ECE-GY 6123 Image and Video Processing

Computer Assignment 2

Abirami Sivakumar

as16288

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

Problem 1 (Implementing 2D convolution in Python)

(a) Write 2D convolution function conv2, which implements “same”-padding convolution. For simplicity, you can assume the filter has an odd length in both vertical and horizontal directions. The input image should be grayscale. Assume pixel values outside the image are zero

```
def conv2(img, filter):
    imageHeight, imageWidth = img.shape     #Storing image height and width
    filterHeight, filterWidth = filter.shape    #Storing filter height and width

    paddingHeight = filterHeight // 2
    paddingWidth = filterWidth // 2         #Calculating padding pixels width required

    paddedImage = np.pad(img, ((paddingHeight, paddingHeight), (paddingWidth, paddingWidth)), mode='constant')    #Using the np.pad to p
    if imageHeight < 10 and imageWidth < 10:
      print("Padded Image: \n", paddedImage, "\n")
      print("Filter: \n", filter, "\n")

    convolvedImage = np.zeros_like(img)     #Creating a numpy array with zeros and the same size as the input image (used for storing th
    for i in range(imageHeight):
        for j in range(imageWidth):
            convolvedImage[i, j] = np.sum(paddedImage[i:i+filterHeight, j:j+filterWidth] * filter)  #Performing convolution. Using pytho
    if imageHeight < 10 and imageWidth < 10:
      print("Convoluted Image: \n",convolvedImage)

    return convolvedImage
```

Initially, the dimensions of the image and filter are obtained. Subsequently, the required padding pixels to maintain consistent padding are calculated. Afterward, an empty NumPy array of the same size as the input image is created to store the convolution results (maintaining the input size due to identical padding). The convolution is then computed by taking sections of the image matching the filter's size, performing element-wise multiplication with the filter, summing the results, and storing the outcome as a pixel value in the empty array. This process is repeated for each pixel to obtain the filtered image.

```
a = conv2(img=np.array([[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3],[4,4,4,4,4],[5,5,5,5,5]]), filter=1/9*np.array([[1,1,1],[1,1,1],[1,1,1]]))
```

```
 Padded Image: 
  [[0 0 0 0 0 0 0]
  [0 1 1 1 1 1 0]
  [0 2 2 2 2 2 0]
  [0 3 3 3 3 3 0]
  [0 4 4 4 4 4 0]
  [0 5 5 5 5 5 0]
  [0 0 0 0 0 0 0]] 

 Filter: 
  [[0.11111111 0.11111111 0.11111111]
  [0.11111111 0.11111111 0.11111111]
  [0.11111111 0.11111111 0.11111111]] 

 Convoluted Image: 
  [[0 0 0 0 0]
  [1 2 2 2 1]
  [2 3 3 3 1]
  [2 4 4 4 2]
  [2 3 3 3 2]]
```

We can see an example of using an average filter over an image above.

(b) Write a function plot filtering that uses your conv2 function to filter a given input image with a given input filter and plots the following:

• The original input image and filtered image

• The log-magnitude spectrum of the original image, filter, and output image.

Use a grayscale colormap and a colorbar for each plot. For your filter-response, use an FFT size equal to the size of your image.

```
def plot_filtering(img, filter):
    filteredImage = conv2(img, filter)   #Filtering the image using the convolution function

    logImage = fft.fftshift(fft.fft2(img))      #Calculating the fourier transform of the image using fft.fft2. But this produces result
    logImage = np.log(np.abs(logImage) + 1)    #Calculating magnitude using np.abs, calculating log using np.log and adding 1 to avoid l
    logFilter = fft.fftshift(fft.fft2(filter))
    logFilter = np.log(np.abs(logFilter) + 1)
    logFilteredImage = fft.fftshift(fft.fft2(filteredImage))
    logFilteredImage = np.log(np.abs(logFilteredImage) + 1)

    fig, axis = plt.subplots(2, 3, figsize=(15, 10))  #creating the structure for the plots (2,3)

    axis[0, 0].imshow(img, cmap='gray')           #Assigning the original image to 0,0 subplot
    axis[0, 0].set_title('Original Image')        #setting title for 0,0
    plt.colorbar(ax=axis[0, 0], mappable=axis[0, 0].get_images()[0], orientation='vertical')    #creating colorbar
    axis[0, 1].imshow(filteredImage, cmap='gray')
    axis[0, 1].set_title('Filtered Image')
    plt.colorbar(ax=axis[0, 1], mappable=axis[0, 1].get_images()[0], orientation='vertical')
    axis[0, 2].imshow(logImage, cmap='gray')
    axis[0, 2].set_title('Original Spectrum')
    plt.colorbar(ax=axis[0, 2], mappable=axis[0, 2].get_images()[0], orientation='vertical')
    axis[1, 0].imshow(logFilter, cmap='gray')
    axis[1, 0].set_title('Filter Spectrum')
    plt.colorbar(ax=axis[1, 0], mappable=axis[1, 0].get_images()[0], orientation='vertical')
    axis[1, 1].imshow(logFilteredImage, cmap='gray')
    axis[1, 1].set_title('Filtered Spectrum')
    plt.colorbar(ax=axis[1, 1], mappable=axis[1, 1].get_images()[0], orientation='vertical')
    axis[1, 2].imshow(filter, cmap='gray')
    axis[1, 2].set_title('Filter')
    plt.colorbar(ax=axis[1, 2], mappable=axis[1, 2].get_images()[0], orientation='vertical')

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

Initially, the image is subjected to filtering using the convolution function. Subsequently, the Fourier transform of the image is calculated using fft.fft2. However, results are generated with low frequencies positioned in the corners and high frequencies positioned in the center. For visualization purposes, fft.fftshift is employed to relocate the zero frequency components to the center of the visualization. Following this step, the magnitude is calculated using np.abs, the logarithm is computed using np.log, and 1 is added to prevent log(0). Lastly, the results are plotted.

(c) Use plot filtering on an image of your choice with the following 3 filters:

H1 = 1/16 *

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

H2 =

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

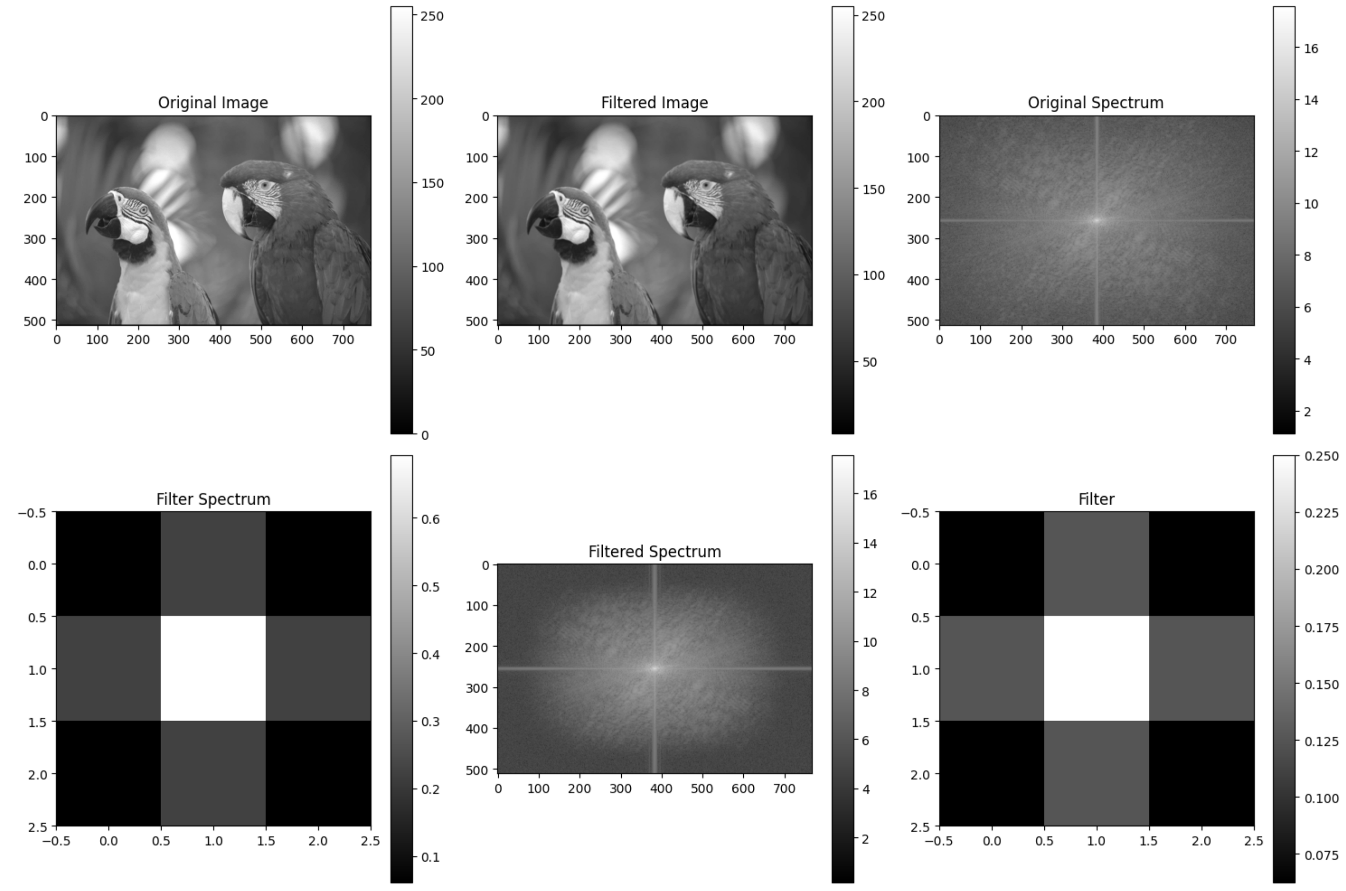
H3 =

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

For each filter, discuss the result of filtering in both spatial and frequency domains. Explain how the filtering effect in the image domain correlates with the filter and its frequency response.

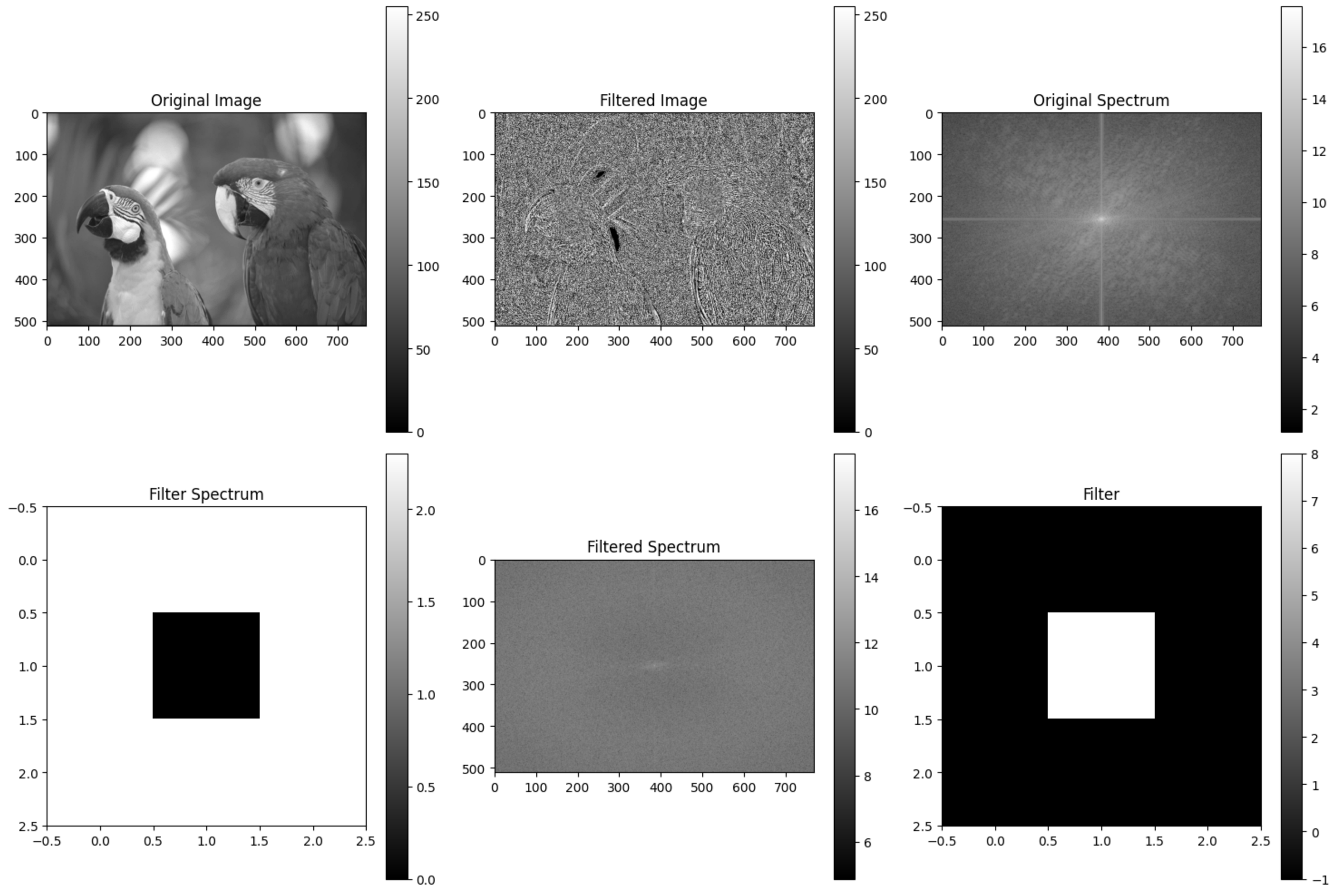
```
imgRGB = cv2.imread("/color image.png")
img = cv2.cvtColor(imgRGB, cv2.COLOR_BGR2GRAY)

filter1 = 1/16 * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])    #Filter H1
filter2 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])   #Filter H2
filter3 = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])       #Filter H3
```

```
plot_filtering(img, filter1)
```



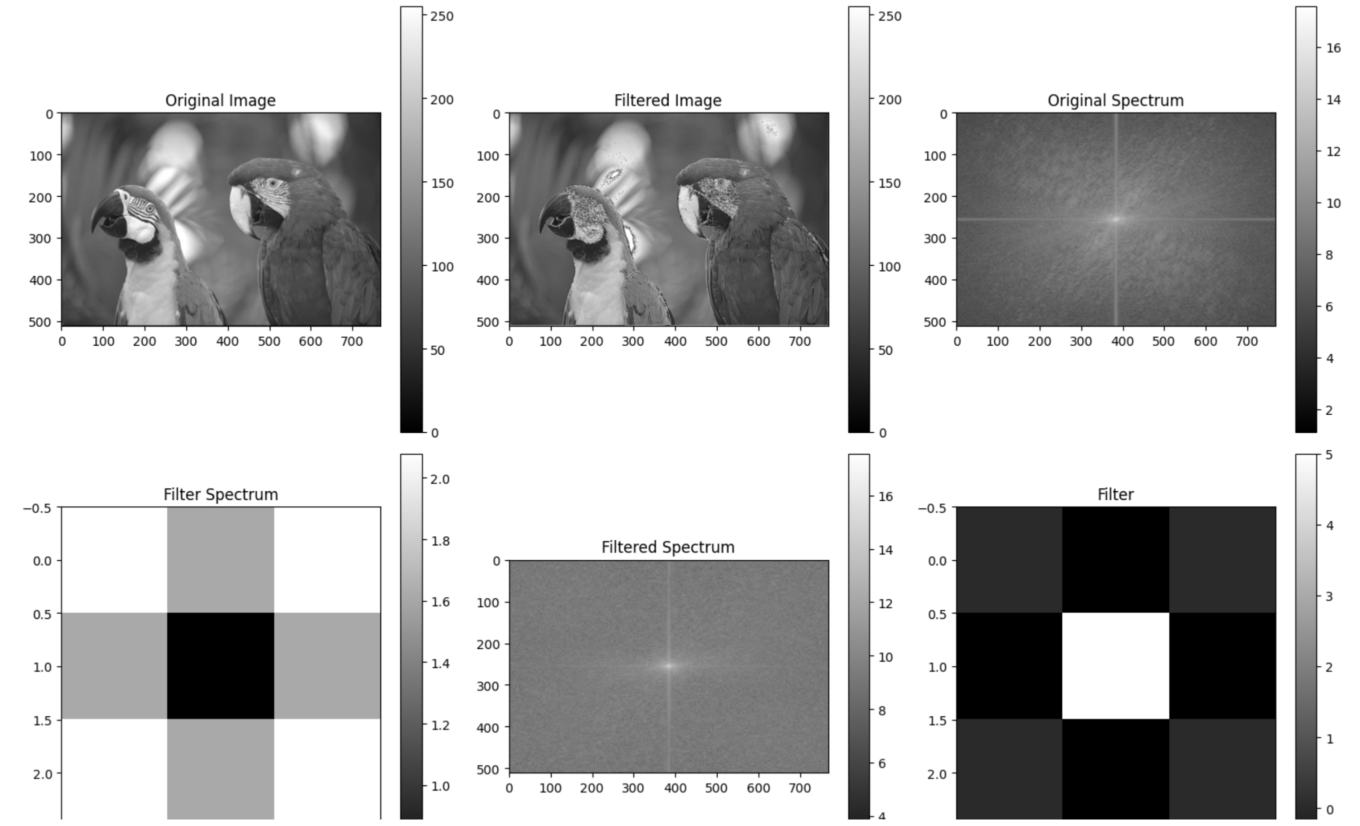
This is a low-pass smoothing filter. From the image perspective, it can be observed that smoothing has been achieved. From the frequency response perspective, it is evident that the high-frequency components, which are distant from the center, have been significantly diminished.

```
plot_filtering(img, filter2)
```



This is a LoG (Laplacian of Gaussian) filter, and it also functions as a high-pass filter. From the image perspective, it can be observed that sharp edges are accentuated, while smoother areas are subdued. From the frequency response perspective, it is evident that the high-frequency components, which are distant from the center, have been significantly amplified, whereas low-frequency components have been reduced.

```
plot_filtering(img, filter3)
```



This is a High Emphasis filter, and the coefficient is also set to 1. From the image perspective, it can be observed that sharp edges are accentuated, while smooth areas remain unchanged. From the frequency response perspective, it is evident that the high-frequency components, which are distant from the center, have been significantly amplified, and low-frequency components remain nearly unaltered.

Problem 2 (Image denoising with average and Gaussian filters)

(a) Write a function awgn that takes in an input image and noise-level σn and adds i.i.d zero-mean Gaussian random noise with standard-deviation σn. (see numpy.random.randn)

```
def awgn(img, sigma):
    noise = sigma * np.random.randn(*img.shape)  #Using np.random.randn where random samples are generated from the standard normal dist
    noisyImage = img + noise    #Adding the noise to the image
    noisyImage = np.clip(noisyImage, 0, 1)    #Clipping values if the pixels exceeds 1 or becomes less than 0
    return noisyImage
```

The standard normal distribution is utilized for generating random samples using np.random.randn. The function necessitates arguments spanning from d0 to dn, hence *img.shape is passed to create random samples corresponding to the image's height multiplied by its width. Ultimately, the generated noise is scaled by the standard deviation and added to the image, with subsequent value clipping to ensure that it falls within the range of 0 to 1.

(b) Write a function gaussian_filter which returns a 2D Gaussian filter of size m × m with standard deviation σ, where m = ⌈5σ⌉.Be sure to normalize your filter to sum to 1 so that filtering does not shift the mean of your image.

```
def gaussian_filter(sigma):
    m = int(np.ceil(5 * sigma))  # K which is >=5*sigma for a 5x5 gaussian filter. np.ceil is used to make sure K value is always >=5*si
    if m % 2 == 0:        #Also making sure m is not an odd number. If even, 1 is added to make it odd.
        m += 1

    x = np.linspace(-m // 2, m // 2, m)   #Create the 2d grid with 0 in the center and symmetric negative and positive values around it.
    y = np.linspace(-m // 2, m // 2, m)
    x, y = np.meshgrid(x, y)

    h = (1 / (2 * np.pi * sigma**2)) * np.exp(-(x**2 + y**2) / (2 * sigma**2))  #Using 2-D Gaussian equation
    h /= h.sum()  #Finally, the filter values are normalized between 0 and 1.
    print("Filter Shape: ",h.shape)
    return h
```

The calculation of the value for K, which must be greater than or equal to 5 times sigma for a 5x5 Gaussian filter, is performed. To ensure that the K value is always greater than or equal to 5 times sigma, np.ceil is employed. Additionally, it is ensured that m is not an odd number; if it is even, 1 is added to make it odd. A 2D grid with 0 at the center and symmetric negative and positive values around it is generated. The values of the Gaussian filter are computed using the 2D Gaussian filter equation. As x^2 and y^2 increase, the exponent's value decreases, ensuring that the values indeed decrease from the center of the Gaussian filter.

Equation:

$$h(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

(c) Generate a noisy version of an image using awgn, with a noise-level σn = 0.1 on an image intensity scale of [0, 1]. Apply separately a Gaussian filter of size 5 × 5 and average-filter of size 5 × 5, and comment on how the filters compare in their noiseremoval, qualitatively. Repeat this for a few different noise-levels and filter sizes. Each time, calculate the PSNR of your noisy and denoised images via the formula

$$\text{PSNR} = 10\log_{10}\left(\frac{\max^2}{\text{MSE}}\right) = -10\log_{10}(\text{MSE}), \quad \text{(when images are on range [0,1])}$$

where MSE is the mean-squared-error with respect to your original image, and max is the maximum of your image intensity range (1 for images on range [0,1], 255 for images on range [0,255]).

```
from scipy.signal import convolve2d

img = cv2.imread("/content/colourimg.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)   #Loading and converting to grayscale
img = img.astype(np.float32) / 255.0        #Normalizing the image

noiseSigma = 0.1       #Noise level
noisyImage = awgn(img, noiseSigma)    #Generating noisy image

gaussianFilter = gaussian_filter(sigma=1)   #Creating the gaussian filter
gaussianDenoised = convolve2d(noisyImage, gaussianFilter, mode='same')  #Applying the filter

averageFilter = np.ones((5, 5)) / 25    #Creating the averaging filter
averageDenoised = convolve2d(noisyImage, averageFilter, mode='same') #Applying the filter

print("Noise Level: ", noiseSigma)
print("PSNR after Gaussian filtering:", -10 * np.log10(np.mean((img - gaussianDenoised) ** 2)))
print("PSNR after Average filtering:", -10 * np.log10(np.mean((img - averageDenoised) ** 2)))
```

```
     Filter Shape:  (5, 5)
     Noise Level:  0.1
     PSNR after Gaussian filtering: 28.712128697459825
     PSNR after Average filtering: 31.94802119910737
```

```
noiseLevels = [0.05, 0.1, 0.15, 0.2]    #Varying noise levels
filterSizes = [3, 5, 7, 9]              #Varying filter sizes

for noiseSigma in noiseLevels:
    noisyImage = awgn(img, noiseSigma)

    for fSize in filterSizes:
        gaussianFilter = gaussian_filter(sigma=fSize/5)  #Since m = ceil(5*sigma)
        gaussianDenoised = convolve2d(noisyImage, gaussianFilter, mode='same')

        averageFilter = np.ones((fSize, fSize)) / (fSize**2)
        averageDenoised = convolve2d(noisyImage, averageFilter, mode='same')

        print("Noise Level: ", noiseSigma)
        print("PSNR after Gaussian filtering:", -10 * np.log10(np.mean((img - gaussianDenoised) ** 2)))
        print("PSNR after Average filtering:", -10 * np.log10(np.mean((img - averageDenoised) ** 2)))
        print()
```

```
     Filter Shape:  (3, 3)
     Noise Level:  0.05
     PSNR after Gaussian filtering: 30.176149047440845
     PSNR after Average filtering: 34.335222610428595
```

```
Filter Shape:  (5, 5)
Noise Level:  0.05
PSNR after Gaussian filtering: 33.638110139245526
PSNR after Average filtering: 34.77207007828129

Filter Shape:  (7, 7)
Noise Level:  0.05
PSNR after Gaussian filtering: 34.87803479938296
PSNR after Average filtering: 33.88004420366614

Filter Shape:  (9, 9)
Noise Level:  0.05
PSNR after Gaussian filtering: 34.80635330037787
PSNR after Average filtering: 32.979229941330615

Filter Shape:  (3, 3)
Noise Level:  0.1
PSNR after Gaussian filtering: 24.394439598625198
PSNR after Average filtering: 29.270519207107032

Filter Shape:  (5, 5)
Noise Level:  0.1
PSNR after Gaussian filtering: 28.712467778971046
PSNR after Average filtering: 31.93998638286638

Filter Shape:  (7, 7)
Noise Level:  0.1
PSNR after Gaussian filtering: 31.26254802171943
PSNR after Average filtering: 32.395939687767665

Filter Shape:  (9, 9)
Noise Level:  0.1
PSNR after Gaussian filtering: 32.374937569185974
PSNR after Average filtering: 32.12131479333784

Filter Shape:  (3, 3)
Noise Level:  0.15
PSNR after Gaussian filtering: 21.051480826905347
PSNR after Average filtering: 26.030835417544456

Filter Shape:  (5, 5)
Noise Level:  0.15
PSNR after Gaussian filtering: 25.513468709586128
PSNR after Average filtering: 29.356310206708596

Filter Shape:  (7, 7)
Noise Level:  0.15
PSNR after Gaussian filtering: 28.3901548306666
PSNR after Average filtering: 30.556128778085636

Filter Shape:  (9, 9)
Noise Level:  0.15
PSNR after Gaussian filtering: 29.962863779879996
```

Observations:

By considering varying noise levels and different filter sizes, it can be observed that Gaussian filters surpass average filters as the size of the Gaussian filter is enlarged. However, to surpass the performance of the average filter, a substantial increase in the filter size is required as noise levels rise. Otherwise, the average filter performs effectively.

When employing filter shapes of (7,7) and (9,9) with a noise level of 0.05, and when using a filter shape of (9,9) with a noise level of 0.1, the Gaussian filter exhibits superior performance. In other cases, the average filter proves to be more effective.