S19355_Assignment_06_CSC_3141

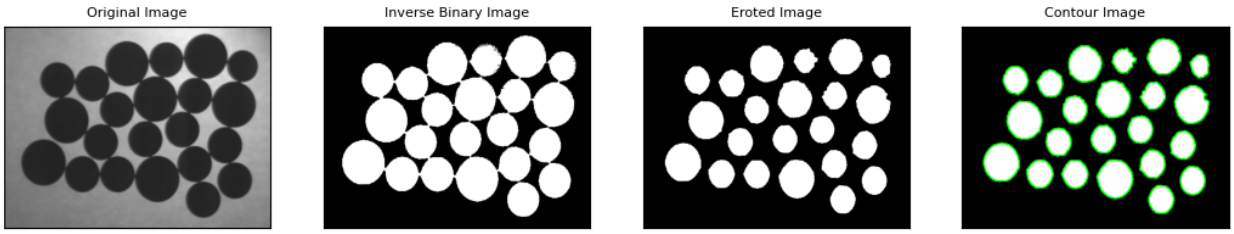
1. Use the image: overlap_coins.jpg and perform the following tasks.

    a. Count the number of coins available in the image.

```
# Import Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Import the original image
img = cv2.imread(r'overlap_coins.jpg', cv2.IMREAD_COLOR)

# Convert into grayscale
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Perform Thresholding
ret, thresh = cv2.threshold(imgGray, 50, 255, 0)

# Take the inverse of the image to make the background to black
imgInverse = cv2.bitwise_not(thresh)

# Create a Rectangular Kernel as the SE
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))

# Perform Erosion to seperate overlaps of the objects
erosion = cv2.erode(imgInverse, kernel, iterations=3)

# Get eroted image copies
eroCopy_contours = erosion.copy()
eroCopy_drawContours = erosion.copy()
eroCopy_drawContoursBGR = cv2.cvtColor(eroCopy_drawContours,cv2.COLOR_GRAY2BGR)

# Find Contours
contours, hierarchy = cv2.findContours(eroCopy_contours, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# Draw Contours
imgWithContours = cv2.drawContours(eroCopy_drawContoursBGR,contours,-
1,(0,255,0),2)

# Number of coins available in the image
totalCoins = len(contours)
```

```
print("Number of Coins = ", totalCoins)

# Plots

titles = ["Original Image", "Inverse Binary Image", "Eroted Image", "Contour
Image"]
images = [img, imgInverse, erosion, imgWithContours]

plt.figure(figsize=(13,10), num='test.img')

for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.title(titles[i], fontsize = 8)
    plt.xticks([]), plt.yticks([])
    plt.imshow(images[i], 'gray')
```

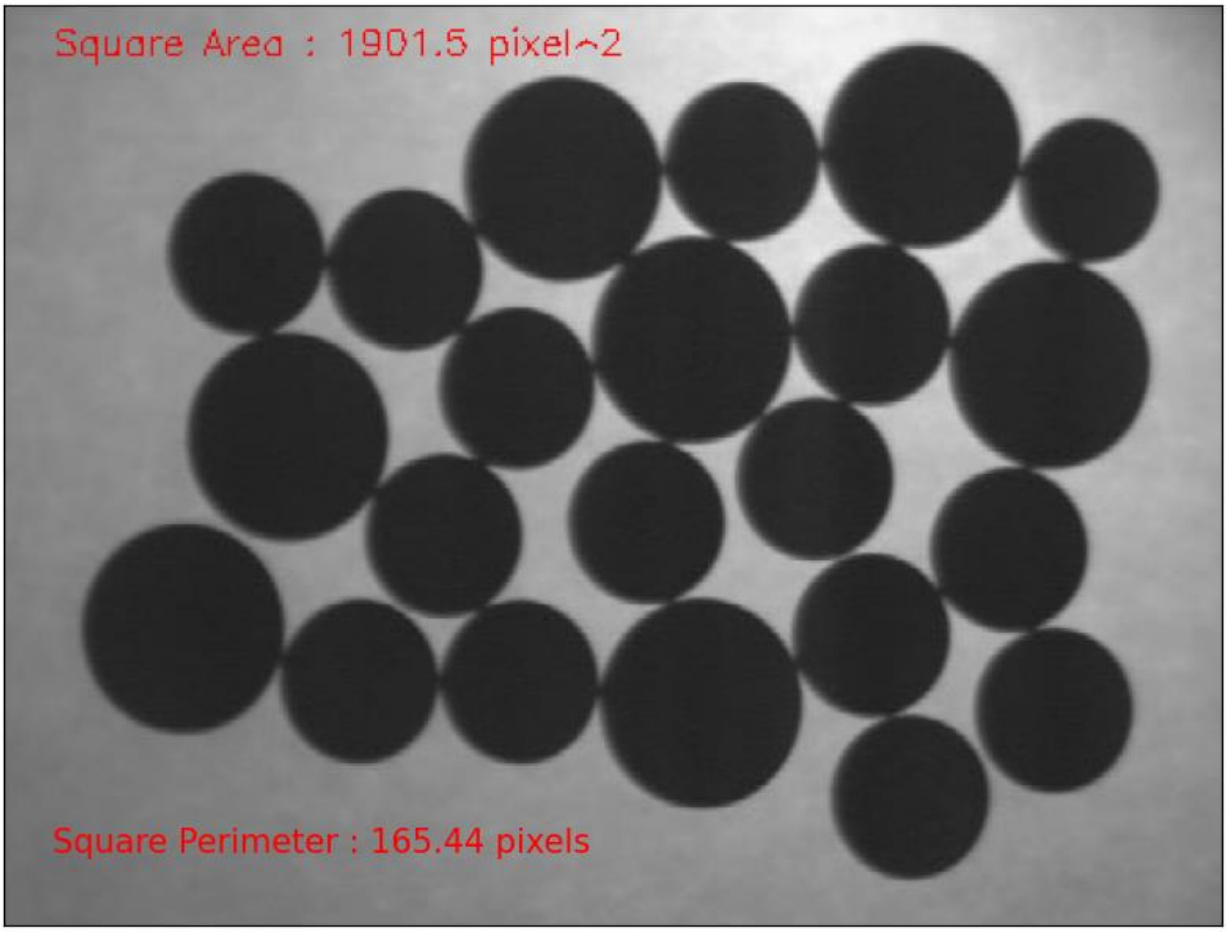
Number of Coins = 22



b. Calculate the total area covered by the coins.

```
# Area
area = cv2.contourArea(contours[0])

# writing texts
position =(20, 20)      # (y, x)
text = "Square Area : " + str(area) + " pixel^2"

cv2.putText(img,        # numpy array on which text is written
            text,       # text
            position,   # position at which writing has to start
            cv2.FONT_HERSHEY_SIMPLEX,   # font family
            0.5,        # font size
            (255,0,0,255),            # font color
            1)          # font stroke
# Perimeter
# Second argument - whether a closed contour or just a curve
```

```
perimeter = cv2.arcLength(contours[1], True)
perimeter = "{0:.2f}".format(perimeter)
position1 = (20, 30)   # (x, y)
text1 = "Square Perimeter : " + str(perimeter) + " pixels"

# x, y plotting

plt.figure(figsize=(10,10), num='test.img')

plt.text(20, 350, text1, fontsize=15, color='r')
plt.xticks([]), plt.yticks([])
plt.imshow(img, 'gray')
```



2. Compare and contrast cv2.Sobel, cv2.Laplacian and cv2.Canny with some examples. You may paste the screen shots of resulting images for each function.

A. Sobel Operator (cv2.Sobel)

The Sobel operator is used for edge detection by calculating the gradient of the image intensity. It uses two convolution kernels to compute the gradients in the x and y directions.

- Computes the first-order derivatives.
- Sensitive to noise but less than the basic gradient.
- Can be used to find edges in both horizontal and vertical directions.

B. Laplacian Operator (cv2.Laplacian)

The Laplacian operator detects edges by calculating the second-order derivatives. It highlights regions of rapid intensity change and is therefore more sensitive to noise.

- Computes the second-order derivatives.
- More sensitive to noise compared to Sobel.
- Detects edges regardless of their direction.

C. Canny Edge Detector (cv2.Canny)

The Canny edge detector is a multi-stage algorithm that provides robust edge detection by combining gradient calculation, non-maximum suppression, and hysteresis thresholding.

- Multi-step process: gradient calculation, non-maximum suppression, and hysteresis thresholding.
- Provides precise and strong edges.
- Less sensitive to noise due to the initial Gaussian smoothing step.

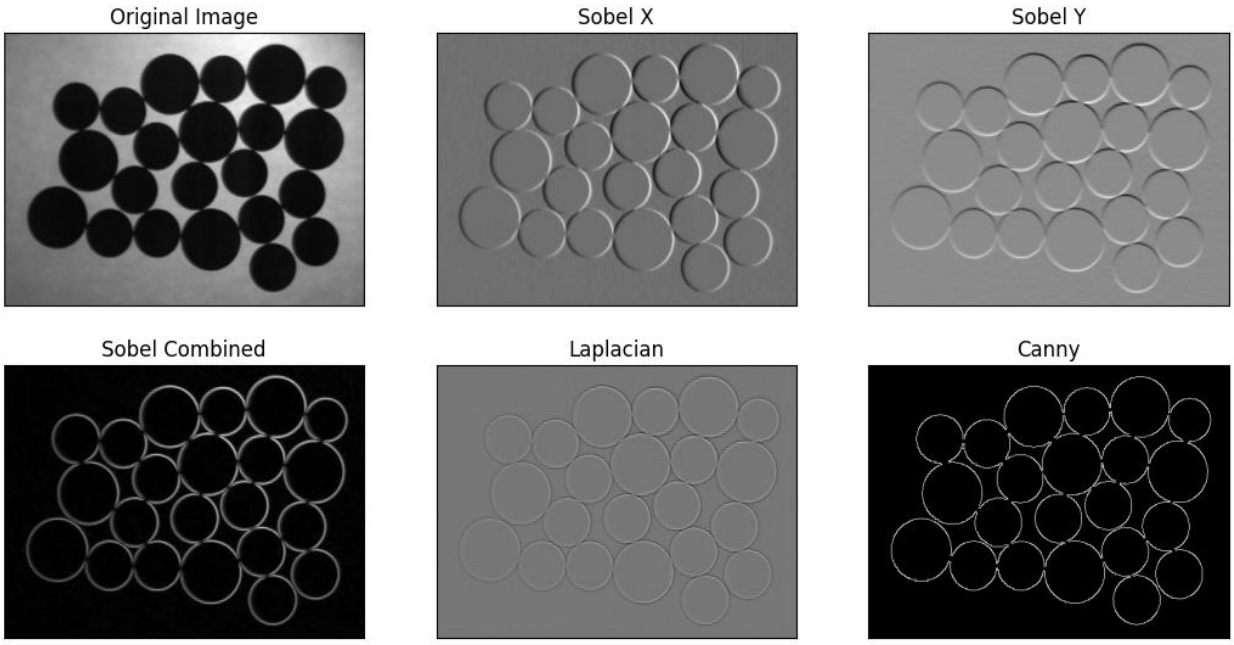
Summary

Feature	Sobel	Laplacian	Canny
Derivative Order	First	Second	First
Direction Sensitivity	X and Y directional	Non-directional	Non-directional
Noise Sensitivity	Moderate	High	Low
Edge Detecting Quality	Good for detecting edges in specific directions	Good for overall edge detection	Excellent, precise, and robust
Computation	Fast	Moderate	Slow

- Sobel is useful for detecting edges in specific directions and is moderately sensitive to noise.

- Laplacian provides overall edge detection but is highly sensitive to noise, as it uses second-order derivatives.
- Canny is a more sophisticated and robust edge detection method that combines several steps to provide precise edge maps with reduced noise sensitivity.

```
# Load coins image
img = cv2.imread(r'overlap_coins.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Sobel operator
sobelx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # Sobel edge detection on the
X axis
sobely = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # Sobel edge detection on the
Y axis
sobel = cv2.magnitude(sobelx, sobely)               # Combine both directions

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

# Apply Canny edge detector
canny = cv2.Canny(img, 100, 200)

# Plotting

titles = ["Original Image", "Sobel X", "Sobel Y", "Sobel Combined", "Laplacian",
"Canny"]
images = [img, sobelx, sobely, sobel, laplacian, canny]

plt.figure(figsize=(13,13), num='test.img')

for i in range(6):
    if i<3:
        plt.subplot(1, 3, i+1)
        plt.title(titles[i])
        plt.xticks([]), plt.yticks([])
        plt.imshow(images[i], 'gray')
    else:
        plt.subplot(2, 3, i+1)
        plt.title(titles[i])
        plt.xticks([]), plt.yticks([])
        plt.imshow(images[i], 'gray')
```



3. Compare the results of applying spatial domain soothing operations and frequency domain smoothing operation to the clown.jpg image which is corrupted by patterned noise. Paste the screen shots of resulting images for each operation along with the codes.

A. Averaging (Mean Filter)

Computes the average of all the pixels in the neighborhood of the target pixel.

- Simple and easy to implement.
- Effective for uniform noise reduction.
- Can blur edges significantly.
- Use Case : Basic noise reduction.

B. Gaussian Blur

Applies a Gaussian function to weight the pixels in the neighborhood, giving more weight to the central pixels.

- Smooths the image while preserving some edges better than averaging.
- Effective for Gaussian noise reduction.
- Requires choice of kernel size and standard deviation.
- Use Case : General noise reduction with less edge blurring compared to averaging.

C. Median Blur

Replaces each pixel with the median value of its neighborhood.

- Very effective at removing "salt-and-pepper" noise.
- Preserves edges better than averaging and Gaussian blur.
- Computationally more expensive.
- Use Case : Removing "salt-and-pepper" noise and preserving edges.

D. Bilateral Filter

Combines domain and range filtering; considers both spatial distance and intensity difference.

- Smooths images while preserving edges.
- More complex and computationally expensive.
- Requires tuning of spatial and intensity parameters.
- Use Case : Smoothing while preserving edges, useful in applications requiring edge preservation like image abstraction and stylization.

E. Frequency Domain Low-pass Filter

Reduces high-frequency components in the frequency domain (FFT).

- Effective at removing high-frequency noise.
- Can blur the entire image.
- More complex processing involving FFT.
- Use Case : General noise reduction in the frequency domain.

F. Frequency Domain High-pass Filter

Reduces low-frequency components, enhancing high-frequency details.

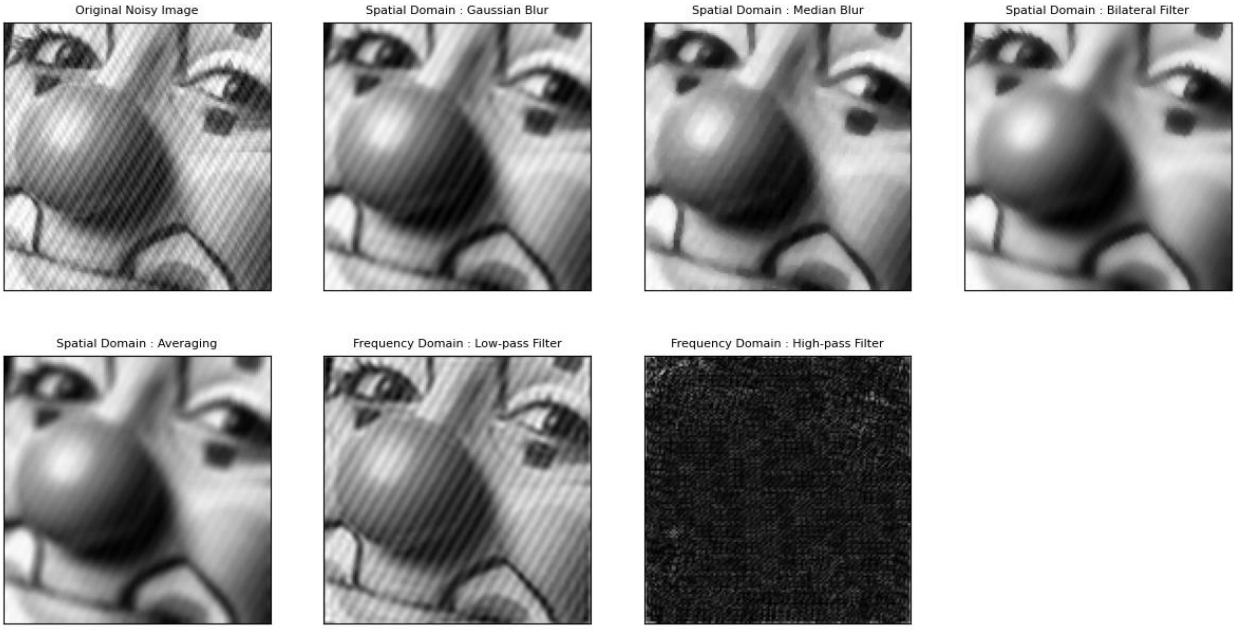
- Enhances edges and fine details.
- Can amplify noise.
- Requires understanding of frequency domain processing.
- Use Case : Edge detection and image sharpening.

Summary

- Averaging and Gaussian Blur are suitable for simple noise reduction tasks where edge preservation is less critical.
- Median Blur is ideal for images with salt-and-pepper noise and requires edge preservation.
- Bilateral Filter is excellent for applications needing edge preservation alongside noise reduction.
- Frequency Domain Filters (Low-pass and High-pass) are powerful but require more complex processing and understanding of the frequency domain. They are used for specialized tasks like noise reduction in high-frequency regions or enhancing edges.

```
# Load coins image
noisyImg = cv2.imread(r'clown.jpg', cv2.IMREAD_GRAYSCALE)

# Spatial Domain Smoothing Operations
# 1. Apply Gaussian blur
gaussianBlur = cv2.GaussianBlur(noisyImg, (5, 5), 0)

# 2. Apply Median blur
medianBlur = cv2.medianBlur(noisyImg, 5)

# 3. Apply Bilateral filter
bilateralFilter = cv2.bilateralFilter(noisyImg, 9, 75, 75)

# 4. Apply Averaging
averaging = cv2.blur(noisyImg, (5,5))

# Frequency Domain Smoothing Operation
# 1. Low-pass Filter

# Perform DFT
dft = cv2.dft(np.float32(noisyImg), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

# Create a mask with a low-pass filter
rows, cols = noisyImg.shape
crow, ccol = rows // 2, cols // 2
mask = np.zeros((rows, cols, 2), np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1
```

```
# Apply the mask and inverse DFT
fshift = dft_shift * mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv2.idft(f_ishift)
img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

# 2. High-pass Filter
# Create a mask with a high-pass filter
mask = np.ones((rows, cols, 2), np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 0

# Apply the mask and inverse DFT
fshift = dft_shift * mask
f_ishift = np.fft.ifftshift(fshift)
img_back_hp = cv2.idft(f_ishift)
img_back_hp = cv2.magnitude(img_back_hp[:, :, 0], img_back_hp[:, :, 1])

# Plots

titles = ["Original Noisy Image", "Spatial Domain : Gaussian Blur", "Spatial
Domain : Median Blur", "Spatial Domain : Bilateral Filter",
           "Spatial Domain : Averaging", "Frequency Domain : Low-pass Filter",
"Frequency Domain : High-pass Filter"]
images = [noisyImg, gaussianBlur, medianBlur, bilateralFilter, averaging,
img_back, img_back_hp]

plt.figure(figsize=(15, 15), num='test.img')

for i in range(7):
    if i<4:
        plt.subplot(1, 4, i+1)
        plt.title(titles[i], fontsize=8)
        plt.xticks([]), plt.yticks([])
        plt.imshow(images[i], 'gray')
    else:
        plt.subplot(2, 4, i+1)
        plt.title(titles[i], fontsize=8)
        plt.xticks([]), plt.yticks([])
        plt.imshow(images[i], 'gray')
```


Original Noisy Image
Spatial Domain : Gaussian Blur
Spatial Domain : Median Blur
Spatial Domain : Bilateral Filter
Spatial Domain : Averaging
Frequency Domain : Low-pass Filter
Frequency Domain : High-pass Filter