2018 컴퓨터비전

컴퓨터공학전공
오예진

개요

영상 개선 함수들을 구현하여 코드와 함께 적용한 이미지를 보인다.

Noisy image들을 개선하는 spatial-domain filter들을 구현하여 코드와 함께 개선된 이미지를 보인다.

2D FFT를 구현한다.

1. 영상개선함수

다음의 영상 개선 함수들을 구현하여,

코드와 함께 적용한 이미지를 보인다.

- 1. Negative transformation
- 2. Log, Inverse-log transformation
- 3. Root, Power transformation
- 4. Histogram equalization

Negative Transformation

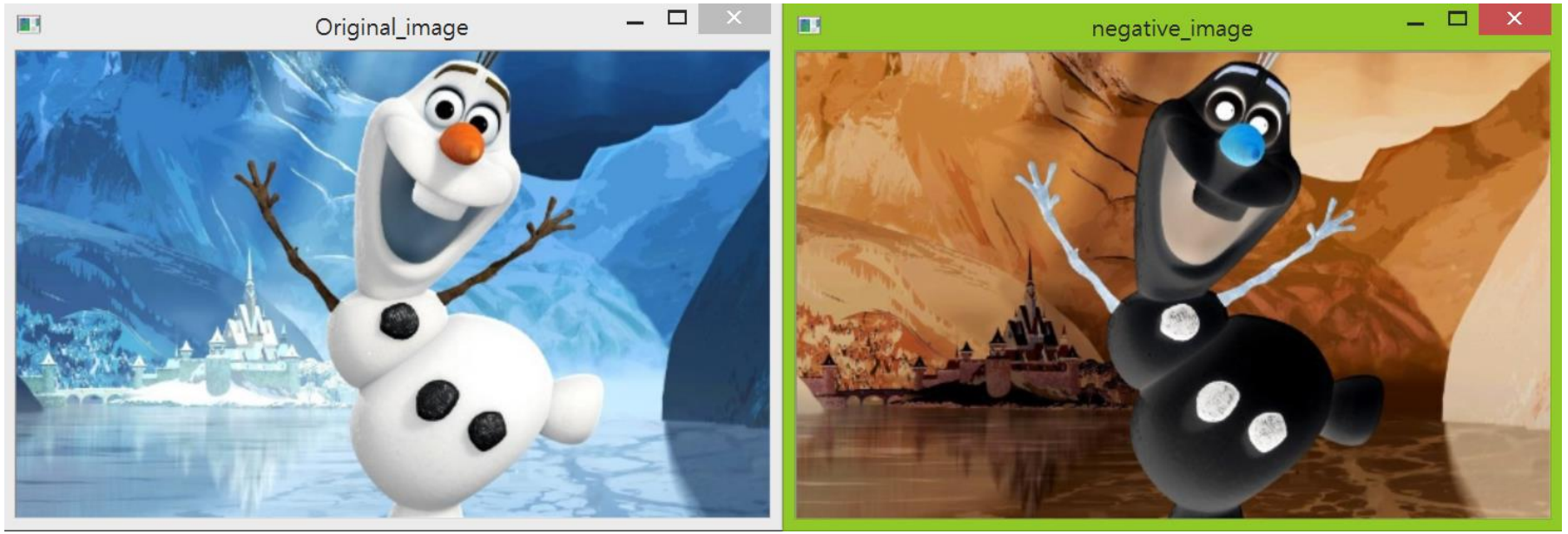
컬러 이미지는 하나의 픽셀마다 RGB 정보를 가지고 있고, 이 때 각각의 색깔 정보를 Channel이라고 한다. 일반적으로 각 채널은 8bit를 사용하므로 256가지의 색을 표현할 수 있고, 0~255까지의 값을 가질 수 있다.

negative transformation에서는 이미지의 색을 반전시키기 때문에, 각 채널의 값 v를 255-v로 대체한다.

```
Mat negative_img = original_img.clone();

for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        for (int channel = 0; channel < original_img.channels(); channel++) {
            negative_img.at<Vec3b>(row, col)[channel] = 255 - original_img.at<Vec3b>(row, col)[channel];      //v=255-v
        }
    }
}
```

Negative Transformation

원본 이미지와 비교하면 아래와 같다.



Log, Inverse-log

1. Log Transformation

Log transformation은 다음과 같은 식을 따른다

$$s = c * \log(1 + |r|)$$

source code는 다음과 같다. Log Transformation을 적용한 결과 이미지가 밝아진 것을 확인할 수 있다.

```
for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        for (int channel = 0; channel < original_img.channels(); channel++) {
            log_img.at<Vec3b>(row, col)[channel] = c * log(1 + original_img.at<Vec3b>(row, col)[channel]) / log(256);
        }
    }
}
```

Log, Inverse-log

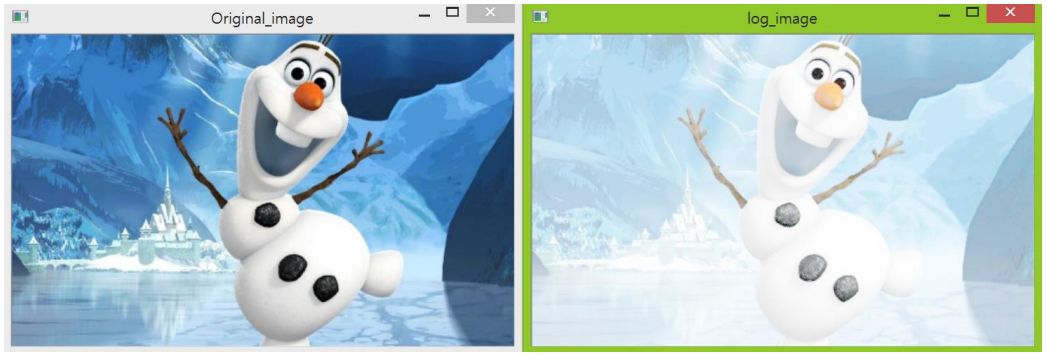
2. Inverse-log Transformation

Source code는 다음과 같다.

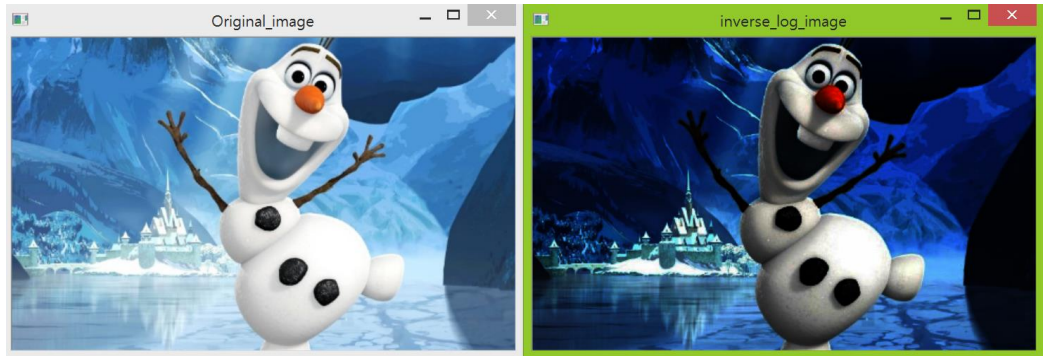
```
for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        for (int channel = 0; channel < original_img.channels(); channel++) {
            inverse_log_img.at<Vec3b>(row, col)[channel] = exp(log(256) / 255 * original_img.at<Vec3b>(row, col)[channel]) - 1;
        }
    }
}
```

Log, Inverse-log

1. Log transformation



2. Inverse-log transformaion



Root, Power Transformation.

비선형 연산에 해당하는 감마 수정의 식은 아래와 같다.

$$f_{out}(j,i) = (L-1) \times (\hat{f}(j,i))^{\gamma} \qquad \text{이때} \quad \hat{f}(j,i) = \frac{f(j,i)}{(L-1)}$$

Source code는 아래와 같다.

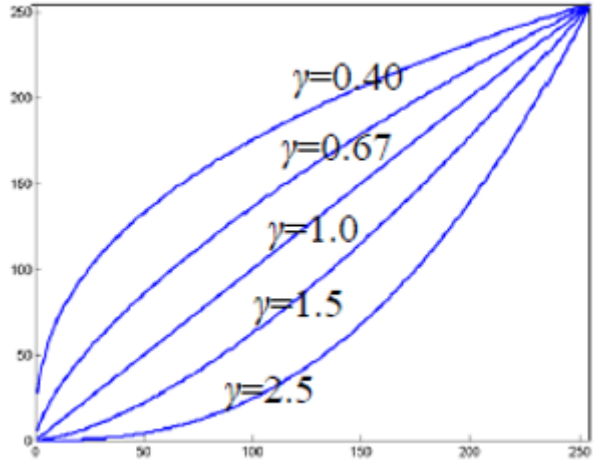
```
float c = 255;
float gamma = 0.5;        //gamma의 값은 유동적

for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        gamma_img.at<uchar>(row, col) = c * pow(original_img.at<uchar>(row, col) / c, gamma);
    }
}
```

Root, Power Transformation.

float변수 gamma의 값이 작을수록 이미지가 밝아지고, gamma의 값이 클수록 이미지가 어두워진다.

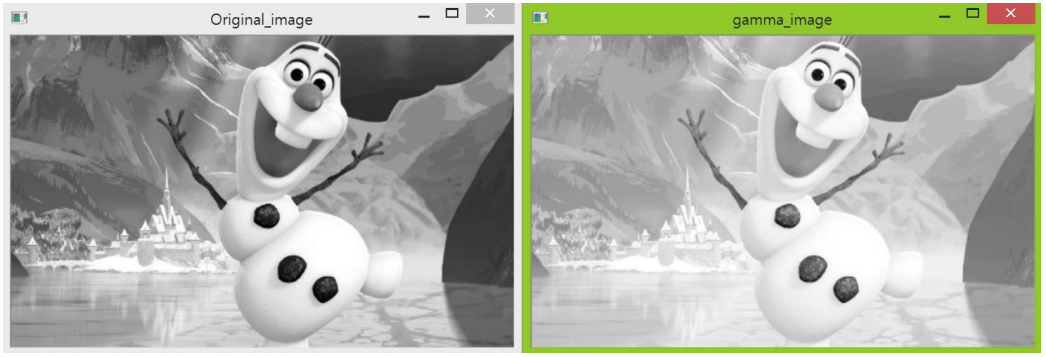
그렇기 때문에 code에서 gamma의 값은 유동적이고, 그에 따른 이미지 결과값은 아래와 같다. 감마수정에서는 이미지를 gray scale로 읽었다.

감마 수정을 통해서, 감마값에 따라 다음과 같은 변환함수의 모양을 확인할 수 있다.

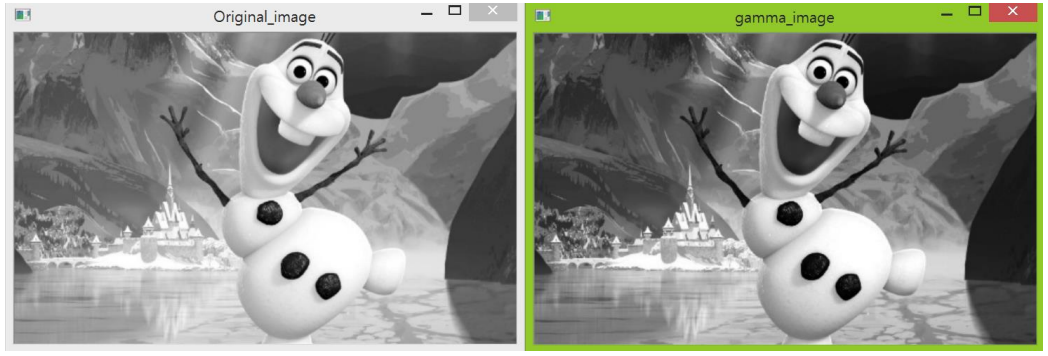


Root, Power Transformation.

GAMMA=0.5



GAMMA=1.5



Histogram Equalization

히스토그램, 정규화 히스토그램, 누적 히스토그램을 통해 히스토그램 평활화를 할 수 있다.

히스토그램 평활화 식은 아래와 같다.

$$l_{out} = T(l_{in}) = round(c(l_{in}) \times (L - 1))$$

$$\text{이때 } c(l_{in}) = \sum_{l=0}^{l_{in}} \hat{h}(l)$$

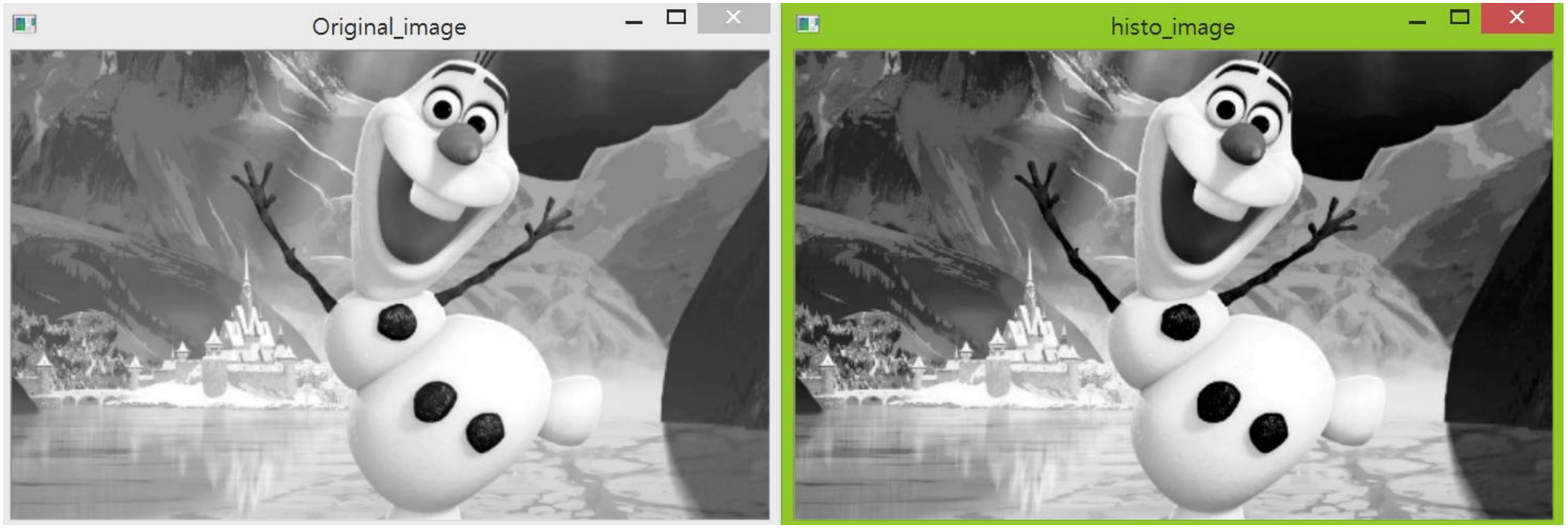
source code는 다음과 같다. 주어진 이미지의 히스토그램을 구한 후, 히스토그램 정규화를 거쳐 누적 히스토그램을 구하여 round함수와 함께 히스토그램 평활화를 한다.

Histogram Equalization

```
Mat histo_img = original_img.clone();

float histogram[256] = { 0 };        //히스토그램
float n_histogram[256] = { 0 };      //정규화된 히스토그램
float c_histogram[256] = { 0 };      //누적 히스토그램


for (int row = 0; row < original_img.rows; row++) {          // 1. 히스토그램
    for (int col = 0; col < original_img.cols; col++) {
        histogram[original_img.at<uchar>(row, col)]++;
    }
}

for (int i = 0; i < 256; i++) {              // 2.히스토그램 정규화
    n_histogram[i] = histogram[i] / (double)(original_img.rows*original_img.cols);
}

c_histogram[0] = n_histogram[0];      //3. 누적 히스토그램. 누적값이므로 처음 c_histogram[0]=n_histogram[0]이다.
for (int i = 1; i < 256; i++) {
    c_histogram[i] = c_histogram[i - 1] + n_histogram[i];
}

for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        histo_img.at<uchar>(row, col) = round(c_histogram[original_img.at<uchar>(row, col)] * 255);
    }
}
```

Histogram Equalization

히스토그램 평활화를 마친 결과는 다음과 같다.



2. Spatial domain filter

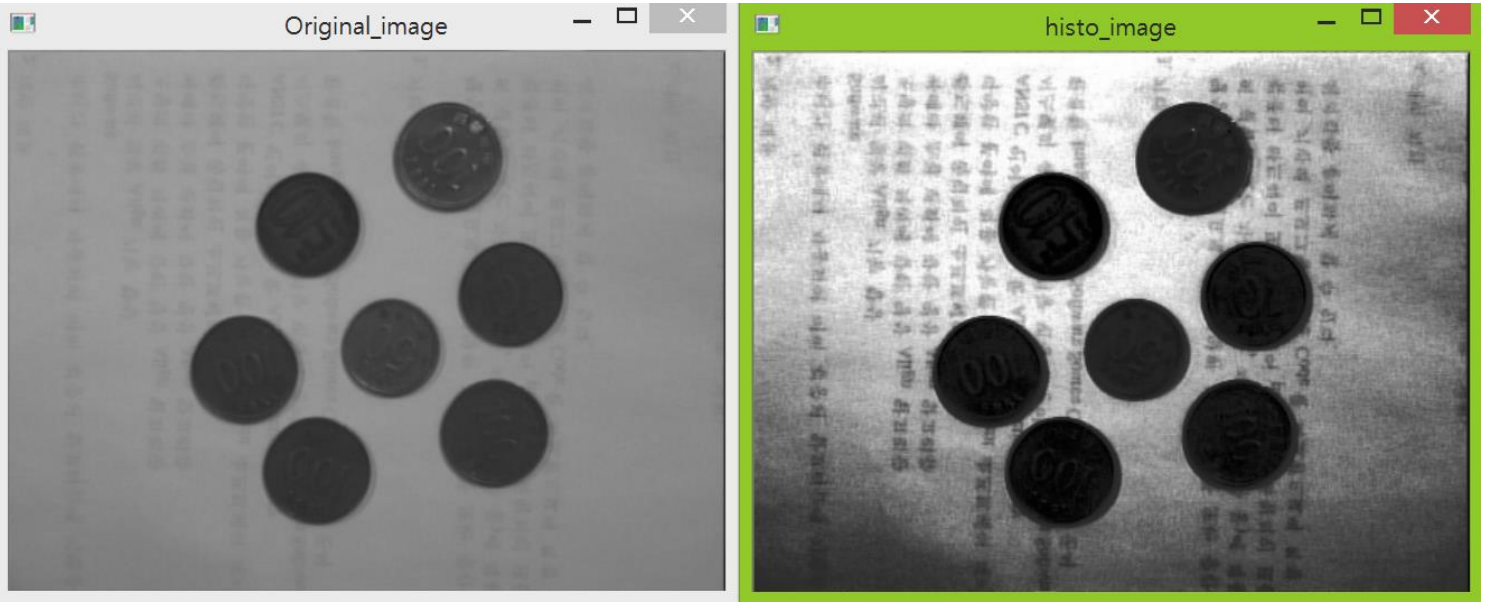
Noisy image들을 개선하는 spatial-domain filter들을 구현하여,

코드와 함께 개선된 이미지를 보인다.

low contrast 개선

low contrast를 개선하기 위해 영상개선함수로 만든 histogram equalization을 한다.

히스토그램 평활화를 사용하여 대조비가 높아진 것을 확인 할 수 있다. 실제 coin 이미지를 통해 나타난 결과값은 아래와 같다.



blur noise 개선

이미지의 blur noise를 개선하기 위해 샤프닝을 한다.

사용할 mask는 라플라시안 mask이며 왼쪽과 같다. 이 mask를 사용하면 edge를 찾아낼 수 있고, blur noise가 개선된 이미지 결과값을 얻기 위해 배열의 중앙값에 1씩 더한다. 이렇게 만들어진 mask는 오른쪽과 같다.

−1	−1	−1
−1	8	−1
−1	−1	−1

-1	-1	-1
-1	9	-1
-1	-1	-1

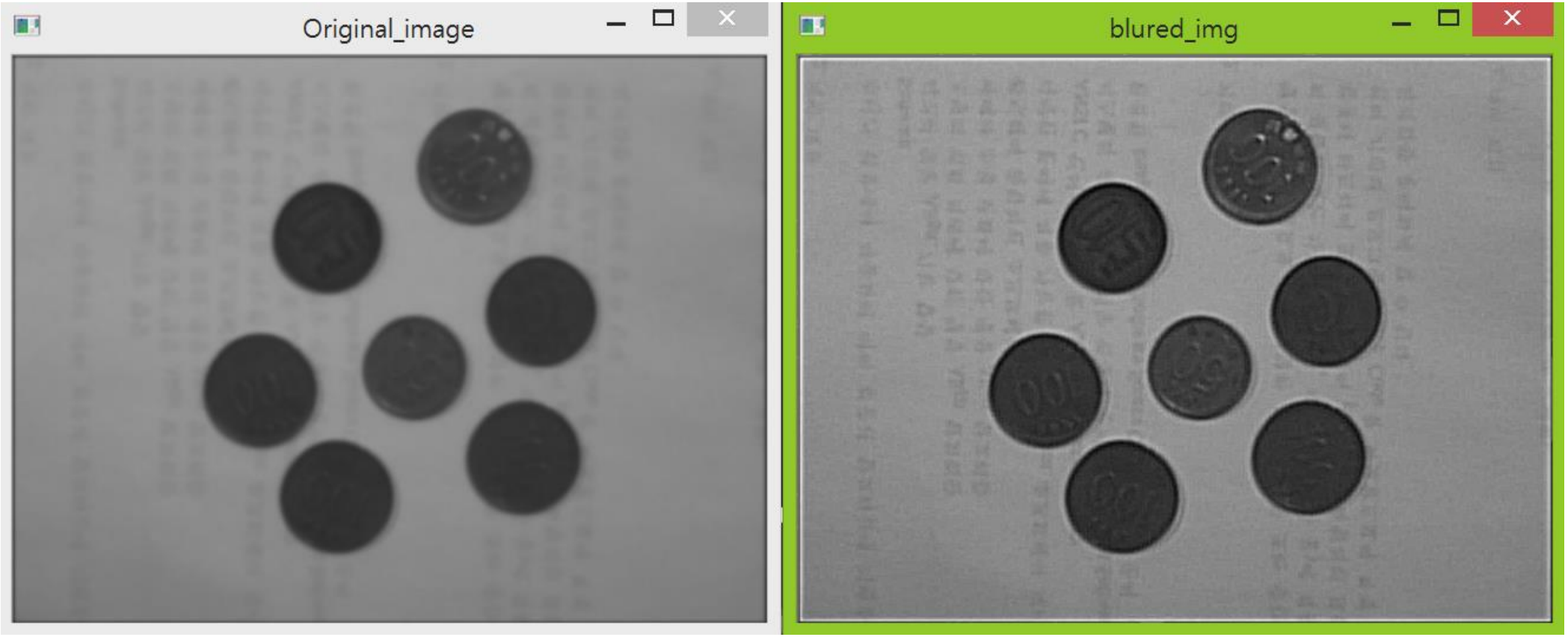
blur noise 개선

source code는 다음과 같다. 이미지 테두리에서는 계산을 할 수 없기 때문에, 마스크 크기/2만큼 테두리 영역을 무시하고 계산하고, mask_size/2가 이를 의미한다.

```
void blur_noise(Mat &original_img) {

    Mat blured_img = original_img.clone();

    int mask[3][3] = { {-1,-1,-1},{-1,9,-1},{-1,-1,-1} };     //라플라시안 +1

    long int sum;
    int mask_size = 3;          //mask[3][3]

    for (int row = 0 + mask_size / 2; row < original_img.rows - mask_size / 2; row++){
        for (int col = 0 + mask_size / 2; col < original_img.cols - mask_size / 2; col++){
            sum = 0;
            for (int i = -1 * mask_size / 2; i <= mask_size / 2; i++){
                for (int j = -1 * mask_size / 2; j <= mask_size / 2; j++){
                    sum += original_img.at<uchar>(row + i, col + j) * mask[mask_size / 2 + i][mask_size / 2 + j];
                }
            }

            if (sum > 255)          //명암은 255를 넘지 않게 한다.
                sum = 255;
            if (sum < 0)
                sum = 0;

            blured_img.at<uchar>(row, col) = sum;
        }
    }
```

blur noise 개선

샤프닝을 마친 이미지는 아래와 같다



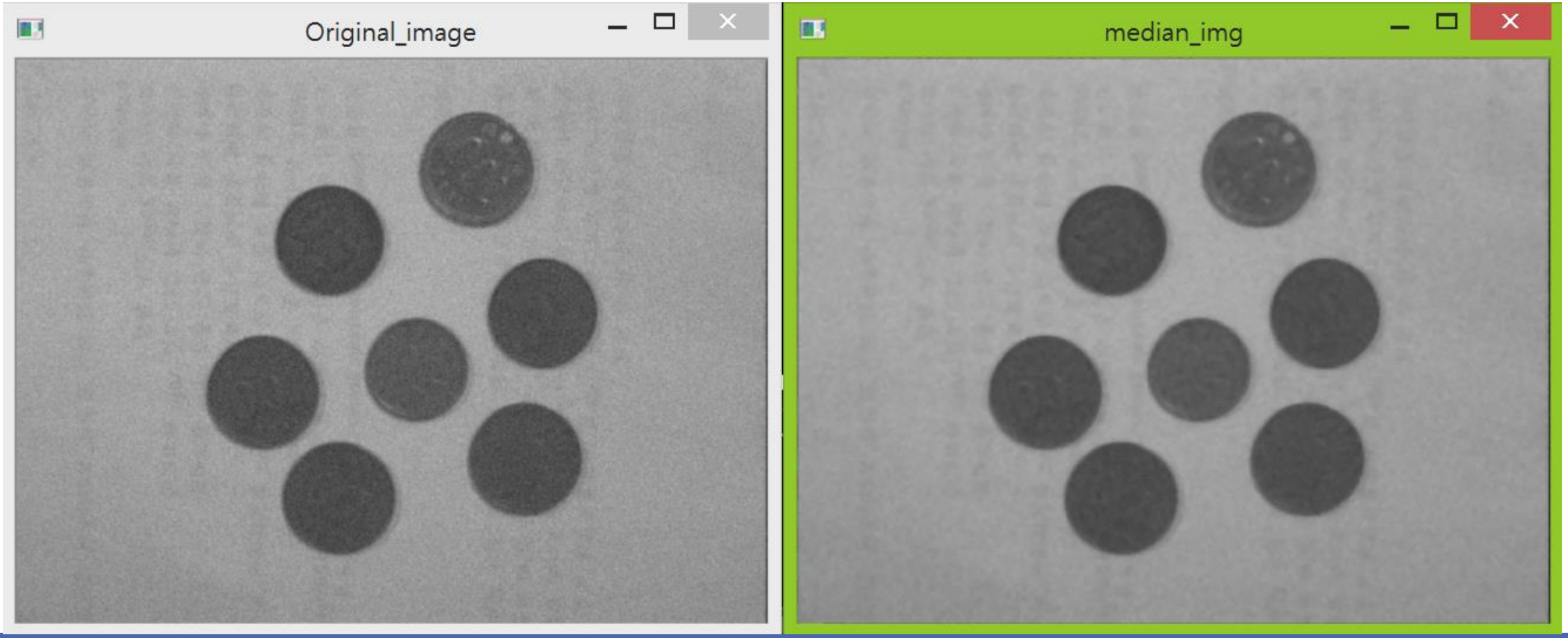
gaussian noise 개선
-median filter

가우시안 노이즈를 제거하기 위해 메디안필터를 사용해본다. 자세한 설명은 salt-pepper noise에서 설명하도록 한다. 우선 코드는 아래와 같다.

```
Mat median_img = original_img.clone();

int median_mask[9] = { 0 };     //3X3의 median mask
int mask_size = 3;          //mask[3][3]
int median_value = 0;    //한 pixel마다 구해진 median value

for (int row = 0 + mask_size / 2; row < original_img.rows - mask_size / 2; row++) {
    for (int col = 0 + mask_size / 2; col < original_img.cols - mask_size / 2; col++) {

        //median filter에 각 pixel의 값을 넣는다.
        median_mask[0] = original_img.at<uchar>(row - 1, col - 1);
        median_mask[1] = original_img.at<uchar>(row - 1, col);
        median_mask[2] = original_img.at<uchar>(row - 1, col + 1);
        median_mask[3] = original_img.at<uchar>(row, col - 1);
        median_mask[4] = original_img.at<uchar>(row, col);
        median_mask[5] = original_img.at<uchar>(row, col + 1);
        median_mask[6] = original_img.at<uchar>(row + 1, col - 1);
        median_mask[7] = original_img.at<uchar>(row + 1, col);
        median_mask[8] = original_img.at<uchar>(row + 1, col + 1);

        sort(median_mask, median_mask + 9);       //크기순으로 정렬
        median_value = median_mask[4];     //median_mask[4] = median_mask의 중앙값
        median_img.at<uchar>(row, col) = median_value;
    }
}
```

gaussian noise 개선
-median filter

가우시안 노이즈가 있는 이미지에 메디안 필터를 적용한 결과는 아래와 같다.



gaussian noise 개선
-mean filter

noise를 제거하기 위해 mean filter를 사용한다. mean filter는 아래와 같은데, 3X3 mean filter의 경우 인접한 9개 픽셀들의 값을 합쳐서 평균을 낸다.

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

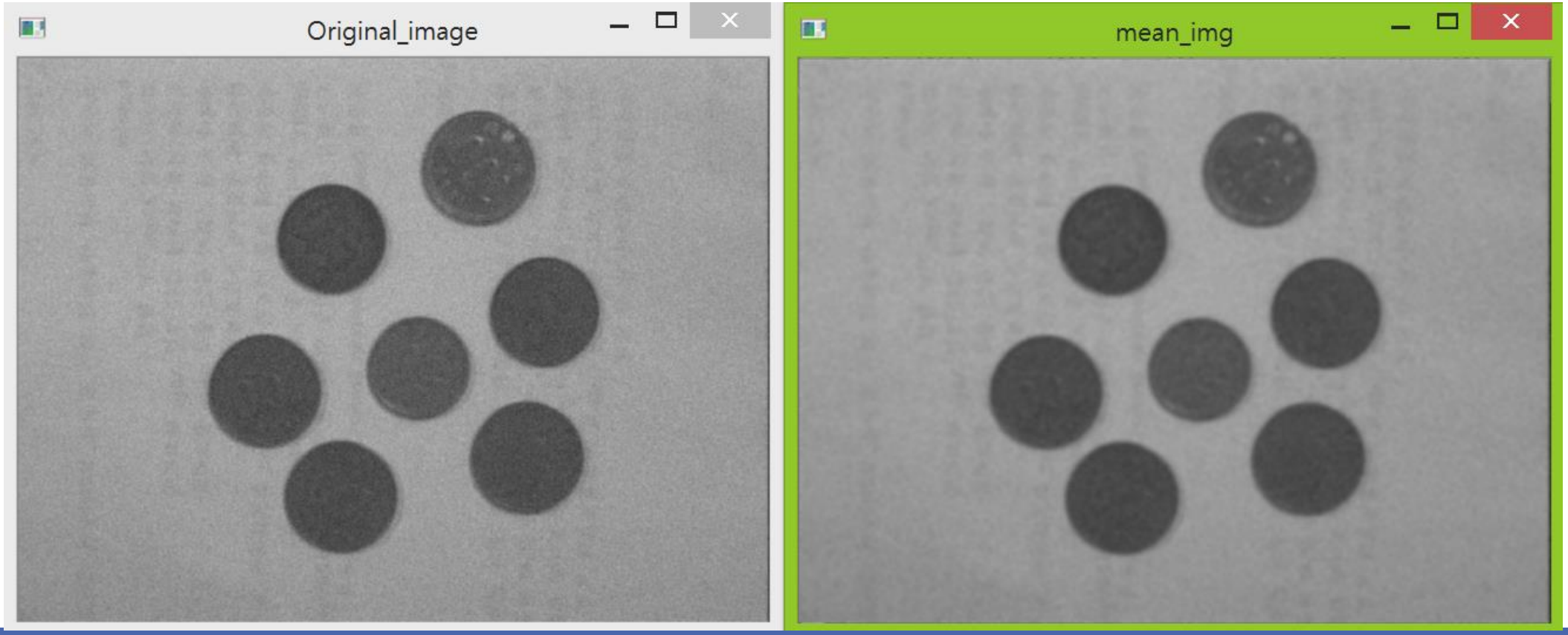
gaussian noise 개선 -mean filter

mean filter를 구현하는 코드는 아래와 같다.

```
Mat mean_img = original_img.clone();

int mean_mask[9] = { 0 };        //3X3의 mean
int mask_size = 3;          //mask[3][3]
int mean_value = 0;  //총 9개의 계산값을 모두 더하는 value

for (int row = 0 + mask_size / 2; row < original_img.rows - mask_size / 2; row++) {
    for (int col = 0 + mask_size / 2; col < original_img.cols - mask_size / 2; col++) {
        mean_value = 0;

        //mean filter에 각 pixel의 값을 넣는다.
        mean_mask[0] = original_img.at<uchar>(row - 1, col - 1)/9;
        mean_mask[1] = original_img.at<uchar>(row - 1, col)/9;
        mean_mask[2] = original_img.at<uchar>(row - 1, col + 1)/9;
        mean_mask[3] = original_img.at<uchar>(row, col - 1)/9;
        mean_mask[4] = original_img.at<uchar>(row, col)/9;
        mean_mask[5] = original_img.at<uchar>(row, col + 1)/9;
        mean_mask[6] = original_img.at<uchar>(row + 1, col - 1)/9;
        mean_mask[7] = original_img.at<uchar>(row + 1, col)/9;
        mean_mask[8] = original_img.at<uchar>(row + 1, col + 1)/9;

        for (int k = 0; k < 9; k++) {
            mean_value += mean_mask[k];
        }
        mean_img.at<uchar>(row, col) = mean_value;
    }
}
```

gaussian noise 개선
-mean filter

이와 같은 코드를 사용하여 노이즈를 제거한 결과는 다음과 같다.



salt-pepper noise 개선 -median filter

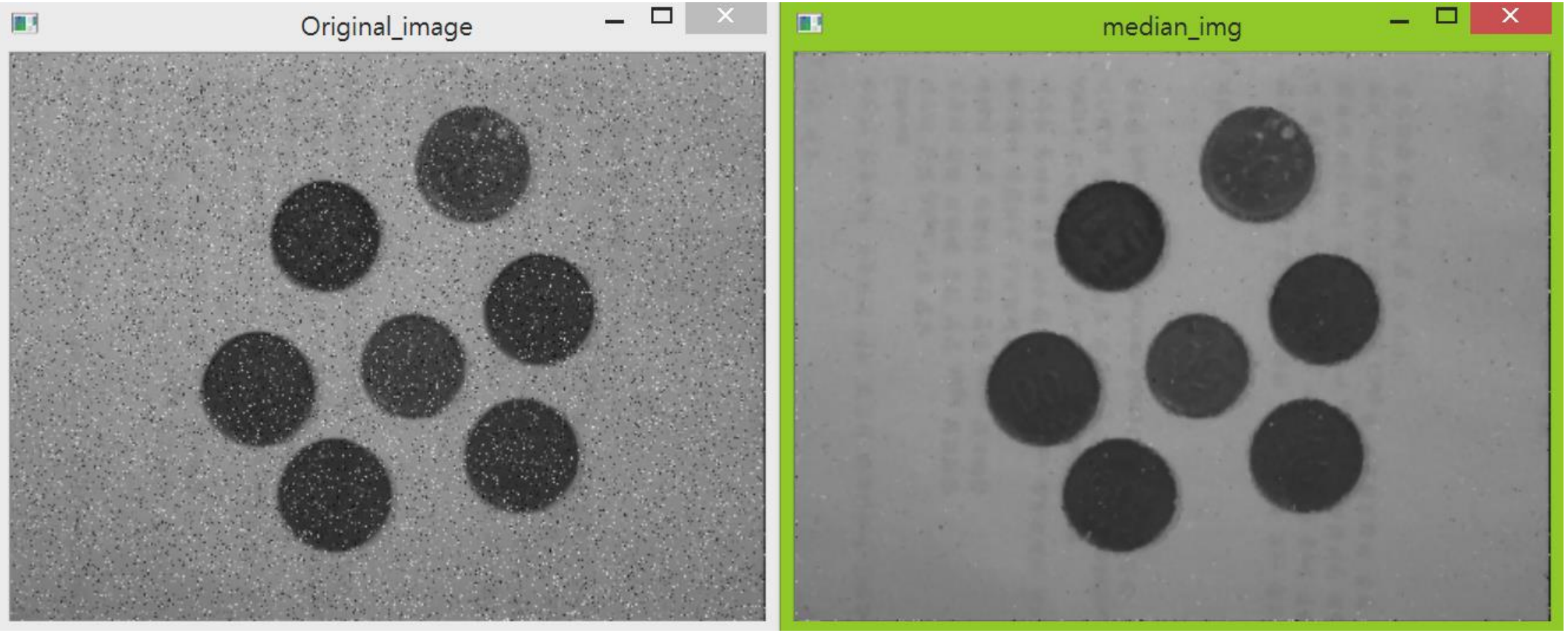
3X3 median filter에서는 9개의 pixel값 중 가운데 값에 해당하는 pixel을 구하여 계산하는데, 이렇게 하면 impulse noise에 해당하는 salt-pepper noise에 효과적이다.

source code는 다음과 같다.

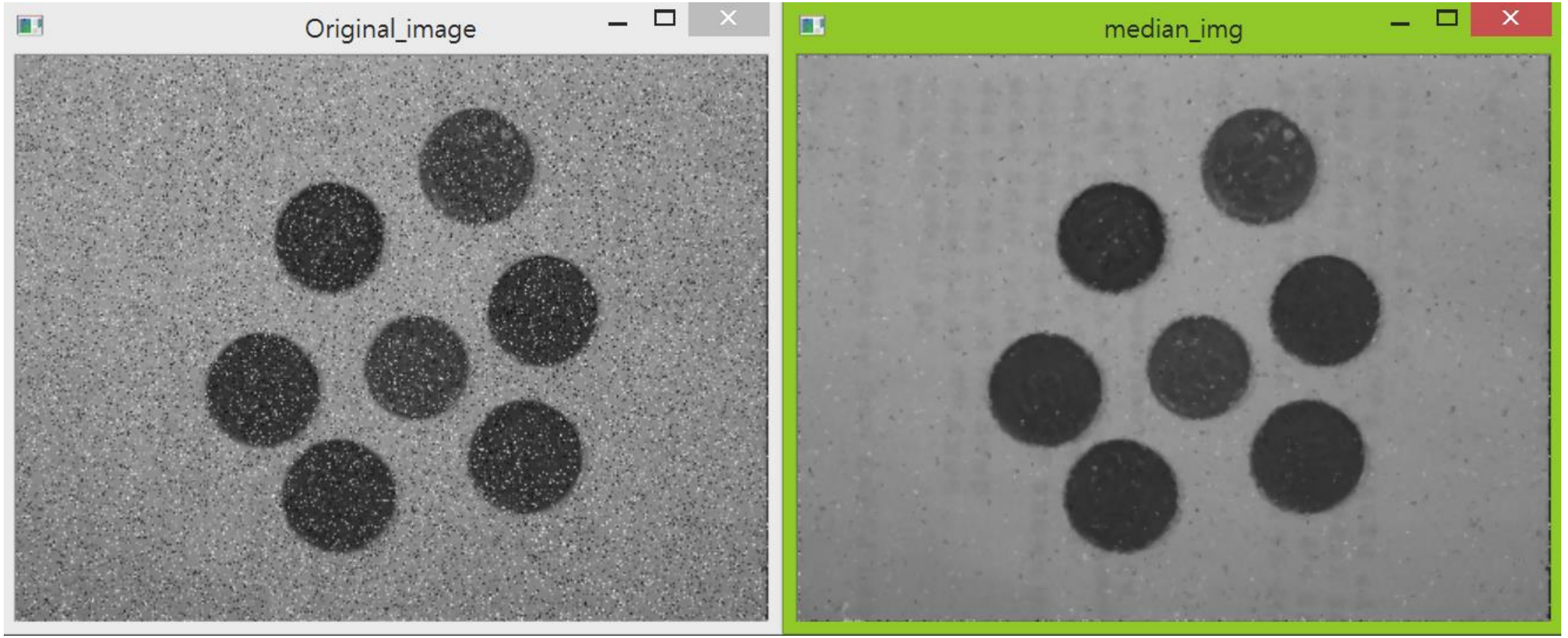
배열에 9개의 pixel값을 구하여 이들 중 가운데 값을 찾아 median_img에 대입하여 최종값을 얻는 과정이다.

```
Mat median_img = original_img.clone();

int median_mask[9] = { 0 };     //3X3의 median mask
int mask_size = 3;          //mask[3][3]
int median_value = 0;    //한 pixel마다 구해진 median value

for (int row = 0 + mask_size / 2; row < original_img.rows - mask_size / 2; row++) {
    for (int col = 0 + mask_size / 2; col < original_img.cols - mask_size / 2; col++) {

        //median filter에 각 pixel의 값을 넣는다.
        median_mask[0] = original_img.at<uchar>(row - 1, col - 1);
        median_mask[1] = original_img.at<uchar>(row - 1, col);
        median_mask[2] = original_img.at<uchar>(row - 1, col + 1);
        median_mask[3] = original_img.at<uchar>(row, col - 1);
        median_mask[4] = original_img.at<uchar>(row, col);
        median_mask[5] = original_img.at<uchar>(row, col + 1);
        median_mask[6] = original_img.at<uchar>(row + 1, col - 1);
        median_mask[7] = original_img.at<uchar>(row + 1, col);
        median_mask[8] = original_img.at<uchar>(row + 1, col + 1);

        sort(median_mask, median_mask + 9);       //크기순으로 정렬
        median_value = median_mask[4];     //median_mask[4] = median_mask의 중앙값
        median_img.at<uchar>(row, col) = median_value;
    }
}
```

salt-pepper noise 개선
-median filter

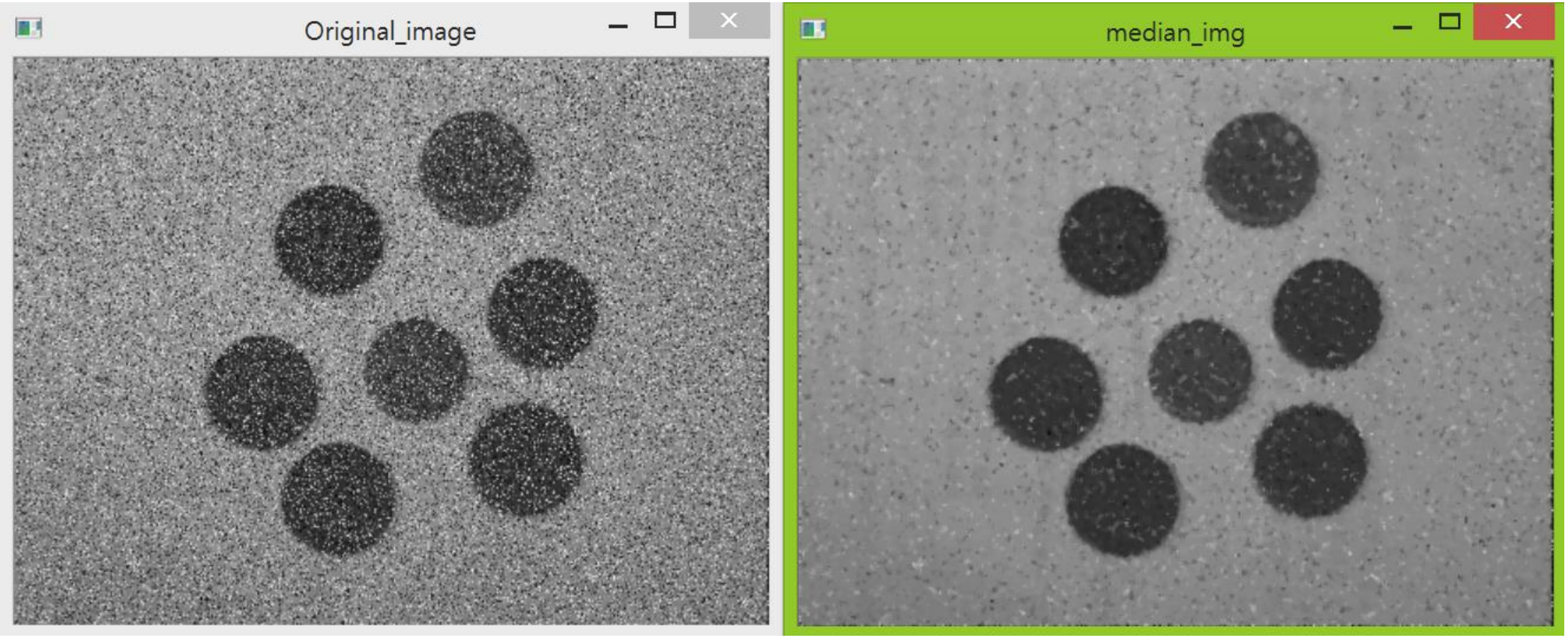


salt-pepper noise 개선
-median filter



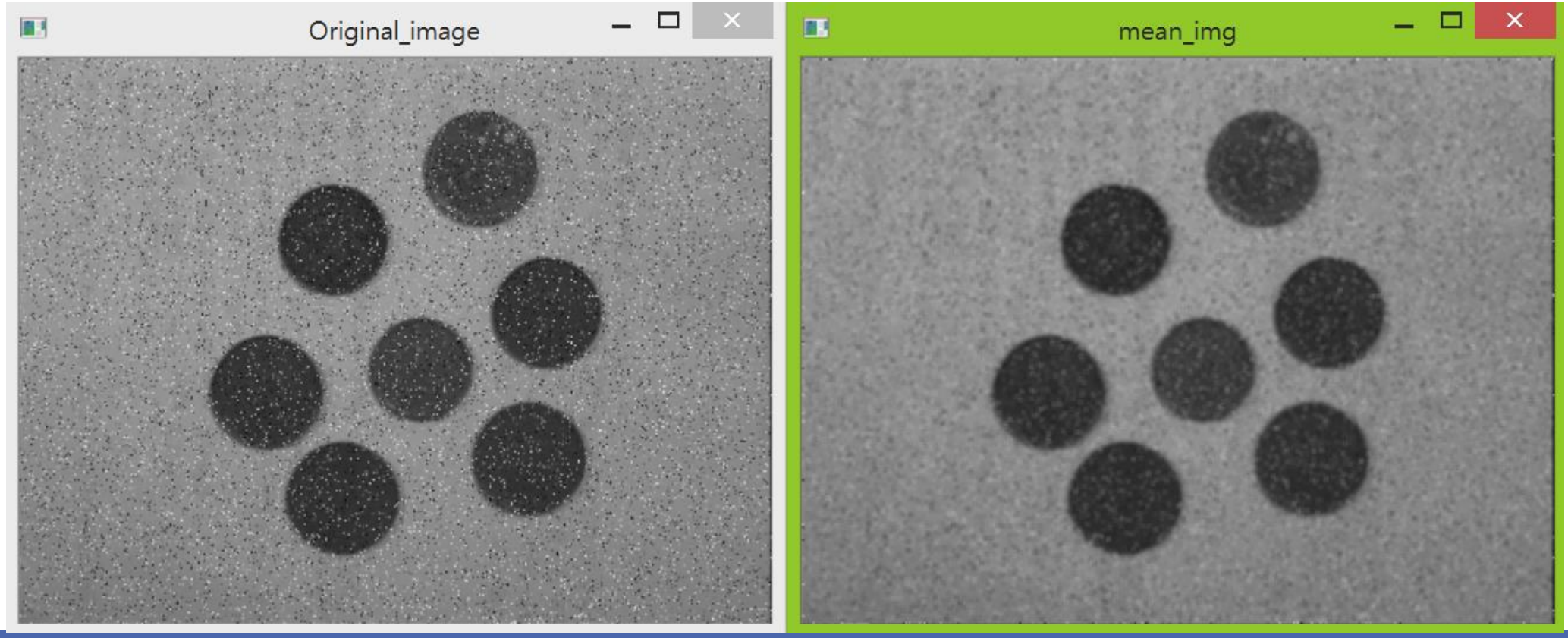
salt-pepper noise 개선
-median filter

다음의 3장의 사진으로부터 salt-pepper noise의 정도가 점점 커질 때 그에 따른 smoothing 결과를 확인할 수 있다.



salt-pepper noise 개선 -mean filter

앞에서 구현한 mean filter를 salt-pepper noise에 적용해보면 다음과 같다. 인접한 모든 픽셀을 더한 후 평균을 내는 filter이기 때문에 salt-pepper noise에는 적합하지 않음을 알 수 있다.



3. FFT

2D FFT를 구현한다.

FFT

푸리에 변환이란, 임의의 신호를 다양한 주파수를 가지는 sin, cos함수들의 합으로 표현하는 것이다. 이미지는 2차원의 신호이므로 2D 푸리에 변환이 필요하며 그 식은 다음과 같다.

$$\Im\{f(x,y)\} = F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\exp[-j2\pi(ux+vy)]dxdy$$

$$\Im^{-1}\{f(u,v)\} = F(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(u,v)\exp[-j2\pi(ux+vy)]dudv$$

FFT

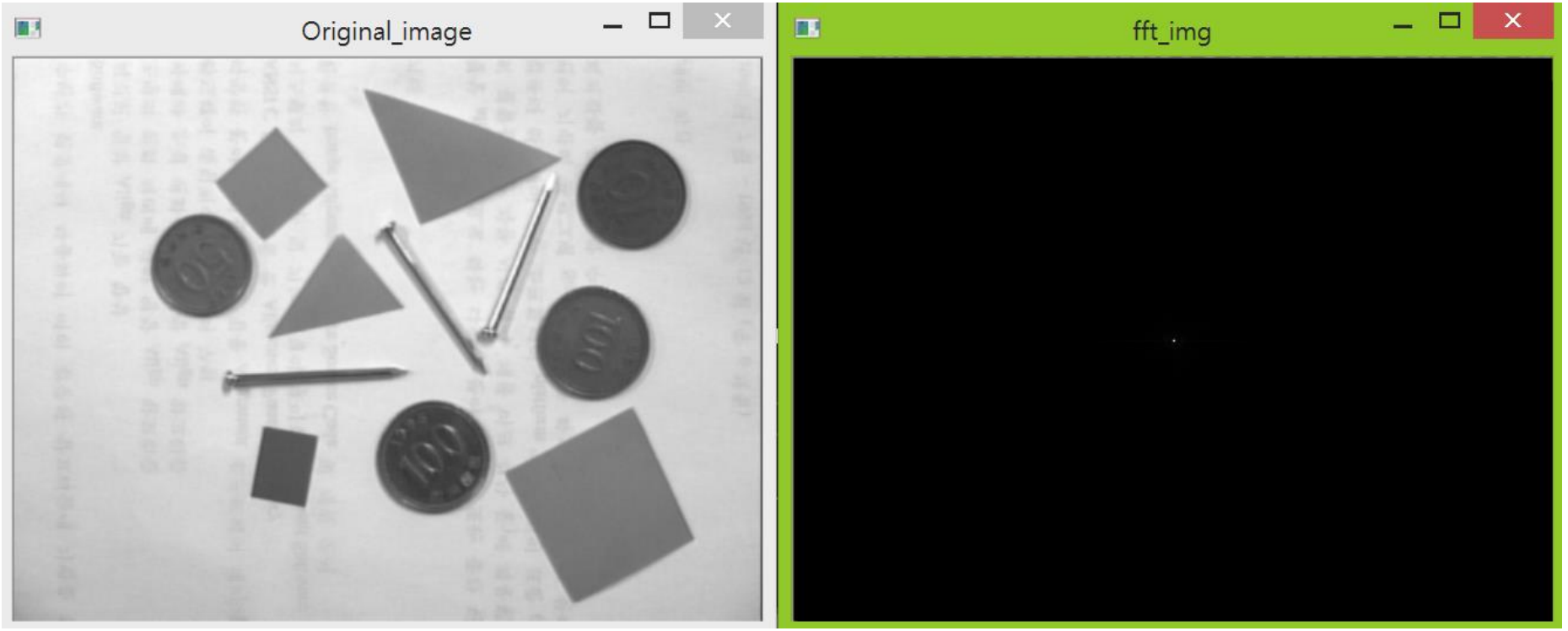
FFT code는 다음과 같다.

https://docs.opencv.org/2.4/doc/tutorials/core/discrete_fourier_transform/discrete_fourier_transform.html 을 참고하여 구현하였다.

```
//FFT(Fast Fourier Transform)
//opencv 함수를 사용한다

Mat padded_img;                         //getOptimalDFTSize를 사용해 Optimalsize로 만들기 위한 img

int row = getOptimalDFTSize(original_img.rows);
int col = getOptimalDFTSize(original_img.cols);
copyMakeBorder(original_img, padded_img, 0, row - original_img.rows, 0, col - original_img.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = { Mat_<float>(padded_img), Mat::zeros(padded_img.size(), CV_32F) };
Mat complexI;
merge(planes, 2, complexI);

//result=>fit matrix
dft(complexI, complexI);

split(complexI, planes);
magnitude(planes[0], planes[1], planes[0]);
Mat magI = planes[0];

magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

//center에 위치하도록 사분면을 재배치한다
int cx = magI.cols / 2;
int cy = magI.rows / 2;

Mat q0(magI, Rect(0, 0, cx, cy));
Mat q1(magI, Rect(cx, 0, cx, cy));
Mat q2(magI, Rect(0, cy, cx, cy));
Mat q3(magI, Rect(cx, cy, cx, cy));

Mat tmp;
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);
q2.copyTo(q1);
tmp.copyTo(q2);

normalize(magI, magI, 0, 1, CV_MINMAX);      //value=>0~1

//namedWindow("fft_img", CV_WINDOW_AUTOSIZE);
imshow("fft_img", magI);
waitKey(0);
//destroyWindow("fft_img");
```

FFT

위의 코드를 통해 영상에 FFT를 적용시킨 결과는 아래와 같다. log scale을 하기 전이기 때문에 제대로 결과값을 분석하기 어렵다.



FFT

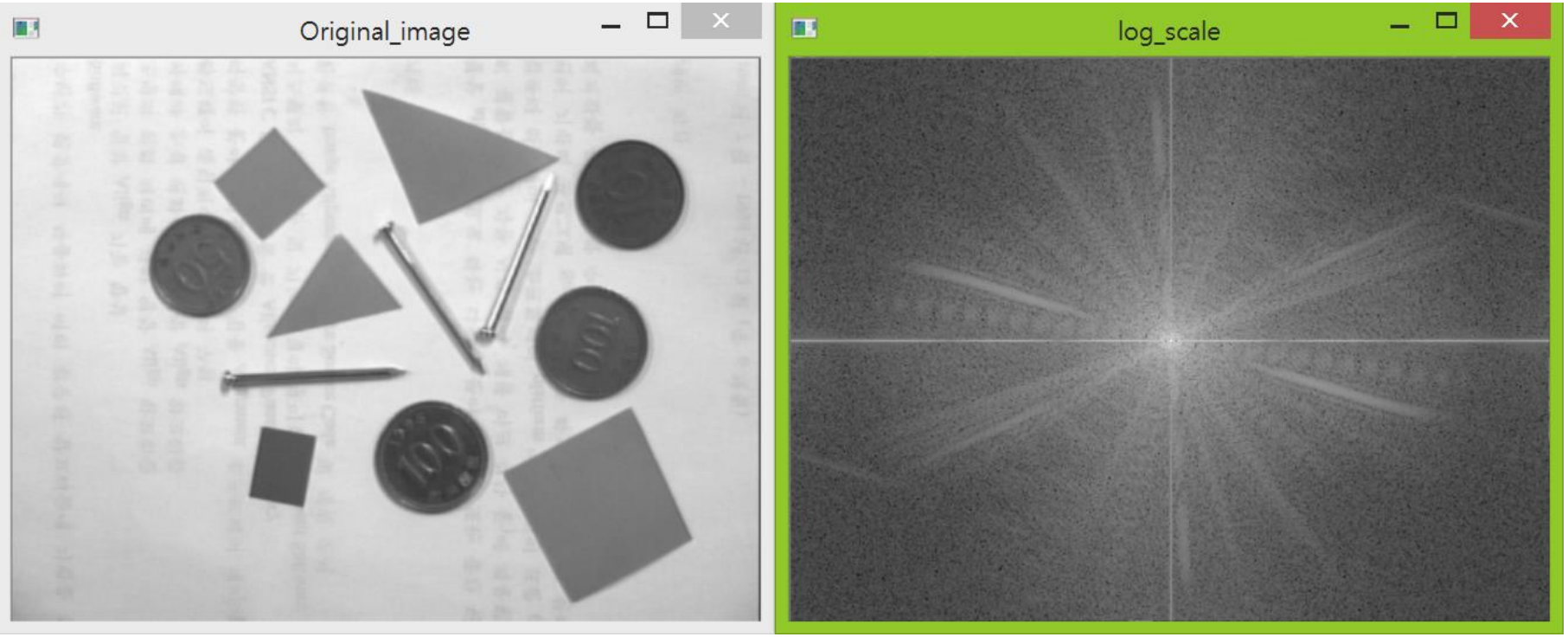
이를 해결하기 위해 다음과 같은 log식을 참고하여 log scale 한다. 코드는 아래와 같다.

$$M_1 = \log(1 + M)$$

```
//Log scale한다
//log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
magI += Scalar::all(1);
log(magI, magI);
```

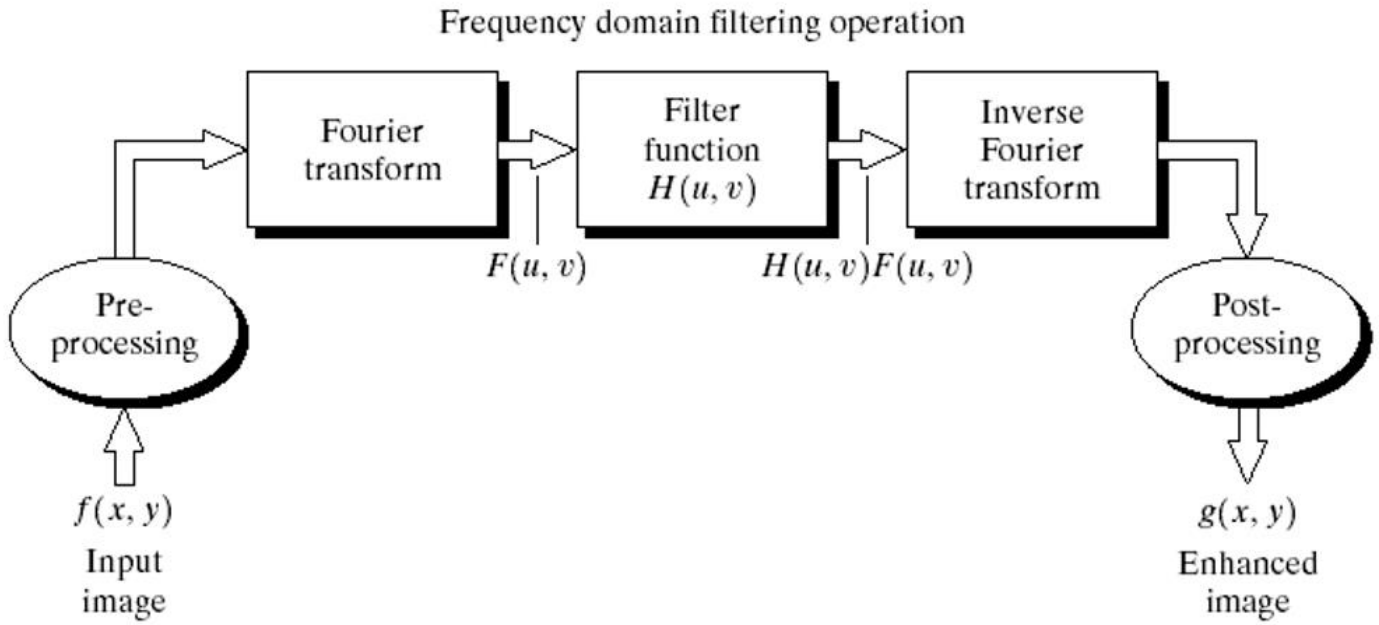
FFT

log scale한 영상을 다음과 같이 확인할 수 있다.



LPF

LPF와 같은 주파수 영역에서의 영상 개선은 다음과 같은 절차로 이루어진다.



$$f(x,y) \xrightarrow{F.T} F(u,v) \xrightarrow[H(u,v)]{filtering} G(u,v) = F(u,v) \cdot H(u,v) \xrightarrow{IFT} g(x,y)$$

LPF

구현코드는 다음와 같다.LPF(H(u,v))는 일정 값을 기준으로 0(제거),1(그대로)을 구별하는데 여기선 100*100으로 마스크사이즈를 지정하여 이를 수행하였다.

LPF구현을 마친 후에 이를 FFT를 마친 영상에 적용하고(G(u,v)), 마지막으로 Inverse FFT하면(g(x,y)) 개선된 영상을 구할 수 있다.

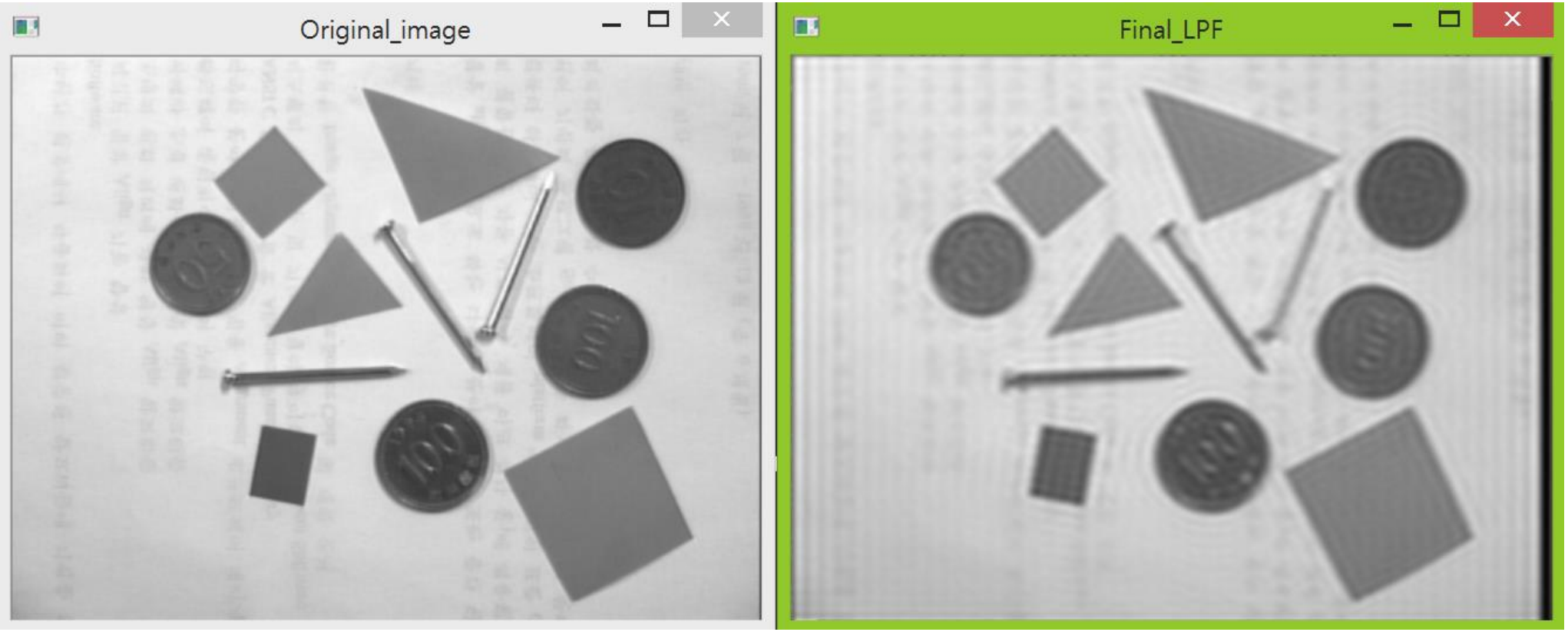
LPF

```
//image->FFT->LPF적용->Inverse LPF의 순서를 거친다.

Mat padded_img;                         //getOptimalDFTSize를 사용해 Optimalsize로 만들기 위한 img

int row = getOptimalDFTSize(original_img.rows);
int col = getOptimalDFTSize(original_img.cols);
copyMakeBorder(original_img, padded_img, 0, row - original_img.rows, 0, col - original_img.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = { Mat_<float>(padded_img), Mat::zeros(padded_img.size(), CV_32F) };
Mat complexI;
merge(planes, 2, complexI);

//result=>fit matrix
dft(complexI, complexI);
split(complexI, planes);

Shift(planes[0]);
Shift(planes[1]);

Mat Mask(complexI.rows, complexI.cols, CV_32FC1, Scalar(0));
int MaskSize = 100;       //mask size는 변경 가능하다.
Mask(Rect(complexI.cols / 2 - (MaskSize / 2), complexI.rows / 2 - (MaskSize / 2), MaskSize, MaskSize)) = 1;

planes[0] = planes[0].mul(Mask);
planes[1] = planes[1].mul(Mask);

Shift(planes[0]);
Shift(planes[1]);

Mat Final_LPF;
merge(planes, 2, Final_LPF);

dft(Final_LPF, Final_LPF, DFT_INVERSE | DFT_REAL_OUTPUT);
normalize(Final_LPF, Final_LPF, 0, 1, CV_MINMAX);
convertScaleAbs(Final_LPF, Final_LPF, 255.0);

//namedWindow("Final_LPF", CV_WINDOW_AUTOSIZE);
imshow("Final_LPF", Final_LPF);
waitKey(0);
//destroyWindow("Final_LPF");
```

```
void Shift(Mat &planes) {
    planes = planes(Rect(0, 0, planes.cols & -2, planes.rows & -2));
    int cy = planes.rows / 2;
    int cx = planes.cols / 2;

    //Rect(x,y,width, height)
    Mat q0(planes, Rect(0, 0, cx, cy));
    Mat q1(planes, Rect(cx, 0, cx, cy));
    Mat q2(planes, Rect(0, cy, cx, cy));
    Mat q3(planes, Rect(cx, cy, cx, cy));

    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
}
```

LPF

LPF를 적용한 영상은 아래와 같다. LPF는 저주파를 통과시키기 때문에, blur처리된 image를 확인할 수 있다.



HPF

영상의 high frequency부분은 보통 명암이 급격히 변하는 edge부분에 해당하므로 HPF를 통해 영상의 edge를 추출할 수 있다.

코드는 다음과 같다. LPF와 반대로 주파수가 높은 부분만을 1로한다(통과시킨다).

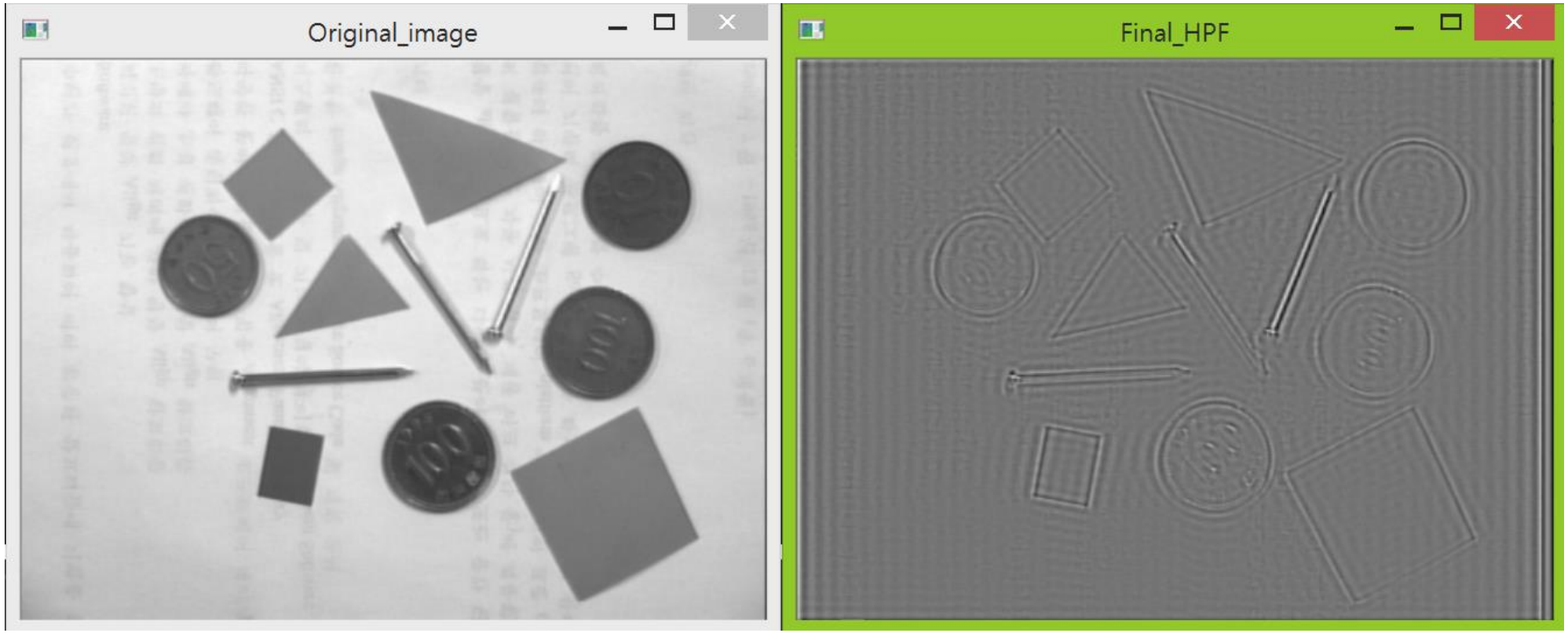
HPF

```
//image->FFT->HPF적용->Inverse HPF의 순서를 거친다.
Mat padded_img;                         //getOptimalDFTSize를 사용해 Optimalsize로 만들기 위한 img

int row = getOptimalDFTSize(original_img.rows);
int col = getOptimalDFTSize(original_img.cols);
copyMakeBorder(original_img, padded_img, 0, row - original_img.rows, 0, col - original_img.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = { Mat_<float>(padded_img), Mat::zeros(padded_img.size(), CV_32F) };
Mat complexI;
merge(planes, 2, complexI);

//result=>fit matrix
dft(complexI, complexI);
split(complexI, planes);

Shift(planes[0]);
Shift(planes[1]);

Mat Mask(complexI.rows, complexI.cols, CV_32FC1, Scalar(1));
int MaskSize = 100;
Mask(Rect(complexI.cols / 2 - (MaskSize / 2), complexI.rows / 2 - (MaskSize / 2), MaskSize, MaskSize)) = 0;

planes[0] = planes[0].mul(Mask);
planes[1] = planes[1].mul(Mask);

Shift(planes[0]);
Shift(planes[1]);

Mat Final_HPF;
merge(planes, 2, Final_HPF);

dft(Final_HPF, Final_HPF, DFT_INVERSE | DFT_REAL_OUTPUT);
normalize(Final_HPF, Final_HPF, 0, 1, CV_MINMAX);
convertScaleAbs(Final_HPF, Final_HPF, 255.0);

//namedWindow("Final_HPF", CV_WINDOW_AUTOSIZE);
imshow("Final_HPF", Final_HPF);
waitKey(0);
//destroyWindow("Final_HPF");
```

```
void Shift(Mat &planes) {
    planes = planes(Rect(0, 0, planes.cols & -2, planes.rows & -2));
    int cy = planes.rows / 2;
    int cx = planes.cols / 2;

    //Rect(x,y,width, height)
    Mat q0(planes, Rect(0, 0, cx, cy));
    Mat q1(planes, Rect(cx, 0, cx, cy));
    Mat q2(planes, Rect(0, cy, cx, cy));
    Mat q3(planes, Rect(cx, cy, cx, cy));

    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
}
```

HPF

High Pass Filter(H(u,v))를 FFT된 영상에 적용시켜(G(u,v)) Inverse FFT한 결과(g(x,y))는 다음과 같다.
edge부분이 드러난 것을 확인할 수 있다.



HPE

고주파를 강조해주는 HPE를 구현하기 위해서, 앞서 구현한 HPF필터를 사용한다.
고주파가 강조된 영상과 원본영상을 합하여 고주파를 강조하는 영상을 결과로 한다.
구현 코드는 다음와 같다.

HPE

```
//image->FFT->HPF적용->Inverse HPF의 순서를 거친다.
Mat padded_img;                         //getOptimalDFTSize를 사용해 Optimalsize로 만들기 위한 img
Mat HPE_img = original_img.clone();

int row = getOptimalDFTSize(original_img.rows);
int col = getOptimalDFTSize(original_img.cols);
copyMakeBorder(original_img, padded_img, 0, row - original_img.rows, 0, col - original_img.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = { Mat_<float>(padded_img), Mat::zeros(padded_img.size(), CV_32F) };
Mat complexI;
merge(planes, 2, complexI);

//result=>fit matrix
dft(complexI, complexI);
split(complexI, planes);

Shift(planes[0]);
Shift(planes[1]);

Mat Mask(complexI.rows, complexI.cols, CV_32FC1, Scalar(1));
int MaskSize = 100;
Mask(Rect(complexI.cols / 2 - (MaskSize / 2), complexI.rows / 2 - (MaskSize / 2), MaskSize, MaskSize)) = 0;

planes[0] = planes[0].mul(Mask);
planes[1] = planes[1].mul(Mask);

Shift(planes[0]);
Shift(planes[1]);

Mat Final_HPF;
merge(planes, 2, Final_HPF);

dft(Final_HPF, Final_HPF, DFT_INVERSE | DFT_REAL_OUTPUT);
normalize(Final_HPF, Final_HPF, 0, 1, CV_MINMAX);
convertScaleAbs(Final_HPF, Final_HPF, 255.0);

//원본영상 + 고주파가 강조된 영상
for (int row = 0; row < original_img.rows; row++) {
    for (int col = 0; col < original_img.cols; col++) {
        HPE_img.at<uchar>(row, col) = original_img.at<uchar>(row, col) + Final_HPF.at<uchar>(row, col);
    }
}

//namedWindow("HPE_img", CV_WINDOW_AUTOSIZE);
imshow("HPE_img", HPE_img);
waitKey(0);
//destroyWindow("HPE_img");
```

```
void Shift(Mat &planes) {
    planes = planes(Rect(0, 0, planes.cols & -2, planes.rows & -2));
    int cy = planes.rows / 2;
    int cx = planes.cols / 2;

    //Rect(x,y,width, height)
    Mat q0(planes, Rect(0, 0, cx, cy));
    Mat q1(planes, Rect(cx, 0, cx, cy));
    Mat q2(planes, Rect(0, cy, cx, cy));
    Mat q3(planes, Rect(cx, cy, cx, cy));

    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);

    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
}
```

HPE

적용한 결과는 아래와 같다. 원본영상에서 엣지부분이 눈에 띄는 모습을 볼 수 있다.