap='gray')
    axes[1, 1].set_title(f'Bilateral Filter\nCorr: {corr_bilateral:.4f}')
    axes[1, 1].axis('off')

    plt.tight_layout()
    plt.show()

    return corr_median, corr_gaussian, corr_bilateral

# Example usage
correlations = correlation_with_filters('input_image.jpg')
```

- **Output:**



4. Creating Custom Filters:

In image processing, **custom filters** allow us to design **specific convolution kernels** that produce desired **visual or analytical effects** beyond standard built-in options. By defining our own **filter matrices**, we can **simulate real-world phenomena**—such as **motion blur** or **focus shifts**—or **highlight particular image features** like **edges**, **textures**, and **contours**.
In the **examples below**, we **explore** how to build and apply **custom filters**, beginning with **motion blur simulation**, **Gaussian motion blur with boundary handling**, and a **gallery of commonly used custom kernels**.

4.1 Motion Blur Filter

A **motion blur filter** simulates the effect of **camera or object movement** during image capture. This effect commonly occurs when the **shutter speed is slow** and either the **camera shakes** or the **subject moves** rapidly. This helps simulate **directional movement** and can also be used artistically to **emphasize speed or motion** in an image.
Example 6: Motion Blur Filter

- **Code:**

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

def create_motion_blur_kernel(size, angle):
    """
    Create a motion blur kernel

    Args:
        size: Size of the kernel
        angle: Angle of motion in degrees
    """
    kernel = np.zeros((size, size))
    center = size // 2

    # Convert angle to radians
    angle_rad = np.deg2rad(angle)

    # Create line along the angle
    for i in range(size):
        offset = i - center
        x = int(center + offset * np.cos(angle_rad))
        y = int(center + offset * np.sin(angle_rad))

        if 0 <= x < size and 0 <= y < size:
            kernel[y, x] = 1

    # Normalize
    kernel /= np.sum(kernel)

    return kernel

# Load image
originalRGB = cv2.imread('peppers.png')
originalBGR = cv2.cvtColor(originalRGB, cv2.COLOR_BGR2RGB)

# Create motion blur kernels with different angles
angles = [0, 45, 90, 135]
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Original
axes[0, 0].imshow(originalBGR)
axes[0, 0].set_title('Original Image')
axes[0, 0].axis('off')
```

```
# Apply motion blur at different angles
for idx, angle in enumerate(angles):
    kernel = create_motion_blur_kernel(30, angle)
    blurred = cv2.filter2D(originalRGB, -1, kernel)
    blurred_rgb = cv2.cvtColor(blurred, cv2.COLOR_BGR2RGB)

    row = (idx + 1) // 3
    col = (idx + 1) % 3
    axes[row, col].imshow(blurred_rgb)
    axes[row, col].set_title(f'Motion Blur - {angle}°')
    axes[row, col].axis('off')

axes[1, 2].axis('off')

plt.tight_layout(pad=3.0)
plt.show()
```

- **Output:**

4.2 Gaussian Motion Blur with Boundary Handling

A **Gaussian motion blur** applies a **smooth, natural-looking blur** that follows a **Gaussian distribution**, giving softer transitions compared to a simple linear blur. **Boundary handling** ensures that edges of the image are processed correctly without creating unwanted dark borders or distortions.

Example 7: Gaussian Motion Blur with Boundary Handling

- **Code:**

```
import cv2
```

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

def apply_gaussian_motion_blur(image, kernel_size=50, sigma=10,
border_type=cv2.BORDER_REPLICATE):
    """
    Apply Gaussian motion blur with specified boundary handling
    """
    # Create Gaussian kernel
    kernel = cv2.getGaussianKernel(kernel_size, sigma)
    kernel = np.outer(kernel, kernel.transpose())

    # Apply filter with specified border type
    filtered = cv2.filter2D(image, -1, kernel, borderType=border_type)

    return filtered

# Load image
originalRGB = cv2.imread('peppers.png')

# Apply with different boundary options
border_types = {
    'Replicate': cv2.BORDER_REPLICATE,
    'Reflect': cv2.BORDER_REFLECT,
    'Wrap': cv2.BORDER_WRAP,
    'Constant': cv2.BORDER_CONSTANT
}

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

axes[0, 0].imshow(cv2.cvtColor(originalRGB, cv2.COLOR_BGR2RGB))
axes[0, 0].set_title('Original')
axes[0, 0].axis('off')

for idx, (name, border_type) in enumerate(border_types.items()):
    filtered = apply_gaussian_motion_blur(originalRGB,
border_type=border_type)
    filtered_rgb = cv2.cvtColor(filtered, cv2.COLOR_BGR2RGB)

    row = (idx + 1) // 3
    col = (idx + 1) % 3
    axes[row, col].imshow(filtered_rgb)
    axes[row, col].set_title(f'Boundary: {name}')
    axes[row, col].axis('off')
```

```
# Hide unused subplot
axes[1, 2].axis('off')
plt.tight_layout(pad=3.0)
plt.show()
```

- **Output:**

4.3 Custom Kernel Gallery

A **custom kernel gallery** showcases various **user-defined convolution filters** that can be applied to achieve specific image-processing effects. By adjusting the values within a kernel matrix, we can enhance or modify features such as **edges**, **sharpening**, **embossing**, **smoothing**, or **edge detection**.

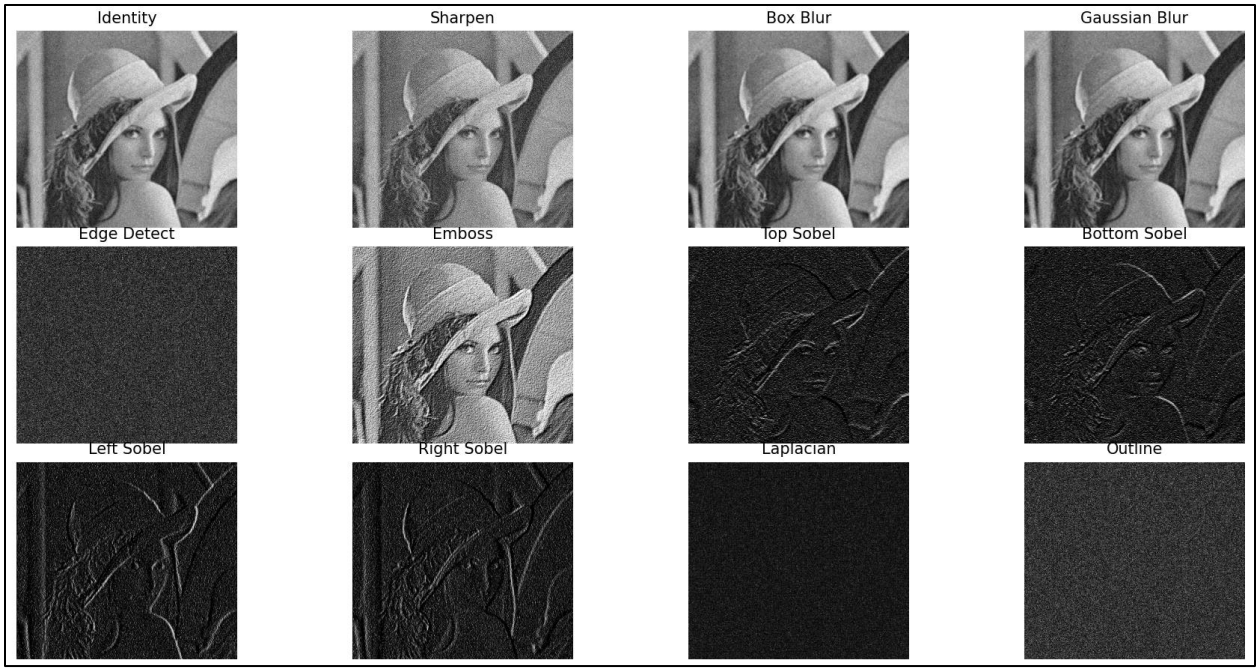
Example 8: Custom Kernel Gallery

- **Code:**

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

def create_kernel_gallery():
    """
    Create and display a gallery of common kernels
    """
    kernels = {
        'Identity': np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]]),
        'Sharpen': np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]),
        'Box Blur': np.ones((3, 3)) / 9,
```

```
        'Gaussian Blur': np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) / 16,
        'Edge Detect': np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -
1]]),
        'Emboss': np.array([[-2, -1, 0], [-1, 1, 1], [0, 1, 2]]),
        'Top Sobel': np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]]),
        'Bottom Sobel': np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]]),
        'Left Sobel': np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]]),
        'Right Sobel': np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]),
        'Laplacian': np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]]),
        'Outline': np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    }

    # Load test image
    image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

    # Create subplot grid
    n_kernels = len(kernels)
    n_cols = 4
    n_rows = (n_kernels + n_cols - 1) // n_cols

    fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, 4*n_rows))
    axes = axes.flatten()

    for idx, (name, kernel) in enumerate(kernels.items()):
        filtered = cv2.filter2D(image, -1, kernel)
        axes[idx].imshow(filtered, cmap='gray')
        axes[idx].set_title(name)
        axes[idx].axis('off')

    # Hide unused subplots
    for idx in range(len(kernels), len(axes)):
        axes[idx].axis('off')


    plt.tight_layout(pad=3.0)
    plt.show()

create_kernel_gallery()
```

- **Output:**



5. Noise Reduction Techniques:

Digital images often suffer from various types of noise introduced during acquisition, transmission, or processing. Noise reduction techniques aim to minimize these unwanted variations while preserving important image details such as edges and textures. In this section, we explore common noise types and filtering methods used to restore image quality.
Section 5.1 introduces Salt and Pepper Noise, a common type of impulsive noise, while 5.2 compares different noise reduction filters and their effectiveness in smoothing noisy images.

5.1 Salt and Pepper Noise

Salt and pepper noise (impulse noise) appears as randomly occurring white and black pixels.

Example 9: Salt and Pepper Noise

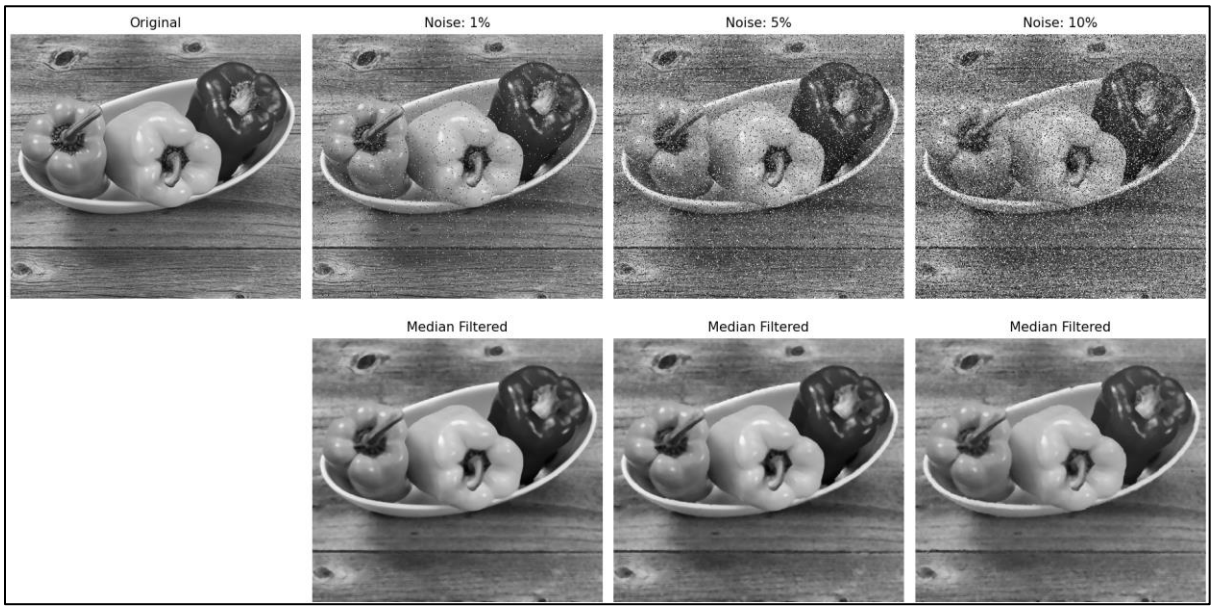
- **Code:**

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

def add_salt_pepper_noise(image, salt_prob=0.01, pepper_prob=0.01):
    """
    Add salt and pepper noise to an image
    """
    noisy = image.copy()
    height, width = image.shape[:2]

    # Add salt (white pixels)
```

```
    n_salt = int(height * width * salt_prob)
    salt_coords = [np.random.randint(0, i, n_salt) for i in (height,
width)]
    noisy[salt_coords[0], salt_coords[1]] = 255

    # Add pepper (black pixels)
    n_pepper = int(height * width * pepper_prob)
    pepper_coords = [np.random.randint(0, i, n_pepper) for i in (height,
width)]
    noisy[pepper_coords[0], pepper_coords[1]] = 0

    return noisy

# Load image
I = cv2.imread('peppers.png', cv2.IMREAD_GRAYSCALE)

# Add different levels of noise
noise_levels = [0.01, 0.05, 0.10]

fig, axes = plt.subplots(2, len(noise_levels)+1, figsize=(16, 8))

# Original
axes[0, 0].imshow(I, cmap='gray')
axes[0, 0].set_title('Original')
axes[0, 0].axis('off')
axes[1, 0].axis('off')

for idx, noise_level in enumerate(noise_levels):
    # Add noise
    noisy = add_salt_pepper_noise(I, noise_level, noise_level)

    # Apply median filter
    filtered = cv2.medianBlur(noisy, 5)

    # Display noisy
    axes[0, idx+1].imshow(noisy, cmap='gray')
    axes[0, idx+1].set_title(f'Noise: {noise_level*100:.0f}%')
    axes[0, idx+1].axis('off')

    # Display filtered
    axes[1, idx+1].imshow(filtered, cmap='gray')
    axes[1, idx+1].set_title('Median Filtered')
    axes[1, idx+1].axis('off')

plt.tight_layout()
```

```
plt.show()
```

- **Output:**



5.2 Comparing Noise Reduction Filters

Different noise reduction filters vary in how effectively they remove noise while preserving image details. This section compares several filtering techniques to highlight their strengths and trade-offs in image restoration.

Example 10: Comparing Noise Reduction Filters

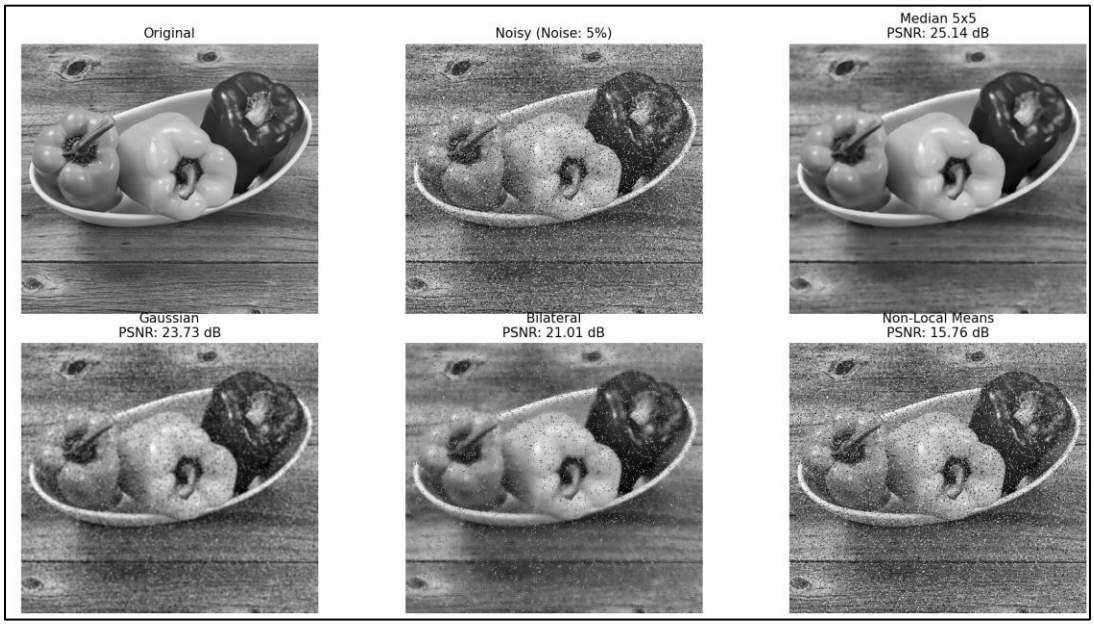
- **Code:**

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

def add_salt_pepper_noise(image, salt_prob=0.01, pepper_prob=0.01):
    """
    Add salt and pepper noise to an image
    """
    noisy = image.copy()
    height, width = image.shape[:2]

    # Add salt (white pixels)
    n_salt = int(height * width * salt_prob)
    salt_coords = [np.random.randint(0, i, n_salt) for i in (height,
width)]
    noisy[salt_coords[0], salt_coords[1]] = 255
```

```
    # Add pepper (black pixels)
    n_pepper = int(height * width * pepper_prob)
    pepper_coords = [np.random.randint(0, i, n_pepper) for i in (height,
width)]
    noisy[pepper_coords[0], pepper_coords[1]] = 0

    return noisy

def compare_noise_reduction_filters(image, noise_level=0.05):
    """
    Compare different noise reduction techniques
    """
    # Add noise
    noisy = add_salt_pepper_noise(image, noise_level, noise_level)

    # Apply different filters
    median_5 = cv2.medianBlur(noisy, 5)
    median_7 = cv2.medianBlur(noisy, 7)
    gaussian = cv2.GaussianBlur(noisy, (5, 5), 0)
    bilateral = cv2.bilateralFilter(noisy, 9, 75, 75)
    nlm = cv2.fastNlMeansDenoising(noisy, None, 10, 7, 21)

    # Calculate PSNR (Peak Signal-to-Noise Ratio)
    def calculate_psnr(original, filtered):
        mse = np.mean((original.astype(float) - filtered.astype(float)) **
2)
        if mse == 0:
            return float('inf')
        max_pixel = 255.0
        psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
        return psnr

    psnr_median_5 = calculate_psnr(image, median_5)
    psnr_median_7 = calculate_psnr(image, median_7)
    psnr_gaussian = calculate_psnr(image, gaussian)
    psnr_bilateral = calculate_psnr(image, bilateral)
    psnr_nlm = calculate_psnr(image, nlm)

    # Display results
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    axes[0, 0].imshow(image, cmap='gray')
    axes[0, 0].set_title('Original')
    axes[0, 0].axis('off')
```

```
    axes[0, 1].imshow(noisy, cmap='gray')
    axes[0, 1].set_title(f'Noisy (Noise: {noise_level*100:.0f}%)')
    axes[0, 1].axis('off')

    axes[0, 2].imshow(median_5, cmap='gray')
    axes[0, 2].set_title(f'Median 5x5\nPSNR: {psnr_median_5:.2f} dB')
    axes[0, 2].axis('off')

    axes[1, 0].imshow(gaussian, cmap='gray')
    axes[1, 0].set_title(f'Gaussian\nPSNR: {psnr_gaussian:.2f} dB')
    axes[1, 0].axis('off')

    axes[1, 1].imshow(bilateral, cmap='gray')
    axes[1, 1].set_title(f'Bilateral\nPSNR: {psnr_bilateral:.2f} dB')
    axes[1, 1].axis('off')

    axes[1, 2].imshow(nlm, cmap='gray')
    axes[1, 2].set_title(f'Non-Local Means\nPSNR: {psnr_nlm:.2f} dB')
    axes[1, 2].axis('off')


    plt.tight_layout(pad=3.0)
    plt.show()

# Example usage
I = cv2.imread('peppers.png', cv2.IMREAD_GRAYSCALE)
compare_noise_reduction_filters(I, noise_level=0.05)
```

- **Output:**



6. Image Smoothing:

Image smoothing is a fundamental technique used to reduce noise and minor variations in pixel intensity, producing a cleaner and more visually appealing image. By averaging neighboring pixel values, smoothing filters help to blur fine details and soften edges. This section explores different smoothing methods, including **Gaussian smoothing** with varying parameters and a comparison between **Box Blur** and **Gaussian Blur** techniques.

6.1 Gaussian Smoothing with Different Parameters

Gaussian smoothing applies a weighted averaging filter based on a Gaussian function to reduce image noise and detail. Changing parameters such as kernel size and sigma value affects the level of blur and smoothness in the resulting image.

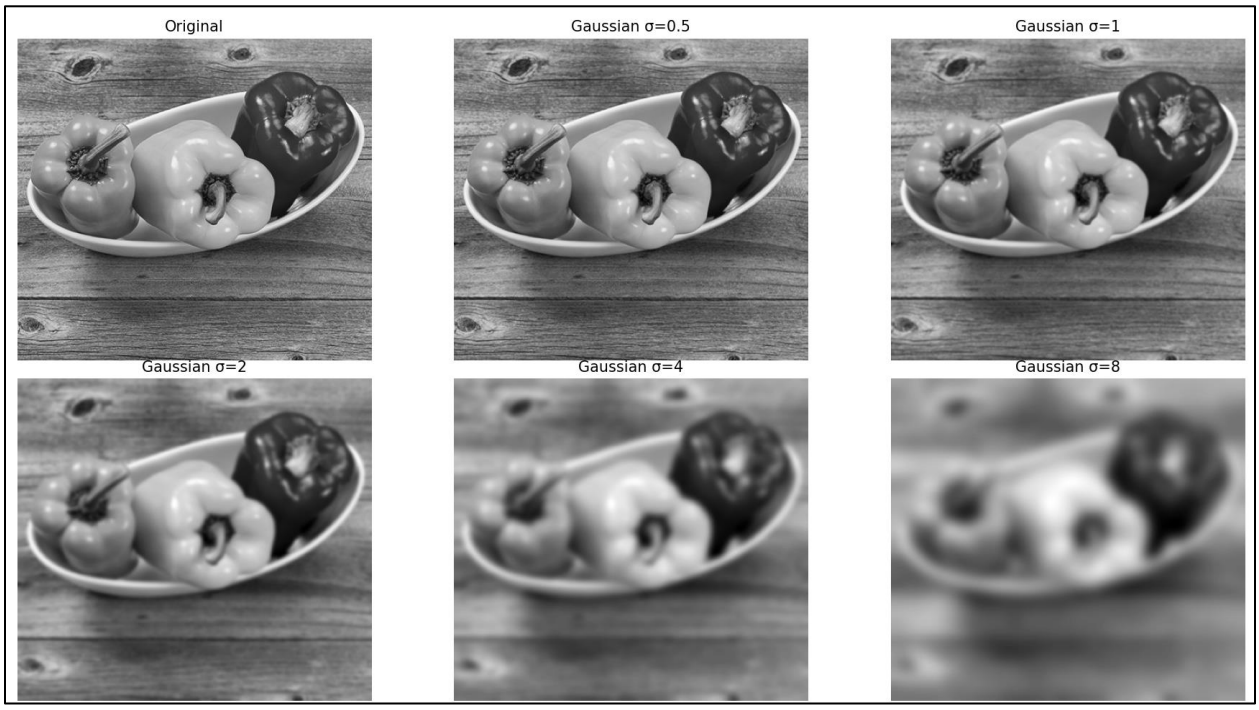
Example 11: Gaussian Smoothing with Different Parameters

- **Code:**

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

def gaussian_smoothing_comparison(image_path):
    """
    Compare Gaussian smoothing with different parameters
    """
    # Read image
    I = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Different sigma values
    sigmas = [0.5, 1, 2, 4, 8]

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Original
    axes[0, 0].imshow(I, cmap='gray')
    axes[0, 0].set_title('Original')
    axes[0, 0].axis('off')

    # Apply Gaussian blur with different sigmas
    for idx, sigma in enumerate(sigmas):
        blurred = cv2.GaussianBlur(I, (0, 0), sigma)

        row = (idx + 1) // 3
        col = (idx + 1) % 3
        axes[row, col].imshow(blurred, cmap='gray')
        axes[row, col].set_title(f'Gaussian σ={sigma}')
        axes[row, col].axis('off')

    plt.tight_layout(pad=3.0)
```

```
    plt.show()

gaussian_smoothing_comparison('peppers.png')
```

- **Output:**



6.2 Box Blur vs Gaussian Blur

Box Blur applies uniform averaging over neighboring pixels, while Gaussian Blur uses a weighted approach that gives more importance to central pixels. This section compares their effects on image clarity and smoothness.

Example 12: Box Blur vs Gaussian Blur

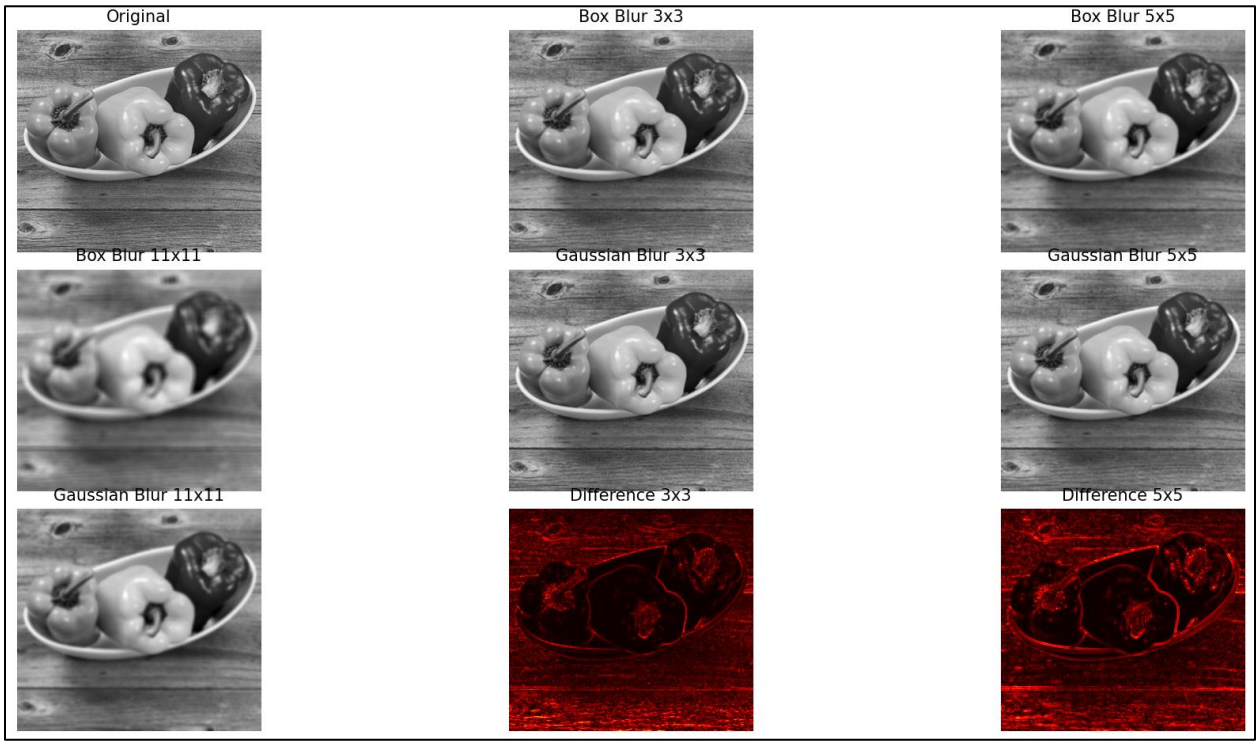
- **Code:**

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

def compare_blur_methods(image_path):
    """
    Compare different blurring methods
    """
    I = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Different blur methods
    box_blur_3 = cv2.blur(I, (3, 3))
    box_blur_5 = cv2.blur(I, (5, 5))
```

```
box_blur_11 = cv2.blur(I, (11, 11))

gaussian_blur_3 = cv2.GaussianBlur(I, (3, 3), 0)
gaussian_blur_5 = cv2.GaussianBlur(I, (5, 5), 0)
gaussian_blur_11 = cv2.GaussianBlur(I, (11, 11), 0)

# Display results
fig, axes = plt.subplots(3, 3, figsize=(15, 15))

axes[0, 0].imshow(I, cmap='gray')
axes[0, 0].set_title('Original')
axes[0, 0].axis('off')

# Box blur
axes[0, 1].imshow(box_blur_3, cmap='gray')
axes[0, 1].set_title('Box Blur 3x3')
axes[0, 1].axis('off')

axes[0, 2].imshow(box_blur_5, cmap='gray')
axes[0, 2].set_title('Box Blur 5x5')
axes[0, 2].axis('off')

axes[1, 0].imshow(box_blur_11, cmap='gray')
axes[1, 0].set_title('Box Blur 11x11')
axes[1, 0].axis('off')

# Gaussian blur
axes[1, 1].imshow(gaussian_blur_3, cmap='gray')
axes[1, 1].set_title('Gaussian Blur 3x3')
axes[1, 1].axis('off')

axes[1, 2].imshow(gaussian_blur_5, cmap='gray')
axes[1, 2].set_title('Gaussian Blur 5x5')
axes[1, 2].axis('off')

axes[2, 0].imshow(gaussian_blur_11, cmap='gray')
axes[2, 0].set_title('Gaussian Blur 11x11')
axes[2, 0].axis('off')

# Difference images
diff_3 = cv2.absdiff(box_blur_3, gaussian_blur_3)
diff_5 = cv2.absdiff(box_blur_5, gaussian_blur_5)

axes[2, 1].imshow(diff_3, cmap='hot')
axes[2, 1].set_title('Difference 3x3')
```

```
    axes[2, 1].axis('off')

    axes[2, 2].imshow(diff_5, cmap='hot')
    axes[2, 2].set_title('Difference 5x5')
    axes[2, 2].axis('off')

    plt.tight_layout(pad=3.0)
    plt.show()

compare_blur_methods('peppers.png')
```

- **Output:**



7. Frequency Domain Filtering:

Frequency domain filtering analyzes, and processes images based on their frequency components rather than pixel intensities. By transforming an image into the frequency domain, we can selectively enhance or suppress specific patterns, edges, or noise. This section covers techniques such as **wavelet transforms**, **Fourier filtering**, and **high-pass filtering**, including both standard implementations and custom approaches for advanced image processing.

7.1 Discrete Wavelet Transform with PyWavelets

The Discrete Wavelet Transform (DWT) decomposes an image into different frequency subbands, allowing multi-resolution analysis and efficient feature extraction.

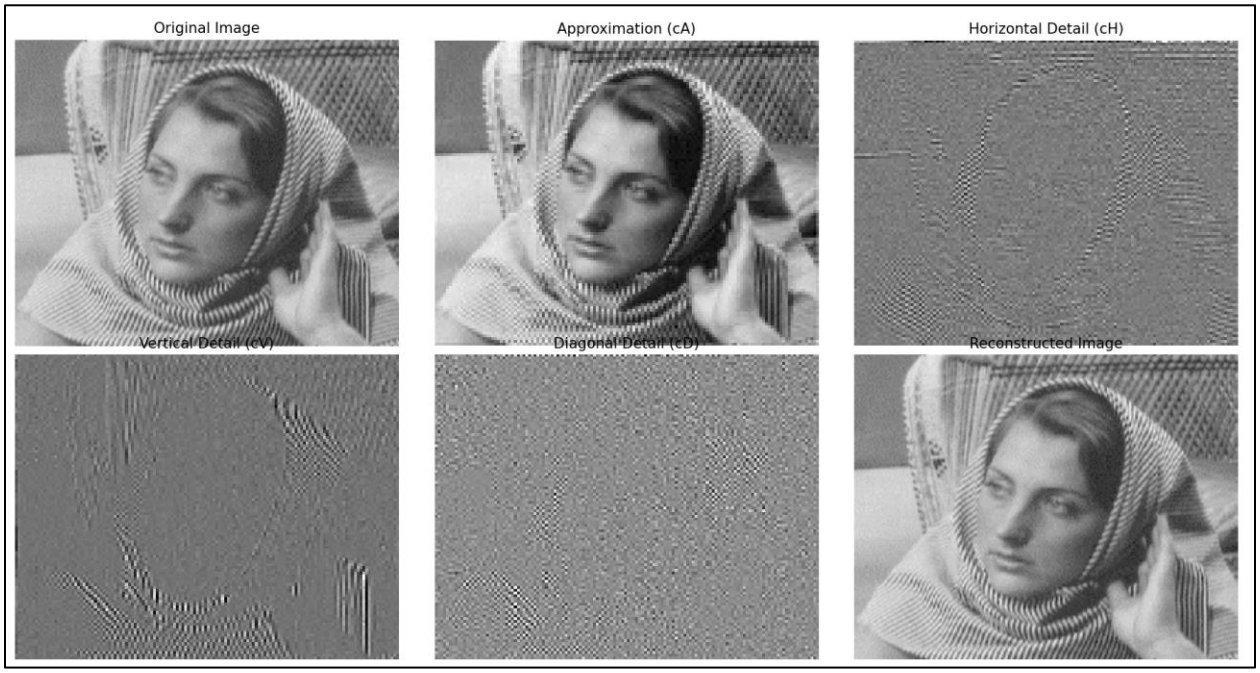
Example 13: Discrete Wavelet Transform with PyWavelets

- **Code:**

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

def wavelet_decomposition_analysis(image_path):
    """
    Perform and visualize wavelet decomposition
    """
    # Load image
    X = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Perform 2D DWT
    coeffs = pywt.dwt2(X, 'sym4', mode='per')
    cA, (cH, cV, cD) = coeffs

    # Display results
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    axes[0, 0].imshow(X, cmap='gray')
    axes[0, 0].set_title('Original Image')
    axes[0, 0].axis('off')

    axes[0, 1].imshow(cA, cmap='gray')
    axes[0, 1].set_title('Approximation (cA)')
    axes[0, 1].axis('off')

    axes[0, 2].imshow(cH, cmap='gray')
    axes[0, 2].set_title('Horizontal Detail (cH)')
    axes[0, 2].axis('off')

    axes[1, 0].imshow(cV, cmap='gray')
    axes[1, 0].set_title('Vertical Detail (cV)')
    axes[1, 0].axis('off')

    axes[1, 1].imshow(cD, cmap='gray')
    axes[1, 1].set_title('Diagonal Detail (cD)')
    axes[1, 1].axis('off')

    # Reconstruct image
    reconstructed = pywt.idwt2(coeffs, 'sym4', mode='per')
    axes[1, 2].imshow(reconstructed, cmap='gray')
    axes[1, 2].set_title('Reconstructed Image')
    axes[1, 2].set_title('Reconstructed Image')
    axes[1, 2].axis('off')
```

```
    plt.tight_layout(pad=3.0)
    plt.show()

    return coeffs

# Example usage
coeffs = wavelet_decomposition_analysis('woman.png')
```

- **Output:**



7.2 Multi-Level Wavelet Decomposition

Multi-level decomposition applies DWT iteratively to capture image details at multiple scales, enhancing both coarse and fine structures.

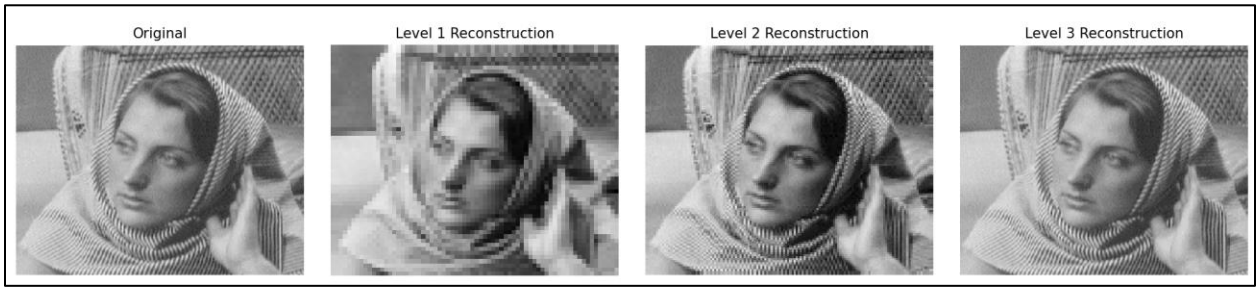
Example 14: Multi-Level Wavelet Decomposition

- **Code:**

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

def multilevel_wavelet_decomposition(image_path, levels=3):
    """
    Perform multi-level wavelet decomposition
    """
```

```
    # Load image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Perform multi-level decomposition
    coeffs = pywt.wavedec2(image, 'db2', level=levels)

    # Create visualization
    fig, axes = plt.subplots(1, levels+1, figsize=(5*(levels+1), 5))

    # Original image
    axes[0].imshow(image, cmap='gray')
    axes[0].set_title('Original')
    axes[0].axis('off')

    # Each level of approximation
    for i in range(1, levels+1):
        # Reconstruct from level i
        reconstructed = pywt.waverec2(coeffs[:i+1], 'db2')
        # Crop to original size
        reconstructed = reconstructed[:image.shape[0], :image.shape[1]]

        axes[i].imshow(reconstructed, cmap='gray')
        axes[i].set_title(f'Level {i} Reconstruction')
        axes[i].axis('off')

    plt.tight_layout(pad=3.0)
    plt.show()

    return coeffs

# Example usage
coeffs = multilevel_wavelet_decomposition('woman.png', levels=3)
```

- **Output:**



7.3 Fourier Transform Filtering

Fourier Transform converts an image into its frequency components, enabling filtering based on spatial frequencies to remove noise or enhance patterns.

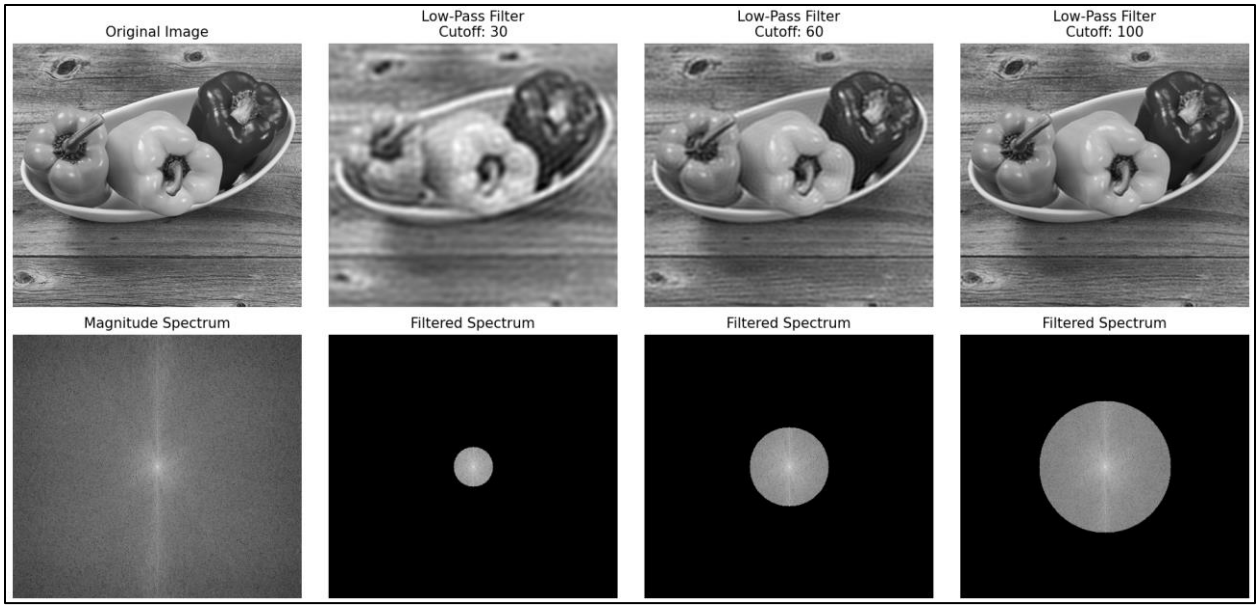
Example 15: Fourier Transform Filtering

- **Code:**

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

def fourier_transform_filtering(image_path):
    """
    Demonstrate frequency domain filtering using Fourier Transform
    """
    # Load image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Perform Fourier Transform
    f_transform = np.fft.fft2(image)
    f_shift = np.fft.fftshift(f_transform)

    # Compute magnitude spectrum
    magnitude_spectrum = 20 * np.log(np.abs(f_shift) + 1)

    # Create low-pass filter
    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2

    # Different cutoff frequencies
    cutoffs = [30, 60, 100]

    fig, axes = plt.subplots(2, len(cutoffs)+1, figsize=(20, 10))

    # Original image
    axes[0, 0].imshow(image, cmap='gray')
    axes[0, 0].set_title('Original Image')
    axes[0, 0].axis('off')

    # Magnitude spectrum
    axes[1, 0].imshow(magnitude_spectrum, cmap='gray')
    axes[1, 0].set_title('Magnitude Spectrum')
    axes[1, 0].axis('off')
```

```
    for idx, cutoff in enumerate(cutoffs):
        # Create mask
        mask = np.zeros((rows, cols), np.uint8)
        cv2.circle(mask, (ccol, crow), cutoff, 1, -1)

        # Apply mask
        f_shift_filtered = f_shift * mask

        # Inverse Fourier Transform
        f_ishift = np.fft.ifftshift(f_shift_filtered)
        image_filtered = np.fft.ifft2(f_ishift)
        image_filtered = np.abs(image_filtered)

        # Display filtered image
        axes[0, idx+1].imshow(image_filtered, cmap='gray')
        axes[0, idx+1].set_title(f'Low-Pass Filter\nCutoff: {cutoff}')
        axes[0, idx+1].axis('off')

        # Display filtered spectrum
        magnitude_filtered = 20 * np.log(np.abs(f_shift_filtered) + 1)
        axes[1, idx+1].imshow(magnitude_filtered, cmap='gray')
        axes[1, idx+1].set_title(f'Filtered Spectrum')
        axes[1, idx+1].axis('off')

    plt.tight_layout(pad=3.0)
    plt.show()

# Example usage
fourier_transform_filtering('peppers.png')
```

- **Output:**



7.4 High-Pass Filtering in Frequency Domain

High-pass filters allow high-frequency components, such as edges and fine details, to pass while attenuating low-frequency information like smooth areas.

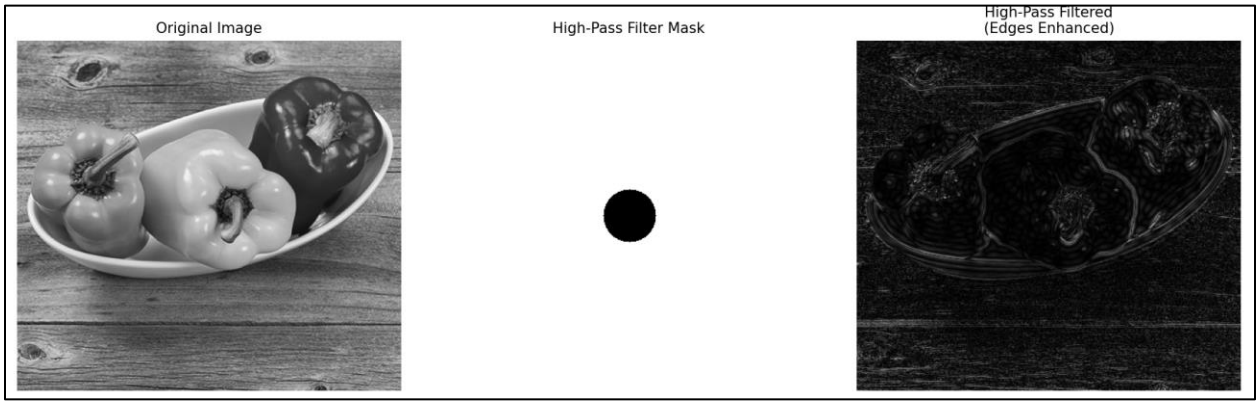
Example 16: High-Pass Filtering in Frequency Domain

- **Code:**

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

def high_pass_filter_demo(image_path):
    """
    Demonstrate high-pass filtering for edge enhancement
    """
    # Load image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Fourier Transform
    f_transform = np.fft.fft2(image)
    f_shift = np.fft.fftshift(f_transform)

    rows, cols = image.shape
    crow, ccol = rows // 2, cols // 2

    # Create high-pass filter (inverse of low-pass)
    cutoff = 30
    mask = np.ones((rows, cols), np.uint8)
```

```
    cv2.circle(mask, (ccol, crow), cutoff, 0, -1)

    # Apply filter
    f_shift_filtered = f_shift * mask

    # Inverse transform
    f_ishift = np.fft.ifftshift(f_shift_filtered)
    image_filtered = np.fft.ifft2(f_ishift)
    image_filtered = np.abs(image_filtered)

    # Display results
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    axes[0].imshow(image, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')

    axes[1].imshow(mask * 255, cmap='gray')
    axes[1].set_title('High-Pass Filter Mask')
    axes[1].axis('off')

    axes[2].imshow(image_filtered, cmap='gray')
    axes[2].set_title('High-Pass Filtered\n(Edges Enhanced)')
    axes[2].axis('off')

    plt.tight_layout(pad=3.0)
    plt.show()

# Example usage
high_pass_filter_demo('peppers.png')
```

- **Output:**



7.5 Custom Haar Wavelet Implementation

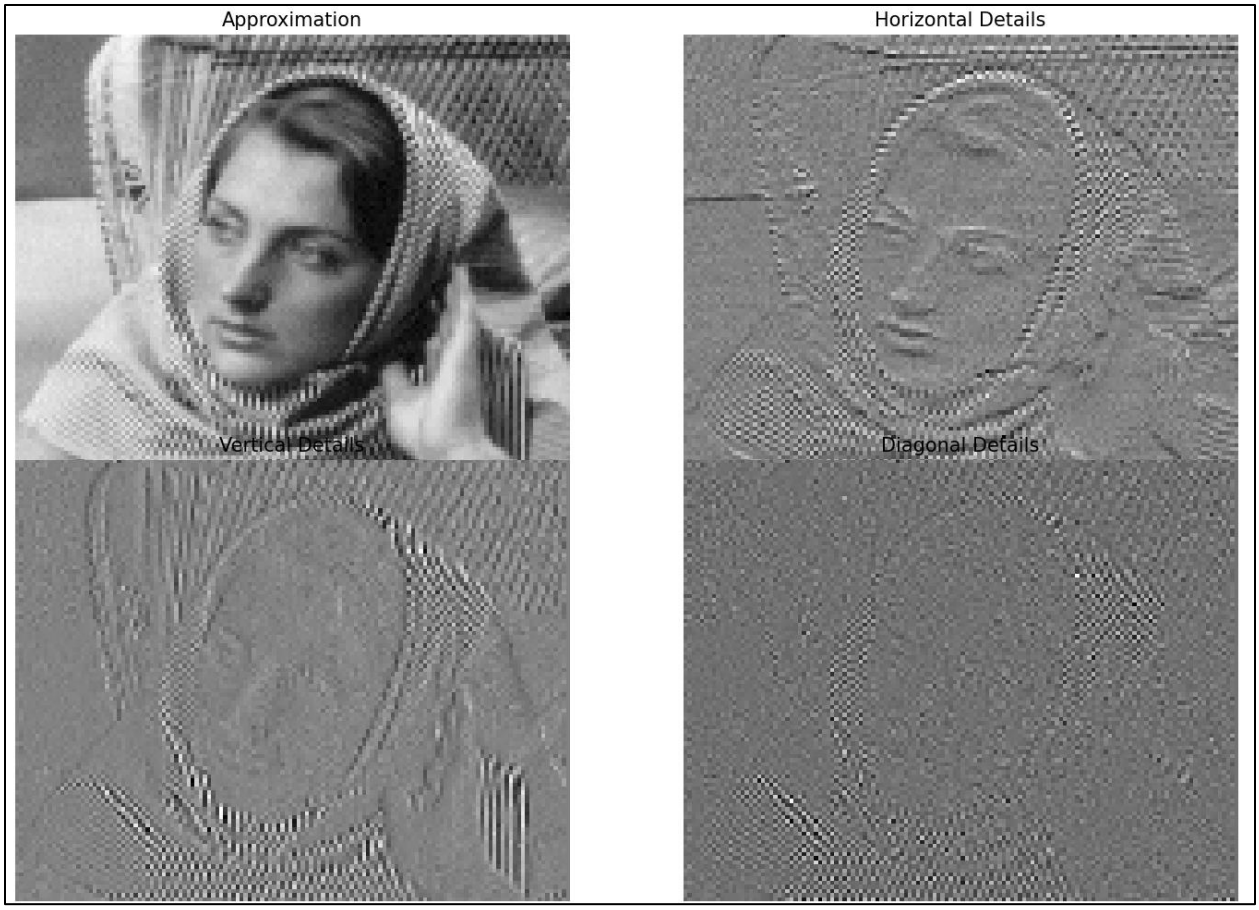
A custom Haar wavelet implementation demonstrates the fundamentals of wavelet-based filtering, showing how basic wavelet operations can be applied to image processing.

Example 17: Custom Haar Wavelet Implementation

- **Code:**

```
import cv2
import pywt
import matplotlib.pyplot as plt

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

# Perform single-level Haar wavelet decomposition
coeffs2 = pywt.dwt2(image, 'haar')
approximation, (horizontal, vertical, diagonal) = coeffs2

# Display
fig, axes = plt.subplots(2, 2, figsize=(12, 12))

axes[0, 0].imshow(approximation, cmap='gray')
axes[0, 0].set_title('Approximation')
axes[0, 0].axis('off')

axes[0, 1].imshow(horizontal, cmap='gray')
axes[0, 1].set_title('Horizontal Details')
axes[0, 1].axis('off')

axes[1, 0].imshow(vertical, cmap='gray')
axes[1, 0].set_title('Vertical Details')
axes[1, 0].axis('off')

axes[1, 1].imshow(diagonal, cmap='gray')
axes[1, 1].set_title('Diagonal Details')
axes[1, 1].axis('off')

plt.tight_layout(pad=3.0)
plt.show()
```

- **Output:**



8. Practice Tasks:

Task 1: Edge Detection Comparison

Objective: **The objective of this task is to understand and compare different edge detection operators by applying them to the same image and analyzing their effectiveness in detecting edges at various orientations and strengths.**

Steps:

1. **Load a grayscale image of your choice (preferably one with clear edges and structures).**
2. **Implement edge detection using three different operators:**
    - **Sobel operator (horizontal and vertical)**
    - **Prewitt operator (horizontal and vertical)**
    - **Scharr operator (horizontal and vertical)**
3. **For each operator, compute the magnitude of edges by combining horizontal and vertical components using: magnitude = sqrt(horizontal² + vertical²)**
4. **Display all results in a grid layout showing:**

- **Original image**
- **Sobel edge detection result**
- **Prewitt edge detection result**
- **Scharr edge detection result**

5. **Compare the three operators and observe which one detects edges more accurately and produces stronger responses.**

Task 2: Custom Kernel Design and Application

Objective: The objective of this task is to design and apply custom convolution kernels to achieve specific image processing effects, helping you understand how kernel values influence the output image characteristics.

Steps:

1. **Load a grayscale or color image of your choice.**
2. **Create at least four custom kernels with different effects:**
   - **A sharpening kernel (e.g., [[0, -1, 0], [-1, 5, -1], [0, -1, 0]])**
   - **An edge enhancement kernel**
   - **An emboss kernel (e.g., [[-2, -1, 0], [-1, 1, 1], [0, 1, 2]])**
   - **A custom blur kernel of your own design**
3. **Apply each kernel to your image using cv2.filter2D().**
4. **Display the original image alongside all filtered versions in a grid layout with appropriate titles.**
5. **Experiment by modifying kernel values and observe how changes affect the output. Document your observations.**

Task 3: Noise Addition and Reduction Analysis

Objective: The objective of this task is to understand different types of image noise and evaluate the effectiveness of various noise reduction filters in restoring image quality while preserving important details.

Steps:

1. **Load a clean grayscale image (e.g., 'peppers.png' or any image of your choice).**
2. **Add salt-and-pepper noise to the image at three different noise levels: 1%, 5%, and 10%.**
3. **Apply the following noise reduction filters to each noisy image:**
   - **Median filter (5x5)**
   - **Gaussian filter (5x5)**

- Bilateral filter
- Non-Local Means denoising (optional)

4. Calculate the Peak Signal-to-Noise Ratio (PSNR) between the original clean image and each filtered result to quantitatively measure restoration quality.
5. Display a comprehensive comparison showing:
    - Original clean image
    - Noisy images at different levels
    - All filtered results with their PSNR values
6. Analyze which filter performs best for different noise levels and explain why.

Task 4: Motion Blur Simulation and Analysis

Objective: The objective of this task is to simulate realistic motion blur effects at different angles and understand how directional kernels affect image appearance, which is useful for understanding camera motion and image restoration techniques.

Steps:

1. Load a color image (preferably with distinct objects or patterns).
2. Implement a function to create motion blur kernels at different angles (0°, 45°, 90°, 135°) with a kernel size of at least 20x20.
3. Apply each motion blur kernel to your image using cv2.filter2D().
4. Create a visualization showing:
    - The original image
    - Motion blurred versions at each angle
    - The kernel used for each blur (visualized as a small image)
5. Experiment with different kernel sizes (e.g., 15, 30, 50) and observe how the blur intensity changes.
6. Try combining horizontal and vertical motion blur kernels to create diagonal motion effects.

Task 5: Frequency Domain Filtering with Fourier Transform

Objective: The objective of this task is to understand frequency domain image processing by applying low-pass and high-pass filters in the Fourier domain, enabling selective enhancement or suppression of image frequencies for noise removal or edge enhancement.

Steps:

1. Load a grayscale image of your choice.

2. **Perform a 2D Fourier Transform on the image using np.fft.fft2() and shift the zero-frequency component to the center using np.fft.fftshift().**
3. **Compute and display the magnitude spectrum using: magnitude_spectrum = 20 * np.log(np.abs(f_shift) + 1)**
4. **Create circular masks for both low-pass and high-pass filtering:**
    - **Low-pass: Create a circular mask that keeps only the center frequencies (try cutoff radii of 30, 60, and 100 pixels)**
    - **High-pass: Create a mask that removes center frequencies (inverse of low-pass)**
5. **Apply each mask to the shifted Fourier transform, then perform inverse FFT to obtain filtered images.**
6. **Display a comprehensive comparison showing:**
    - **Original image and its magnitude spectrum**
    - **Low-pass filtered images at different cutoffs**
    - **High-pass filtered image**
    - **The corresponding filtered magnitude spectra**
7. **Analyze how different cutoff frequencies affect the image smoothness and edge preservation.**