

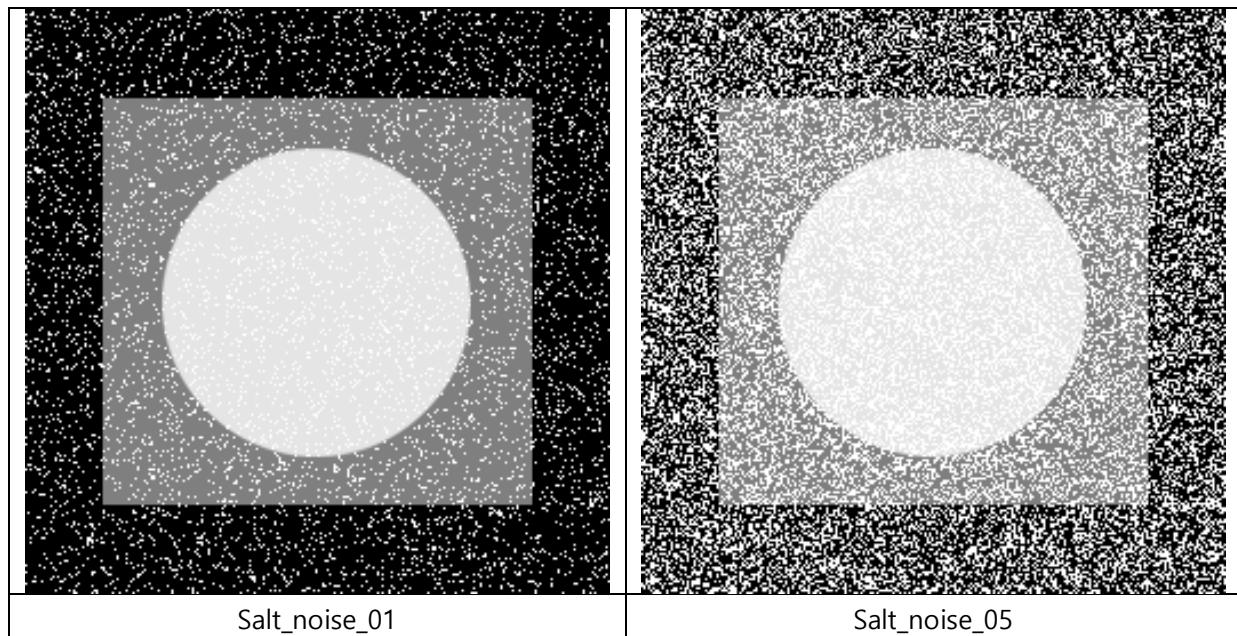
Course Title (과목명)	디지털영상처리개론 (01)
HW Number (HW 번호)	HW04
Submit Date (제출일)	2021-06-03
Grade (학년)	4 th Grade
ID (학번)	20161482
Name (이름)	박준용

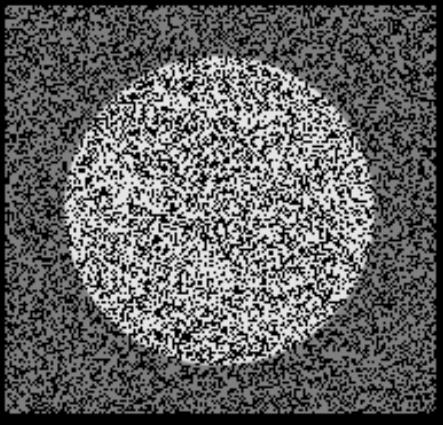
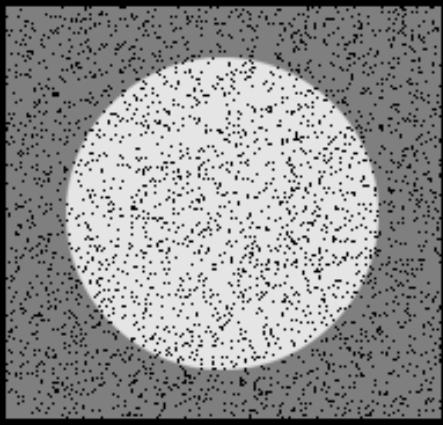
Image Restoration

- Spatial non-linear filter와 frequency domain filter를 이용한 image restoration을 구현.
 - ✓ OpenCV의 Mat class method 및 입/출력 함수(imread와 imwrite)만을 사용함.
 - ✓ dft.cpp내에 있는 함수 모두 사용 가능.
 - ✓ Frequency response는 dft.cpp의 fftshift2d()를 사용해서 원점이 영상의 중앙에 위치하도록 출력하고 Fourier spectrum의 모든 값들이 잘 보일 수 있도록 log transformation ($s = \log(1 + r)$)과 min-max scaling을 통해 dynamic range를 조절.
 - ✓ 제공된 HW04.cpp 및 dft.cpp, utils.h 파일은 수정하지 않고, utils.cpp 파일을 작성하여 제출함.

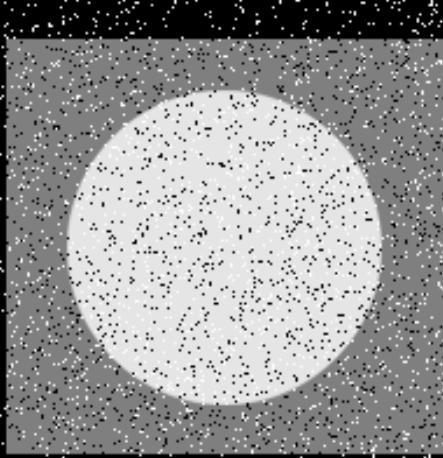
(1) 구현 1

- a) *.zip에 포함된 “Pattern.tif”에 다음 조건을 만족하는 salt & pepper noise를 추가하시오. (모든 noise는 additive noise라고 가정. p_s : a probability of the salt noise. p_p : a probability of the pepper noise.) 그리고 각 image의 histogram을 보이시오.

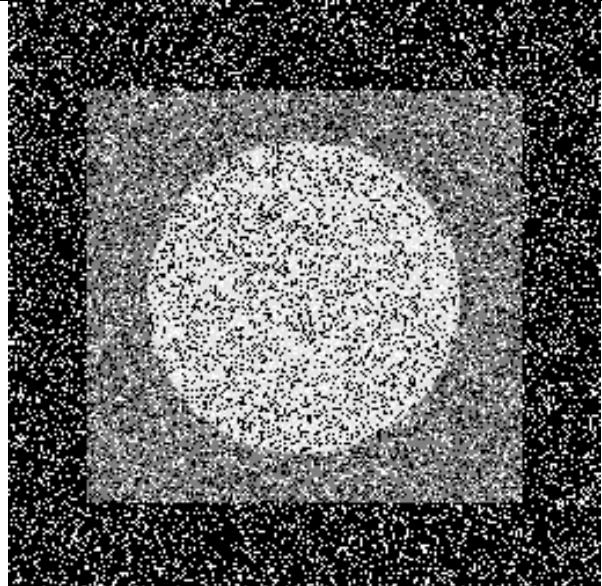




Pepper_noise_01

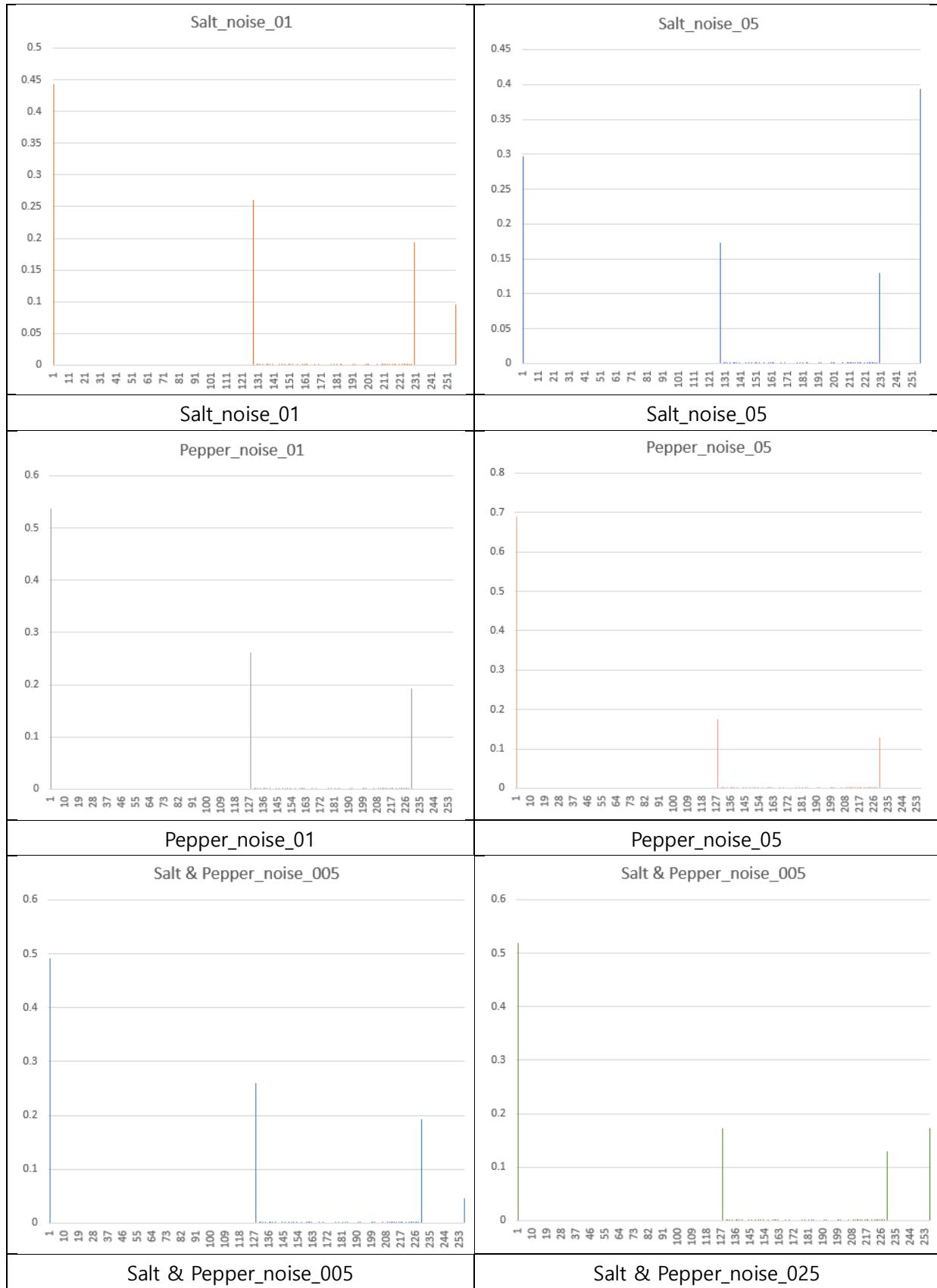


Pepper_noise_05



Salt & Pepper_noise_005

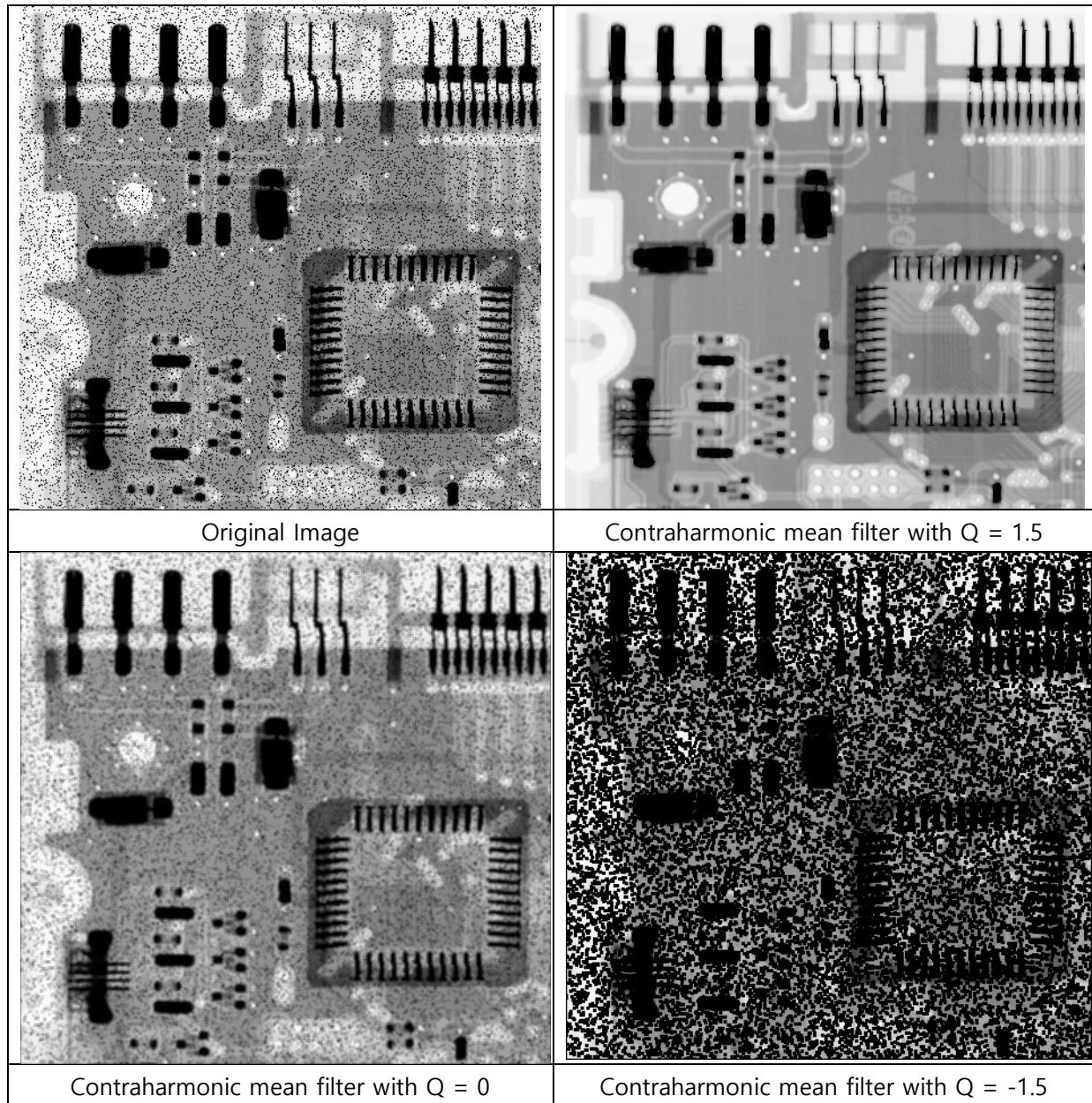
Salt & Pepper_noise_025

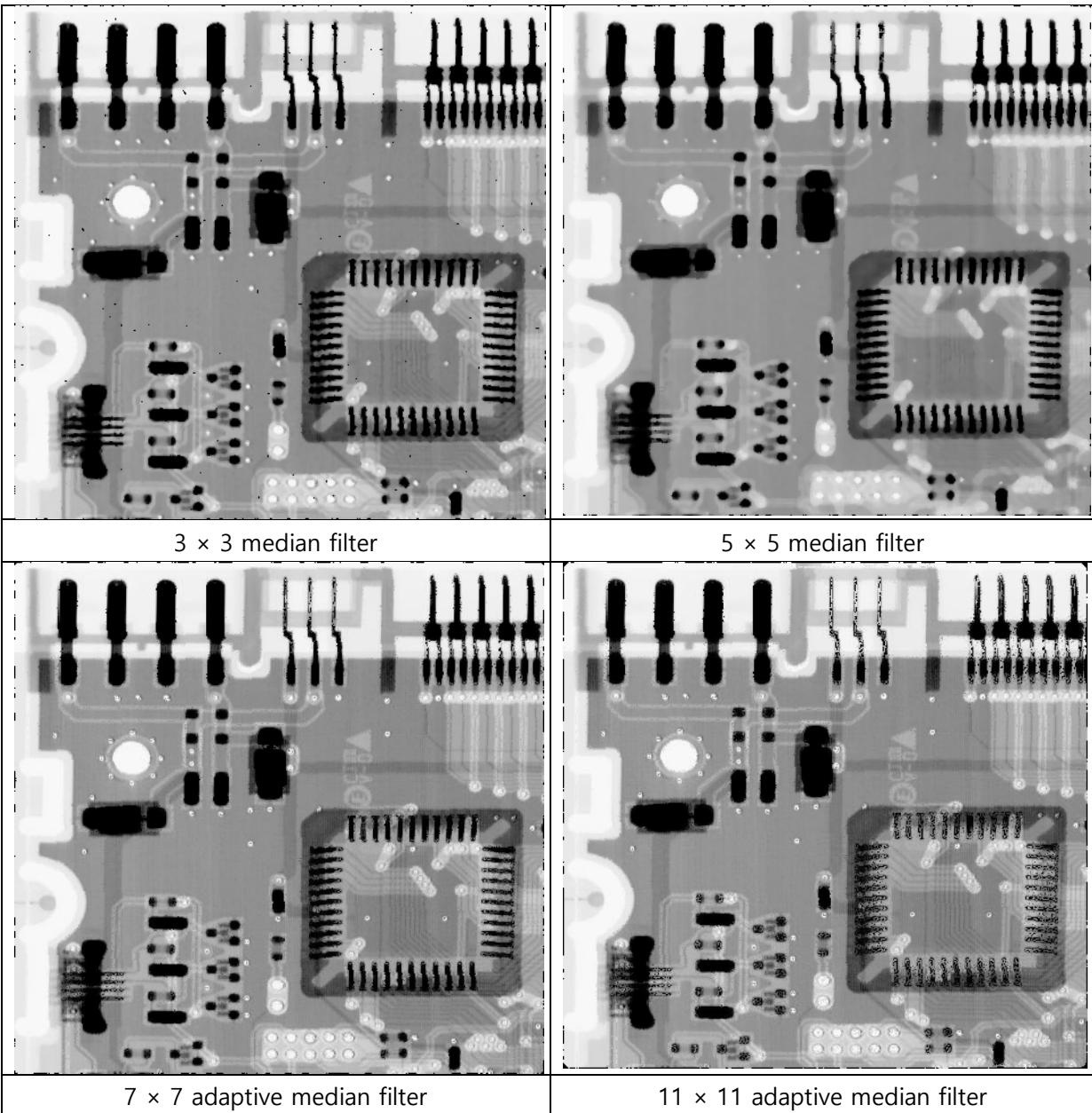


b) *.zip에 포함된 “Ckt_board.tif”에 salt & pepper noise를 추가한 image에 대하여 다음의 모든 filter를 거친 결과를 출력하시오.

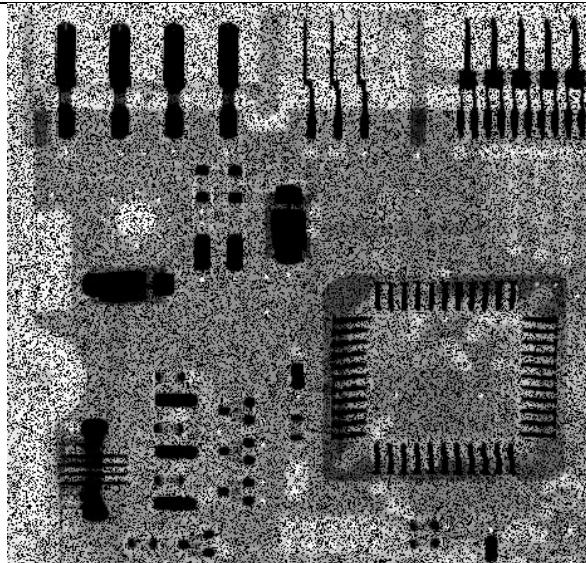
- 3×3 contraharmonic mean filter with $Q = 1.5$, $Q = 0$, and $Q = -1.5$.
- 3×3 and 5×5 median filter
- 7×7 and 11×11 adaptive median filter

i) Ckt_board_pepp_0.1

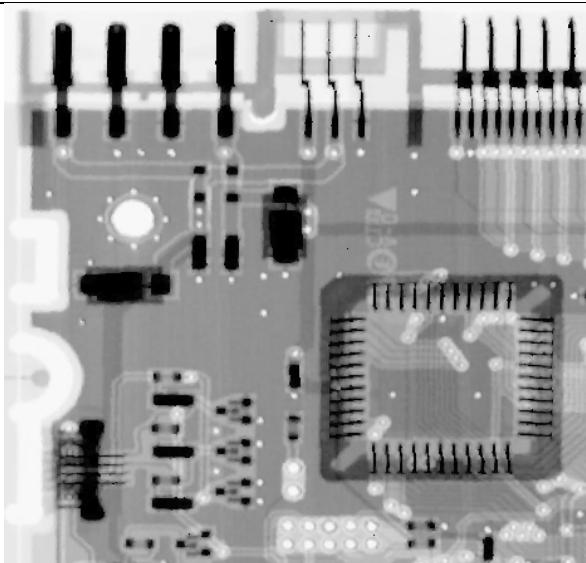




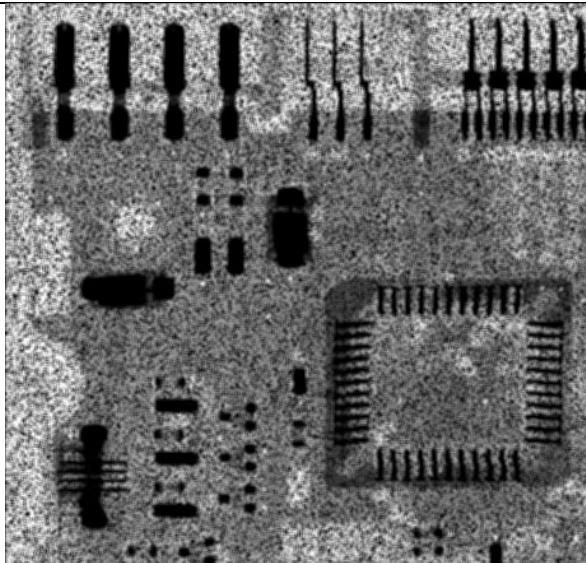
ii) Ckt_board_pepp_0.3



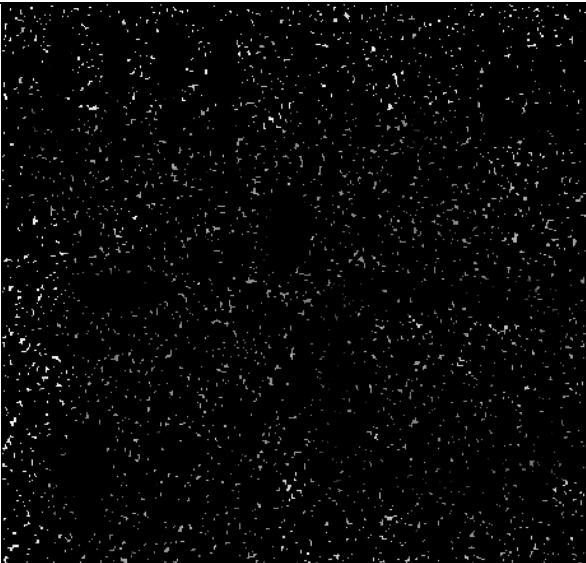
Original Image



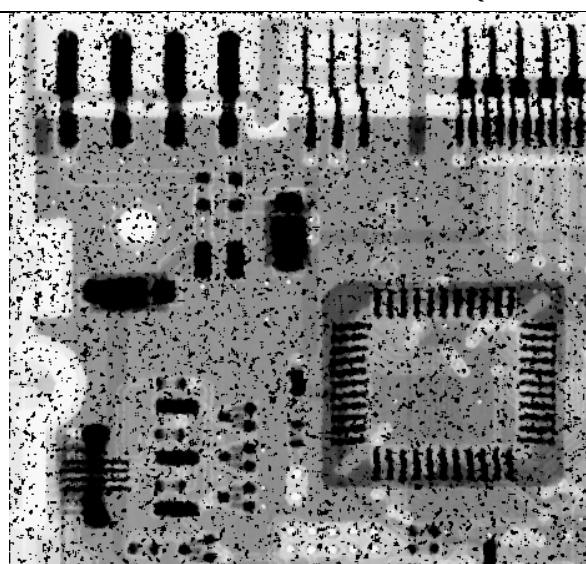
Contraharmonic mean filter with $Q = 1.5$



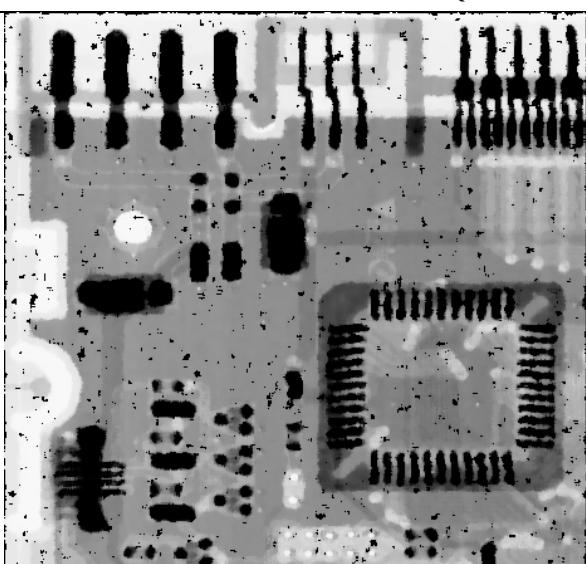
Contraharmonic mean filter with $Q = 0$



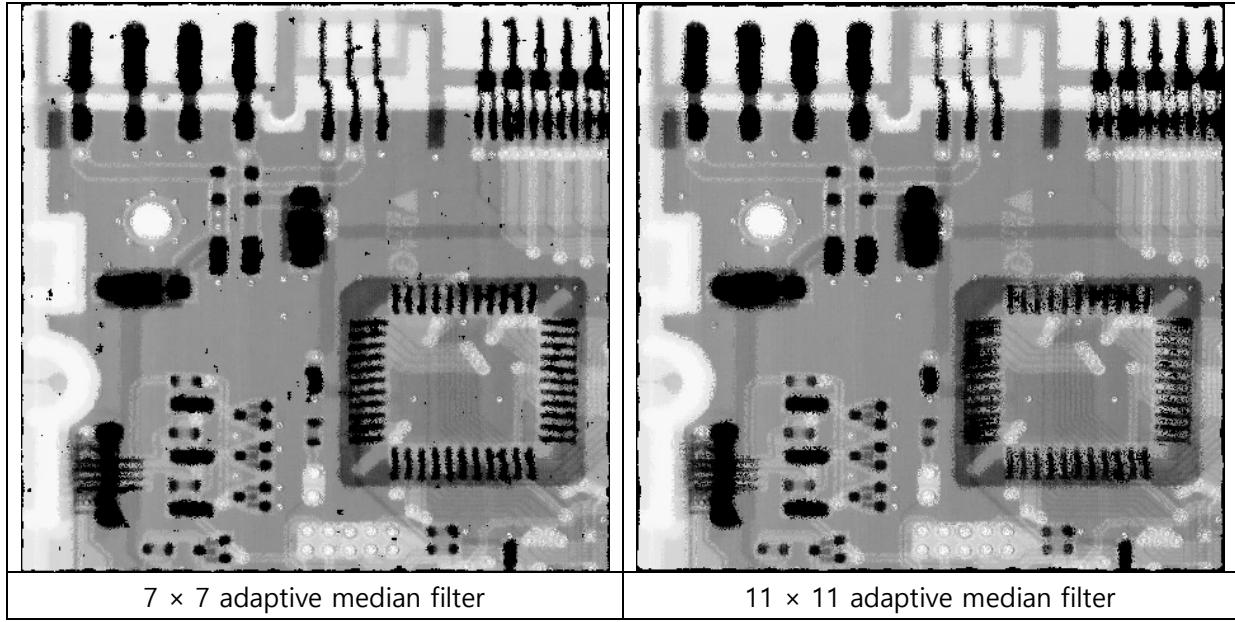
Contraharmonic mean filter with $Q = -1.5$



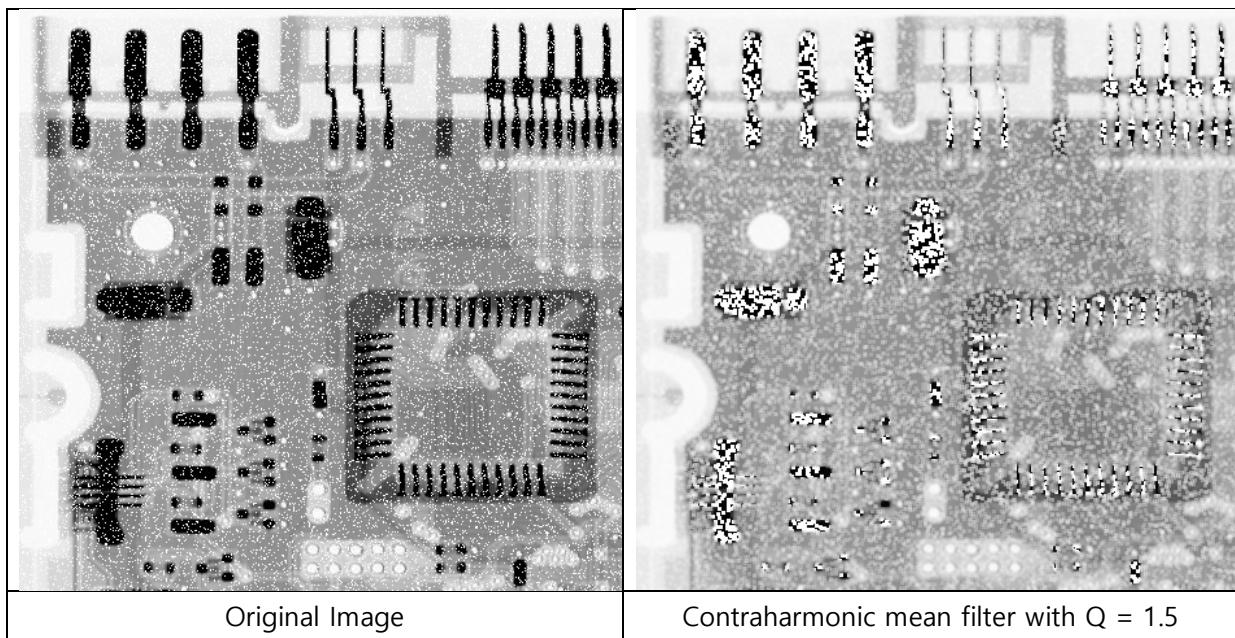
3×3 median filter

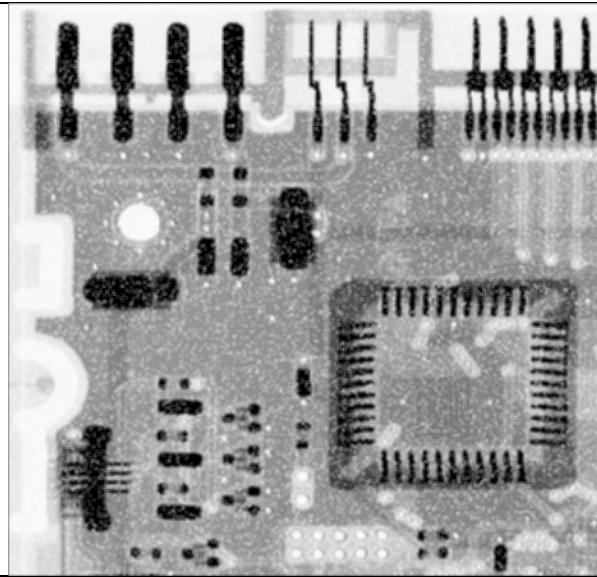


5×5 median filter

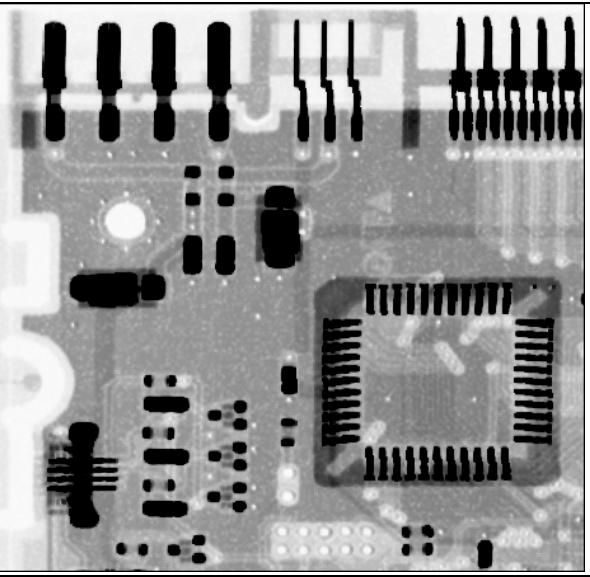


iii) Ckt_board_salt_0.1

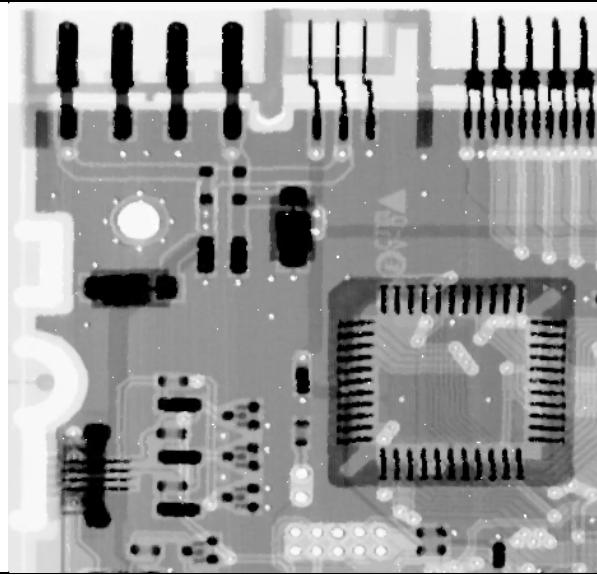




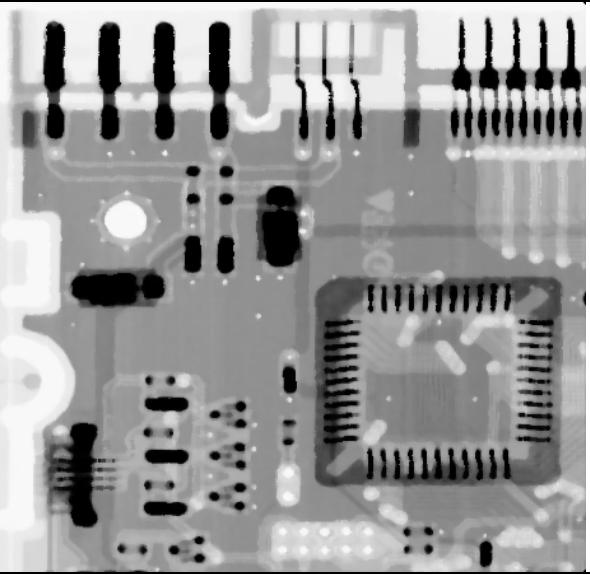
Contraharmonic mean filter with $Q = 0$



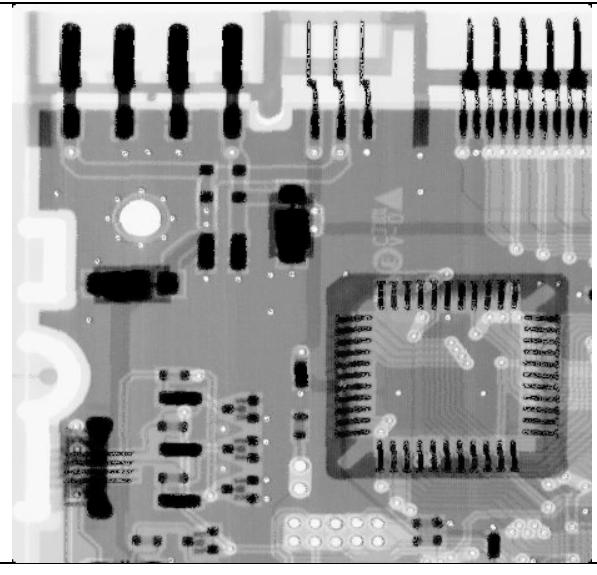
Contraharmonic mean filter with $Q = -1.5$



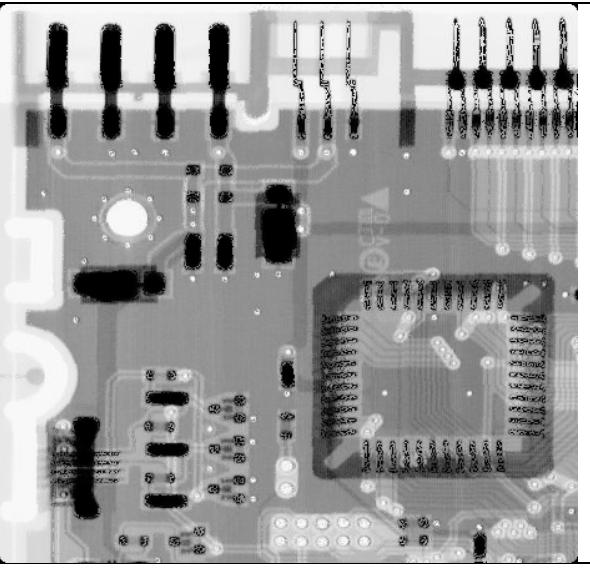
3×3 median filter



5×5 median filter

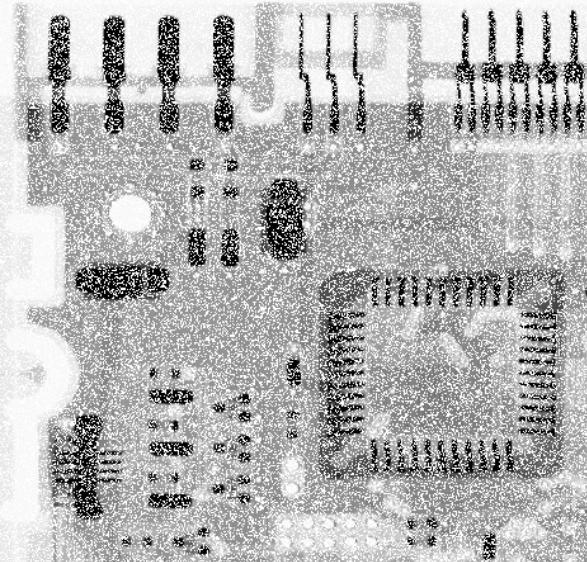
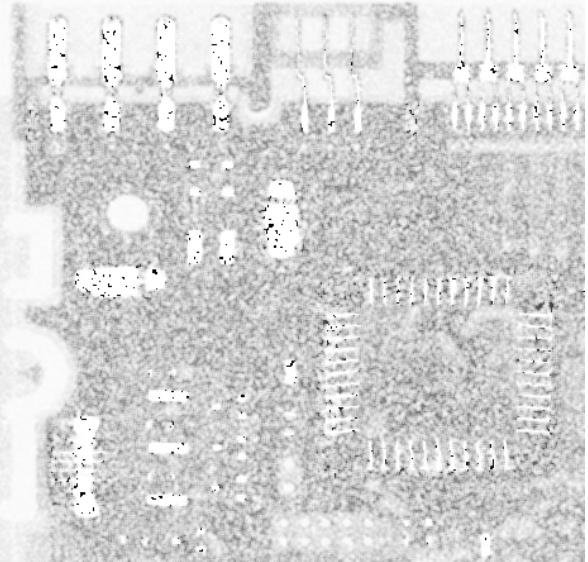
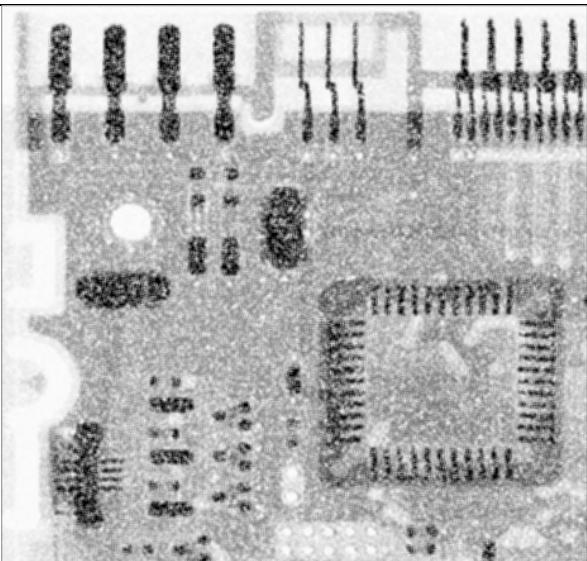
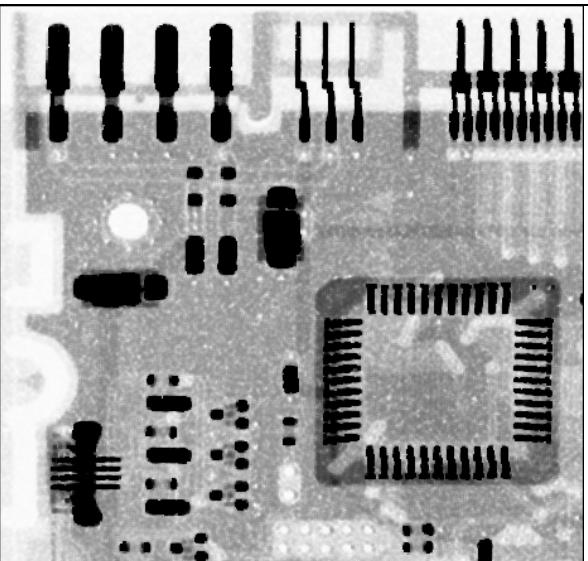


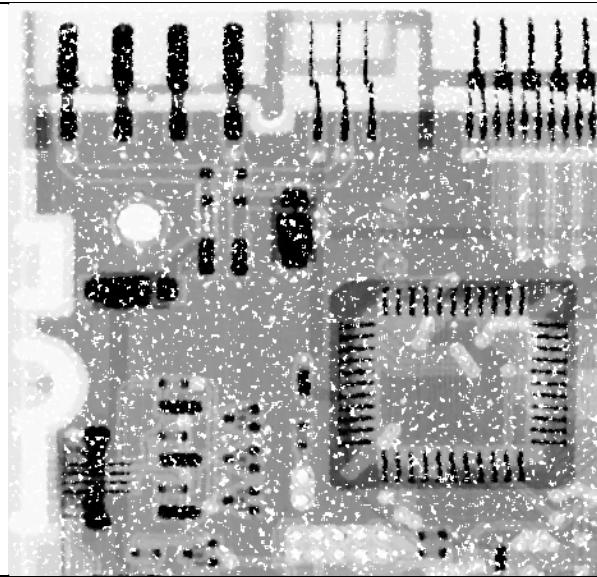
7×7 adaptive median filter



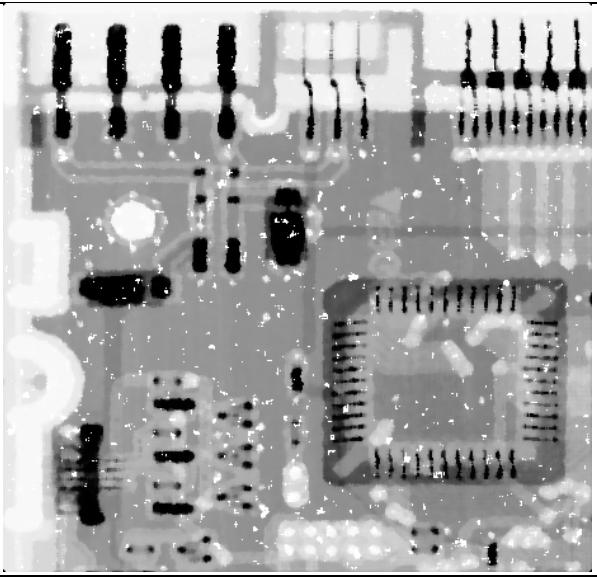
11×11 adaptive median filter

iv) Ckt_board_salt_0.3

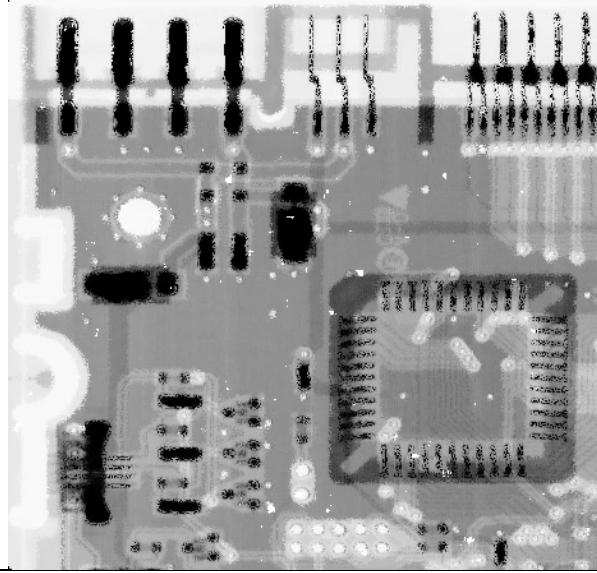
	
Original Image	Contraharmonic mean filter with $Q = 1.5$
	
Contraharmonic mean filter with $Q = 0$	Contraharmonic mean filter with $Q = -1.5$



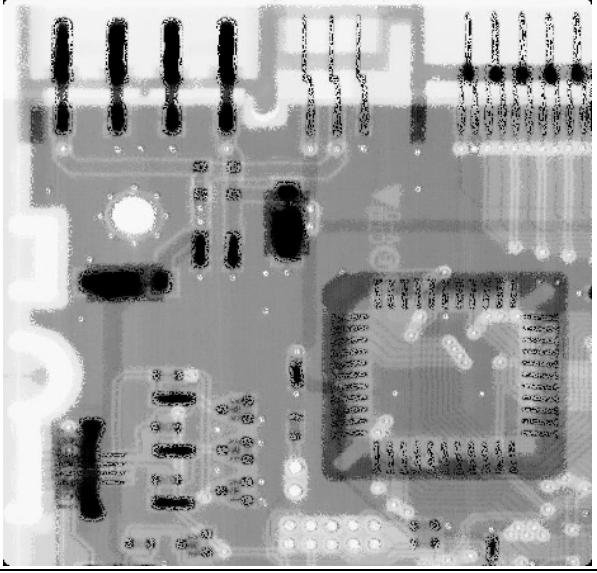
3×3 median filter



5×5 median filter

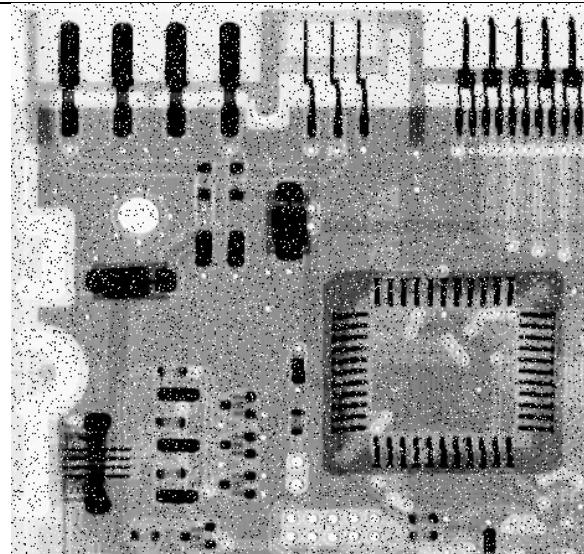


7×7 adaptive median filter

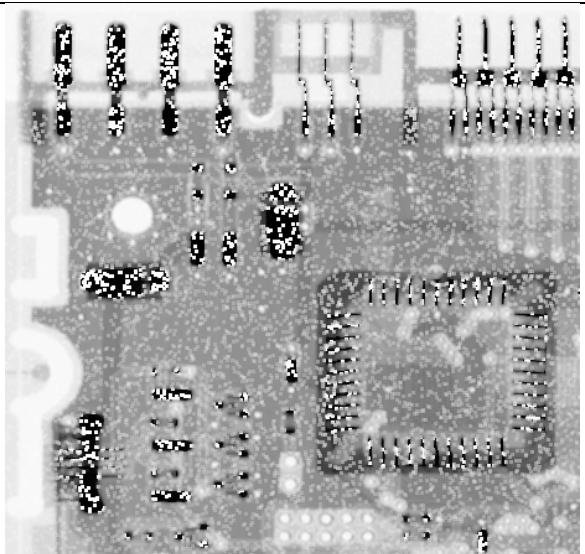


11×11 adaptive median filter

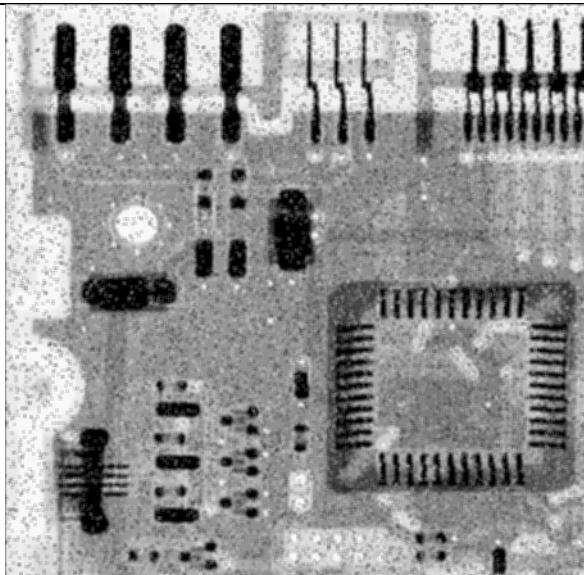
v) Ckt_board_salt&pepper_0.1



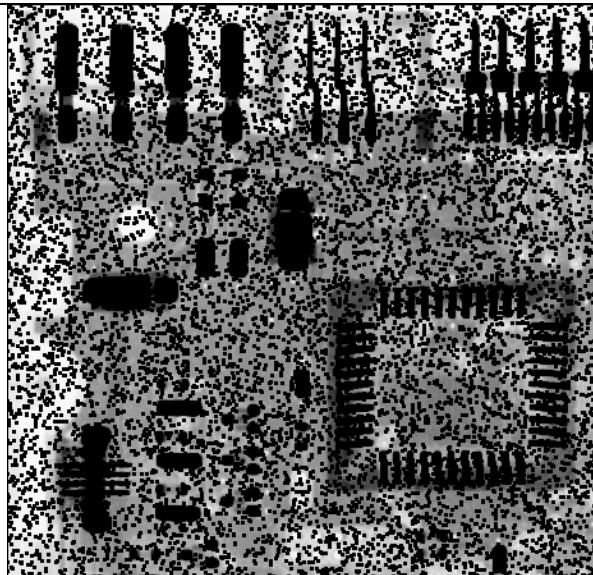
Original Image



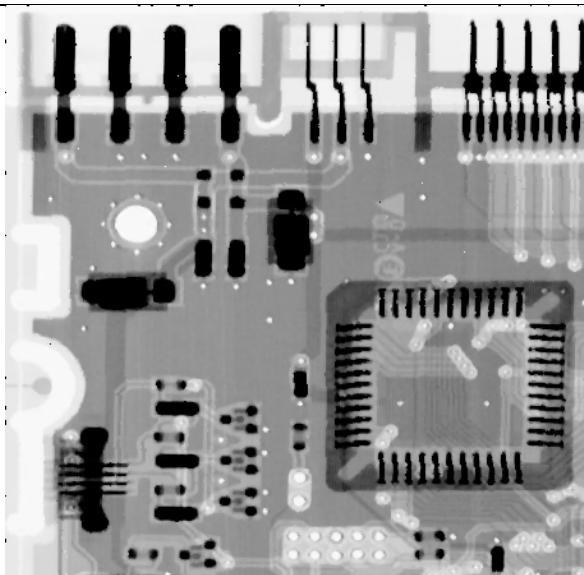
Contraharmonic mean filter with $Q = 1.5$



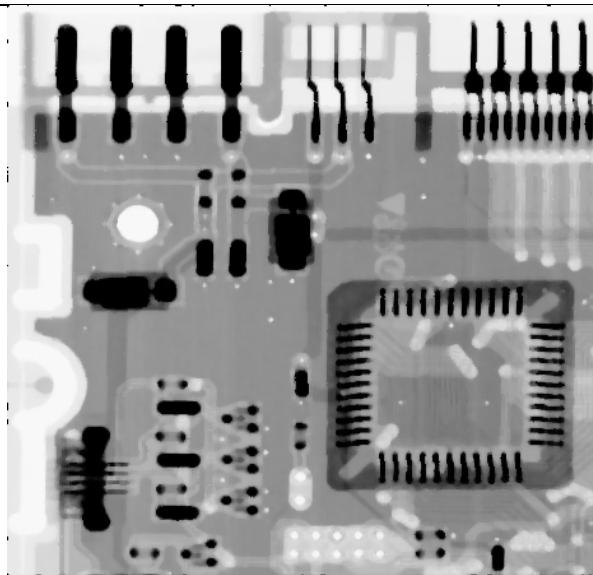
Contraharmonic mean filter with $Q = 0$



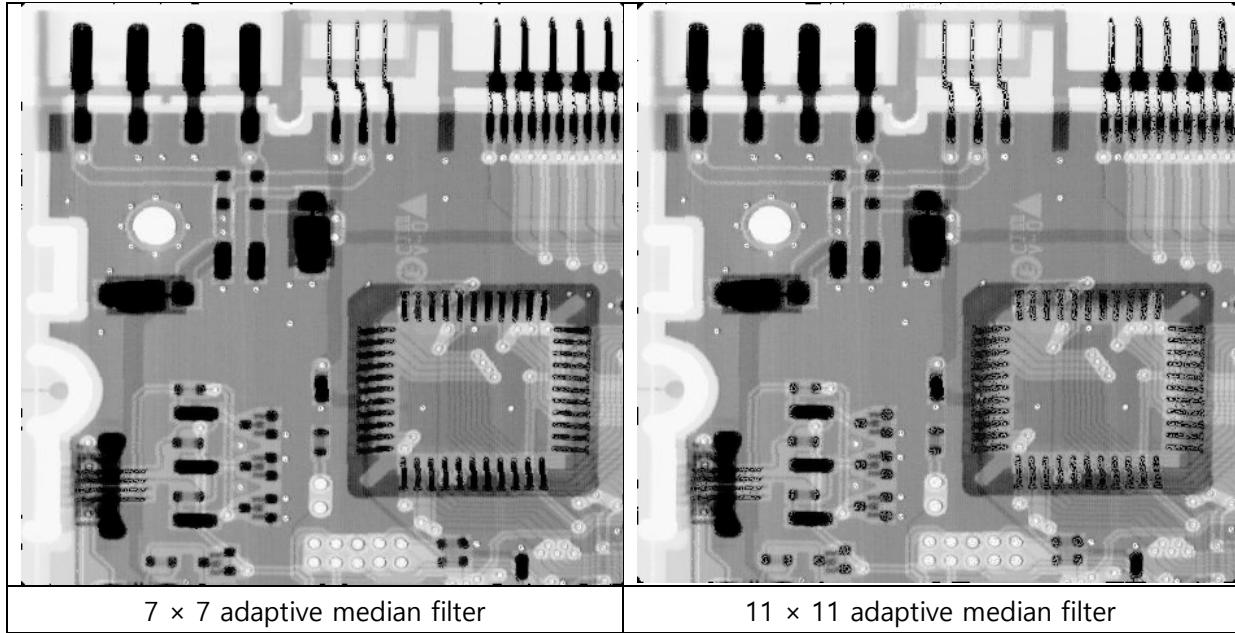
Contraharmonic mean filter with $Q = -1.5$



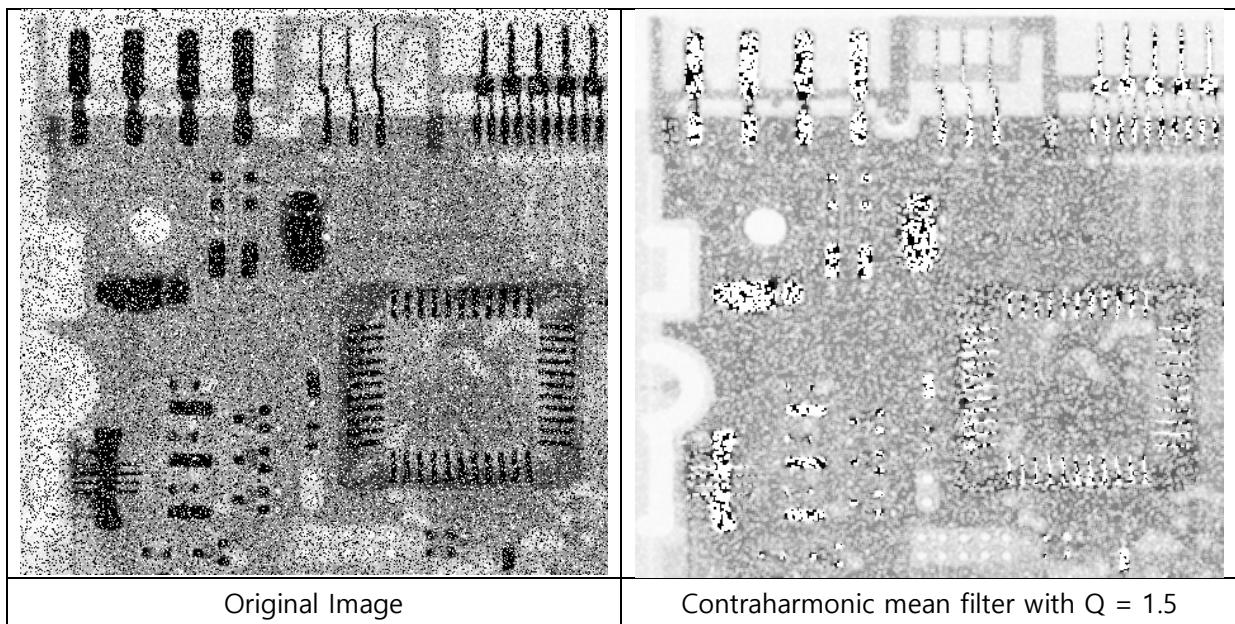
3×3 median filter

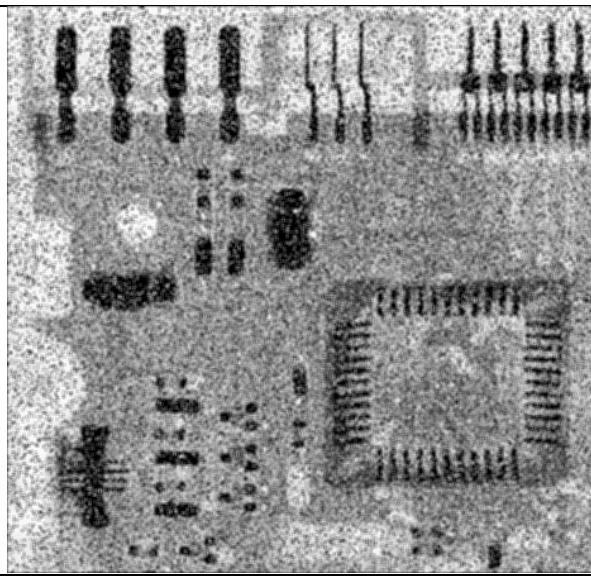


5×5 median filter

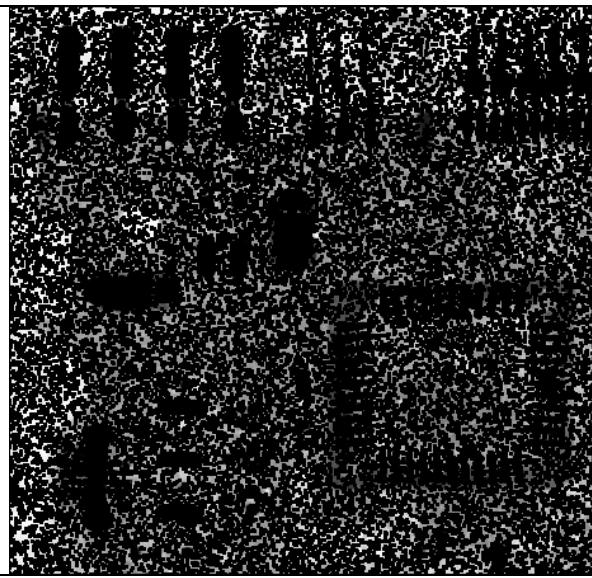


vi) Ckt_board_salt&pepper_0.3

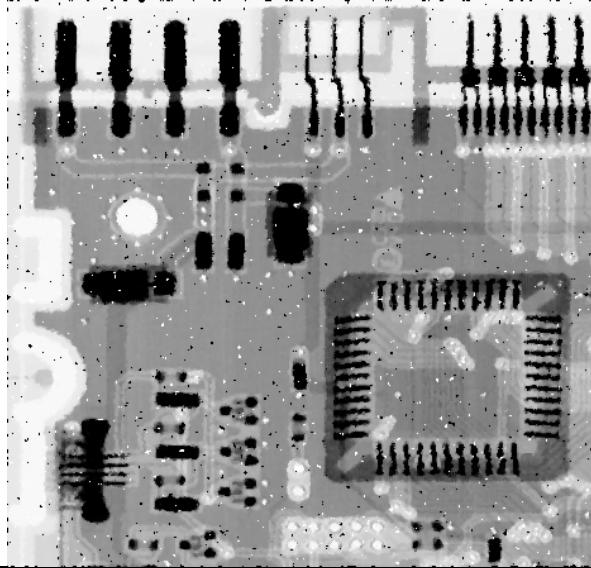




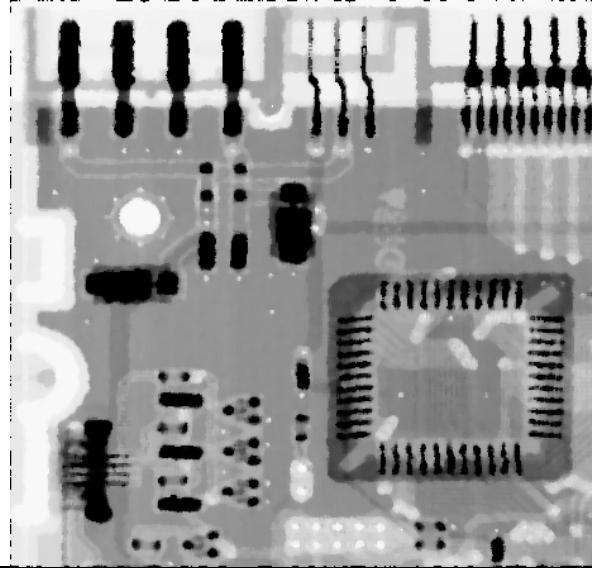
Contraharmonic mean filter with $Q = 0$



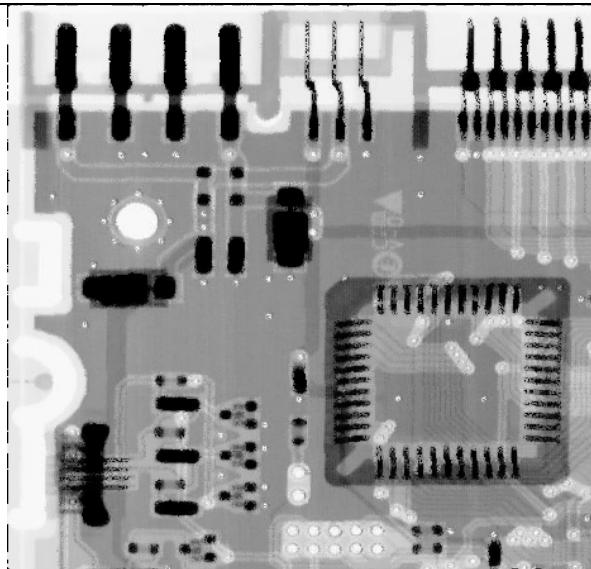
Contraharmonic mean filter with $Q = -1.5$



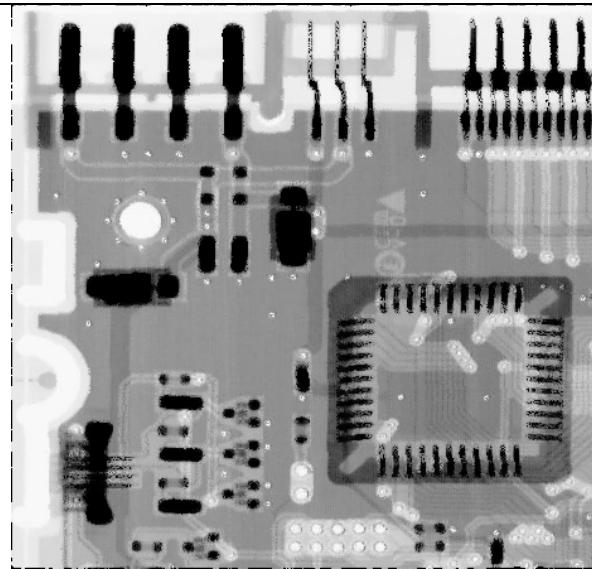
3×3 median filter



5×5 median filter



7×7 adaptive median filter



11×11 adaptive median filter

c) 6.2)를 참고하여 addSaltPepper(), contraharmonic(), median(), adaptiveMedian()을 작성하시오.

```
float* getHist(Mat src) {
    float* hist = (float*)calloc(256, sizeof(hist));
    for (int y = 0; y < src.cols; y++) {
        for (int x = 0; x < src.rows; x++) {
            hist[src.at<uchar>(x, y)]++;
        }
    }
    for (int i = 0; i < 256; i++) {
        hist[i] = hist[i] / (src.rows * src.cols);
    }
    return hist;
}

Mat addSaltPepper(Mat src, float ps, float pp, float salt_val, float pepp_val)
{
    Mat dst(src.size(), src.type());
    dst = src;
    dst.convertTo(dst, CV_32F);
    int val_ps = ((src.rows * src.cols) * ps);
    int val_pp = ((src.rows * src.cols) * pp);
    srand((int)time(NULL));

    int dst_row;
    int dst_col;
    for (int k = 0; k < val_ps; k++) {
        dst_row = rand() % src.rows;
        dst_col = rand() % src.cols;
        dst.at<float>(dst_row, dst_col) = salt_val;
    }

    for (int k = 0; k < val_pp; k++) {
        dst_row = rand() % src.rows;
        dst_col = rand() % src.cols;
        dst.at<float>(dst_row, dst_col) = pepp_val;
    }

    dst.convertTo(dst, CV_8U);

    std::ofstream saltPepperHist;
    float* dst_hist = getHist(dst);

    saltPepperHist.open("saltPepperHist.csv");
    for (int i = 0; i < 256; i++) {
        saltPepperHist << i << "," << dst_hist[i] << "\n";
    }
    saltPepperHist.close();

    return dst;
}
```

우선, pattern 이미지에 salt&pepper noise를 추가할 때 사용되는 함수인 addSaltPepper () 함수를 정의할 수 있다. 함수는 input source image와 salt, pepper noise 각각의 확률, salt, pepper noise의 값을 입력으로 받는다.

먼저, float ps와 pp를 사용하여 noise pixel을 적용할 개수를 val_ps, val_pp를 통하여 지정 할 수 있다. 이후, srand() 함수를 사용하여 input source image의 임의의 pixel을 지정하여 intensity를 변경함으로서 salt 또는 pepper noise를 적용할 수 있다.

또한, histogram을 작성하기 위하여 getHist() 함수 역시 작성하였다. histogram 값을 저장 하기 위하여 float array를 만들어 memory 할당을 한 뒤 변수를 초기화시킨 이후, input image의 pixel 개수를 count하여 $h(r_k)$ 를 얻은 뒤, 이를 input image의 height와 width를 곱한 값을 통하여 normalized histogram $p(r_k) = \frac{h(r_k)}{MN}$ 을 구할 수 있도록 구현하였다. (HW2에 사용하였던 함수 사용)

```
Mat contraharmonic(Mat src, Size kernel_size, float Q)
{
    Mat padded(src.rows + kernel_size.height - 1, src.cols + kernel_size.width - 1,
CV_32F);

    float k_row1 = (kernel_size.height - 1) / 2;
    float k_col1 = (kernel_size.width - 1) / 2;
    for (int x_prime = 0; x_prime < padded.rows; x_prime++) {
        for (int y_prime = 0; y_prime < padded.cols; y_prime++) {
            if (x_prime < (int)k_row1 || y_prime < (int)k_col1 || x_prime >
padded.rows - 1 - ((int)k_row1) || y_prime > padded.cols - 1 - ((int)k_col1)) {
                padded.at<float>(x_prime, y_prime) = 0;
            }
            else {
                padded.at<float>(x_prime, y_prime) =
(float)src.at<uchar>(x_prime - ((int)k_row1), y_prime - ((int)k_col1));
            }
        }
    }
    Mat dst(src.size(), CV_32F);
    for (int x = 0; x < src.rows; x++) {
        for (int y = 0; y < src.cols; y++) {
            float num = 0;
            float den = 0;
            for (int x_kernel = 0; x_kernel < kernel_size.height; x_kernel++) {
                for (int y_kernel = 0; y_kernel < kernel_size.width;
y_kernel++) {
                    num += pow((int)padded.at<float>(x + x_kernel, y +
y_kernel), Q + 1.0);
                    den += pow((int)padded.at<float>(x + x_kernel, y +
y_kernel), Q);
                }
            }
            dst.at<float>(x, y) = num / den;
        }
    }
    dst.convertTo(dst, CV_8U);

    return dst;
}
```

```

}

Mat median(Mat src, Size kernel_size)
{
    Mat padded(src.rows + kernel_size.height - 1, src.cols + kernel_size.width - 1,
src.type()));

    float k_row1 = (kernel_size.height - 1) / 2;
    float k_col1 = (kernel_size.width - 1) / 2;
    for (int x_prime = 0; x_prime < padded.rows; x_prime++) {
        for (int y_prime = 0; y_prime < padded.cols; y_prime++) {
            if (x_prime < (int)k_row1 || y_prime < (int)k_col1 || x_prime >
padded.rows - 1 - ((int)k_row1) || y_prime > padded.cols - 1 - ((int)k_col1)) {
                padded.at<uchar>(x_prime, y_prime) = 0;
            }
            else {
                padded.at<uchar>(x_prime, y_prime) = src.at<uchar>(x_prime -
((int)k_row1), y_prime - ((int)k_col1));
            }
        }
    }

    Mat dst(src.size(), src.type());
    for (int x = 0; x < src.rows; x++) {
        for (int y = 0; y < src.cols; y++) {
            int* median_array = new int[kernel_size.height * kernel_size.width];
            int i = 0;
            for (int x_kernel = 0; x_kernel < kernel_size.height; x_kernel++) {
                for (int y_kernel = 0; y_kernel < kernel_size.width;
y_kernel++) {
                    median_array[i] = padded.at<uchar>(x + x_kernel, y +
y_kernel);
                    i = i + 1;
                }
            }
            sort(median_array, median_array + kernel_size.height *
kernel_size.width);
            dst.at<uchar>(x, y) = median_array[(kernel_size.height *
kernel_size.width) / 2];
            delete[] median_array;
        }
    }
    return dst;
}

Mat adaptiveMedian(Mat src, Size kernel_size)
{
    Size kernel = Size(1, 1);
    Mat dst(src.size(), src.type());
    Mat padded(src.rows + kernel_size.height - 1, src.cols + kernel_size.width - 1,
src.type()));

    float k_row1 = (kernel_size.height - 1) / 2;
    float k_col1 = (kernel_size.width - 1) / 2;
    for (int x_prime = 0; x_prime < padded.rows; x_prime++) {
        for (int y_prime = 0; y_prime < padded.cols; y_prime++) {

```

```

        if (x_prime < (int)k_row1 || y_prime < (int)k_col1 || x_prime >
padded.rows - 1 - ((int)k_row1) || y_prime > padded.cols - 1 - ((int)k_col1)) {
            padded.at<uchar>(x_prime, y_prime) = 0;
        }
        else {
            padded.at<uchar>(x_prime, y_prime) = src.at<uchar>(x_prime -
((int)k_row1), y_prime - ((int)k_col1));
        }
    }
}

for_loop:
    for (int x = 0; x < src.rows; x++) {
        for (int y = 0; y < src.cols; y++) {
            unsigned char* median_array = (unsigned char*)malloc(kernel.height *
kernel.width * sizeof(unsigned char));
            int i = 0;
            for (int x_kernel = 0; x_kernel < kernel.height; x_kernel++) {
                for (int y_kernel = 0; y_kernel < kernel.width; y_kernel++) {
                    median_array[i] = padded.at<uchar>(x +
(kernel_size.height - kernel.height) / 2 + x_kernel, y + (kernel_size.width - kernel.width) / 2
+ y_kernel);
                    i = i + 1;
                }
            }
            sort(median_array, median_array + kernel.height * kernel.width);
            int kernel_min = median_array[0];
            int kernel_max = median_array[kernel.height * kernel.width - 1];
            int kernel_med = median_array[(kernel.height * kernel.width) / 2];

            float k_row2 = kernel_size.height / 2;
            float k_col2 = kernel_size.width / 2;
            if (kernel_min < kernel_med && kernel_med < kernel_max) {
                if (kernel_min < padded.at<uchar>(x + k_row2, y + k_col2) &&
padded.at<uchar>(x + k_row2, y + k_col2) < kernel_max) {
                    dst.at<uchar>(x, y) = padded.at<uchar>(x + k_row2, y
+ k_col2);
                }
                else {
                    dst.at<uchar>(x, y) = kernel_med;
                }
            }
            else {
                if (kernel.height < kernel_size.height) {
                    kernel += Size(2, 2);
                    goto for_loop;
                }
                else {
                    dst.at<uchar>(x, y) = kernel_med;
                }
            }
            free(median_array);
        }
    }
    return dst;
}

```

filter 함수들의 경우 input source image에 대해 공통적으로 zero padding을 해 주었다.

$$\hat{f}(x, y) = \frac{\sum_{(r,c) \in S_{xy}} g(r, c)^{Q+1}}{\sum_{(r,c) \in S_{xy}} g(r, c)^Q}$$

이후, contraharmonic() 함수의 경우 input source image, kernel_size와 float value Q를 입력으로 받으면 이 값을 사용하여 위와 같은 식을 통해 dst pixel value를 구한다. 이를 numerator와 denominator value를 하나씩 늘려가며 sum을 구하고, 이를 나누는 형식을 통해 code로 구현하였다.

$$\hat{f}(x, y) = \underset{(r,c) \in S_{xy}}{\text{median}}\{g(r, c)\}$$

median() 함수의 경우 input source image, kernel_size를 입력으로 받으며, kernel 안에 존재하는 모든 pixel intensity를 읽은 뒤 이를 sort 함수를 통하여 분류한다. 이후 median 값을 계산하여 해당 pixel의 값을 median 값으로 대체할 수 있다. adaptiveMedian() 함수의 경우, 해당 median intensity를 계산한 뒤 이 값이 kernel_min 값과 kernel_max 값 사이에 있지 않을 경우에는 kernel size를 maximum kernel size까지 증가시키게 된다. 이러한 과정을 통하여 kernel_med 값이 min 값과 max 값 사이에 있을 경우에만 해당 pixel의 값을 median 값으로 대체하게 된다.

(2) 구현 2

- a) Turbulence에 의해 발생하는 image degradation은 frequency domain에서 다음 식으로 표현할 수 있다.

$$H(u, v) = \exp(-k(u^2 + v^2)^{5/6})$$

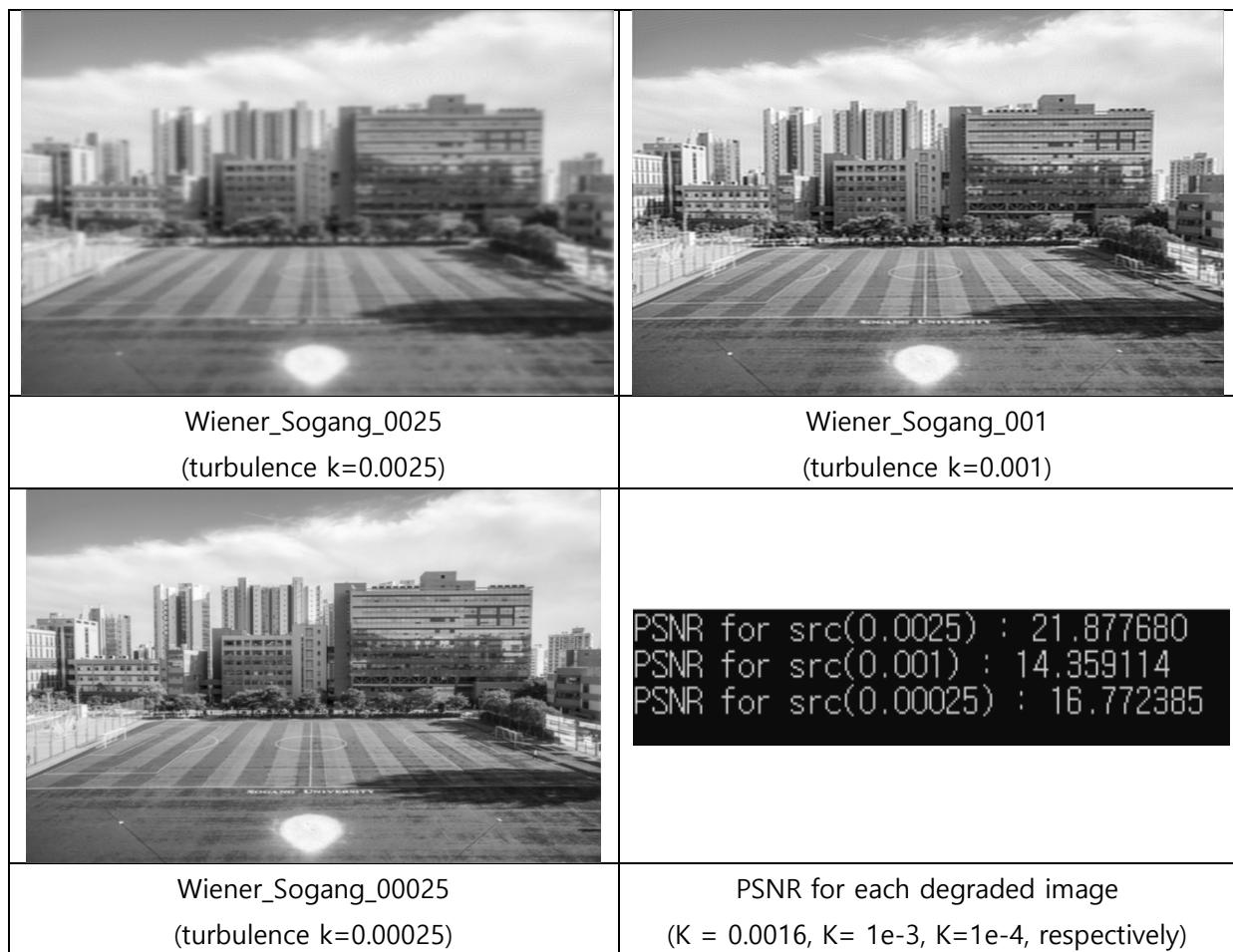
*.zip에 포함된 “**Sogang.tif**” 대하여 k 가 각각 0.0025와 0.001, 0.00025 일 때의 degradation된 image를 각각 출력하시오.





b) Wiener filter를 사용하여 degradation된 image를 복원하시오

- 복원에 사용되는 Wiener filter는 원본 영상과의 PSNR이 최대가 되도록 설정



c) 6.2)를 참고하여 add_turbulence(), wienerFilter()를 작성하시오.

```
Mat add_turbulence(Mat src, float k)
{
    Mat src_fft;
```

```

src_fft = fftshift2d(fft2d(src));

Mat tmp[2];
Mat src_magnitude;
Mat src_phase;
split(src_fft, tmp);
magnitude(tmp[0], tmp[1], src_magnitude);
phase(tmp[0], tmp[1], src_phase);

Mat H(src.size(), CV_32FC2);
for (int u = 0; u < src.rows; u++) {
    for (int v = 0; v < src.cols; v++) {
        H.at<Vec2f>(u, v)[0] = exp(-k * pow((u - floor(src.rows/2)) * (u -
floor(src.rows / 2)) + (v - floor(src.cols / 2)) * (v - floor(src.cols / 2)), 5.0 / 6));
        H.at<Vec2f>(u, v)[1] = 0;
    }
}

Mat dst(src.size(), CV_32FC2);
for (int x = 0; x < src.rows; x++) {
    for (int y = 0; y < src.cols; y++) {
        dst.at<Vec2f>(x, y)[0] = src_fft.at<Vec2f>(x, y)[0] * H.at<Vec2f>(x,
y)[0] - src_fft.at<Vec2f>(x, y)[1] * H.at<Vec2f>(x, y)[1];
        dst.at<Vec2f>(x, y)[1] = src_fft.at<Vec2f>(x, y)[1] * H.at<Vec2f>(x,
y)[0] + src_fft.at<Vec2f>(x, y)[0] * H.at<Vec2f>(x, y)[1];
    }
}

dst = ifft2d(fftshift2d(dst));
dst.convertTo(dst, CV_8U);
return dst;
}

Mat wienerFilter(Mat src, float K)
{
    Mat src_fft;
    src_fft = fftshift2d(fft2d(src));

    Mat tmp[2];
    Mat src_magnitude;
    Mat src_phase;
    split(src_fft, tmp);
    magnitude(tmp[0], tmp[1], src_magnitude);
    phase(tmp[0], tmp[1], src_phase);

    Mat H(src.size(), CV_32FC2);
    for (int u = 0; u < src.rows; u++) {
        for (int v = 0; v < src.cols; v++) {
            H.at<Vec2f>(u, v)[0] = exp(-K * pow((u - floor(src.rows / 2)) * (u -
floor(src.rows / 2)) + (v - floor(src.cols / 2)) * (v - floor(src.cols / 2)), 5.0 / 6));
            //H.at<Vec2f>(u, v)[0] = exp(-0.0025 * pow((u - floor(src.rows / 2))
* (u - floor(src.rows / 2)) + (v - floor(src.cols / 2)) * (v - floor(src.cols / 2)), 5.0 / 6));
            //H.at<Vec2f>(u, v)[0] = exp(-0.001 * pow((u - floor(src.rows / 2)) *
(u - floor(src.rows / 2)) + (v - floor(src.cols / 2)) * (v - floor(src.cols / 2)), 5.0 / 6));
            //H.at<Vec2f>(u, v)[0] = exp(-0.00025 * pow((u - floor(src.rows / 2))
* (u - floor(src.rows / 2)) + (v - floor(src.cols / 2)) * (v - floor(src.cols / 2)), 5.0 / 6));
        }
    }
}

```

```

        H.at<Vec2f>(u, v)[1] = 0;
    }

}

Mat H_magnitude;
Mat H_phase;
split(H, tmp);
magnitude(tmp[0], tmp[1], H_magnitude);
phase(tmp[0], tmp[1], H_phase);

Mat dst(src.size(), CV_32FC2);
Mat dst_magnitude(src.size(), CV_32F);
Mat dst_phase(src.size(), CV_32F);
Mat dst_real(src.size(), CV_32F);
Mat dst_imaginary(src.size(), CV_32F);

for (int u = 0; u < src.rows; u++) {
    for (int v = 0; v < src.cols; v++) {
        dst_magnitude.at<float>(u, v) = (1 / H_magnitude.at<float>(u, v)) *
            (H_magnitude.at<float>(u, v) * H_magnitude.at<float>(u, v)) /
            (H_magnitude.at<float>(u, v) * H_magnitude.at<float>(u, v) + K) *
            src_magnitude.at<float>(u, v);
        dst_phase.at<float>(u, v) = src_phase.at<float>(u, v);
    }
}

polarToCart(dst_magnitude, dst_phase, dst_real, dst_imaginary);

for (int x = 0; x < src.rows; x++) {
    for (int y = 0; y < src.cols; y++) {
        dst.at<Vec2f>(x, y)[0] = dst_real.at<float>(x, y);
        dst.at<Vec2f>(x, y)[1] = dst_imaginary.at<float>(x, y);
    }
}

dst = ifft2d(fftshift2d(dst));
dst.convertTo(dst, CV_8U);

return dst;
}

```

$$H(u, v) = \exp(-k(u^2 + v^2)^{5/6})$$

add_turbulence() 함수의 경우 input source image와 위의 수식에서 사용되는 k값을 입력으로 받는다. 우선, input source image에 fft2d와 fftshift2d를 적용한 뒤, magnitude와 phase를 구하기 위하여 Fourier Transform된 matrix를 openCV에서 제공하는 split 함수를 통하여 분리한 후 magnitude와 phase 함수를 적용하여 분리된 matrix에 해당하는 값을 저장할 수 있다. 이를 frequency domain의 함수인 $H(u,v)$ 를 구현한 뒤 2채널 image의 형태로 곱해주며 fftshift2d와 ifft2d를 사용하여 원래의 spatial domain으로 되돌린 후 output image를 반환하게 된다.

$$\hat{F}(u, v) = G(u, v) \frac{H(u, v)^2}{H(u, v)(H(u, v)^2 + \frac{S_N(u, v)}{S_s(u, v)})}$$

wienerFilter() 함수의 경우 input source image와 위에서 사용되는 float $K = \frac{S_N(u, v)}{S_s(u, v)}$ 값을 입력으로 받는다. 위와 같이, input source image에 fft2d와 fftshift2d를 적용한 뒤 해당 함수와 H 함수에 split 함수를 통하여 magnitude와 phase 함수를 각각 저장한 뒤, 분리된 matrix에 해당하는 값을 저장할 수 있다. (이때 H함수의 경우 add_turbulence 함수에서 사용한 3가지 경우를 주석을 통해 on/off 형식으로 사용하였다) 이를 위의 수식과 같은 형태로 계산한 뒤 fftshift2d와 ifft2d를 사용하여 spatial domain으로 되돌린 후 output image를 반환한다.

4. 검토사항

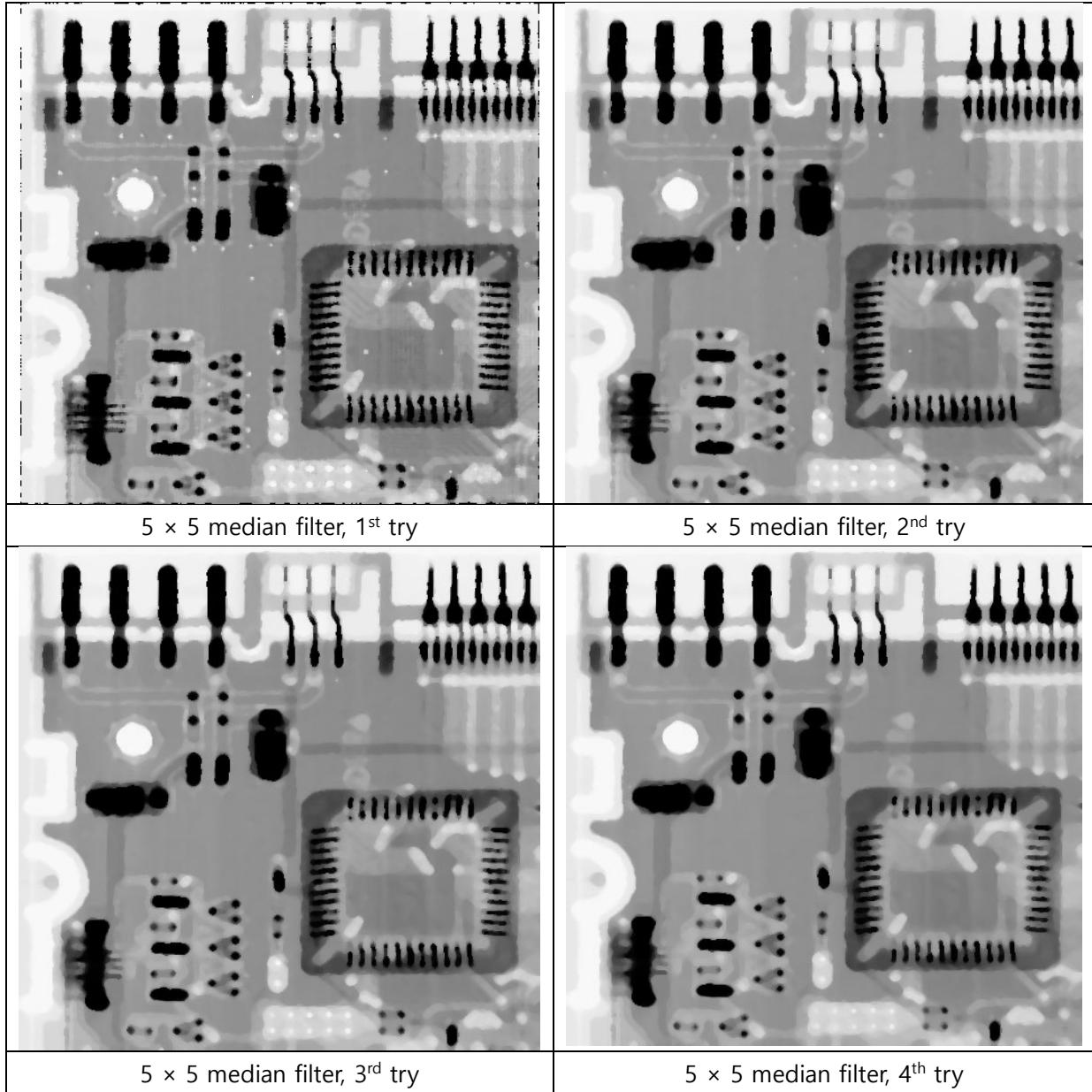
- 1) 구현 1 ~ 2에 대한 결과를 분석하여 보고서에 포함하시오.
- 2) 구현 1의 결과를 참고하여 각 filter의 장단점을 자세히 설명하시오.

우선, Contraharmonic filter의 경우 Q가 0인 경우, 일반 arithmetic mean filter와 동일한 결과를 가진다. 또한, 음수인 Q에 대해서는 salt noise에 대해 효과적이며 양수인 Q에 대해서는 pepper noise에 대해 효과적인 등 noise에 따라 Q의 값을 조정할 수 있는 장점이 존재한다. 그러나, 두 종류의 노이즈가 같이 있는 salt&pepper noise와 같은 경우에는 효율이 매우 떨어지는 단점이 있다.

Median filter의 경우 salt&pepper noise 등의 극단적인 intensity를 배제할 수 있어 일반적인 kernel size와 noise에 대해 대응할 수 있으므로 범용성이 크며, 연산량이 적어 처리가 빠르고 사용이 용이한 장점이 있다. 그러나 kernel size가 매우 작거나 noise가 많을 경우 다른 필터에 비해 효과적이지 못하며, 여러 번 적용할 경우 이미지의 손상이 나타나기 쉬운 단점이 존재한다.

Adaptive median filter의 경우 다른 필터들에 비해 거의 모든 noise의 경우에서 원본에 가까운 영상이 구해지는 장점이 있지만, 다른 필터들에 비해 연산의 횟수가 많아 수행하는데 상대적으로 긴 시간이 소모되는 단점이 있다. 또한 영상의 intensity가 급격하게 바뀌는 부분에서 그 부분이 잡을 경우 온전히 복원되지 않았으며, 이는 kernel size가 클수록 더욱 심해졌다.

- 3) 구현 1의 median filter를 반복해서 적용했을 때 어떤 차이점이 존재하는지 확인하고 차이점이 발생한다면(또는 발생하지 않는다면) 그러한 현상이 발생하는 이유에 대하여 자세히 설명하시오.



Salt&Pepper image에 5×5 median filter를 여러 번 적용한 결과를 통하여 filter를 적용할 수록 존재하는 noise가 감소하는 것을 확인할 수 있다. 그러나, detail한 부분의 intensity가 주변의 intensity와 비슷해지며 회로도 부분과 같은 부분 역시 noise와 함께 blur되는 결과 역시 나타나게 된다. 이로 인하여 median filter를 반복하여 계속 적용하게 된다면 영상에 필요한 정보 역시 손실되는 결과를 불러일으킬 수 있다. 이를 통하여 median filter의 filter 크기와 빈도를 조절하여 low pass의 정도를 조절하는 것이 필요하다는 것을 알 수 있다.

- 4) 구현 2에서 Wiener filter의 상수 K 값을 조절하면서 복원 영상의 **PSNR**을 측정하고 복원 영상의 품질과 어떤 연관이 있는지 분석하시오. 그리고 PSNR 외에 영상의 품질 측정에 자주 사용하는 **SSIM**에 대해서 조사하고 PSNR과 장단점을 비교하시오.

$$\text{PSNR} = 10 \log \frac{s^2}{\text{MSE}} \text{ (dB)}$$

PSNR은 peak signal to noise ratio의 약자로 위와 같이 정의된다. 이 때 MSE는 mean square error, s는 영상에서의 최대값을 의미하며, loss인 MSE가 낮을수록, s의 값이 높을수록 PSNR은 높은 값을 가지게 되며, 이는 변형된 영상이 원본영상에 가깝다는 것을 의미한다.

```
Mat Wiener_filter_image = wienerFilter(add_turbulence_0025, 1e-6);
imshow("Wiener_filter_image_0025", Wiener_filter_image);
imwrite("Wiener_Sogang_0025.tif", Wiener_filter_image);
float PSNR;
int s = 0;
int MSE = 0;

for (int x = 0; x < src3.rows; x++) {
    for (int y = 0; y < src3.cols; y++) {
        if (s < src3.at<uchar>(x, y))
            s = src3.at<uchar>(x, y);
        MSE = MSE + pow(src3.at<uchar>(x, y) -
Wiener_filter_image.at<uchar>(x, y), 2);
    }
}

MSE = MSE / (src3.rows * src3.cols);

PSNR = 10 * log(pow(s, 2) / MSE) / log(10);
printf("PSNR for src(0.0025) : %f\n", PSNR);

Wiener_filter_image = wienerFilter(add_turbulence_001, 1e-6);
imshow("Wiener_filter_image_001", Wiener_filter_image);
imwrite("Wiener_Sogang_001.tif", Wiener_filter_image);
int s2 = 0;
int MSE2 = 0;

for (int x = 0; x < src3.rows; x++) {
    for (int y = 0; y < src3.cols; y++) {
        if (s2=0 < src3.at<uchar>(x, y))
            s2 = src3.at<uchar>(x, y);
        MSE2 = MSE2 + pow(src3.at<uchar>(x, y) -
Wiener_filter_image.at<uchar>(x, y), 2);
    }
}

MSE2 = MSE2 / (src3.rows * src3.cols);

PSNR = 10 * log(pow(s2, 2) / MSE2) / log(10);
printf("PSNR for src(0.001) : %f\n", PSNR);

Wiener_filter_image = wienerFilter(add_turbulence_00025, 1e-6);
imshow("Wiener_filter_image_00025", Wiener_filter_image);
imwrite("Wiener_Sogang_00025.tif", Wiener_filter_image);
int s3 = 0;
int MSE3 = 0;

for (int x = 0; x < src3.rows; x++) {
    for (int y = 0; y < src3.cols; y++) {
        if (s3 = 0 < src3.at<uchar>(x, y))
            s3 = src3.at<uchar>(x, y);
```

```

        MSE3 = MSE3 + pow(src3.at<uchar>(x, y) -
Wiener_filter_image.at<uchar>(x, y), 2);
    }
MSE3 = MSE3 / (src3.rows * src3.cols);

PSNR = 10 * log(pow(s3, 2) / MSE3) / log(10);
printf("PSNR for src(0.00025) : %f\n", PSNR);

```

Original image와 Wiener filtered image를 비교하여 PSNR을 구하기 위하여 util.cpp가 아닌 HW04.cpp에 PSNR을 계산하는 code를 작성하였다. 모든 src image의 pixel의 값 중의 maximum 값을 s값으로 반환한 뒤, 해당 pixel 값에서 Wiener_filter_image의 pixel 값을 뺀 값을 square하여 MSE를 구한다. 이후 위의 식을 사용하여 PSNR을 구할 수 있다.

PSNR for src(0.0025) : 21.981831 PSNR for src(0.001) : 11.371732 PSNR for src(0.00025) : 3.597886	PSNR for src(0.0025) : 21.877680 PSNR for src(0.001) : 12.027817 PSNR for src(0.00025) : 3.644170
K = 0.0016	K = 0.0015
PSNR for src(0.0025) : 21.345619 PSNR for src(0.001) : 14.359114 PSNR for src(0.00025) : 4.460870	PSNR for src(0.0025) : 20.687874 PSNR for src(0.001) : 11.258332 PSNR for src(0.00025) : 16.772385
K = 1e-3	K = 1e-4
PSNR for src(0.0025) : 20.656685 PSNR for src(0.001) : 11.082836 PSNR for src(0.00025) : 14.867783	PSNR for src(0.0025) : 20.656685 PSNR for src(0.001) : 11.082836 PSNR for src(0.00025) : 14.741158
K = 1e-5	K = 1e-6

위와 같이 6가지 경우에 대하여 PSNR을 확인하였으며, 이 때 turbulence k는 위에서부터 각각 0.0025, 0.001, 0.00025로 설정하였다. (K = 0.0016 이상의 값에 대해서는 측정 불가하다) 이를 확인한 결과, 각각의 Weiner filter image에 따라 최대값이 되는 PSNR은 전부 다르며, 위에서부터 각각 K = 0.0016, K= 1e-3, K=1e-4일 때 PSNR이 최대값이 되는 것을 확인할 수 있다.

SSIM은 structural similarity index이며, 구조적 유사 지수를 의미한다. 이 식은 원본영상 A와 왜곡된 영상 B를 가지고 나타낸다. SSIM은 시각의 화질 차이 및 유사도를 평가하기 위한 방법으로, 두 이미지의 휘도, 대비, 구조를 비교하여 분석한다.

i) 휘도 비교

$$I(A, B) = \frac{2\mu_A\mu_B + C_1}{\mu_A^2 + \mu_B^2 + C_1}$$

ii) 대비 비교

$$C(A, B) = \frac{2\sigma_A\sigma_B + C_2}{\sigma_A^2 + \sigma_B^2 + C_2}$$

iii) 구조 비교

$$s(A, B) = \frac{\sigma_{AB} + C_3}{\sigma_A \sigma_B + 3}$$

이러한 비교 요소를 곱하여 아래와 같이 SSIM을 나타낼 수 있다.

$$\text{SSIM}(A, B) = l(A, B)c(A, B)s(A, B) = \frac{(2\mu_A\mu_B + C_1)(2\sigma_{AB} + C_2)}{(\mu_A^2 + \mu_B^2 + C_1)(\sigma_A^2 + \sigma_B^2 + C_2)}$$

이 때, 각각의 C의 값 역시 아래와 같이 나타낼 수 있다.

$$C_1 = (0.01 * L)^2, C_2 = (0.03 * L)^2, C_3 = \frac{C_2}{2}, L = \text{dynamic range value}$$

- 5) 직접적으로 더해지는 noise가 없음에도 불구하고 full inverse filter를 사용해서 원래 image로 복구하는 것이 거의 불가능한 이유에 대해서 자세히 설명하시오.

구현하였던 결과와 같이 noise 또는 degradation process가 적용된 경우 image는 다음과 같다.

$$G(u, v) = F(u, v)H(u, v) + N(u, v)$$

이를 Inverse filtering의 과정을 거치면 다음과 같다.

$$\hat{F}(u, v) = F(u, v) + \frac{N(u, v)}{H(u, v)}$$

이 식에서, $N(u, v)/H(u, v)$ 의 경우, 작은 $N(u, v)$ 가 존재하더라도 $H(u, v)$ 가 0과 아주 가까운 경우 그 값이 매우 커질 수 있다. 이러한 경우 degrade된 noise를 복원하는 과정에서 해당 noise 성분이 매우 커져 이러한 오차로 인하여 완벽한 복원이 거의 불가능하다.

참조문헌

DIP_Homework4.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.