

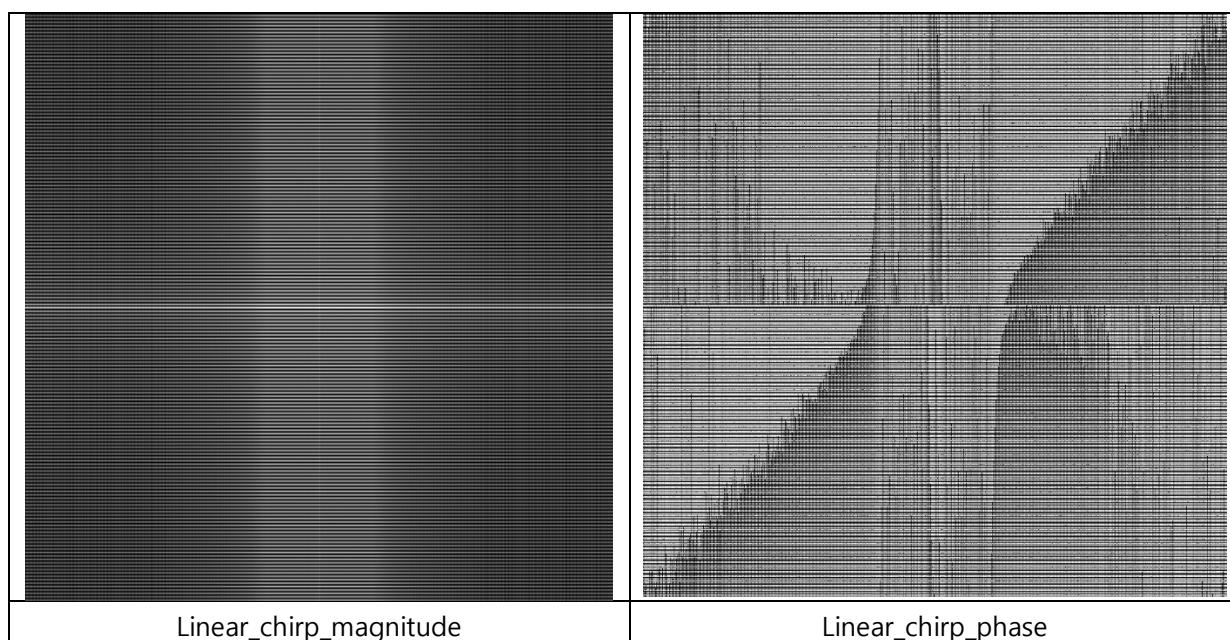
Course Title (과목명)	디지털영상처리개론 (01)
HW Number (HW 번호)	HW03
Submit Date (제출일)	2021-05-20
Grade (학년)	4 th Grade
ID (학번)	20161482
Name (이름)	박준용

Frequency domain filtering of gray scale images

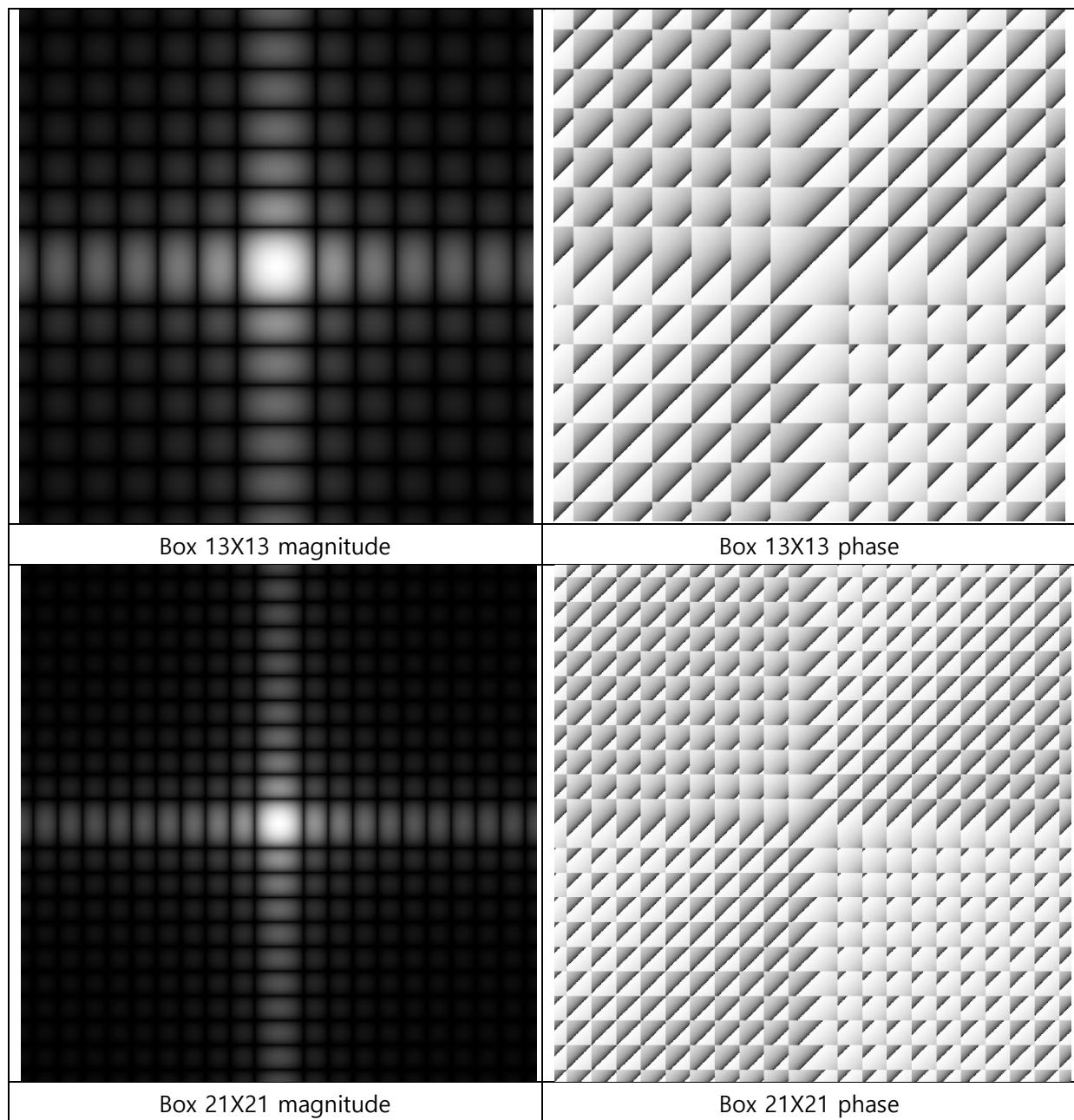
- DFT와 frequency domain kernel을 이용한 frequency domain filtering을 구현
- Spatial kernel과 convolution을 이용한 spatial filtering을 구현
 - ✓ OpenCV의 Mat class method 및 입/출력 함수(imread와 imwrite)만을 사용함.
 - ✓ dft.cpp내에 있는 함수 모두 사용가능.
 - ✓ 제공된 HW03.cpp 및 dft.cpp, utils.h 파일은 수정하지 않고, utils.cpp파일을 작성하여 제출함.

(1) 구현 1

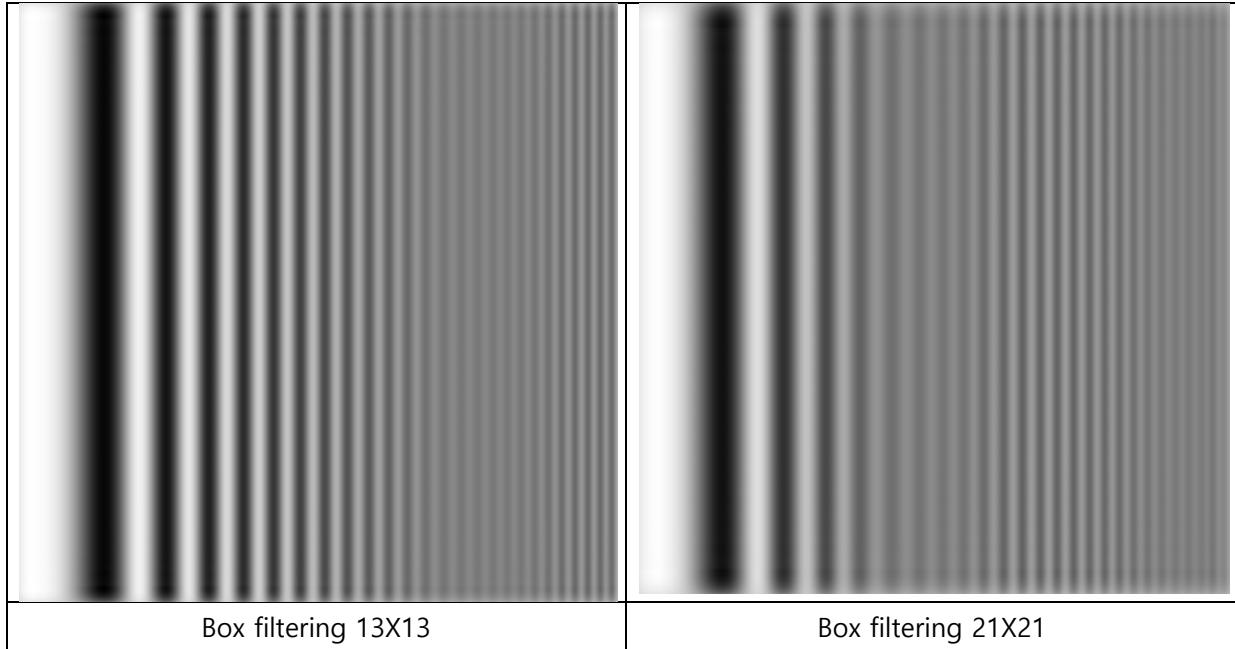
- a) *.zip에 포함된 “**Linear_chirp.tif**”(500×500 크기의 영상)과 **spatial domain**에서 13×13 , 21×21 의 크기를 갖는 **low pass box kernel**이 주어졌을 때, frequency domain에서의 filtering한 결과를 출력하시오. (4th ed.의 **Figure 4.35**의 방법 참조)
- ✓ “**Linear_chirp.tif**”에 대한 **Fourier spectrum**과 **phase**를 영상으로 출력하시오. 이 때, 입력 영상에 zero padding을 수행하여 frequency domain에서 1000×1000 의 크기를 갖도록 하시오.



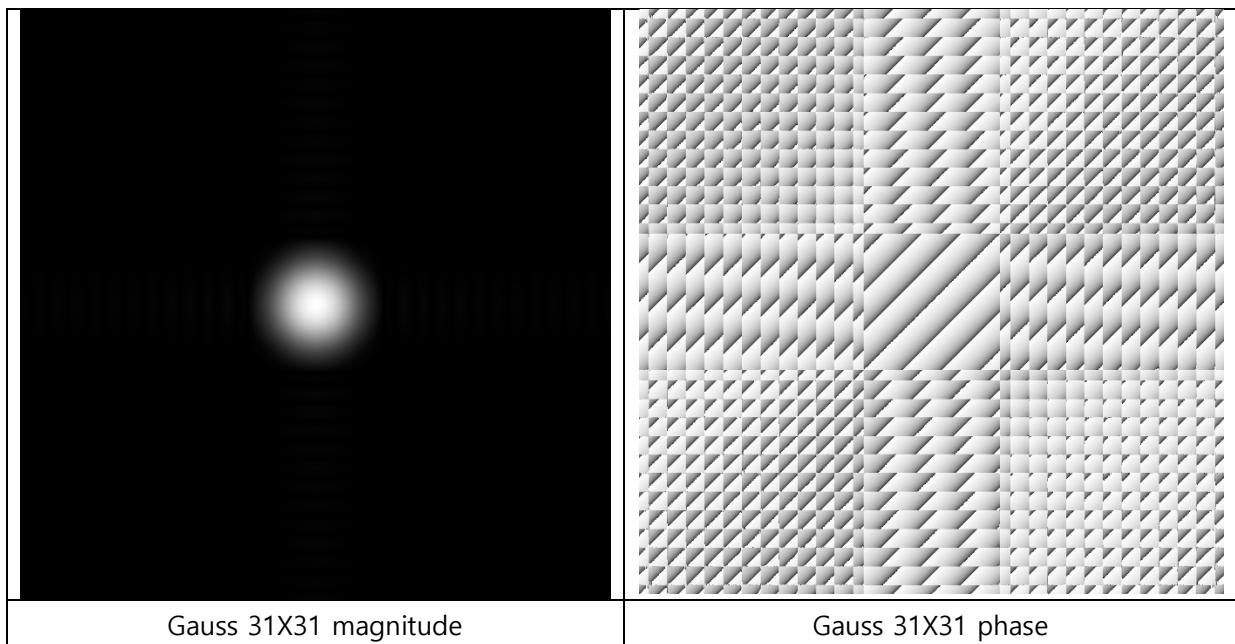
- ✓ 13×13 , 21×21 의 크기를 갖는 low pass box kernel의 Fourier spectrum과 phase를 영상으로 출력하시오.

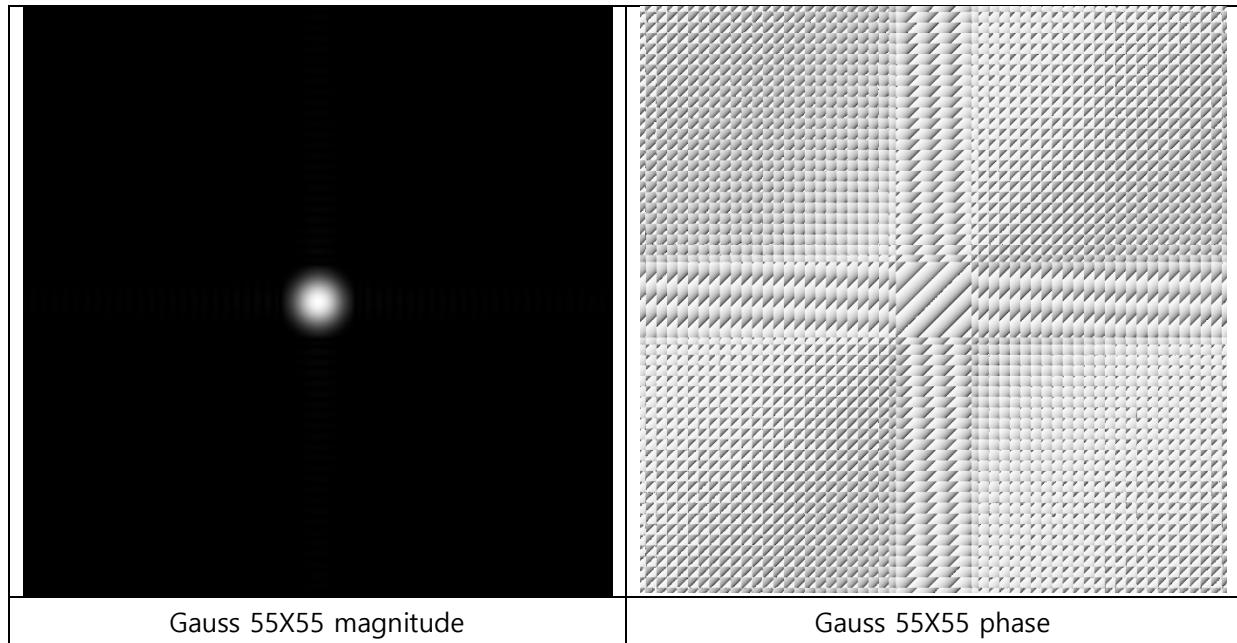


- ✓ “**Linear_chirp.tif**”를 frequency domain에서 filtering한 결과를 출력하시오.

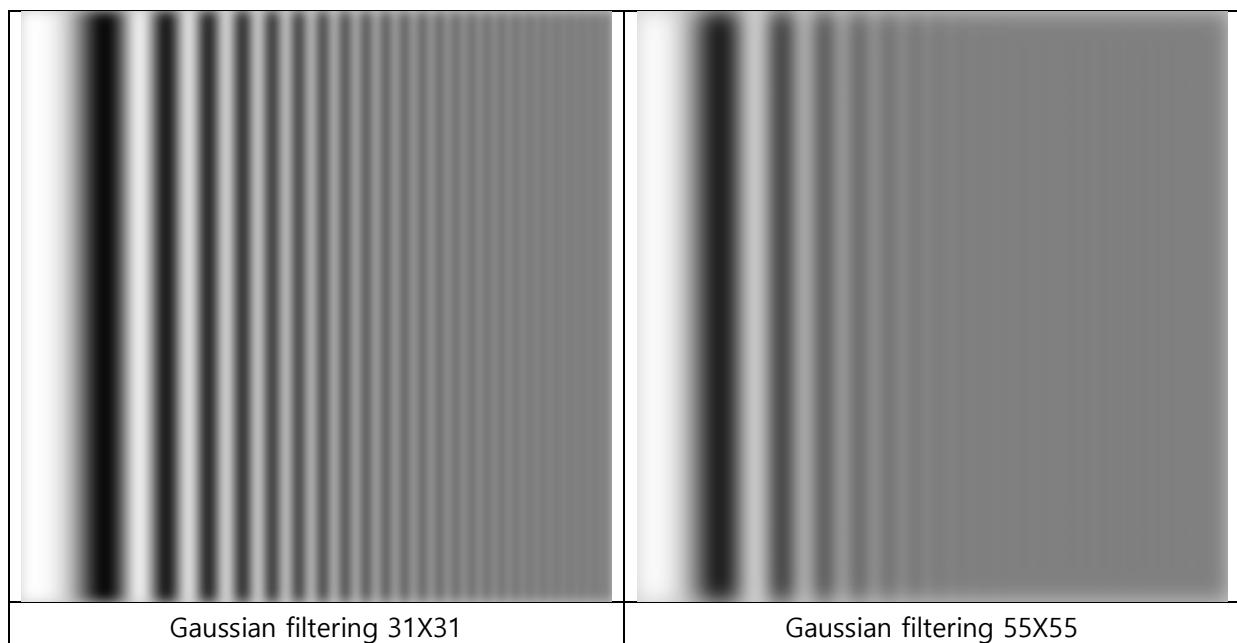


- b) 위 조건에서 box kernel을 spatial domain에서 $31 \times 31(\sigma = 5)$, $55 \times 55(\sigma = 9)$ 의 크기
를 갖는 **low pass Gaussian kernel**로 변경했을 때의, frequency domain에서의
filtering 결과를 출력하시오.
- ✓ $31 \times 31(\sigma = 5)$, $55 \times 55(\sigma = 9)$ 의 크기를 갖는 **low pass Gaussian kernel**의
Fourier spectrum과 **phase**를 영상으로 출력하시오.





- ✓ “**Linear_chirp.tif**”를 frequency domain에서 filtering한 결과를 출력하시오.



- c) 6.2)와 6.3)를 참고하여 freqFilter() 와 freqBoxFilter(), freqGaussFilter()
을 작성하시오.

```
Mat freqFilter(Mat src, Mat kernel) {
    Mat src_padded(src.rows * 2, src.cols * 2, src.type());
    Mat dst(src.rows, src.cols, CV_32F, src.type());
    Mat dst_padded(src.rows * 2, src.cols * 2, CV_32FC2);

    if (src.size() != kernel.size()) {
        Mat kernel_padded(dst_padded.size(), CV_32F, src.type());
    }
}
```

```

        for ( int x = 0; x < kernel.rows; x++) {
            for ( int y = 0; y < kernel.cols; y++) {
                kernel_padded.at<float>(x, y) = kernel.at<float>(x, y);
            }
        }

        for ( int x = 0; x < src.rows; x++) {
            for ( int y = 0; y < src.cols; y++) {
                src_padded.at<uchar>(x, y) = src.at<uchar>(x, y);
            }
        }

src_padded = fft2d(src_padded);
src_padded = fftshift2d(src_padded);
kernel = fftshift2d(kernel);

Mat kernel_mag;
Mat kernel_phase;
Mat tmp[2];
split(kernel, tmp);
magnitude(tmp[0], tmp[1], kernel_mag);
phase(tmp[0], tmp[1], kernel_phase);

Mat src_padded_mag;
Mat src_padded_phase;
split(src_padded, tmp);
magnitude(tmp[0], tmp[1], src_padded_mag);
phase(tmp[0], tmp[1], src_padded_phase);

Mat dst_padded_mag(dst_padded.size(), CV_32F);
Mat dst_padded_phase(dst_padded.size(), CV_32F);
Mat dst_padded_re(dst_padded.size(), CV_32F);
Mat dst_padded_im(dst_padded.size(), CV_32F);

for ( int x = 0; x < dst_padded.rows; x++) {
    for ( int y = 0; y < dst_padded.cols; y++) {
        dst_padded_mag.at<float>(x, y) = src_padded_mag.at<float>(x, y) *
kernel_mag.at<float>(x, y);
        dst_padded_phase.at<float>(x, y) = src_padded_phase.at<float>(x, y);
    }
}

polarToCart(dst_padded_mag, dst_padded_phase, dst_padded_re, dst_padded_im);

for ( int x = 0; x < dst_padded.rows; x++) {
    for ( int y = 0; y < dst_padded.cols; y++) {
        dst_padded.at<Vec2f>(x, y)[0] = dst_padded_re.at<float>(x, y);
        dst_padded.at<Vec2f>(x, y)[1] = dst_padded_im.at<float>(x, y);
    }
}

dst_padded = fftshift2d(dst_padded);
dst_padded = ifft2d(dst_padded);

for ( int x = 0; x < dst.rows; x++) {

```

```

        for ( int y = 0; y < dst.cols; y++) {
            dst.at<float>(x, y) = dst_padded.at<float>(x, y);
        }
    }
    return dst;
}

```

우선, filter를 적용할 때에 사용되는 함수인 freqFilter() 함수를 정의할 수 있다. 함수는 input source와 kernel matrix를 입력으로 받는다.

먼저, kernel의 size와 input source의 사이즈가 다를 경우에는 주파수 domain kernel을 생성하기 위하여 kernel에 zero padding and fft2d를 적용시켜 kernel_padded를 제작한다. 이후, input source를 원하는 크기 지정된 src_padded matrix로 zero padding한 뒤, fft2d와 fftshift2d를 적용한다. 다음으로는 kernel과 input source의 magnitude와 phase를 구하기 위하여, Fourier Transform된 matrix를 openCV에서 제공하는 split 함수를 통하여 분리한 후, magnitude와 phase 함수를 적용하여 각자 분리된 matrix에 해당하는 값을 저장할 수 있다.

이후, output image의 출력을 위하여 magnitude는 input source와 kernel의 magnitude를 곱한 값을, phase는 input source의 phase를 적용하여 출력할 수 있다. 또한, inverse Fourier Transform을 적용하기 위하여 real value와 imaginary value를 구분하여 2채널 이미지의 형태로 dst_padded matrix에 입력한다. (이 때 cartesian 함수로 입력하기 위하여 polarToCart 함수를 사용하였다) 마지막으로, fftshift2d와 ifft2d를 사용하여 원래의 spatial domain으로 되돌린 후 output image를 반환하게 된다.

```

Mat freqBoxFilter(Mat src, Size kernel_size) {

    Mat src_padded(src.rows * 2, src.cols * 2, src.type());
    for ( int x = 0; x < src.rows; x++) {
        for ( int y = 0; y < src.cols; y++) {
            src_padded.at<uchar>(x, y) = src.at<uchar>(x, y);
        }
    }

    Mat src1;
    src1 = fft2d(src_padded);

    fftshift2d(src1, src1);
    Mat temp[2];
    split(src1, temp);

    Mat src_mag;
    Mat src_phase;
    magnitude(temp[0], temp[1], src_mag);
    phase(temp[0], temp[1], src_phase);
    log(1 + src_mag, src_mag);
    normalize(src_mag, src_mag, 0, 1, NORM_MINMAX);
    log(1 + src_phase, src_phase);
    normalize(src_phase, src_phase, 0, 1, NORM_MINMAX);

    imshow("Linear_chirp_mag", src_mag);
    imshow("Linear_chirp_phase", src_phase);
}

```

```

imwrite("Linear_chirp_mag.tif", src_mag);
imwrite("Linear_chirp_phase.tif", src_phase);

Mat padded(src.cols * 2, src.rows * 2, CV_32FC2);

Mat kernel(kernel_size, CV_32F);
for (int x = 0; x < kernel.rows; x++) {
    for (int y = 0; y < kernel.cols; y++) {
        kernel.at<float>(x, y) = 1 / pow(kernel.rows, 2);
    }
}

Mat result(padded.size(), CV_32F, src.type());
for (int x = 0; x < kernel.rows; x++) {
    for (int y = 0; y < kernel.cols; y++) {
        result.at<float>(x, y) = kernel.at<float>(x, y);
    }
}

Mat dst;
dst = fft2d(result);
padded = freqFilter(src, dst);

fftnshift2d(dst, dst);
Mat tmp[2];
split(dst, tmp);

Mat dst_mag;
Mat dst_phase;
magnitude(tmp[0], tmp[1], dst_mag);
phase(tmp[0], tmp[1], dst_phase);
log(1 + dst_mag, dst_mag);
normalize(dst_mag, dst_mag, 0, 1, NORM_MINMAX);
log(1 + dst_phase, dst_phase);
normalize(dst_phase, dst_phase, 0, 1, NORM_MINMAX);

if (kernel.rows == 13) {
    imshow("Box 13X13 kernel", dst_mag);
    imshow("Box 13X13 phase", dst_phase);
    imwrite("Box 13X13 kernel.tif", dst_mag);
    imwrite("Box 13X13 phase.tif", dst_phase);
} if (kernel.rows == 21) {
    imshow("Box 21X21 kernel", dst_mag);
    imshow("Box 21X21 phase", dst_phase);
    imwrite("Box 21X21 kernel.tif", dst_mag);
    imwrite("Box 21X21 phase.tif", dst_phase);
}
return padded;
}

Mat freqGaussFilter(Mat src, Size kernel_size) {
    Mat padded(src.cols * 2, src.rows * 2, CV_32FC2, src.type());
    Mat kernel(kernel_size, CV_32F);
    float sigma = NULL;
    if (kernel.rows == 31) {
        sigma = 5;
    }
}

```

```

}if (kernel.rows == 55) {
    sigma = 9;
}

float k_row1 = (kernel_size.height - 1) / 2;
float k_row2 = kernel_size.height / 2;
float k_col1 = (kernel_size.width - 1) / 2;
float k_col2 = kernel_size.width / 2;

float N = NULL;
for (int i = -k_row1; i <= k_row2; i++) {
    for (int j = -k_col1; j <= k_col2; j++) {
        N += exp(-((i * i + j * j) / (2 * pow(sigma, 2))));
    }
}

for (int x = -k_row1; x <= k_row2; x++) {
    for (int y = -k_col1; y <= k_col2; y++) {
        kernel.at<float>(x + k_row1, y + k_col1) = exp(-(float)(x * x + y * y) / (2 * pow(sigma, 2))) / N;
    }
}

Mat result(padded.size(), CV_32F, src.type());
for (int x = 0; x < kernel.rows; x++) {
    for (int y = 0; y < kernel.cols; y++) {
        result.at<float>(x, y) = kernel.at<float>(x, y);
    }
}

Mat dst;
dst = fft2d(result);
padded = freqFilter(src, dst);

fftshift2d(dst, dst);
Mat tmp[2];
split(dst, tmp);

Mat dst_mag;
Mat dst_phase;
magnitude(tmp[0], tmp[1], dst_mag);
phase(tmp[0], tmp[1], dst_phase);
log(1 + dst_mag, dst_mag);
normalize(dst_mag, dst_mag, 0, 1, NORM_MINMAX);
log(1 + dst_phase, dst_phase);
normalize(dst_phase, dst_phase, 0, 1, NORM_MINMAX);

if (kernel.rows == 31) {
    imshow("Gauss 31X31 kernel", dst_mag);
    imshow("Gauss 31X31 phase", dst_phase);
    imwrite("Gauss 31X31 kernel.tif", dst_mag);
    imwrite("Gauss 31X31 phase.tif", dst_phase);
}if (kernel.rows == 55) {
    imshow("Gauss 55X55 kernel", dst_mag);
    imshow("Gauss 55X55 phase", dst_phase);
    imwrite("Gauss 55X55 kernel.tif", dst_mag);
}

```

```

        imwrite("Gauss 55X55 phase.tif", dst_phase);
    }
    return padded;
}

```

freqBoxFilter() 함수의 경우 input source matrix와 kernel_size를 입력으로 받는다.

$$h(x, y) = \frac{1}{k^2} \left\{ -\left\lfloor \frac{k-1}{2} \right\rfloor \leq x, y \leq \left\lfloor \frac{k}{2} \right\rfloor \right\}$$

위와 같은 식을 code로 구현하여 사용하였으며, 이를 토대로 zero padding 및 filtering한 뒤 fft2d를 사용한 뒤 이를 freqFilter() 함수의 kernel 입력으로 받는다. 또한 Linear Chirp (input source)와 kernel의 Fourier spectrum과 phase를 영상으로 출력하기 위하여 split 함수를 통하여 두 요소를 분리하였으며, 이 때 openCV에서 제공하는 log와 normalize 함수를 이용하여 magnitude와 phase를 구현할 수 있었다. Filter size에 따라 다른 이미지를 출력하고 작성하도록 code를 구체화하였다.

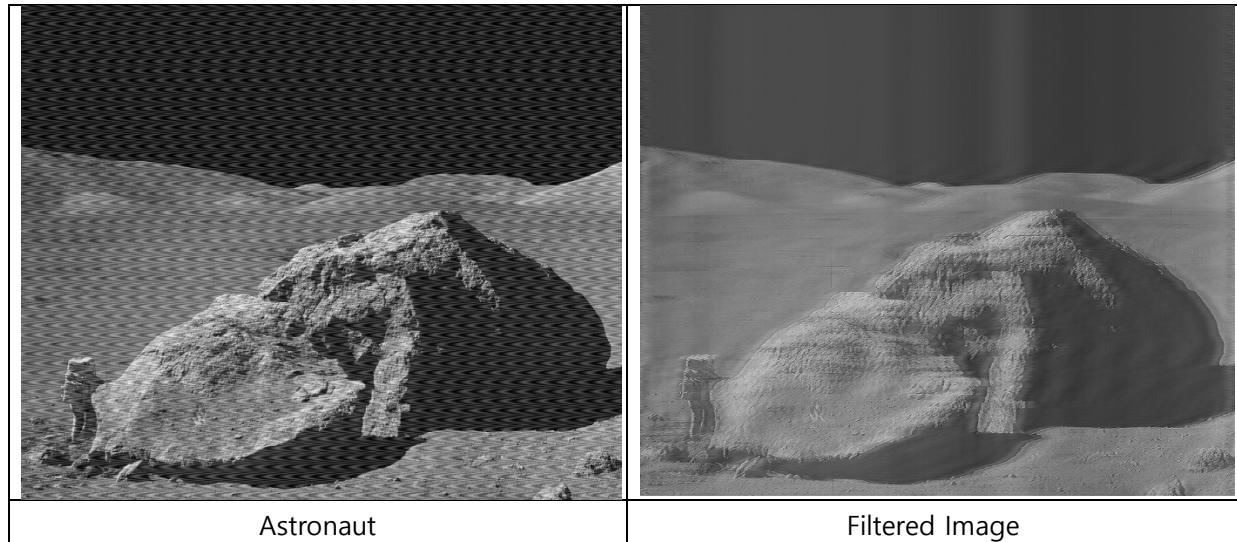
freqBoxFilter() 함수의 경우 input source matrix와 kernel_size를 입력으로 받으며, 아래와 같은 식을 적용하였으며 나머지는 freqBoxFilter() 함수와 동일한 방법으로 작성되었다.

$$h(x, y) = \frac{1}{N} \exp \left(-\frac{x^2 + y^2}{2\sigma^2} \right) \left\{ -\left\lfloor \frac{k-1}{2} \right\rfloor \leq x, y \leq \left\lfloor \frac{k}{2} \right\rfloor \right\}, \quad N = \sum_{i=-\lfloor (k-1)/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{j=-\lfloor (k-1)/2 \rfloor}^{\lfloor k/2 \rfloor} \exp \left(-\frac{i^2 + j^2}{2\sigma^2} \right)$$

이 때 filter size에 따라 sigma값을 조정할 수 있도록 code를 구체화하였다.

(2) 구현 2

- a) *.zip에 포함된 “**Astronaut.tif**”의 노이즈 성분을 효과적으로 제거하기 위해 사용할 수 있는 **Notch filter**를 설계하고 frequency domain에서 filtering한 결과를 출력하시오.



b) 6.3)를 참고하여 NotchFilter()을 작성하시오.

```
Mat NotchFilter(Mat src) {  
  
    Mat Dpk[1000];  
    Mat Dmk[1000];  
    Mat Hpk[1000];  
    Mat Hmk[1000];  
    Mat fliter = Mat::zeros(Sizes, Sizes, CV_32F);  
  
    double N = 0.25;  
    int vk = 30;  
  
    for (int i = 0; i < src.rows; i++) {  
        Dpk[i] = Mat::zeros(Sizes, Sizes, CV_32F);;  
        Dmk[i] = Mat::zeros(Sizes, Sizes, CV_32F);;  
        Hpk[i] = Mat::zeros(Sizes, Sizes, CV_32F);;  
        Hmk[i] = Mat::zeros(Sizes, Sizes, CV_32F);;  
    }  
  
    for (int i = 0; i < 1; i++) {  
        int uk = 0;  
        if (i == 0) {  
            uk = 500;  
        }  
  
        for (int row = 0; row < src.rows; row++) {  
            for (int col = 0; col < src.cols; col++) {  
                int y1 = row - round(Dpk[i].rows / 2) - (uk - 1);  
                int x1 = col - round(Dpk[i].cols / 2) - vk;  
  
                int y2 = row - round(Dpk[i].rows / 2) + uk;  
                int x2 = col - round(Dpk[i].cols / 2) + vk;  
  
                float denom_1 = pow(sqrt(pow(-uk, 2) + pow(-vk, 2)) / sqrt(y1  
* y1 + x1 * x1), N);  
                float denom_2 = pow(sqrt(pow(+uk, 2) + pow(+vk, 2)) / sqrt(y2  
* y2 + x2 * x2), N);  
  
                if (y1 == 0 && x1 == 0)  
                    Hpk[i].at<float>(row, col) = 0;  
                else  
                    Hpk[0].at<float>(row, col) = 1 / (1 + denom_1);  
  
                if (y2 == 0 && x2 == 0)  
                    Hmk[i].at<float>(row, col) = 0;  
                else  
                    Hmk[0].at<float>(row, col) = 1 / (1 + denom_2);  
            }  
        }  
        if (i == 0) {  
            fliter = piecewise(Hpk[0], Hmk[0]);  
        }  
        if (i >= 1) {  
            fliter = piecewise(fliter, (piecewise(Hpk[i], Hmk[i])));  
        }  
    }  
}
```

```

        }

    }

    Mat tmp[4];
    tmp[0] = Mat::zeros(Sizes, Sizes, CV_32F);
    tmp[1] = Mat::zeros(Sizes, Sizes, CV_32F);
    tmp[2] = Mat::zeros(Sizes, Sizes, CV_32F);
    tmp[3] = Mat::zeros(Sizes, Sizes, CV_32F);

    for (int i = 0; i < 5; i++) {
        for (int row = 950; row <= 999; row++) {
            for (int col = 515; col <= 500 + vk - 1; col++) {
                tmp[0].at<float>(row, col) = fliter.at<float>(row, col + 1);
            }
            for (int col = 500 + vk + 1; col <= 550; col++) {
                tmp[1].at<float>(row, col) = fliter.at<float>(row, col - 1);
            }
            for (int col = 515; col <= 500 + vk - 1; col++) {
                fliter.at<float>(row, col) = tmp[0].at<float>(row, col);
            }
            for (int col = 500 + vk + 1; col <= 550; col++) {
                fliter.at<float>(row, col) = tmp[1].at<float>(row, col);
            }
        }
    }
    for (int i = 0; i < 5; i++) {
        for (int row = 0; row < 50; row++) {
            for (int col = 450; col <= 500 - vk - 1; col++) {
                tmp[2].at<float>(row, col) = fliter.at<float>(row, col + 1);
            }
            for (int col = 500 - vk + 1; col <= 485; col++) {
                tmp[3].at<float>(row, col) = fliter.at<float>(row, col - 1);
            }
            for (int col = 450; col <= 500 - vk - 1; col++) {
                fliter.at<float>(row, col) = tmp[2].at<float>(row, col);
            }
            for (int col = 500 - vk + 1; col <= 485; col++) {
                fliter.at<float>(row, col) = tmp[3].at<float>(row, col);
            }
        }
    }
    for (int row = 0; row < src.rows; row++) {
        for (int col = 450; col <= 485; col++) {
            fliter.at<float>(row, col) = fliter.at<float>(0, col);
        }
        for (int col = 515; col <= 550; col++) {
            fliter.at<float>(row, col) = fliter.at<float>(999, col);
        }
    }

    Mat dst = freqFilter_notchkernel(src, fliter);
    float size = -INF;
    for (int row = 0; row < dst.rows; row++) {
        for (int col = 0; col < dst.cols; col++) {
            if (dst.at<float>(row, col) > size)

```

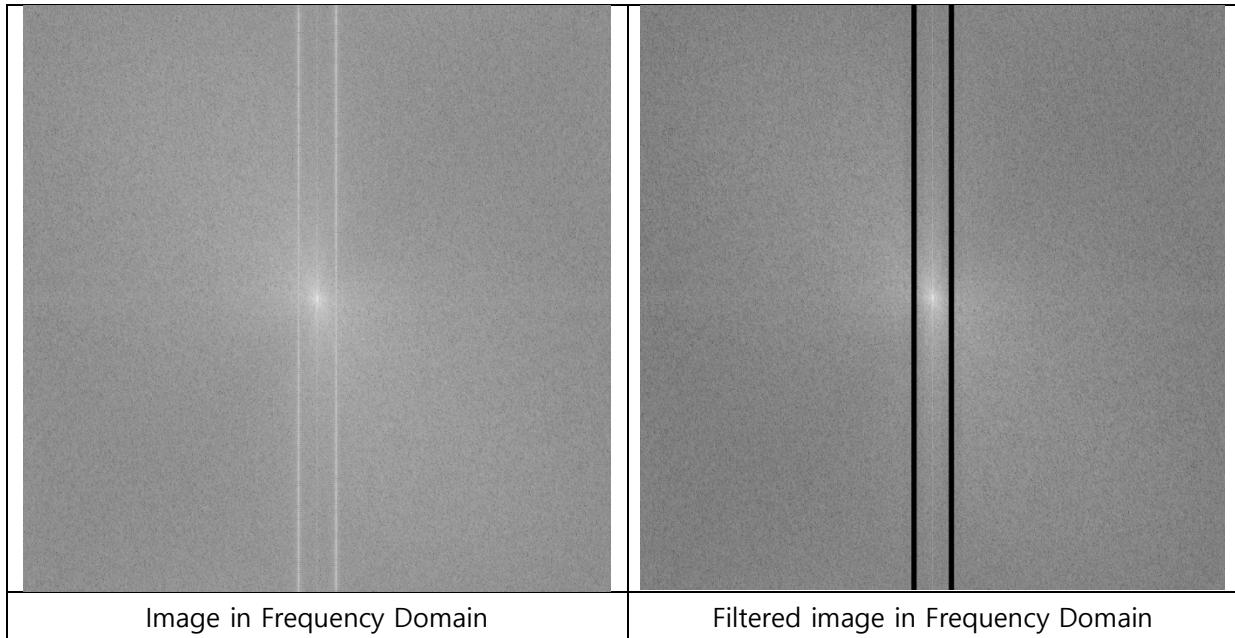
```

        size = dst.at<float>(row, col);
    }
}

Mat result = Mat::zeros(dst.rows, dst.cols, CV_8U);
for (int row = 0; row < dst.rows; row++) {
    for (int col = 0; col < dst.cols; col++) {
        result.at<uchar>(row, col) = (dst.at<float>(row, col) / size) * 255;
    }
}
imshow( "filtering_image.tif", result);
return result;
}

```

우선, NotchFilter() 함수를 작성하기 위하여 NotchFilter에서 사용 가능하도록 수정된 freqFilter_notchkernel 함수와, 두 matrix의 차원이 일치할 때 piecewise multiplication을 수행하는 piecewise 함수, 분할된 real, imaginary factor를 2채널 이미지의 형태로 matrix에 입력하는 merge 함수를 적용하였다.



NotchFilter() 함수는 source image를 입력으로 받아 filtering하게 된다. Filter를 제작하기에 앞서, source image를 Fourier Transform하여 Image를 살펴볼 필요가 있다. 원본 image에 존재하는 noise는 위와 같이 세로 줄무늬의 형태로 frequency domain에 나타나므로, 이를 제거하는 notch filter를 제작할 수 있다.

HPFs whose centers are at (u_k, v_k) and $(-u_k, -v_k)$

$$D_k(u, v) = \sqrt{(u - M/2 - u_k)^2 + (v - N/2 - v_k)^2}$$

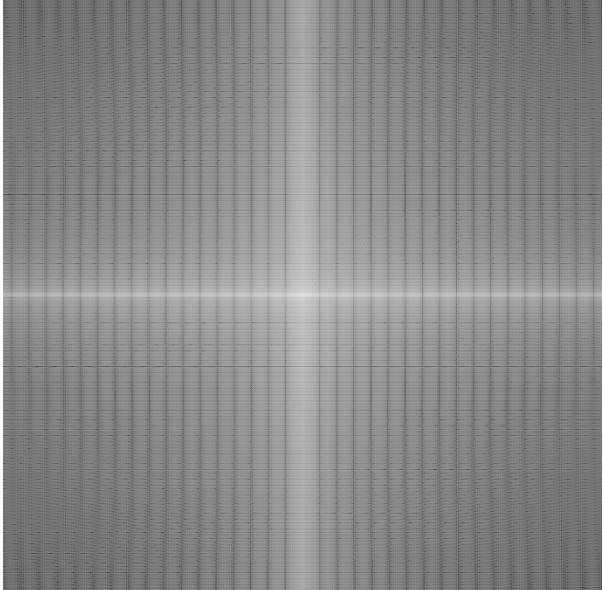
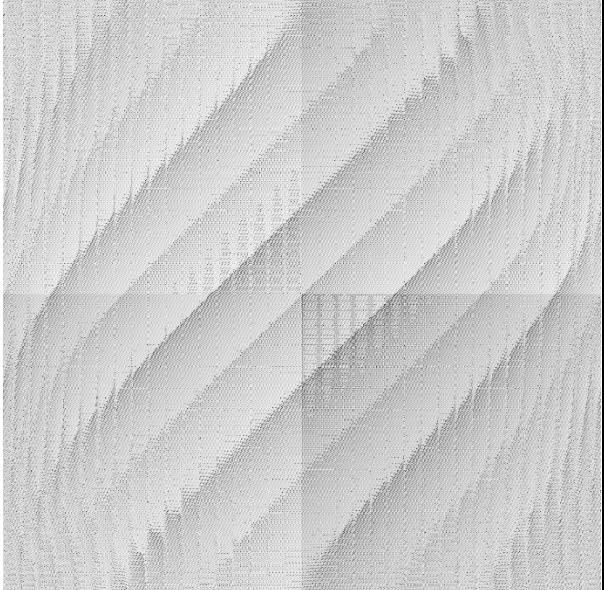
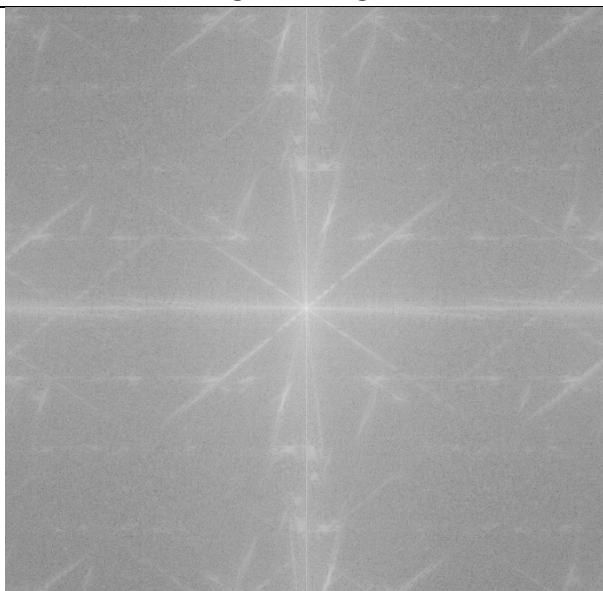
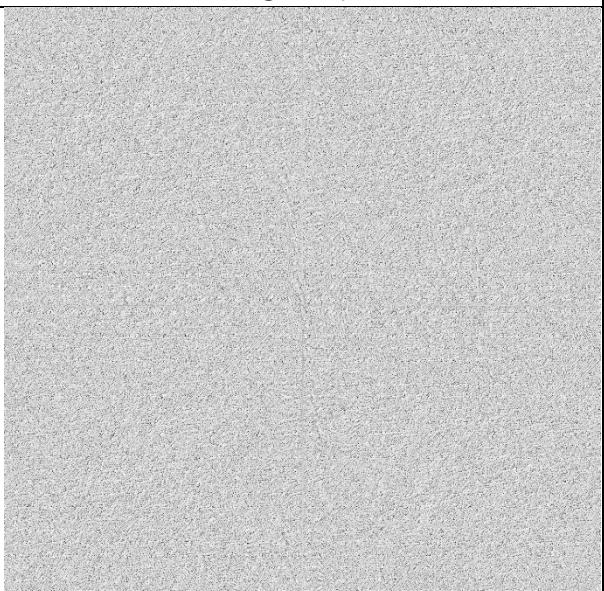
$$D_{-k}(u, v) = \sqrt{(u - M/2 + u_k)^2 + (v - N/2 + v_k)^2}$$

$$H_{NR}(u, v) = \prod_{k=1}^3 \left[\frac{1}{1 + [D_{0k}/D_k(u, v)]^{2n}} \right] \left[\frac{1}{1 + [D_{0k}/D_{-k}(u, v)]^{2n}} \right]$$

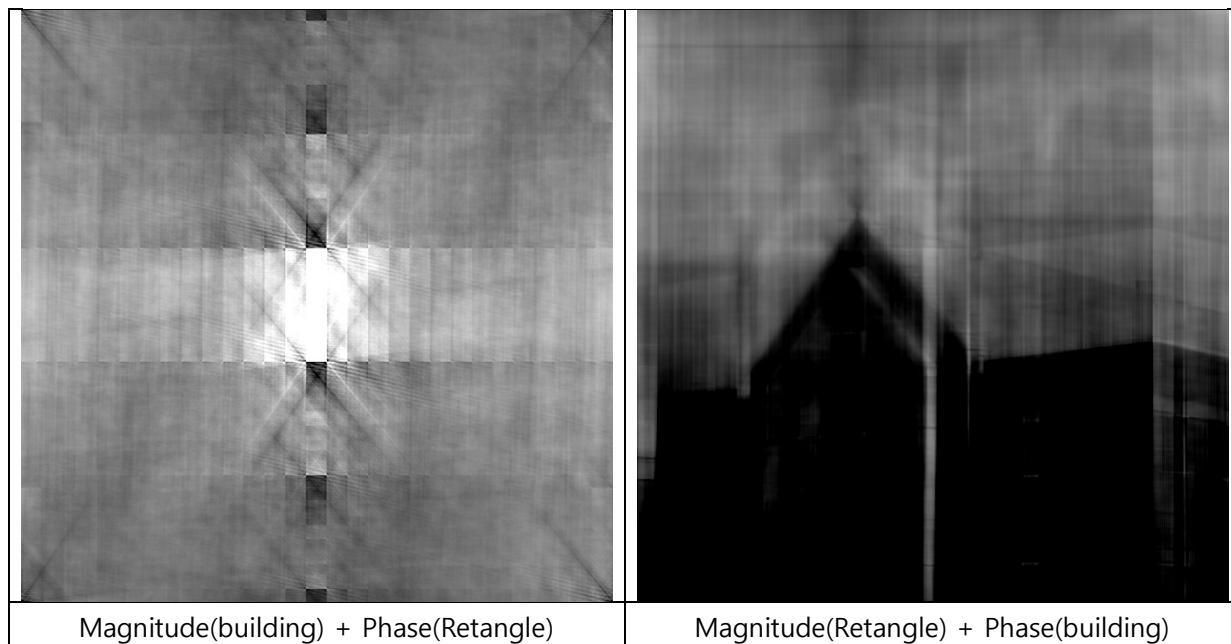
우선, 위의 frequency domain에 존재하는 noise는 width의 중앙인 500px에서 좌우로 30px씩 떨어진 위치에, 모든 height의 범위로 펼쳐져 있다. 이에 HPF의 center가 (0,470) 과 (0,530)인 곳에 위의 수식을 적용하여 HPF를 제작하여 해당하는 곳에 적용한다. (이 때 $2n$ 의 값은 0.25로 설정하였다) 이를 좌우 5px씩 확장하고 상하 길이를 위로 999px만큼 확정하여 notch filter를 제작할 수 있다. 마지막으로 이를 freqFilter_notchkernel 함수를 사용하여 input image와 kernel을 입력으로 하여 filtered된 output을 출력하여 작성할 수 있다.

(3) 구현 3

- a) *.zip에 포함된 “**Rectangle.tif**”와 “**building.tif**”의 **Fourier spectrum**과 **phase**를 영상으로 출력하시오.

	
Rectangle.tif_magnitude	Rectangle.tif_phase
	
building.tif_magnitude	building.tif_phase

- b) “**Rectangle.tif**”의 Fourier spectrum과 “**building.tif**”의 phase angle을 이용하여 영상을 복원하시오. 반대로 “**building.tif**”의 Fourier spectrum과 “**Rectangle.tif**”의 phase angle을 이용하여 영상을 복원하시오. (4th ed.의 EX 4.14의 방법 참조)



- c) 6.3)를 참고하여 `swapPhase()`을 작성하시오.

```

void swapPhase(Mat src1, Mat src2, Mat& dst1, Mat& dst2) {
    Mat src1_padded(src1.rows * 2, src1.cols * 2, src1.type());
    Mat src2_padded(src2.rows * 2, src2.cols * 2, src2.type());

    for (int x = 0; x < src1.rows; x++) {
        for (int y = 0; y < src1.cols; y++) {
            src1_padded.at<uchar>(x, y) = src1.at<uchar>(x, y);
        }
    }
    for (int x = 0; x < src2.rows; x++) {
        for (int y = 0; y < src2.cols; y++) {
            src2_padded.at<uchar>(x, y) = src2.at<uchar>(x, y);
        }
    }

    Mat dft1;
    Mat dft2;

    dft1 = fft2d(src1_padded);
    dft2 = fft2d(src2_padded);

    dft1 = fftshift2d(dft1);
    dft2 = fftshift2d(dft2);

    Mat dft1_re(src1_padded.rows, src1_padded.cols, CV_32F);
    Mat dft1_im(src1_padded.rows, src1_padded.cols, CV_32F);
    Mat src1_mag;
```

```

Mat src1_phase;

Mat dft2_re(src2_padded.rows, src2_padded.cols, CV_32F);
Mat dft2_im(src2_padded.rows, src2_padded.cols, CV_32F);
Mat src2_mag;
Mat src2_phase;

Mat tmp[2];

split(dft1, tmp);
magnitude(tmp[0], tmp[1], src1_mag);
phase(tmp[0], tmp[1], src1_phase);

split(dft2, tmp);
magnitude(tmp[0], tmp[1], src2_mag);
phase(tmp[0], tmp[1], src2_phase);

polarToCart(src1_mag, src2_phase, dft1_re, dft1_im);
polarToCart(src2_mag, src1_phase, dft2_re, dft2_im);

for (int x = 0; x < src1_padded.rows; x++) {
    for (int y = 0; y < src1_padded.cols; y++) {
        dft1.at<Vec2f>(x, y)[0] = dft1_re.at<float>(x, y);
        dft1.at<Vec2f>(x, y)[1] = dft1_im.at<float>(x, y);
    }
}

for (int x = 0; x < src2_padded.rows; x++) {
    for (int y = 0; y < src2_padded.cols; y++) {
        dft2.at<Vec2f>(x, y)[0] = dft2_re.at<float>(x, y);
        dft2.at<Vec2f>(x, y)[1] = dft2_im.at<float>(x, y);
    }
}

Mat dst1_padded(src1_padded.rows, src1_padded.cols, CV_32F);
Mat dst2_padded(src2_padded.rows, src2_padded.cols, CV_32F);

dft1 = fftshift2d(dft1);
dft2 = fftshift2d(dft2);

dst1_padded = ifft2d(dft1);
dst2_padded = ifft2d(dft2);

dst1_padded.convertTo(dst1_padded, CV_8U);
dst2_padded.convertTo(dst2_padded, CV_8U);

Mat dst1_out(src1.rows, src1.cols, src1.type());
Mat dst2_out(src2.rows, src2.cols, src2.type());

for (int x = 0; x < src1.rows; x++) {
    for (int y = 0; y < src1.cols; y++) {
        dst1_out.at<uchar>(x, y) = dst1_padded.at<uchar>(x, y);
    }
}

for (int x = 0; x < src2.rows; x++) {

```

```

        for ( int y = 0; y < src2.cols; y++) {
            dst2_out.at<uchar>(x, y) = dst2_padded.at<uchar>(x, y);
        }
    }
    dst1 = dst1_out;
    dst2 = dst2_out;

    log(1 + src1_mag, src1_mag);
    normalize(src1_mag, src1_mag, 0, 1, NORM_MINMAX);
    log(1 + src1_phase, src1_phase);
    normalize(src1_phase, src1_phase, 0, 1, NORM_MINMAX);

    log(1 + src2_mag, src2_mag);
    normalize(src2_mag, src2_mag, 0, 1, NORM_MINMAX);
    log(1 + src2_phase, src2_phase);
    normalize(src2_phase, src2_phase, 0, 1, NORM_MINMAX);

    imshow("Rectangle.tif_mag", src1_mag);
    imshow("building.tif_mag", src2_mag);
    imshow("Rectangle.tif_phase", src1_phase);
    imshow("building.tif_phase", src2_phase);
    imwrite("Rectangle.tif_mag.tif", src1_mag);
    imwrite("building.tif_mag.tif", src2_mag);
    imwrite("Rectangle.tif_phase.tif", src1_phase);
    imwrite("building.tif_phase.tif", src2_phase);
}

```

우선, swapPhase() 함수는 freqFilter() 함수와 비슷한 방법으로 정의된다. 함수는 두 input source를 입력으로 받으며, 출력 반환을 위한 두 output image 역시 인수로 추가하였다.

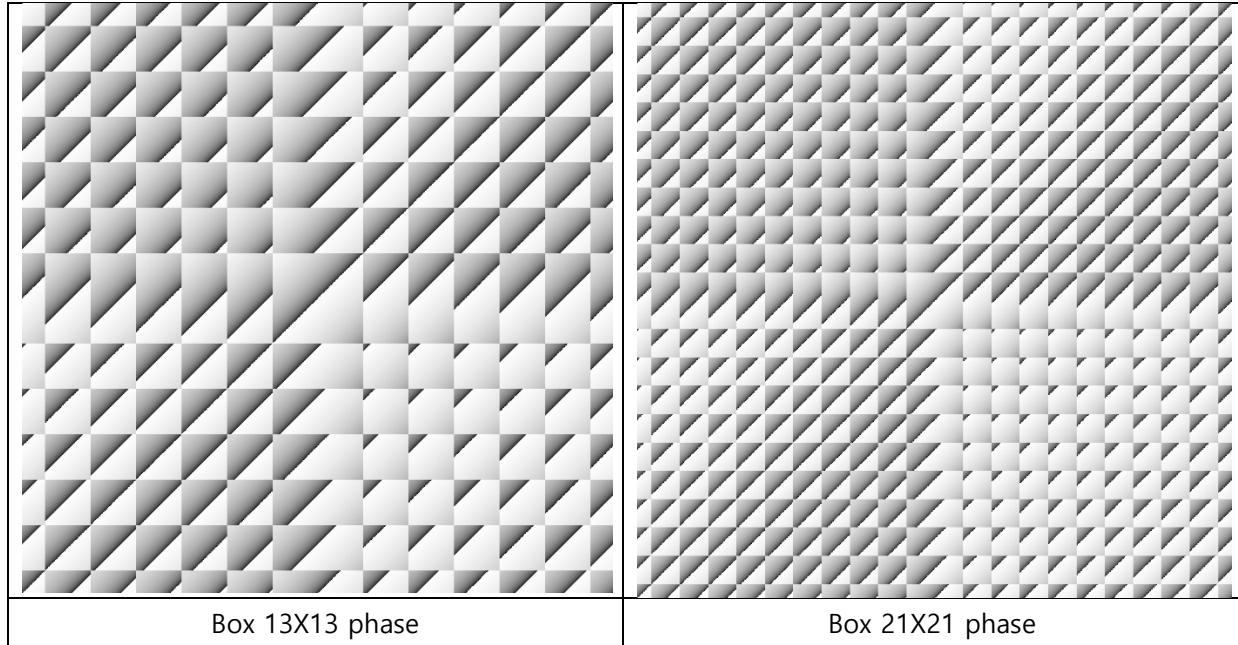
먼저 두 input source를 각각 지정된 크기로 zero padding한 뒤, fft2d와 fftshift2d를 적용한다. 다음으로는 kernel과 input source의 magnitude와 phase를 구하기 위하여, Fourier Transform된 matrix를 openCV에서 제공하는 split 함수를 통하여 분리한 후, magnitude와 phase 함수를 적용하여 각각 분리된 matrix에 해당하는 값을 저장할 수 있다.

이 때, inverse Fourier Transform을 적용하기 위하여 polarToCart 함수를 사용할 때 두 phase를 교차로 적용시켜 적용할 수 있다. 이후, real value와 imaginary value를 구분하여 2채널 이미지의 형태로 dst_padded matrix에 입력한 뒤 fftshift2d와 ifft2d를 사용하여 원래의 domain으로 되돌린 후 output image를 반환하게 된다. 또한, openCV에서 제공하는 log와 normalize 함수를 이용하여 magnitude와 phase를 출력하여 작성할 수 있었다.

4. 검토사항

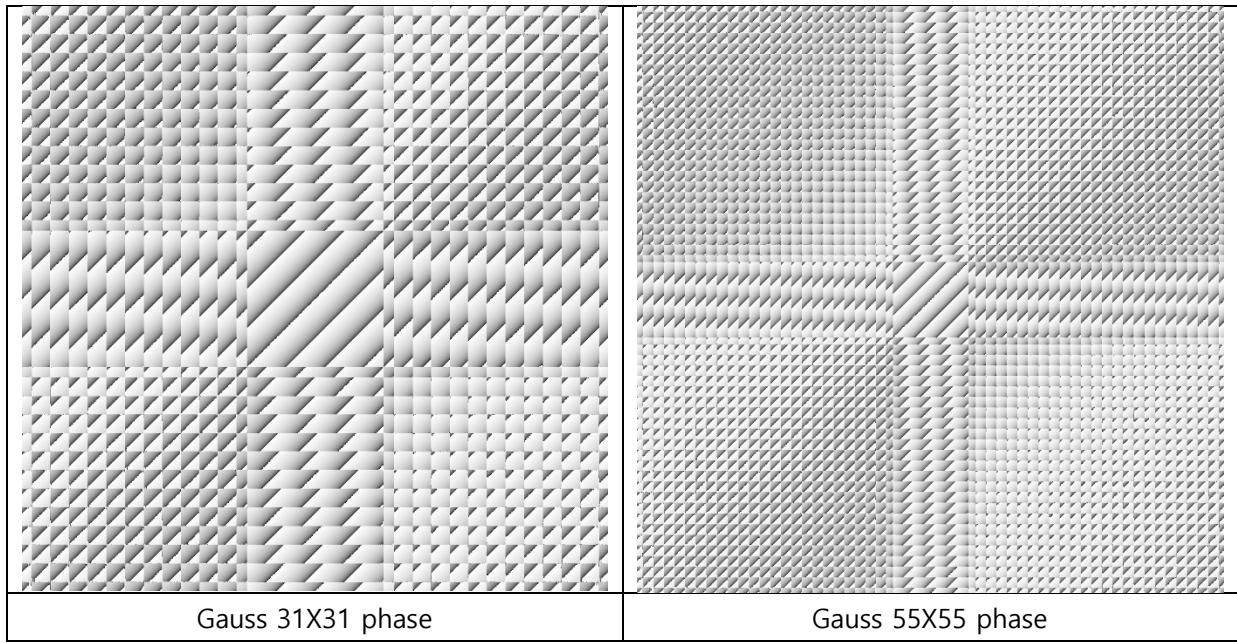
- 1) 구현 1 ~ 3에 대한 결과를 분석하여 보고서에 첨부하시오.
- 2) 구현 1를 통해 **kernel의 size가 커지면 filter의 Fourier spectrum이 어떻게 바뀌는지에 대해 설명하시오. 그리고 box filter와 Gaussian filter의 phase가 어떤 형태로 나오는지 분석하시오.**

우선, kernel의 size가 커지는 경우는 원래의 spatial domain에서 averaging이 더욱 커지는 것과 같으므로, image에 blur가 커지게 된다. 이 경우 frequency domain에서는 낮은 주파수 성분을 pass하므로, LPF에서 pass band의 크기가 감소한다. 따라서, Fourier spectrum의 중앙 low pass 부분의 크기가 작아지게 된다.



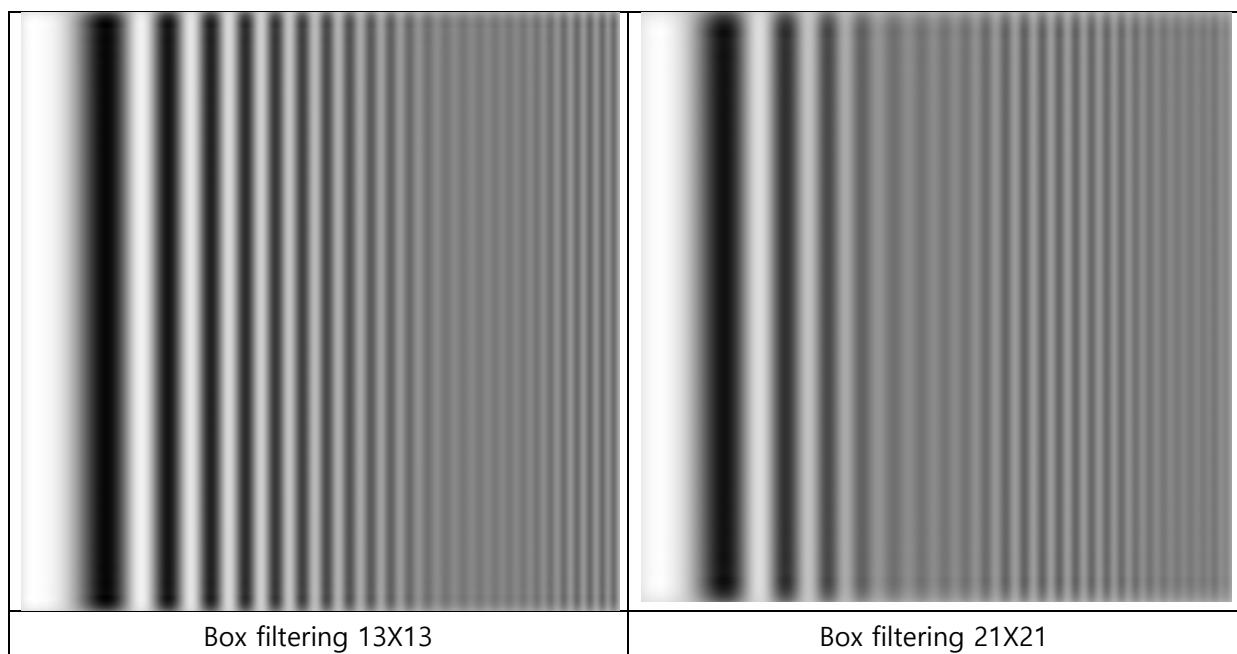
우선, Box filter의 경우를 관측할 경우, 두 경우 전부 사각형 형태로 phase의 변화를 구분 할 수 있다. 이 경우 frequency domain의 spectrum은 원점을 기준으로 N point 2D DFT에 대하여 $\exp(j2\pi(N-1)\frac{uv}{2})$ 의 linear한 phase term을 가지며, u, v의 값에 따라 phase가 변하게 된다.

위의 phase에서는 공통적으로 좌상단의 사각형 그룹 (2사분면 부분)이 우하단의 사각형 그룹 (4사분면 부분)보다 intensity가 낮은 것을 확인할 수 있었다. 또한, 각 사분면 내의 사각형은 우하단에 가까워질수록 intensity가 낮아지게 된다. 이 때, 사각형이 생성되는 지점은 sinc function에서 그 크기가 0인 지점이며, 영상의 intensity는 π 정도의 차이가 발생한다. 또한, 사각형 안쪽 부분의 대각선으로 경계가 구분되는 지점은 phase가 $+\pi$ 와 $-\pi$ 가 교차하는 지점으로, 영상의 intensity는 약 2π 의 정도의 차이가 발생하게 된다.



반면 Gaussian filter의 경우를 관측할 경우, 두 경우 전부 사각형의 형태로 phase의 변화를 구분할 수 있지만, 사각형에 대각선 형태가 더욱 많이 보이는 차이가 있다. 이 경우, sinc function이 0이 되기 전에 phase가 $+\pi$ 와 $-\pi$ 가 여러 번 변경되는 것으로 생각할 수 있다. Gaussian filter 역시 box filter와 같이 사각형으로 구분되는 지점에서는 π 만큼의 크기 차이, 대각선으로 구분되는 지점에서는 2π 만큼의 크기 차이가 발생하게 된다.

- 3) 구현 1에서는 low pass box filter를 사용했을 때 **high frequency** 성분이 제대로 제거되지 않는 현상이 발견된다. 이러한 이유를 low pass box filter가 **frequency domain**에서 갖는 성질과 함께 설명하시오.



Gaussian filter와 비교하여, box filter에 존재하는 high frequency chirp가 blur되지 않고 그대로 남아있는 것을 확인할 수 있다. 이는 box filter의 center stop band와 edge의 pass band의 차이가 크지 않기 때문이다. Low pass box filter에서 얻을 수 있는 frequency response의 pass band내에 영상의 주파수가 있다면 이를 제거할 수 있지만, 영상에서 발생하는 주파수 성분이 pass band 외부에 있다면 해당 주파수 성분이 그대로 출력되게 된다.

- 4) 구현 3의 결과 영상을 통해 **Fourier spectrum**과 **phase angle**이 어떤 역할을 하는지 자세히 설명하시오.

두 결과 영상을 관찰할 경우, Magnitude(building) + Phase(Rectangle)의 경우 영상의 전체적인 형상은 Rectangle의 형태가 남아 있는 것을 볼 수 있지만, intensity와 intensity 변화 주파수 등에 차이를 보여 원 영상보다 더 밝은 영상이 출력되었다. 반대로 Magnitude(Rectangle) + Phase(building)의 경우는 전체적으로 building의 모습이 보이지만, 원 영상보다 더 밝기 변화가 적고 어두운 영상이 출력되었다. 따라서 phase는 영상의 형태와 detail을 결정하지만, magnitude는 단위 길이 당 intensity의 주파수 변화의 amount를 의미한다.

- 5) Frequency domain filtering이 spatial domain filtering과 비교해서 **계산상의 이점을 갖기 위해서는 어떤 조건을 만족해야 하는지 자세히 설명하시오.** (FFT를 사용하여 입력 영상의 크기는 $2^m \times 2^n$, kernel의 크기는 $k \times k$ 라고 가정한다. 그리고 입력 영상에 추가적인 padding은 수행하지 않는다. m, n : 양의 정수)

Non-separable filter의 경우, spatial domain에서의 convolution에서 걸리는 시간은 $MN \times mn$ 이며, 추가적인 padding을 수행하지 않을 때 연산 횟수는 $2^{m+n} \times k^2$ 이다. 반면 separable filter의 경우 걸리는 시간은 $MN(m+n)$ 이며, 이 경우의 연산 횟수는 $2^{m+n+1} \times k$ 로 나타낼 수 있다.

반면 FFT를 사용하는 경우, FFT와 IFFT를 통하여 frequency domain에서 filtering을 할 경우 $(m+n) \times 2^{m+n+1}$ 으로 나타낼 수 있다. 따라서 $2(m+n) < k^2$ 일 때 frequency domain에서 계산상의 이점이 있으며, separable한 filter의 경우 $(m+n) < k$ 일 때 frequency domain에서 연산이 더 유리하다.

참조문헌

DIP_Homework3.pdf, 서강대학교 MMI Lab

OpenCV Documentation (<https://docs.opencv.org/>), Intel Corporation

Gonzalez, Rafael C. Woods, Richard E. - Digital image processing, 4th Edition, Pearson, 2018.