EE326 Lab Report 4: Frquency Domain Filtering

11812418 樊青远[1]

[1] *School of Microelectronics, SUSTech Email: fanqy2018@mail.sustech.edu.cn*

Abstract

This report provides the introduction, theory and the Python code implementation of the filter that could be append on the frequency domain.

Keywords: Frequency domain, Filter, FFT

Contents

EE326 DIP (2021)

DOI: 10.1017/pan.xxxx.xx

Corresponding author
Qingyuan Fan

Advised by
Yajun Yu

1 Objective

Filtering the image with specified kernel could be an important techniques in the field of digital image processing. However, not all transformations can be done in the spacial domain. For instance, the Gaussian filter [3] and the Butterworth notch filter needs to filter the specific frequency, which is hard to implemented in the spacial domain. In order to implemented the filter mentioned above, we introduce the filtering techniques in the frequency domain in this lab.

In this lab, to transform the image from spacial domain to time domain, we needs to implement the Fourier transformation, which decomposes functions depending on space or time into functions depending on spatial or temporal frequency. After the transformation, we will also apply the sobel, gaussian and butterworth notch filter on the different image and analysis their performance.

The filter mentioned above are also widely used in the custom and professional field of image processing, such as sharpen the scanned document, remove the grid caused by the sampling process, or even used as the Pre-processing of deep learning of the image datasets.

2 Mathematical Process of Fourier Transform

2.1 Fourier Transform in 1D Linespace

In 1D linespace, the fourier transformation of the continuous signal can be written as follows, while $f(t)$ is the signal itself, and c_n is the coefficient:

$$c_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t)e^{j\frac{2\pi n}{T}t} \tag{1}$$

$$F(\mu) = \Im\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{j2\pi\mu t} \tag{2}$$

2.2 Fourier Transform in Discrete Signal

In the digital image processing, the image is constructed by discrete pixels, so it is we needs also consider the equation of discrete Fourier Transform.

For arbitrary discrete signal, we could write the transform as:

$$F(u) = \sum_{n=0}^{M-1} f_n e^{-j2\pi un/M} \tag{3}$$

$$f(x) = \frac{1}{M}\sum_{u=0}^{M-1} F(u)e^{j2\pi ux/M} \tag{4}$$

2.3 Extent the Transformation into 2D Space

The 2D Fourier Transformation of countinous signal can be written as:

$$F(\mu, v) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(t,z)e^{-j2\pi(\mu t+vz)}dtdz \tag{5}$$

and

$$f(u,v) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} F(\mu,v)e^{j2\pi(\mu t+vz)}d\mu dv \tag{6}$$

While the signal is in discrete form:

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(ux/M+vy/N)} \tag{7}$$

and

$$f(x,y) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} F(u,v)e^{-j2\pi(ux/M+vy/N)} \tag{8}$$

2.4 Properties of Fourier Transformation

Convolution and Multiplication

$$f(t) * h(t) = H(\mu)F(\mu) \tag{9}$$

and

$$f(t)h(t) = H(\mu) * F(\mu) \tag{10}$$

Periodicity

$$f(x,y)(-1)^{x+y} \Leftrightarrow F(u - M/2, v - N/2) \tag{11}$$

Symmetries

For the discrete signal, the signal satisfies $f(x) = f(N-x)$ can be regarded as even function, the one satisfies $f(x) = -f(N-x)$ can be regarded as odd function.

2.5 Shift and Zero Padding

Observing from the frequency domain of a discrete signal, we can find that its signal is symmetrical along the vertical axis, while the frequency component near the zero point indicate the center of the frequency (the low frequency). Specifically, if we directly apply the FFT transform on the 2D image array, the center of the frequency component will be also lies at the zero point of the image array after FFT. However, in 2D array, the zero point of the array located at the corner of the image according to the FFT theory and the periodically propriety of the frequency domain. Such array is hard for us to design the appropriate filter. Besides, the periodicity of the frequency domain could also brings the issues like wraparound error [1].

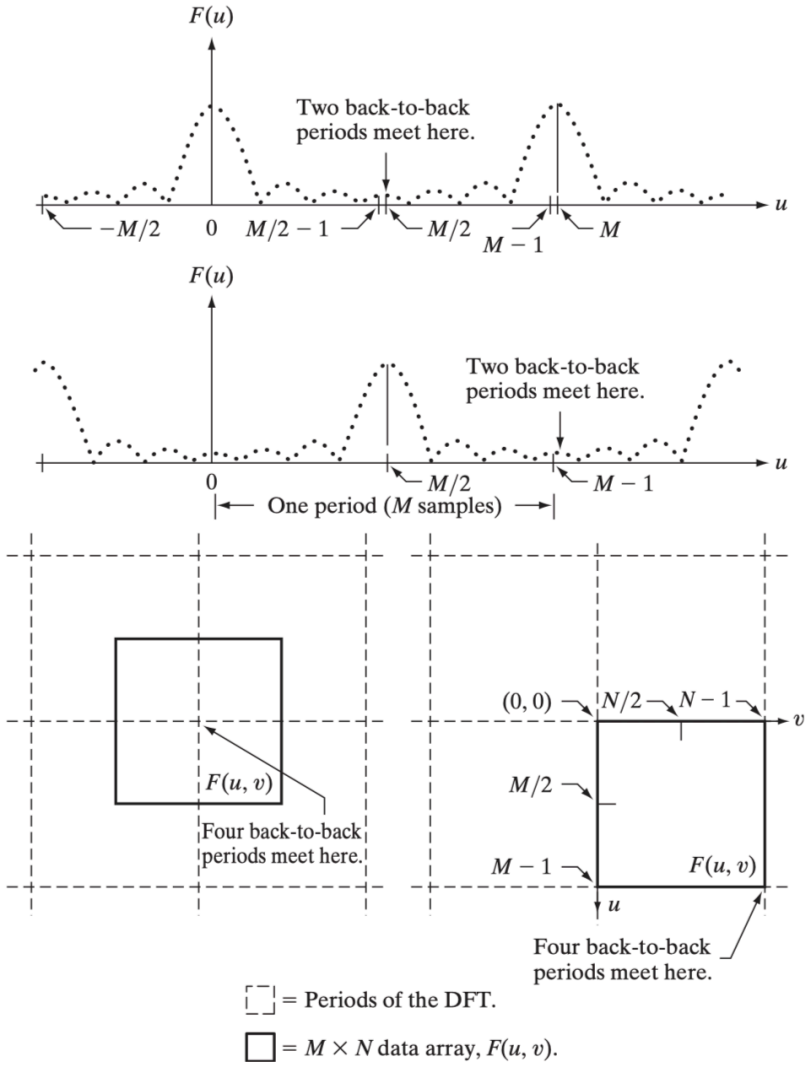


Figure 1. Centering the Fourier transform.

To avoid the wraparound error and for the simplicity to designing the filter kernel, we must pad the image with the circumstance below:

$$P > A + C - 1$$

and

$$Q > B + D - 1$$

with

$$P > 2M - 1$$

and

$$Q > 2N - 1$$

while P and Q is the width and the length of the array after zero padding.

After the shift, the image is also required to multiplied by $(-1)^{x+y}$, which mainly helps visualising the the filtering process.

2.6 Real and Symmetric of the Filter

As the single frequency component of Fourier transform of the 2D discrete image array is symmetric, to filter those frequency component, a symmetric filter is also needed when doing the frequency filtering. Besides, as the frequency filter $H(\mu, v)$ only change the intensity of the frequency component, but not change the phase of the frequency component itself, the filter must also be real.

2.7 Process

Conclusively, the whole process of Fourier Transform on discrete 2D image array and apply the filter to its time domain can be summarized as follows [2]:

1. Given an input image $f(x, y)$ of size $M \times N$, obtain the padding parameters P and Q. Typically, P = 2M and Q = 2N.
2. Form a padded image, $f_p(x, y)$ of size $P \times Q$ by appending the necessary number of zeros to $f(x, y)$
3. Multiply $f_p(x, y)$ by $(-1)^{x+y}$ to center its transform note: as mentioned in property 2, centering helps in visualizing the filtering process and in generating the filter functions themselves, but centering is not a fundamental requirement.
4. Compute the DFT, F(u,v) of the image from step 3
5. Generate a real, symmetric filter function, $H(u, v)$, of size $P \times Q$ with center at coordinates $(P/2, Q/2)$
6. Form the product $G(u, v) = H(u, v)F(u, v)$ using array multiplication
7. Obtain the processed image

$$g_p(x, y) = \{real[\Im^-1[G(u, v)]]\}(-1)^{x+y}$$

8. Obtain the final processed result, $g(x, y)$, by extracting the $M \times N$ region from the top, left quadrant of $g_p(x, y)$

3 Sobel Filter

The sobel filter are mainly used for image edge detection, whose kernel could enhance the edge part of the image. The kernel of the Sobel filter can be written as follows:

$$\mathbf{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \tag{12}$$

$$\mathbf{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{13}$$

To detect the edge on both vertical and horizontal direction, we needs to add these two kernel together as the new kernel for filtering.

3.1 Process Flow

Algorithm 1: Apply Sobel Filter on the image

1. load the image and convert into numpy array.
2. load the kernel and convert into numpy array.
3. Pad the input image and kernel.
4. Multiply $f_p(x, y)$ by $(-1)^{x+y}$ to center its transform.
5. Use np.fft.fft2d to apply the fourier transformation on the image and kernel.
6. Multiply the image and kernel in spectrum domain.
7. Use np.fft.ifft2d to transform the result back to spacial domain.
8. Save the output image.

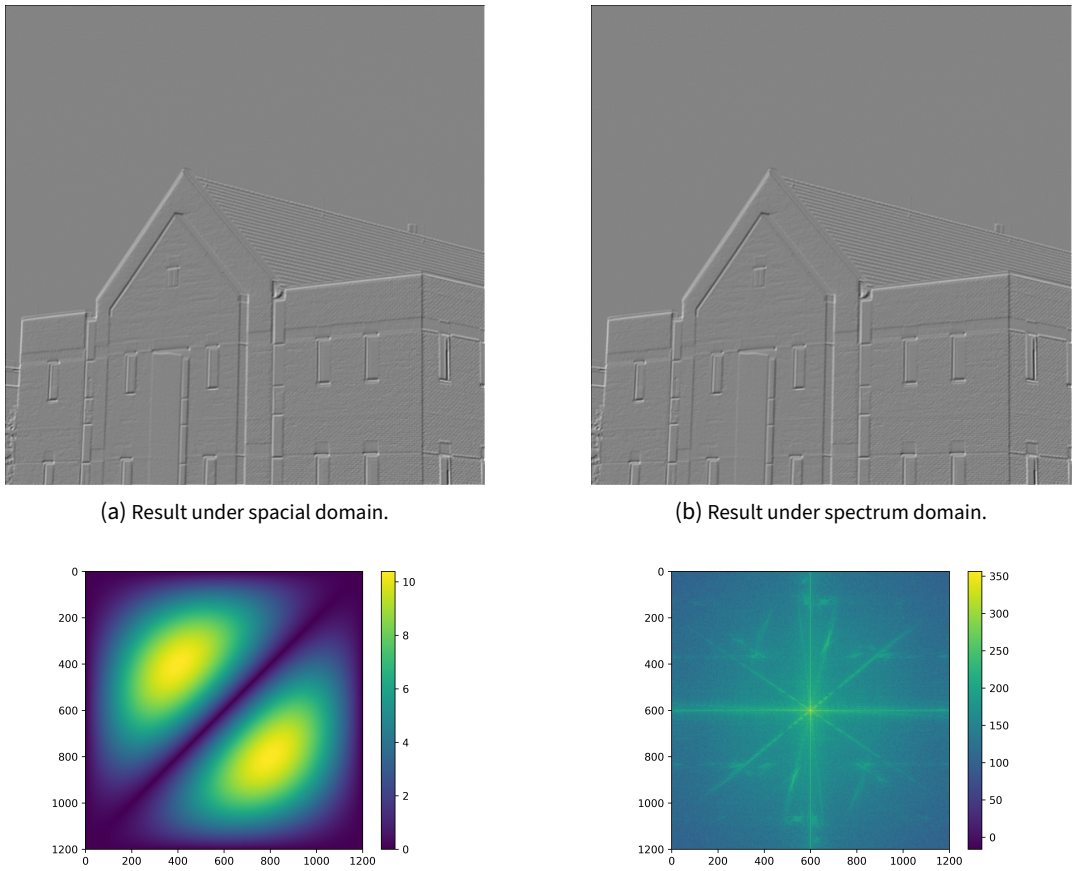
3.2 Python Code

```python
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from joblib import Parallel, delayed
import time
import lab5_lib


def conv(input_img, kernel_flat):
    row, col = input_img.shape

    def pad_with(vector, pad_width, iaxis, kwargs):
        pad_value = kwargs.get('padder', 0)
        vector[:pad_width[0]] = pad_value
        vector[-pad_width[1]:] = pad_value

    input_img_padding = np.pad(input_img, 1, pad_with, padder=0)

    # def par func
    def single_conv(i, j):
        im_arr = input_img_padding[(i - 1):(i + 2), (j - 1):(j +
        ↪  2)].flatten()
        return np.sum(im_arr * kernel_flat)

    par_output = np.array(Parallel(n_jobs=6)(delayed(single_conv)(i, j)
                                             for i in range(1, row + 1)
                                             for j in range(1, col + 1)
                                             ))

    Lap = np.reshape(par_output, input_img.shape)
    return Lap


def sobel_11812418(input_img):
    pic_num = 1
    # Reading of the image into numpy array:
    img = Image.open(input_img)
```

```
37
38      # FFT transform for img arr
39      img_arr = np.asarray(img)
40      input_image_origin = np.asarray(img)
41
42      row, col = img_arr.shape
43
44      # FFT Padding
45      img_arr = lab5_lib.FFT_zero_padding(img_arr)
46
47      img_arr = lab5_lib.multiply_center(img_arr)
48
49      img_arr_fft = np.fft.fft2(img_arr)
50
51      plt.imshow(20 * np.log(np.abs((np.fft.fft2(img_arr)))))
52      heatmap = plt.pcolor(20 * np.log(np.abs((np.fft.fft2(img_arr)))))
53      plt.colorbar(heatmap)
54      plt.savefig('output/Q5_1_F.png', dpi=600)
55
56      kernel1 = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
57      kernel2 = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
58      kernel1 = kernel1 + kernel2
59
60      DFT_kernel_1_fft = lab5_lib.DFT_kernel(img_arr, kernel1)
61      DFT_kernel_fft = DFT_kernel_1_fft
62
63      # plt.imshow(np.abs(DFT_kernel_1_fft))
64      # heatmap = plt.pcolor(np.abs(DFT_kernel_1_fft))
65      # plt.colorbar(heatmap)
66      # plt.savefig('output/sobel_kernel.png', dpi=600)
67
68      # K1
69      filtered_1 = np.real(np.fft.ifft2(img_arr_fft * DFT_kernel_fft))
70      filtered_1 = lab5_lib.FFT_extract_padding(filtered_1)
71      filtered_1 = lab5_lib.multiply_center(filtered_1)
72      # plt.imshow(filtered_1)
73      # plt.show()
74
75      # conv
76      # op_img = np.fft.ifft2(conv(img_arr, kernel_flat))
77      # print(filtered_1)
78      op_img = lab5_lib.normalize(filtered_1) * 255
79
80      plt.imshow(op_img)
81      plt.show()
82
83      op_img_spacial = conv(input_image_origin, np.flip(kernel1.flatten()))
84      op_img_spacial = lab5_lib.normalize(op_img_spacial) * 255
85
```

```
86        return op_img, op_img_spacial
87
88
89    start_time = time.time()
90    A, A_S = sobel_11812418("Q5_1.tif")
91    op_image = Image.fromarray(A.astype(np.uint8))
92    op_image_S = Image.fromarray(A_S.astype(np.uint8))
93    print("--- %s seconds ---" % (time.time() - start_time))
94    op_image.save("output/Q5_1_M_Frequency.tif")
95    op_image_S.save("output/Q5_1_M_Spatial.tif")
96    op_image.save("output/Q5_1_M_Frequency.png")
97    op_image_S.save("output/Q5_1_M_Spatial.png")
```

3.3 Result



(a) Result under spacial domain.

(b) Result under spectrum domain.

(c) Sobel Filter

(d) Time(spectrum) domain of the image.

Figure 2. Process of Sobel Filter.

From the figure 2a and 2b, we could find that the result of filtering the image with the spacial filter and the spectrum domain could are the same. The reason for this phenomenon is as mentioned in the previous chapter by equation 9 and 10, while correlating in the spacial domain is equal to convolution in spectrum domain.

We also noticed that if the kernel is flipped, the unevenness of the result will also be flipped.

4 Gaussian Filter

The key of this section is to generate a Gaussian function in the spectrum domain. The Gaussian function is listed below:

$$H(u,v) = e^{-D^2(u,v)/2\sigma^2} \tag{14}$$

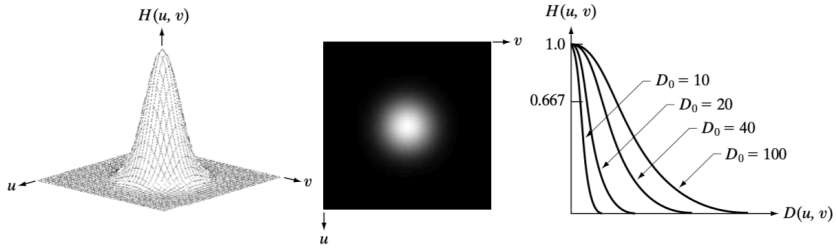


Figure 3. Diagram of Gaussian lowpass filter.

4.1 Process Flow

Algorithm 2: Apply Gaussian Filter on the image

1. load the image and convert into numpy array.
2. Generate the Gaussian function with the width and height of the image.
3. Pad the input image.
4. Multiply $f_p(x,y)$ by $(-1)^{x+y}$ to center its transform.
5. Use np.fft.fft2d to apply the Fourier transformation on the image and kernel.
6. Multiply the image and Gaussian filter in spectrum domain.
7. Use np.fft.ifft2d to transform the result back to spacial domain.
8. Save the output image.

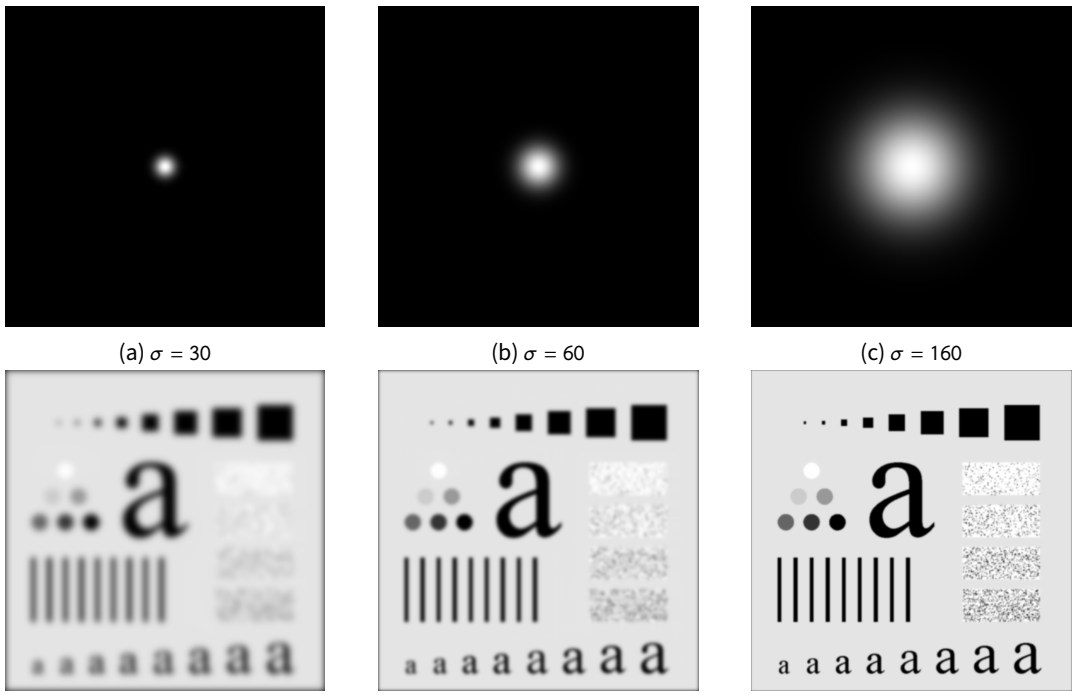
4.2 Python Code

```python
import time
import numpy as np
from PIL import Image
import lab5_lib


def gaussian_pass_11812418(input_img, sigma):
    # Reading of the image into numpy array:
    img = Image.open(input_img)
    # FFT transform for img arr
    input_image_origin = np.asarray(img)
    input_image = np.asarray(img)

    input_image = lab5_lib.FFT_zero_padding(input_image)
    row, col = input_image.shape

```

```
    input_image = np.fft.fft2(input_image)
    input_image = np.fft.fftshift(input_image)
    # define gaussian
    gaussian_filter_HP = lab5_lib.generate_gaussian(row, col, sigma)
    gaussian_filter_LP = np.ones([row, col]) - lab5_lib.generate_gaussian(row, col, sigma)
    # print(gaussian_filter)
    # plt.imshow(gaussian_filter)
    # plt.show()
    filtered_img_HP = np.multiply(input_image, gaussian_filter_HP)
    output_img_HP = np.fft.ifft2(np.fft.ifftshift(filtered_img_HP))
    output_img_HP = lab5_lib.transform_centering(output_img_HP)
    filtered_img_LP = np.multiply(input_image, gaussian_filter_LP)
    output_img_LP = np.fft.ifft2(np.fft.ifftshift(filtered_img_LP))
    output_img_LP = lab5_lib.transform_centering(output_img_LP)
    # print(np.min(np.abs(output_img_HP)))
    # output_img_HP = lab5_lib.normalize(output_img_HP)
    # plt.imshow(np.real(output_img_HP))
    # plt.show()

    return np.abs(output_img_HP), np.abs(output_img_LP), lab5_lib.normalize(
        gaussian_filter_LP) * 255, lab5_lib.normalize(gaussian_filter_HP) * 255


if __name__ == '__main__':
    start_time = time.time()
    for i in [30, 60, 160]:
        A_LP, A_HP, G_HP, G_LP = gaussian_pass_11812418("Q5_2.tif", i)
        op_image_LP = Image.fromarray(A_LP.astype(np.uint8))
        op_image_LP.save("output/Q5_2_LP" + str(i) + ".tif")
        op_image_HP = Image.fromarray(A_HP.astype(np.uint8))
        op_image_HP.save("output/Q5_2_HP" + str(i) + ".tif")
        op_image_GL = Image.fromarray(G_LP.astype(np.uint8))
        op_image_GL.save("output/Q5_2_LP" + str(i) + "_F.tif")
        op_image_GH = Image.fromarray(G_HP.astype(np.uint8))
        op_image_GH.save("output/Q5_2_HP" + str(i) + "_F.tif")
    print("--- %s seconds ---" % (time.time() - start_time))
    for i in [30, 60, 160]:
        A_LP, A_HP, G_HP, G_LP = gaussian_pass_11812418("Q5_2.tif", i)
        op_image_LP = Image.fromarray(A_LP.astype(np.uint8))
        op_image_LP.save("output/Q5_2_LP" + str(i) + ".png")
        op_image_HP = Image.fromarray(A_HP.astype(np.uint8))
        op_image_HP.save("output/Q5_2_HP" + str(i) + ".png")
        op_image_GL = Image.fromarray(G_LP.astype(np.uint8))
        op_image_GL.save("output/Q5_2_LP" + str(i) + "_F.png")
        op_image_GH = Image.fromarray(G_HP.astype(np.uint8))
        op_image_GH.save("output/Q5_2_HP" + str(i) + "_F.png")
```

4.3 Result

Low Pass Filter



(a) $\sigma = 30$ (b) $\sigma = 60$ (c) $\sigma = 160$

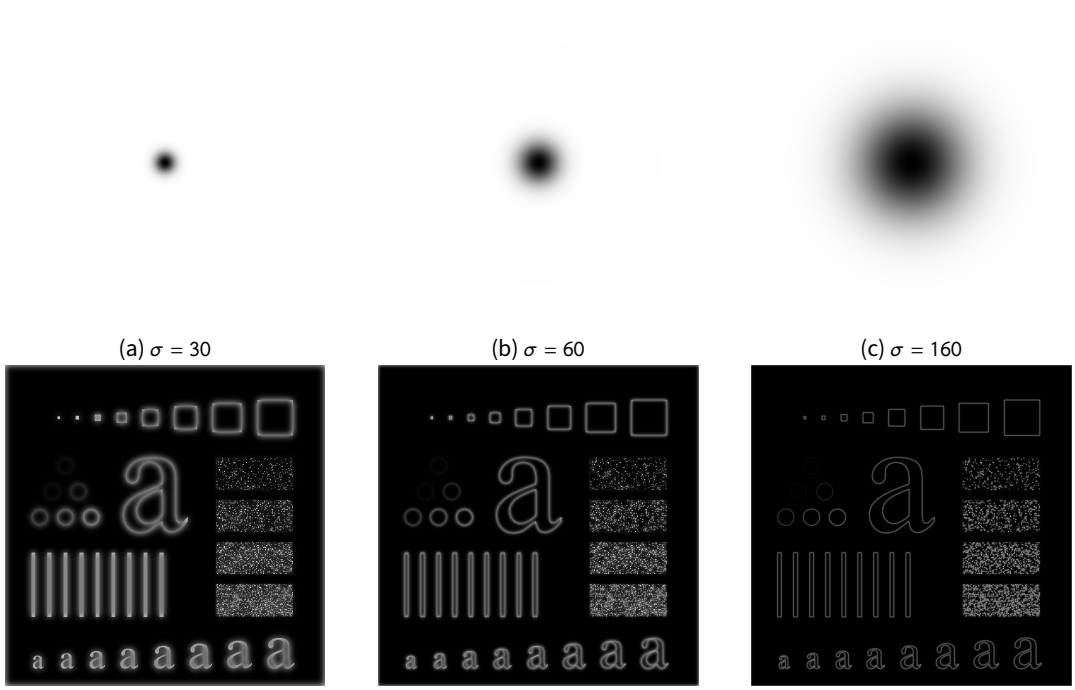
(d) $\sigma = 30$ (e) $\sigma = 60$ (f) $\sigma = 160$

Figure 4. The Gaussian low filter and output images

From the result of low pass Gaussian filter, we could find that with the σ size increases, more high-frequency signals are retained. Intuitively, the image becomes sharper.

High Pass Filter

To get the high pass filter, we subtract the low pass filter from the array filled with one.



(a) $\sigma = 30$ (b) $\sigma = 60$ (c) $\sigma = 160$

(d) $\sigma = 30$ (e) $\sigma = 60$ (f) $\sigma = 160$

Figure 5. The Gaussian highpass filter and output images

From the result of high pass Gaussian filter, we could find that with the σ size increases, less high frequency component has been preserved, which make the edges of the pattern thinner.

Conclusively, we could say that while using the high pass filter, larger σ could bring us a thinner edge, which may useful in edge detection with higher accuracy requirements. Contemporary, while using the low pass filter, larger σ could bring an more mosaic image, which could be useful in photography post-processing.

5 Butterworth Notch Filters

The difference between this section and the Gaussian Filter section is the type of the filter kernel. The Butterworth Notch filters can be written as:

$$H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \tag{15}$$

Comparing with the Gaussian filter, Butterworth Notch filter an extra parameter to adjust, the order of the function. With larger n, or higher order, the more clear the high frequency component will be filtered. In short, the larger the n, the better the selectivity of the filter.

5.1 Process Flow

```
1  import numpy as np
2  from PIL import Image
3  import time
4  import lab5_lib
5
6
```

Algorithm 3: Apply Gaussian Filter on the image

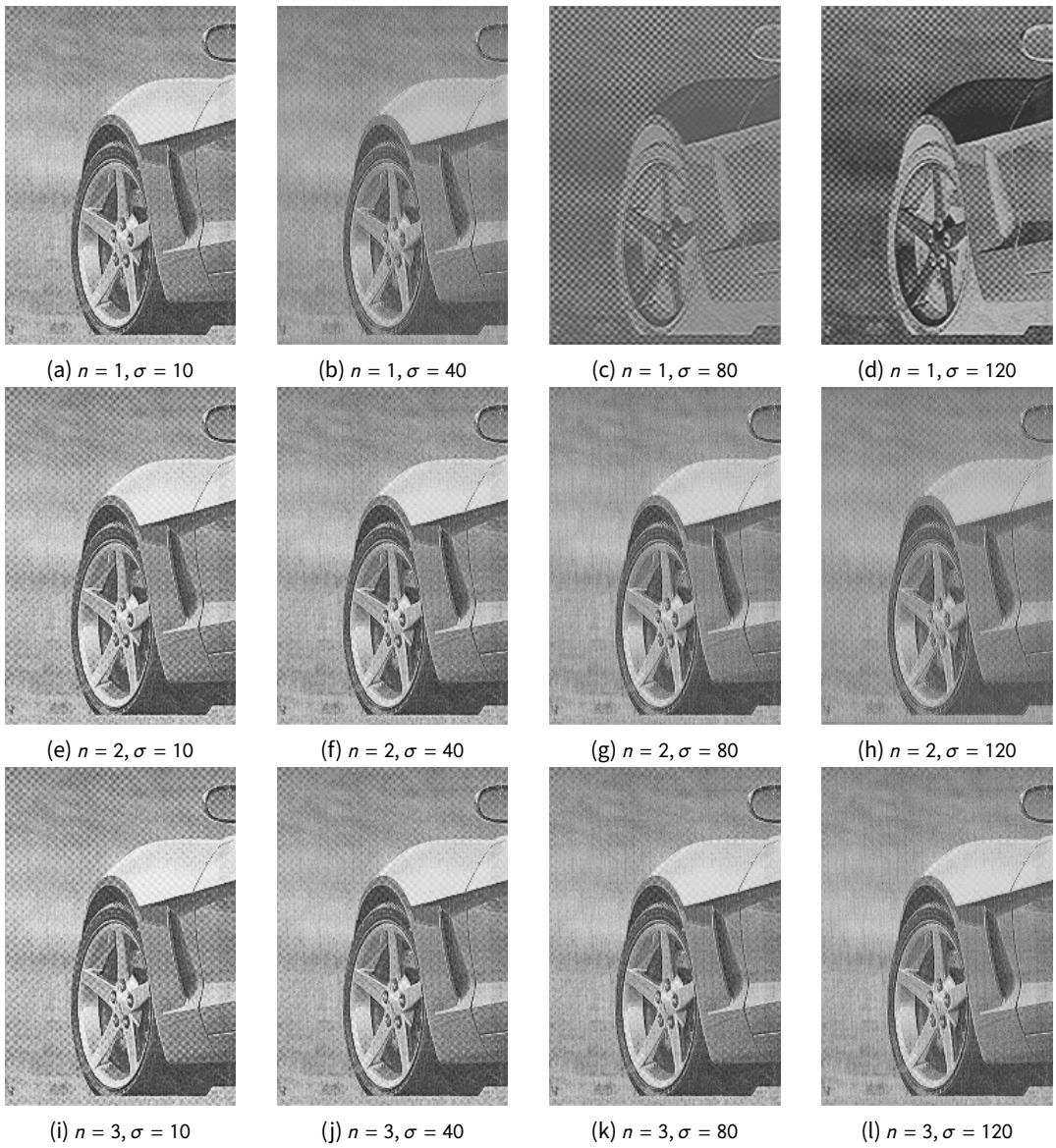
1 load the image and convert into numpy array.
2 Generate the Butterworth filter with the width and height of the image.
3 Pad the input image.
4 Multiply $f_p(x, y)$ by $(-1)^{x+y}$ to center its transform.
5 Use np.fft.fft2d to apply the Fourier transformation on the image and kernel.
6 Multiply the image and Gaussian filter in spectrum domain.
7 Use np.fft.ifft2d to transform the result back to spacial domain.
8 Save the output image.

```
def butterworse_pass_11812418(input_img, sigma, n):
    # Reading of the image into numpy array:
    img = Image.open(input_img)
    # FFT transform for img arr
    input_image = np.asarray(img)

    input_image = lab5_lib.FFT_zero_padding(input_image)
    row, col = input_image.shape

    input_image = np.fft.fft2(input_image)
    input_image = np.fft.fftshift(input_image)
    # 492,336

    # define butterworth
    butterworth_filter = np.ones([row, col])
    # = 1-lab5_lib.generate_butterworth(40, 40, 2, sigma)
    centers = [
        [109, 87],
        [109, 170],
        [115, 330],
        [115, 412],
        [227, 405],
        [227, 325],
        [223, 162],
        [223, 79]
    ]
    for point in centers:
        butterworth_filter = butterworth_filter -
        ↪ lab5_lib.generate_butterworth(row, col, n, sigma, point[0],
        ↪ point[1])

    input_image = np.multiply(input_image, butterworth_filter)

    # plt.imshow(np.log(np.abs(input_image)))
    # plt.show()

    # plt.imshow(input_image)
    # plt.show()
    filtered_img = input_image
```

```
44      # filtered_img = np.multiply(input_image, butterworth_filter)
45      output_img = np.fft.ifft2(np.fft.ifftshift(filtered_img))
46      output_img = lab5_lib.transform_centering(output_img)
47      output_img = lab5_lib.normalize(output_img) * 255
48      # plt.imshow(np.real(output_img))
49      # plt.show()
50
51      return np.abs(output_img), np.log(np.abs(input_image))
52
53
54  if __name__ == '__main__':
55      start_time = time.time()
56      for sigma in [10, 40, 80, 120]:
57          for n in [1, 2, 3, 4]:
58              A, A_S = butterworse_pass_11812418("Q5_3.tif", sigma, n)
59              op_image = Image.fromarray(A.astype(np.uint8))
60              op_image.save("output/Q5_3_M_S" + str(sigma) + "_N" + str(n) +
                ↪  ".png")
61              # plt.imshow(A_S)
62              # plt.axis('off')
63              # plt.savefig("output/Q5_3_M_F" + str(sigma) + "_N" + str(n) +
                ↪  ".png", bbox_inches='tight', pad_inches=0,
64              #             dpi=150)
65      print("--- %s seconds ---" % (time.time() - start_time))
```

5.2 Result

Spacial Domain



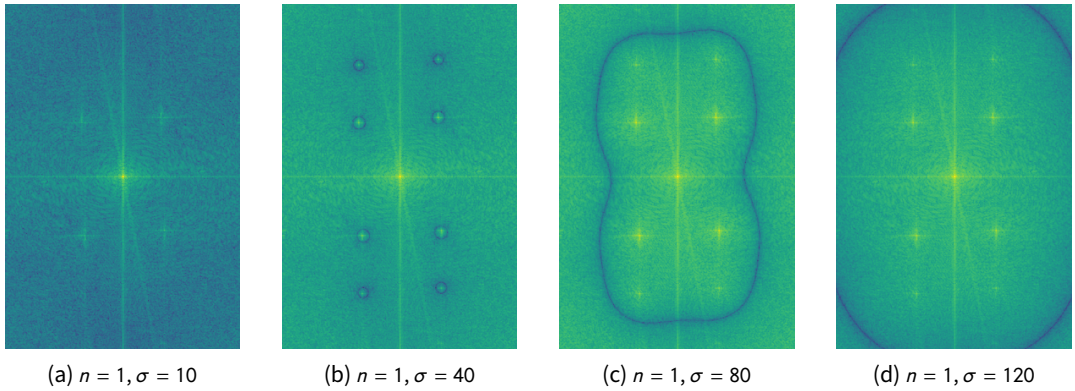
(a) $n = 1, \sigma = 10$ (b) $n = 1, \sigma = 40$ (c) $n = 1, \sigma = 80$ (d) $n = 1, \sigma = 120$

(e) $n = 2, \sigma = 10$ (f) $n = 2, \sigma = 40$ (g) $n = 2, \sigma = 80$ (h) $n = 2, \sigma = 120$

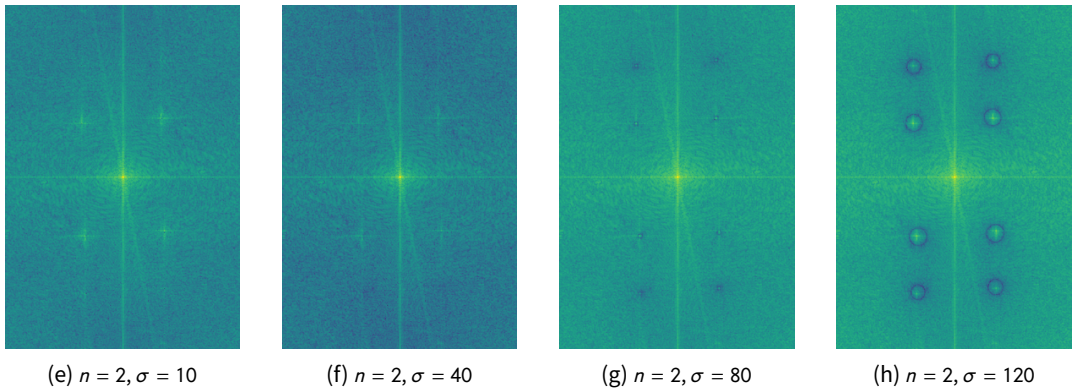
(i) $n = 3, \sigma = 10$ (j) $n = 3, \sigma = 40$ (k) $n = 3, \sigma = 80$ (l) $n = 3, \sigma = 120$

Figure 6. Result after Butterworth Filter (Spacial Domain)

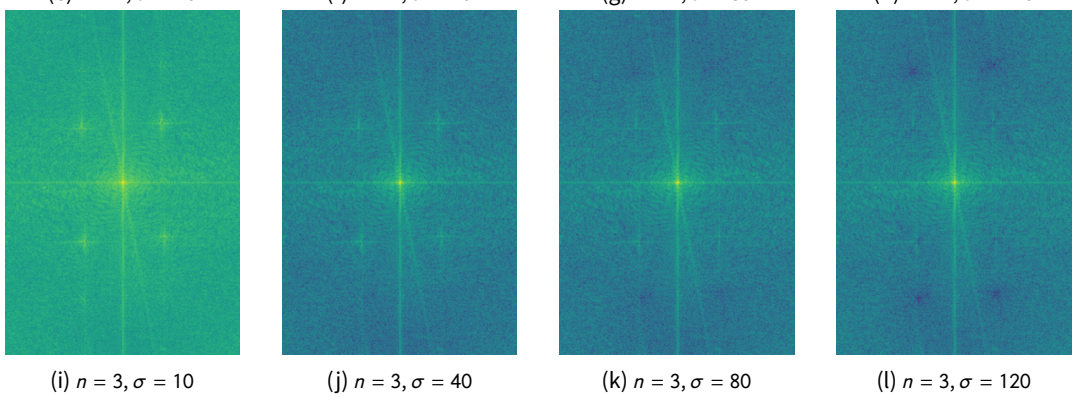
Frequency Domain



(a) $n = 1, \sigma = 10$ (b) $n = 1, \sigma = 40$ (c) $n = 1, \sigma = 80$ (d) $n = 1, \sigma = 120$



(e) $n = 2, \sigma = 10$ (f) $n = 2, \sigma = 40$ (g) $n = 2, \sigma = 80$ (h) $n = 2, \sigma = 120$



(i) $n = 3, \sigma = 10$ (j) $n = 3, \sigma = 40$ (k) $n = 3, \sigma = 80$ (l) $n = 3, \sigma = 120$

Figure 7. Result after Butterworth Filter (Frequency Domain)

From the spacial domain and the frequency domain of the transformed image, we found that with a lower order n, oversized σ will remove too many frequency components on the picture, but also make the picture more blurred while the effect of a Butterworth filter with oversized diameter and low order is similar to that of a Gaussian filter. Therefore, with the increase of the order n, the selectivity of filters has become better, which has a better ability to filter the unwanted signal but not damage other frequency component.

In general, among the images in figure 6, the image with $n = 3, \sigma = 120$ performs the best. Therefore, we can conclude that when we need to remove a certain frequency component accurately, we can use a higher n and a σ that matches the size of the frequency component to remove it.

6 Conclusion

In this lab, we implemented several filter in spatial and frequency domain including the Sobel filter, Gaussian filter and the Butterworth Notch filter. Compare spatial filtering and frequency domain filtering, the frequency filter could deal with more complicated jobs with the price of consuming more time on calculating the Fourier transformation, while the spatial filter could provide us a faster computing speed, but only suitable for relatively limited cases.

Despite this, with the development of the algorithm and the specific instruction set on the matrix multiplication and Fourier transformation, the time difference between running spatial filtering and frequency domain filtering has been greatly reduced. Specifically, the spatial filtering and frequency domain wave of all single pictures in this experiment can be performed within 0.5 seconds. Therefore, with the development of computer performance, frequency domain filtering will be a better choice when performing filtering that requires precise removal of certain frequency components.

7 Appendix1: Python Code of Lab5 Library

In order to modularize the program, we separate some common functions and write them into `lab5_lib.py`

```python
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from joblib import Parallel, delayed
import time


def FFT_zero_padding(input_img):
    row, col = input_img.shape
    output_image = np.zeros([2 * row, 2 * col])
    output_image[0:row, 0:col] = input_img
    return output_image


def FFT_extract_padding(input_img):
    row, col = input_img.shape
    output_image = input_img[int(row / 2):row, int(col / 2):col]
    return output_image


def multiply_center(input_img):
    row, col = input_img.shape
    I, J = np.ogrid[:row, :col]
    mask = np.full((row, col), -1)
    mask[(I + J) % 2 == 0] = 1
    return mask * input_img


def transform_centering(input_img):
    row, col = input_img.shape
```

```
31      return input_img[0:int(row / 2), 0:int(col / 2)]
32
33
34  def DFT_kernel(input_image_padded, kernel):
35      sz = (input_image_padded.shape[0] - kernel.shape[0],
        ↪  input_image_padded.shape[1] - kernel.shape[1])
36      DFT_kernel_1 = np.pad(kernel, (((sz[0] + 1) // 2, sz[0] // 2), ((sz[1]
        ↪  + 1) // 2, sz[1] // 2)), 'constant')
37      DFT_kernel_1 = multiply_center(DFT_kernel_1)
38      DFT_kernel_1_fft = np.fft.fft2(DFT_kernel_1)
39      return DFT_kernel_1_fft
40
41
42  def generate_gaussian(a, b, sigma):
43      x, y = np.meshgrid(np.linspace(0, a-1, a), np.linspace(0, b-1, b))
44      x = x - a/2
45      y = y - b/2
46      d = x * x + y * y
47      g = np.exp(-(d / (2.0 * sigma ** 2)))
48      # g = g/np.sum(g)
49      return g
50
51
52
53  def generate_butterworth(row, col, n, sigma, cr, cc):
54      x, y = np.meshgrid(np.linspace(0, col - 1, col), np.linspace(0, row -
        ↪  1, row))
55      x = x - cr
56      y = y - cc
57      d = np.sqrt(x * x + y * y)
58      h = 1 / ((1 + (d / sigma)) ** (2 * n))
59      return h
60
61
62  def normalize(input_img):
63      return (input_img - np.min(input_img)) / (np.max(input_img) -
        ↪  np.min(input_img))
```

8 Supplementary Information

8.1 Code Repository of EE326 Lab

The source code of this lab can be retrieved from https://github.com/sparkcyf/SUSTech_EE326_Digital_image_Processing/.