CSE4182 - Digital Image Processing Lab

Md. Nahid Hassan - 1710276139

October 15, 2022

Contents

1 Some Basics Works Before Processing the Image

Every time before process the image we need to read, convert and display an image. This job is repetitive. So, at the beginning I will try to explain those script. Later I am only writing the core part of every problems.

1.1 Import Library

To read and process image we will use Python well known library like, matplotlib, opencv and numpy.

```
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import cv2
```

1.2 Read Image

To read image we are using imread function. This function takes the path of the image and return the image.

```
path = 'sunset.jpg'
img = plt.imread(path)
print(img.shape, img.max(), img.min())
```

1.3 Convert Image in Different Formats

To convert RGB to gray we are using cv2.CvtColor() method.

```
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

To convert grayscale to binary we are using cv2.threshold() method.

```
_, binary = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY)
```

And histogram for grayscale image.

```
gray_hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
```

1.4 Plot Images

Here is an example of plotting images. This is part is not well maintained. Several way you can plot the images. You can choose any way what you like Matplotlib Official Document

```
plt.subplots_adjust(hspace=0.5, wspace=.2)
plt.figure(figsize=(10, 10))

plt.subplot(221) # 2 row 2 column and plot no 1
plt.imshow(img)  # for ploting images
plt.title('RGB Image')
plt.subplot(222) # plot no 2
plt.imshow(gray, cmap='gray') # for grayscale image
plt.title('Grayscale Image')
plt.subplot(223) # plot no 2
plt.imshow(binary, cmap='gray') # for grayscale image
plt.title('Grayscale Image')
plt.subplot(224) # plot no 3
plt.plot(gray_hist, 'g') # plot a histogram and color is green
plt.title('Histogram')
plt.xlabel('Pixel')
```

```
plt.ylabel('Frequencies')

plt.savefig('Sample.png')
plt.show()
```

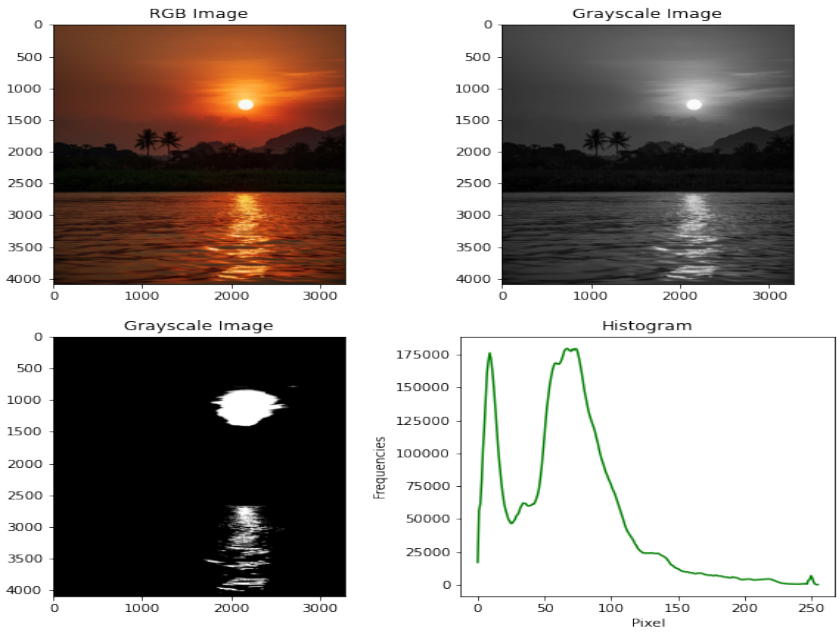

Figure 1: Basic Matplotlib Image Plot

2 Histogram

An image histogram is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image.[1] It plots the number of pixels for each tonal value. By looking at the histogram for a specific image a viewer will be able to judge the entire tonal distribution at a glance.Wiki.

2.1 Using Builtin Method

For this we are using cv2.calcHist() method.

```
red = cv2.calcHist([img], [0], None, [256], [0, 256])
green = cv2.calcHist([img], [1], None, [256], [0, 256])
blue = cv2.calcHist([img], [2], None, [256], [0,256])
```

2.2 Custom Method - Dictionary

```
def pixel_freq(mat):
    d = dict()

    for x in range(256):
        d[x] = 0

    for row in mat:
        for pixel in row:
            d[pixel] += 1
    return d

# call function
red_dict = pixel_freq(img[:,:,0])
green_dict = pixel_freq(img[:,:,1])
blue_dict = pixel_freq(img[:,:,2])
```

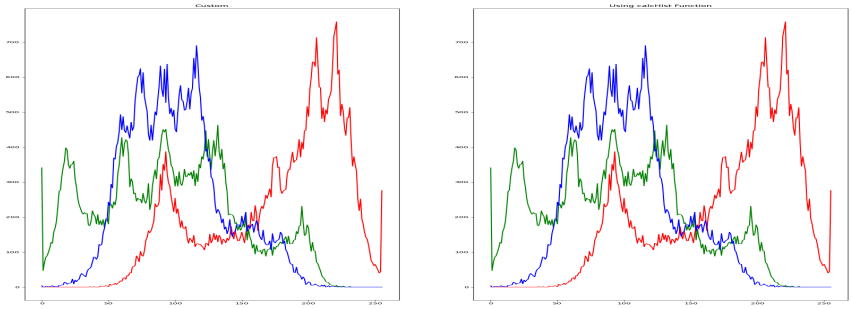

Figure 2: Histogram

In the above figure 2 you I will display two image side by side. First figure is for builtin method and second one is custom. The output of the two methods is exact same.

3 Point Processing

3.1 Point Processing

Single-point processing is a simple method of image enhancement. This technique determines a pixel value in the enhanced image dependent only on the value of the corresponding pixel in the input image.

```
T1, T2, c, p, EPSILON  = 70, 150, 1, 2, 1e-6

row, col = gray.shape
s1 = np.full(gray.shape, 10)
# s = 100, if r >= T1 and r <= T2; otherwise s = 10.
for x in range(row):
    for y in range(col):
        if gray[x, y] >= T1 and gray[x, y] <= T2:
            s1[x, y] = 100

# s = 100, if r >= T1 and r <= T2; otherwise s = r.
s2 = np.zeros(gray.shape, dtype=np.uint8)
for x in range(row):
    for y in range(col):
        if gray[x, y] >= T1 and gray[x, y] <= T2:
            s2[x, y] = 100
        else:
            s2[x, y]  = gray[x][y]

# s = c log(1 + r) .
s3 = c * np.log(1 + gray)
# s = c ( r + epsilon ) ^ p
s4 = c * ((EPSILON + gray) ** p)
```

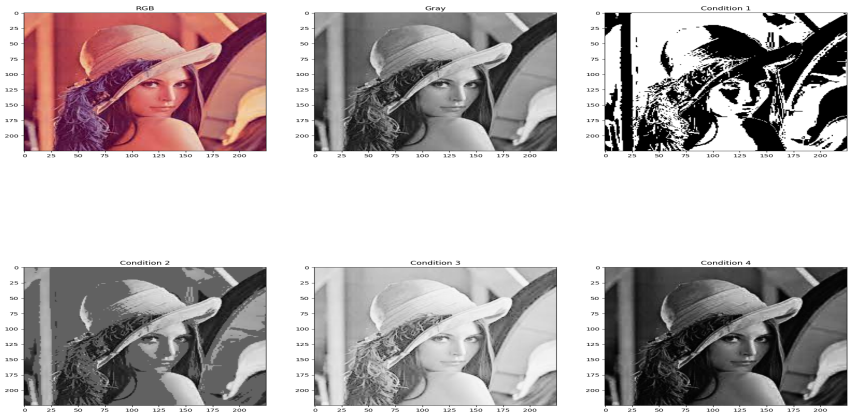

Figure 3: Point Processing

In the figure 3 is showing the direct point processing effect.

3.2 Neighbourhood Processing

3.2.1 Convolution

Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of 'multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality. Glossary - Convolution

3.2.2 Kernel

In image processing, a kernel, convolution matrix, or mask is a small matrix used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between the kernel and an image. Kernel - Image Processing

```
box_blur_kernel = np.ones((3,3)) / 9
sharpen_kernel = np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])
```

3.2.3 Bulitin Method

$filter2D()$ function is used to change the pixel intensity value of an image based on the surrounding pixel intensity values. This method can enhance or remove certain features of an image to create a new image. Syntax to define filter2D() function in python is as follows:

```
resulting_image = cv2.filter2D(src, ddepth, kernel)

box_blur = cv2.filter2D(gray, -1, box_blur_kernel)
sharpen = cv2.filter2D(gray, -1, sharpen_kernel)
```

3.2.4 Custom - Iterative

```
def conv(mat, kernel):
    row, col = mat.shape
    r, c = kernel.shape[0] // 2, kernel.shape[1] // 2
    r, c = r * 2, c * 2

    new_image = np.zeros((row - r, col - c), dtype=np.uint8)
    for i in range(row - r):
        for j in range(col - c):
            temp = np.sum(np.multiply(mat[i:3+i, j:3+j], kernel))
            if temp > 255:
                new_image[i][j] = 255
            elif temp < 0:
                new_image[i][j] = 0
            else:
                new_image[i][j] = temp

    return new_image

box_blur_custom = conv(gray, box_blur_kernel)
sharpen_custom = conv(gray, sharpen_kernel)
```

Figure 4: Convolution - Custom vs Builtin

4 Histogram Shifting

The histogram-shifting method is a well-known method for reversible data hiding. To reach the goal of data hiding, this method shifts all pixel values between peak and zero points to leaves vacant spaces for data hiding.

```
r, c = gray.shape
left, right, narrow_band = gray.copy(), gray.copy(), gray.copy()

# shifting
left = left - 79
right = right + 50

for i in range(r):
    for j in range(c):
        if narrow_band[i][j] <= 100:
            narrow_band[i][j] = 100
        elif narrow_band[i][j] >= 175:
            narrow_band[i][j] = 175

# calcHist
gray_hist = cv2.calcHist([gray], [0], None, [256], [0,256])
left_hist = cv2.calcHist([left], [0], None, [256], [0,256])
right_hist = cv2.calcHist([right], [0], None, [256], [0,256])
narrow_band_hist = cv2.calcHist([narrow_band], [0], None, [256], [0, 256])
```

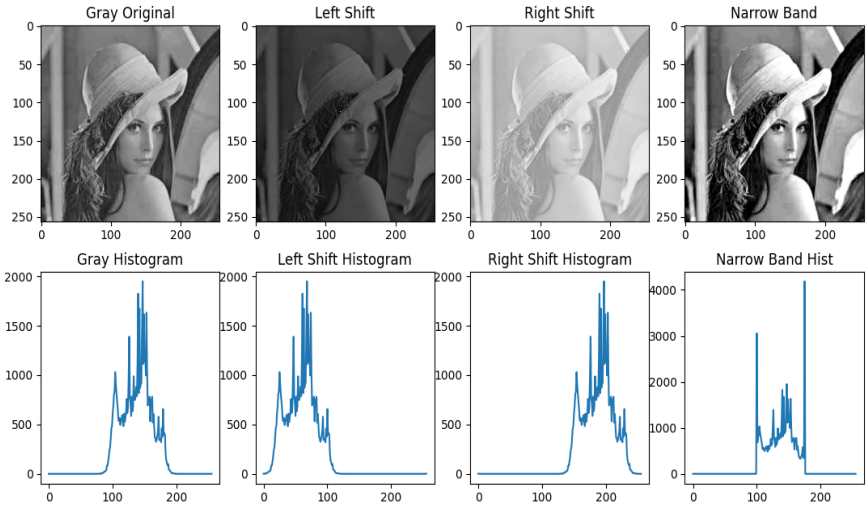

Figure 5: Histogram Shifting

5 Histogram Equalization

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram.

5.1 Using Builtin Method

```
gray_eq = cv2.equalizeHist(gray)
```

5.2 Custom Method

```
L = 256
row, col = gray.shape
img_size = row * col

cdf = gray_hist.cumsum()
cdf_min = cdf.min()

equalize_img = np.zeros((row, col), np.uint8)
for x in range(row):
    for y in range(col):
        equalize_img[x, y] = ((cdf[gray[x, y]] - cdf_min) / (img_size - cdf_min)) * (L-1)

equalize_img_hist = cv2.calcHist([equalize_img], [0], None, [256], [0, 256])
```

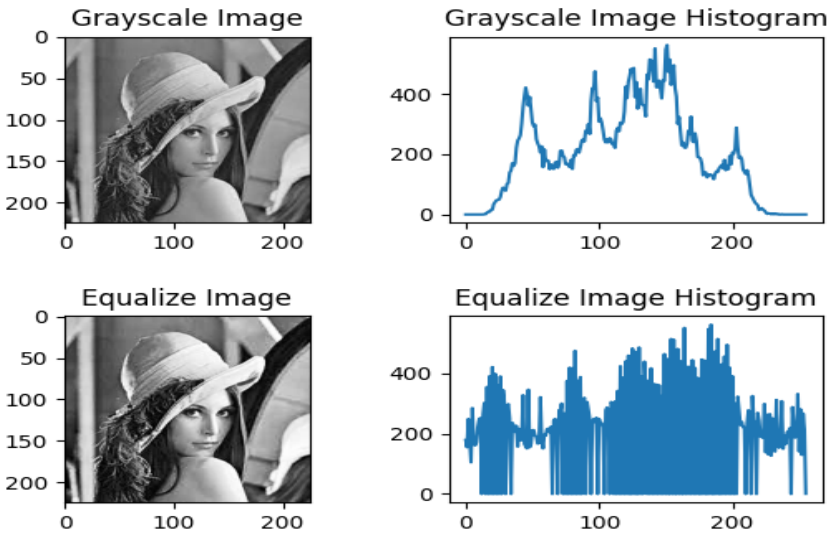

Figure 6: Histogram Equalization - Custom Method

6 Bit Slicing

Bit plane slicing is a method of representing an image with one or more bits of the byte used for each pixel. One can use only MSB to represent the pixel, which reduces the original gray level to a binary image. The three main goals of bit plane slicing is: Converting a gray level image to a binary image.

```
grayscale = cv2.cvtColor(rgb, cv2.COLOR_RGB2GRAY)
r, c = grayscale.shape

bit_1 = np.zeros((r, c), dtype=np.uint8)
bit_2 = np.zeros((r, c), dtype=np.uint8)
bit_3 = np.zeros((r, c), dtype=np.uint8)
bit_4 = np.zeros((r, c), dtype=np.uint8)
bit_5 = np.zeros((r, c), dtype=np.uint8)
bit_6 = np.zeros((r, c), dtype=np.uint8)
bit_7 = np.zeros((r, c), dtype=np.uint8)
bit_8 = np.zeros((r, c), dtype=np.uint8)

k = 1
for i in range(r):
    for j in range(c):
        bit_1[i][j] = grayscale[i][j] & k

# bit_1 = grayscale & k
bit_2 = grayscale & (k << 1)
bit_3 = grayscale & (k << 2)
bit_4 = grayscale & (k << 3)
bit_5 = grayscale & (k << 4)
bit_6 = grayscale & (k << 5)
bit_7 = grayscale & (k << 6)
bit_8 = grayscale & (k << 7)
```

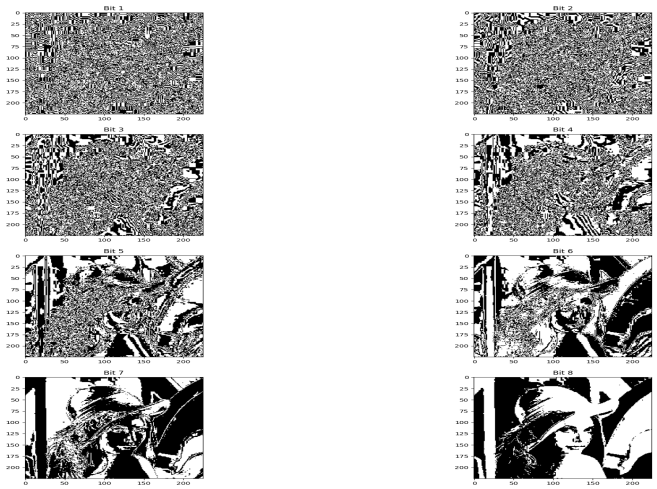

Figure 7: Histogram Equalization - Custom Method

7 Salt and Pepper Noise

Salt-and-pepper noise, also known as impulse noise, is a form of noise sometimes seen on digital images. This noise can be caused by sharp and sudden disturbances in the image signal. It presents itself as sparsely occurring white and black pixels.

```
    # noises
row, col = gray.shape
noises = row * col // 50        # total noises you want to add

# randomly added black and white pixels
for i in range(noises):
    # provide random points
    x, y = np.random.randint(0, row), np.random.randint(0, col)
    # randint(0,2) -> returns 0 or 1 at a time
    # 0 * 255 => 0, and 1 * 255 => 255
    gray[x, y] = np.random.randint(0, 2) * 255

# gaussian kernel
# divide by 16 because of total sum of each weights is equal 16
g_kernel = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]]) / 16
# here total some of each weights is equal 9
avg_kernel = np.ones((3, 3)) / 9

# filter2d
gray_avg_filtered = cv2.filter2D(original, -1, avg_kernel)
# apply gaussian filter
g_filtered = cv2.filter2D(gray, -1, g_kernel)
# apply average filter
avg_filtered = cv2.filter2D(gray, -1, avg_kernel)
# median filter produce best output for salt and pepper noise
# median blur smooth the image
median_filtered = cv2.medianBlur(gray, 3)
```

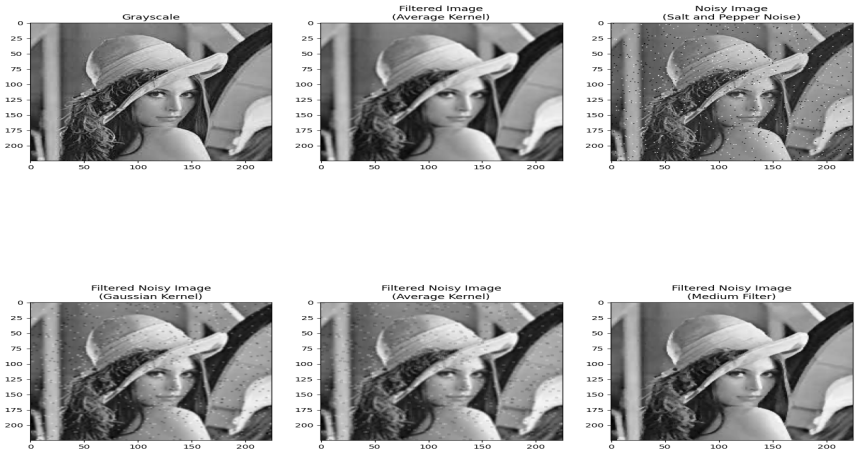

Figure 8: Histogram Equalization - Custom Method

8 Morphological Image Processing

- Erosion - When you perform erosion operation it actually remove the white pixels and vice versa.
- Dilation - when you perform dilation operation it actually adding the white pixels and vice versa.
- Opening - First perform erosion then dilation.
- Closing - First perform dilation then erosion.

```
def main():
    # image_path = "hw.jpeg"
    image_path = "abc.jpeg"
    img = plt.imread(image_path)
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # print(gray.shape) # 416, 416
    kernel = np.ones((3,3))

    erosion = cv2.erode(gray, kernel=kernel, iterations=1)
    dilation = cv2.dilate(gray, kernel=kernel, iterations=1)
    opening = cv2.morphologyEx(gray, cv2.MORPH_OPEN, kernel=kernel)
    closing = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel=kernel)

    image_set = [img, gray, erosion, dilation, opening, closing]
    image_title = ['Original','Gray Image', 'After Erosion', 'After Dilation', 'Opening', 'Closing']

    plt.figure(figsize=(15,15))
    plt.subplots_adjust(hspace=.5)

    for i in range(len(image_set)):
        plt.subplot(2,3,i + 1)
        plt.title(image_title[i])
        if image_set[i].ndim == 3:
```

```
            plt.imshow(image_set[i])
        else:
            plt.imshow(image_set[i], cmap='gray')

    plt.savefig('edoc.png')
    plt.show()

if __name__ == '__main__':
    main()
```

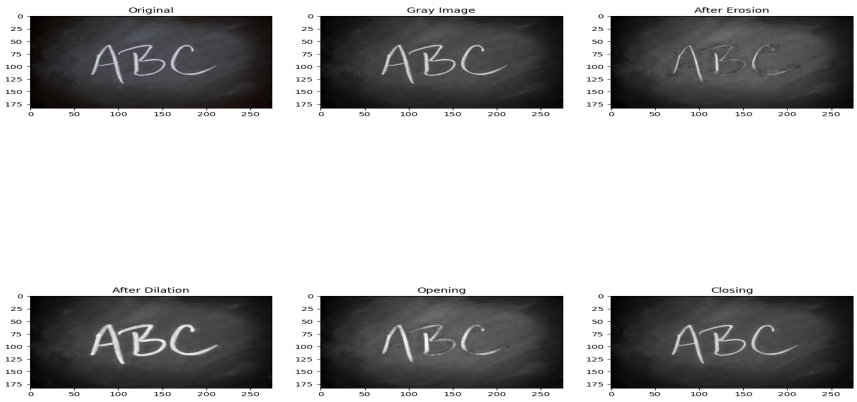

Figure 9: Erosion and Dilation

9 Image Processing in Frequency Domain

In the frequency domain, a digital image is converted from spatial domain to frequency domain. In the frequency domain, image filtering is used for image enhancement for a specific application. A Fast Fourier transformation is a tool of the frequency domain used to convert the spatial domain to the frequency domain.

9.1 Gaussian Filter In Frequency Domain

9.1.1 Gaussian Low Pass Filter

Gaussian low-pass filtering is a common post-process operation which is exploited to blur and conceal these discontinuities at the border of tampered objects introduced by copy paste operation, making the tampered image more realistic.

```
g = np.fft.fft2(gray)
g_shift = np.fft.fftshift(g)

M, N = gray.shape        # M->Row, N->Col
H = np.zeros((M, N), dtype=np.float32)

# parameter D0 control the shape of our gaussian filter.
D0 = 10   # cut off frequency
```

```
for u in range(M):
    for v in range(N):
        D = np.sqrt((u - M/2) ** 2 + (v - N/2) ** 2)
        H[u, v] = np.exp((-D**2) / (2 * D0 * D0))

# low pass
plt.subplot(2,1,1)
plt.imshow(np.abs(np.fft.ifft2(np.fft.ifftshift(g_shift * H))), cmap='gray')
```

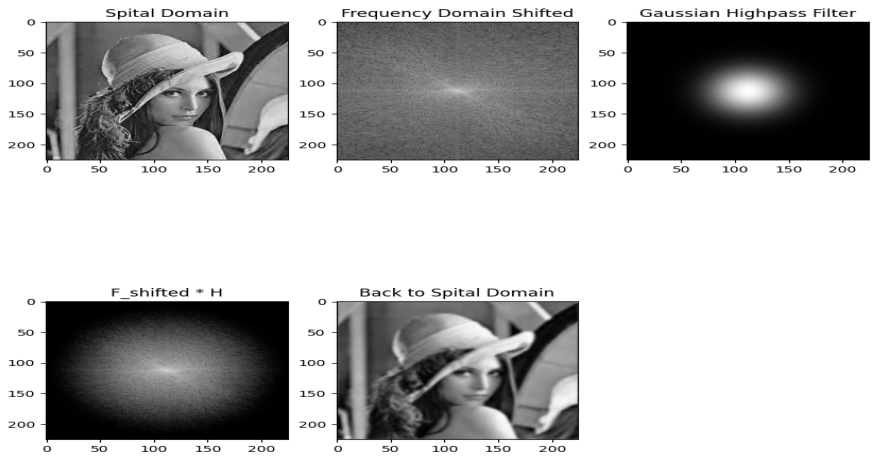

Figure 10: Gaussian Low Pass Filter

9.1.2 Gaussian High Pass Filter

High pass filter give emphasis on the high frequencies in the image. The difference between Butterworth and Gaussian filters is that the former is much sharper than latter. The resultant images by BHPF is much sharper than GHPF ,while analysis the FFT of CT and MRI image, one sharp spike is concentrated in the middle.

```
# high pass
hpf = 1 - H
plt.subplot(2,1,2)
plt.imshow(np.abs(np.fft.ifft2(np.fft.ifftshift(g_shift * hpf))), cmap='gray')
```

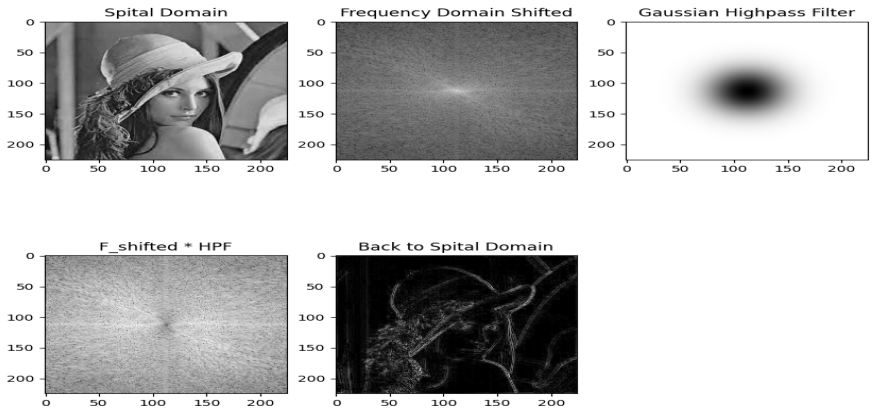

Figure 11: Gaussian High Pass Filter

10 Image Conversion

```
from PIL import Image

img = Image.open("lenna.jpeg")
img.save("lenna.png")

img = Image.open("dog.png")
rgb_img = img.convert('RGB')
rgb_img.save("dog.jpeg")
```