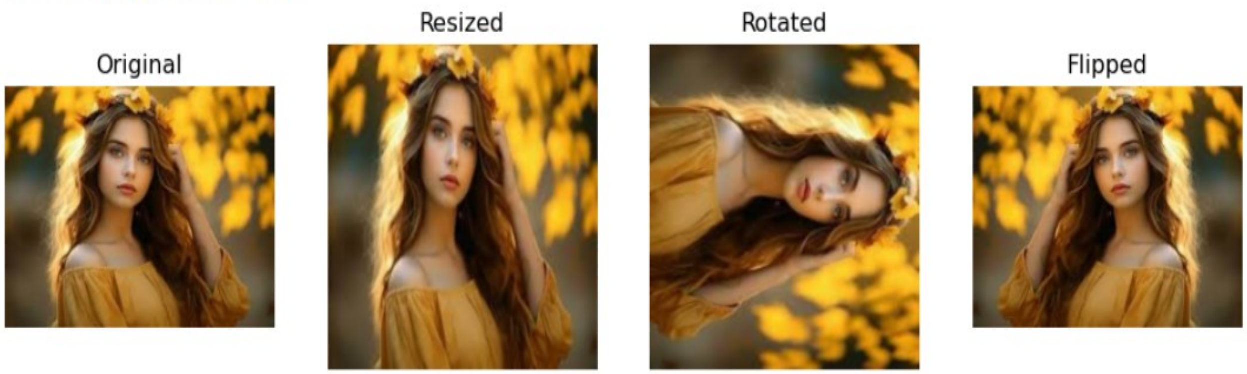
Image Processing Lab List

1. Image Representation and Basic Operations Read, display, save images, and perform pixel access, resizing, rotating, and simple transformations.

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

# Read image
img = cv2.imread('D:/images.jpeg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Access a pixel
px_value = img[50, 50]
print("Pixel at (50,50):", px_value)

# Resize and Rotate
resized = cv2.resize(img, (300, 300))
rotated = cv2.rotate(resized, cv2.ROTATE_90_CLOCKWISE)

# Simple Transformation (Flipping)
flipped = cv2.flip(img, 1)  # Flip horizontally

# Save images
cv2.imwrite('D:/resized.jpg', cv2.cvtColor(resized, cv2.COLOR_RGB2BGR))
cv2.imwrite('D:/rotated.jpg', cv2.cvtColor(rotated, cv2.COLOR_RGB2BGR))
cv2.imwrite('D:/flipped.jpg', cv2.cvtColor(flipped, cv2.COLOR_RGB2BGR))

# Display images using Matplotlib
fig, axes = plt.subplots(1, 4, figsize=(12, 5))
axes[0].imshow(img)
axes[0].set_title("Original")
axes[0].axis("off")

axes[1].imshow(resized)
axes[1].set_title("Resized")
axes[1].axis("off")

axes[2].imshow(rotated)
```

```
axes[2].set_title("Rotated")
axes[2].axis("off")

axes[3].imshow(flipped)
axes[3].set_title("Flipped")
axes[3].axis("off")
plt.show()
```

```
Pixel at (50,50): [104  85  45]
```



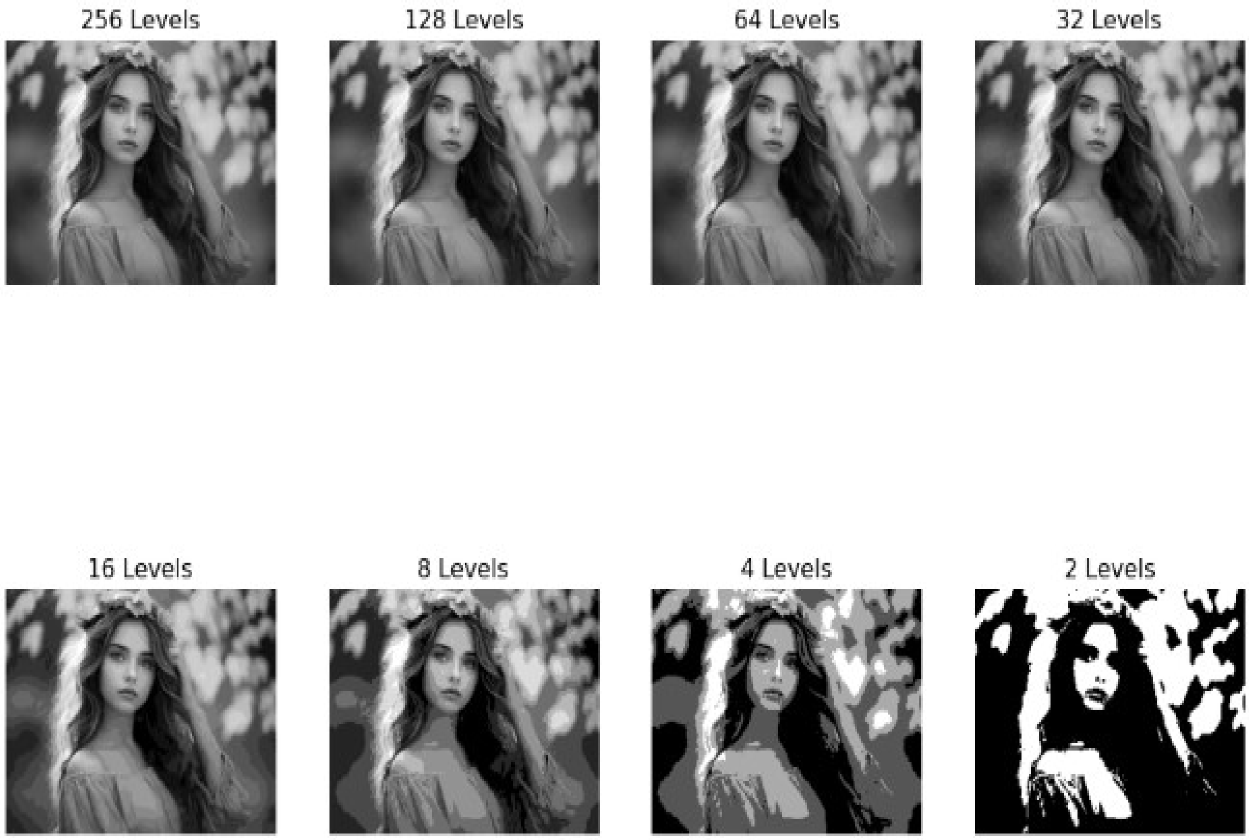
2. Gray-Level Transformation Apply basic gray-level transformations (contrast stretching, logarithmic, and negative transformations) to modify image intensity values.

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

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Contrast stretching
stretched = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX)

# Log transformation (fixed)
img_float = img.astype(np.float64)  # Convert to float to avoid log issues
log_image = np.log1p(img_float)     # Apply log1p
log_image = (255 / np.log1p(255)) * log_image  # Scale to [0,255]
log_transformed = np.uint8(log_image)  # Convert to uint8 for display

# Negative transformation
negative = cv2.bitwise_not(img)

# Display images using Matplotlib
titles = ["Original", "Contrast Stretching", "Log Transformation", "Negative"]
images = [img, stretched, log_transformed, negative]

plt.figure(figsize=(12, 5))
for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")

plt.show()
```


Original
Contrast Stretching
Log Transformation
Negative

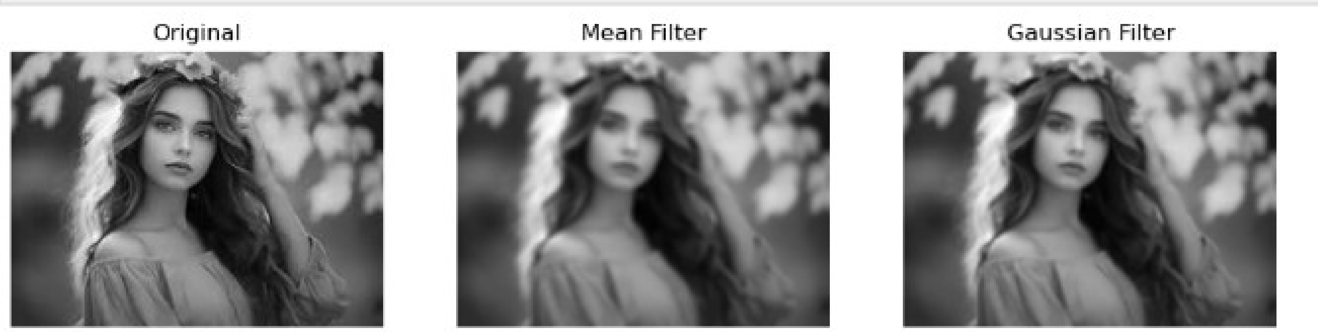
3. Image Quantization Simulate image quantization by reducing the number of bits per pixel and analyze the effects on image quality.

```
import cv2
import matplotlib.pyplot as plt

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Function to quantize image
def quantize_image(image, levels):
    factor = 256 // levels
    return (image // factor) * factor

# Different quantization levels
quant_levels = [256, 128, 64, 32, 16, 8, 4, 2]
quantized_images = [quantize_image(img, level) for level in quant_levels]

titles = [f"{lvl} Levels" for lvl in quant_levels]

# Display images
plt.figure(figsize=(12, 8))
for i in range(len(quantized_images)):
    plt.subplot(2, 4, i+1)
    plt.imshow(quantized_images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")

plt.show()
```


256 Levels
128 Levels
64 Levels
32 Levels
16 Levels
8 Levels
4 Levels
2 Levels

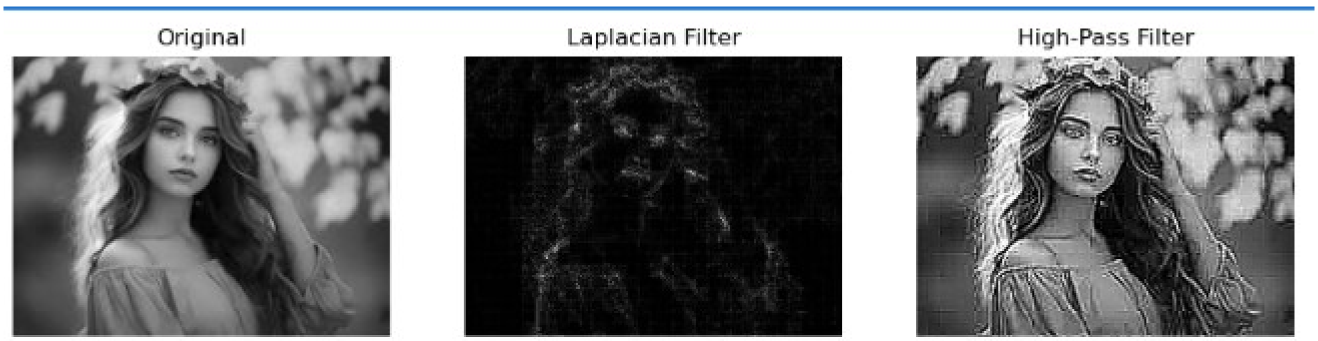
4. Spatial Filtering: Smoothing Filters Apply mean filtering and Gaussian smoothing to reduce noise in images, and compare their performance.

```
import cv2
import matplotlib.pyplot as plt

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Apply Mean Filtering (Averaging Filter)
kernel_size = (5, 5)
mean_filtered = cv2.blur(img, kernel_size)

# Apply Gaussian Smoothing
gaussian_filtered = cv2.GaussianBlur(img, kernel_size, 0)

# Display images using Matplotlib
titles = ["Original", "Mean Filter", "Gaussian Filter"]
images = [img, mean_filtered, gaussian_filtered]

plt.figure(figsize=(12, 5))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")

plt.show()
```



5. Spatial Filtering: Sharpening Filters. Implement sharpening filters like Laplacian and high-pass filters to enhance edges and fine details in the image.

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

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Apply Laplacian Filter
laplacian = cv2.Laplacian(img, cv2.CV_64F)
laplacian = cv2.convertScaleAbs(laplacian)

# Apply High-Pass Filter (Sharpening)
kernel = np.array([[-1, -1, -1],
                   [-1,  9, -1],
                   [-1, -1, -1]])
high_pass = cv2.filter2D(img, -1, kernel)

# Display images using Matplotlib
titles = ["Original", "Laplacian Filter", "High-Pass Filter"]
images = [img, laplacian, high_pass]

plt.figure(figsize=(12, 5))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")
plt.show()
```



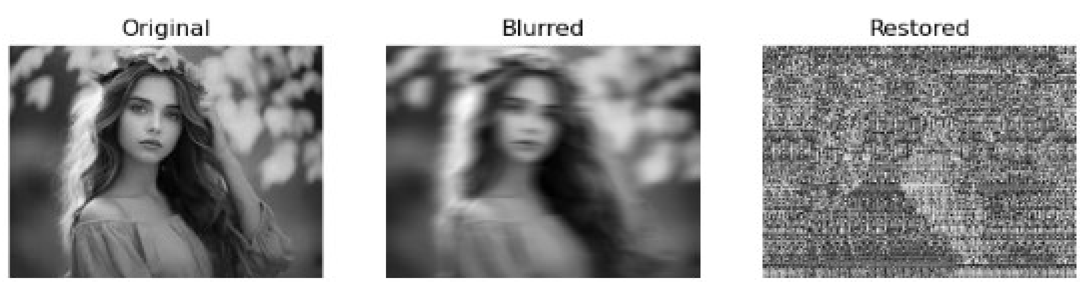
6. Homomorphic Filtering Apply homomorphic filtering to improve contrast and correct lighting variations in images.

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

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Convert image to logarithmic domain
img_log = np.log1p(img.astype(np.float32))

# Perform Fourier Transform
img_fft = np.fft.fft2(img_log)
img_fft_shift = np.fft.fftshift(img_fft)

# Create a high-pass filter
rows, cols = img.shape
x, y = np.meshgrid(np.linspace(-0.5, 0.5, cols), np.linspace(-0.5, 0.5, rows))
d = np.sqrt(x**2 + y**2)
hp_filter = 1 - np.exp(-(d**2) / (0.05**2))

# Apply filter and inverse transform
img_fft_filtered = img_fft_shift * hp_filter
img_ifft = np.fft.ifft2(np.fft.ifftshift(img_fft_filtered))
homomorphic = np.expm1(np.real(img_ifft))
homomorphic = cv2.normalize(homomorphic, None, 0, 255,
cv2.NORM_MINMAX).astype(np.uint8)

# Display images
titles = ["Original", "Homomorphic Filter"]
images = [img, homomorphic]

plt.figure(figsize=(8, 4))
for i in range(2):
    plt.subplot(1, 2, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")

plt.show()
```


Original
Homomorphic Filter

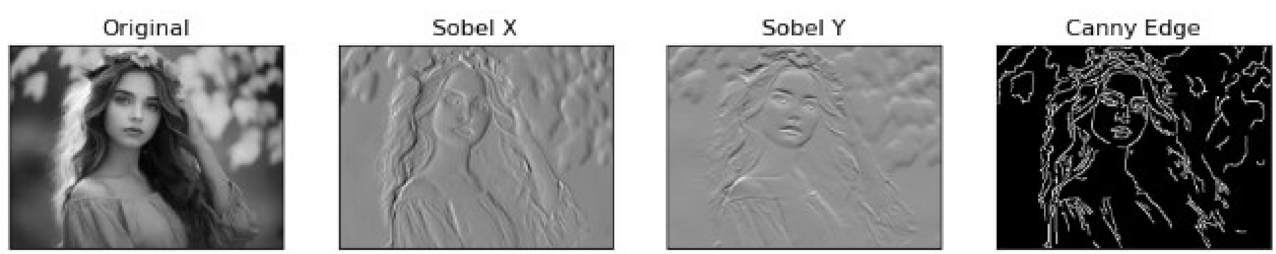
7. Image Restoration Using Wiener Filter Simulate image degradation (e.g., motion blur) and restore the image using Wiener filtering to reduce noise and improve image quality.

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

# Read image in grayscale
img = cv2.imread('D:/images.jpeg', cv2.IMREAD_GRAYSCALE)

# Simulate motion blur
kernel = np.ones((1, 15)) / 15
blurred = cv2.filter2D(img, -1, kernel)

# Wiener filter (simplified)
eps = 1e-6  # Prevent division errors
H = np.fft.fft2(kernel, s=img.shape)  # Blur in frequency domain
restored = np.fft.ifft2(np.fft.fft2(blurred) / (H + eps))
restored = np.abs(restored).astype(np.uint8)  # Convert back

# Display images
titles = ["Original", "Blurred", "Restored"]
images = [img, blurred, restored]

plt.figure(figsize=(10, 4))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis("off")

plt.show()
```



8. Edge Detection: Sobel and Canny Apply Sobel and Canny edge detection algorithms to identify edges and boundaries in an image.

```
import cv2
import matplotlib.pyplot as plt

# Load the image in grayscale
image = cv2.imread('D:/images.jpeg', 0)

# Apply Sobel filters
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# Apply Canny edge detection
canny_edges = cv2.Canny(image, 100, 200)

# Display results
titles = ['Original', 'Sobel X', 'Sobel Y', 'Canny Edge']
images = [image, sobel_x, sobel_y, canny_edges]

plt.figure(figsize=(12, 5))
for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.xticks([])  # Hide X-axis numbers
    plt.yticks([])  # Hide Y-axis numbers

plt.show()
```



9. Thresholding for Image Segmentation Implement global thresholding (Otsu’s method) and adaptive thresholding to segment an image based on pixel intensity values.

```
import cv2
import matplotlib.pyplot as plt

# Load image in grayscale
image = cv2.imread('D:/images.jpeg', 0)

# Global thresholding (Otsu's method)
_, otsu_thresh = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)

# Adaptive thresholding
adaptive_thresh = cv2.adaptiveThreshold(image, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,  cv2.THRESH_BINARY, 11, 2)

# Show results
titles = ['Original', 'Otsu Thresholding', 'Adaptive Thresholding']
images = [image, otsu_thresh, adaptive_thresh]

plt.figure(figsize=(10, 4))
for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')

plt.show()
```

10. Region Growing for Image Segmentation Implement a region-growing algorithm for segmenting regions in an image based on pixel intensity or color similarity.

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

# Load image in grayscale
image = cv2.imread('D:/images.jpeg', 0)

# Initialize seed point
seed = (50, 50)  # Change this based on the image
tolerance = 10  # Intensity difference allowed

# Create mask and perform flood fill
mask = np.zeros((image.shape[0] + 2, image.shape[1] + 2), np.uint8)
flooded = image.copy()
cv2.floodFill(flooded, mask, seed, 255, tolerance, tolerance,
cv2.FLOODFILL_FIXED_RANGE)

# Show results
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original')

plt.subplot(1, 2, 2)
plt.imshow(flooded, cmap='gray')
plt.title('Region Grown')

plt.show()
```