1. salt_and_pepper.cpp

코드 목적:

salt and pepper noise 를 만들고, 이를 meanfilter 를 사용해 제거한다.

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지를 만든다.
2) 컬러 이미지와 흑백 이미지를 각각 Add_salt_pepper_Noise()함수에 넣어 noise가 생긴 이미지를 return 받는다.
3) (2)의 흑백 이미지는 Salt_pepper_noise_removal_Gray() 함수에, 컬러이미지는 Salt_pepper_noise_removal_RGB에 넣어 noise를 제거한다.
4) 모든 과정에서의 이미지를 새로운 창에 띄운다. (원본이미지, 원본이미지 흑백, salt and pepper noise가진 흑백 이미지, salt and pepper noise가진 컬러 이미지, meanfiltering한 흑백 이미지, meanfiltering한 컬러 이미지)

함수 설명: Add_salt_pepper_Noise(const Mat input, float ps, float pp)

매개변수:

input: input 이미지 행렬
ps: density of salt noise (0~1)
pp: density of pepper noise (0~1)

함수 목적:

input 이미지에 ps 비율의 salt noise 와 pp 비율의 pepper noise 를 만들어 반환

Mat Salt_pepper_noise_removal_Gray(const Mat input, int n, const char *opt)

매개변수:

input: input 이미지 행렬(흑백)
n: kernel 의 크기 (2n+1)x(2n+1)
opt: type of boundary processing. image boundary 의 픽셀 처리 방법
"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: meanfilter를 이용해 salt and pepper noise를 제거한 이미지를 반환 (흑백)

Mat Salt_pepper_noise_removal_RGB(const Mat input, int n, const char *opt);

매개변수:

input: input 이미지 행렬(컬러)
n: kernel 의 크기 (2n+1)x(2n+1)
opt: type of boundary processing. image boundary 의 픽셀 처리 방법
"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: meanfilter를 이용해 salt and pepper noise를 제거한 이미지를 반환 (컬러)

코드 설명:

1) Add_salt_pepper_Noise()

salt pepper Noise 는 무작위로 선택된 픽셀의 색 값을 0 또는 255 로 바꿔 흰색 또는 검은 색의 점이 생기게 하는 noise 이다.

```
int amount1 = (int)(output.rows * output.cols * pp);
int amount2 = (int)(output.rows * output.cols * ps);
Mat output = input.clone();
```

전체 픽셀 수에 pp, ps 비율을 곱해 salt noise 가 될 pixel 수와 pepper noise 의 pixel 의 개수를 계산한다.

반환할 이미지의 output 행렬에 input 이미지를 복사하여 넣는다.

```
RNG rng;
if (output.channels() == 1) {
        for (int counter = 0; counter < amount1; ++counter)
                output.at<G>(rng.uniform(0, output.rows), rng.uniform(0, output.cols)) = 0;

        for (int counter = 0; counter < amount2; ++counter)
                output.at<G>(rng.uniform(0, output.rows), rng.uniform(0, output.cols)) = 255;
}
```

int cv::RNG::uniform(int a,int b) 함수는 [a,b) 범위에서 균일하게 분포된 정수의 난수를 반환한다.

흑백 이미지인 경우 channel 은 한개이다.

amount1(salt 개수)만큼 for 문을 돌리며 랜덤으로 정해진 x,y 좌표의 pixel 의 을 0 으로 만든다.

amount2(pepper 개수)만큼 for 문을 돌리며 랜덤으로 정해진 x,y 좌표의 pixel 의 을 255 으로 만든다.

```
else if (output.channels() == 3) {
        for (int counter = 0; counter < amount1; ++counter) {
                x = rng.uniform(0, output.rows);
                y = rng.uniform(0, output.cols);
                output.at<C>(x, y)[0] = 0;

                x = rng.uniform(0, output.rows);
                y = rng.uniform(0, output.cols);
                output.at<C>(x, y)[1] = 0;

                x = rng.uniform(0, output.rows);
                y = rng.uniform(0, output.cols);
                output.at<C>(x, y)[2] = 0;
        }

}
```

```
for (int counter = 0; counter < amount2; ++counter)
{
        x = rng.uniform(0, output.rows);
        y = rng.uniform(0, output.cols);
        output.at<C>(x, y)[0] = 255;

        x = rng.uniform(0, output.rows);
        y = rng.uniform(0, output.cols);
        output.at<C>(x, y)[1] = 255;

        x = rng.uniform(0, output.rows);
        y = rng.uniform(0, output.cols);
        output.at<C>(x, y)[2] = 255;
}
```

컬러 이미지의 경우 channel 은 3 개이다.
컬러 이미지의 경우 랜덤으로 정한 한 픽셀에 대해 RGB 모두 0 또는 255 를 만드는 것이 아니라, R G B 세 색상에 대해 따로따로 난수를 생성한다.

2) Salt_pepper_noise_removal_Gray()

salt and pepper noise 는 색이 주변과 전혀 다른 outlier 이기 때문에, kernel 안의 모든 값에 대한 평균을 사용하면, 그 0 또는 255 의 noise 값이 평균에 큰 영향을 끼치게 된다. 따라서, 이 outlier 가 영향을 끼치지 못하도록 중간값을 사용하는 것이 적당하다.
kernel 크기의 행렬 kernel 에 pixel 들의 모든 색상 값을 저장한 뒤, 오름차순으로 sort 하고, 그 중간값을 output 의 값으로 사용할 것이다.
sort 한 뒤 중간값을 찾기 위해서는 전체 픽셀 수를 알아야 하기에, count 변수를 이용해 픽셀 수를 센다

a. "zero-padding"

```
for (int x = -n; x <= n; x++) {
    for (int y = -n; y <= n; y++) {
        count++;
        if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
            kernel.at<G>((x+n)*kernel_size+(y+n), 0) = input.at<G>(i+x, j+y);
        }
    }
```

zero-padding 은 픽셀이 이미지 범위를 벗어나도 0 으로 반영하기 때문에, kernel 의 모든 pixel 에서 count 해준다. 픽셀이 이미지 범위에 들었을 때, 해당 색 데이터를 kernel 행렬에 저장한다. 정렬을 해야하기 때문에, (2*n+1)X(2*n+1)행렬이 아닌, (2*n+1)*(2*n+1)X1 행렬을 이용한다.

b. "mirroring"

```
for (int x = -n; x <= n; x++) {
        for (int y = -n; y <= n; y++) {
                if (i + x > row - 1)        tempx = i - x;
                else if (i + x < 0)         tempx = -(i + x);
                else                        tempx = i + x;

                if (j + y > col - 1)        tempy = j - y;
                else if (j + y < 0)         tempy = -(j + y);
                else                        tempy = j + y;

            kernel.at<G>((x+n)*kernel_size+(y+n), 0) = input.at<G>(tempx, tempy);
    }
}
```

mirroring 은 image boundary 를 벗어나면 image 안의 pixel 의 값을 mirroring 하여 가져온다. kernel 의 y 좌표 i+a, x 좌표 j+b 가 0 보다 작거나 이미지크기-1 보다 크면 (i,j)를 기준으로 미러링한 좌표를 temp 변수에 저장하고, 그 좌표의 값을 가져와 kernel 에 저장한다. 미러링하여 모든 kernel 의 값을 채우기 때문에 모든 kernel 에서 count 해준다.

c. "adjustkernel"

```
for (int x = -n; x <= n; x++) {
        for (int y = -n; y <= n; y++) {
            if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
                    count++;
                    kernel.at<G>((x+n)*kernel_size+(y+n), 0) = input.at<G>(i+x, j+y);
            }
            else zero++;
        }
}
```

픽셀이 이미지 범위에 들었을 때, 해당 색 데이터를 kernel 행렬에 저장한다.
"adjustkernel"은 이미지 범위에 들어간 것에서만 따진다. 즉, 중앙값을 구할 때도 이미지 범위에 벗어나는 pixel 은 빼고 중앙값을 찾아야 한다. 이미지 범위에서 벗어나는 pixel 의 수를 zero 변수에 저장한다.

```
sort(kernel, kernel, CV_SORT_EVERY_COLUMN + CV_SORT_ASCENDING);
median=zero+count/2;
output.at<G>(i, j) = kernel.at<G>(median, 0);
```

sort 함수를 이용해 색 데이터를 오름차순으로 정렬해준다.
제로패딩, 미러링에선 zero=0 이므로, count 한 pixel 수의 절반이 중앙값의 인덱스이다.
"adjustkernel"의 경우 이미지 범위에서 벗어나는 pixel 은 제외해야 하므로 median 의 인덱스에 zero 를 더해준다. 이미지 범위에서 벗어난 부분은 색 데이터 값이 0 이기에 정렬된 데이터의 맨 앞에 zero 개의 0 들이 저장되어 있을 것이기 때문에 중앙값을 찾을 때 이 0 값들을 제외하는 것이다.
그렇게 구한 median 인덱스에 해당하는 중앙값을 output 의 데이터로 지정해준다.

3) Salt_pepper_noise_removal_RGB()

컬러 이미지에 대한 noise removal 이다. (2)와 동일한 작업을 수행하지만, RGB 각 세 채널에 대해 따로따로 연산한다. kernel 행렬은 (2*n+1)*(2*n+1)X3 크기를 가져 각 열에 R, G, B 값을 저장하고, output 행렬 또한 3 개의 채널을 가진다.

실행 결과

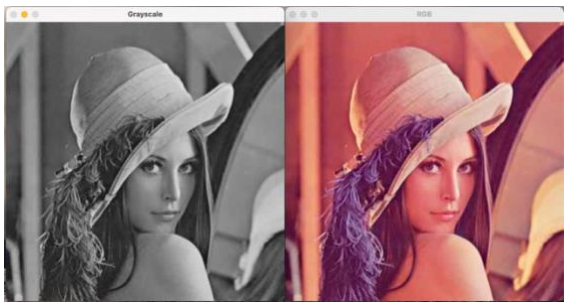
1)

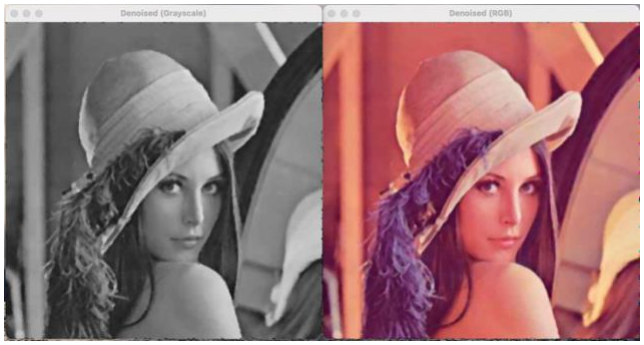
Add_salt_pepper_Noise(input_gray, 0.1f, 0.1f);

Add_salt_pepper_Noise(input, 0.1f, 0.1f);

Salt_pepper_noise_removal_Gray(noise_Gray, window_radius, "zero-padding");

Salt_pepper_noise_removal_RGB(noise_RGB, window_radius, "zero-padding");





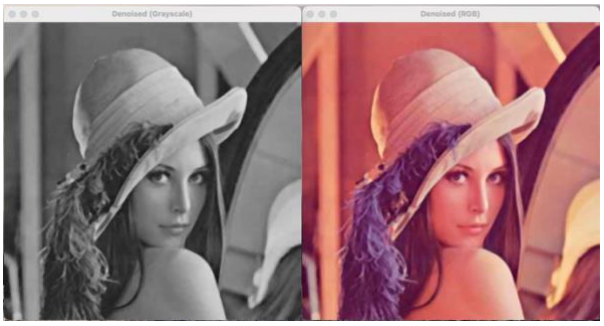
2) (original 이미지 생략)

Add_salt_pepper_Noise(input_gray, 0.01f, 0.01f);

Add_salt_pepper_Noise(input, 0.01f, 0.01f);

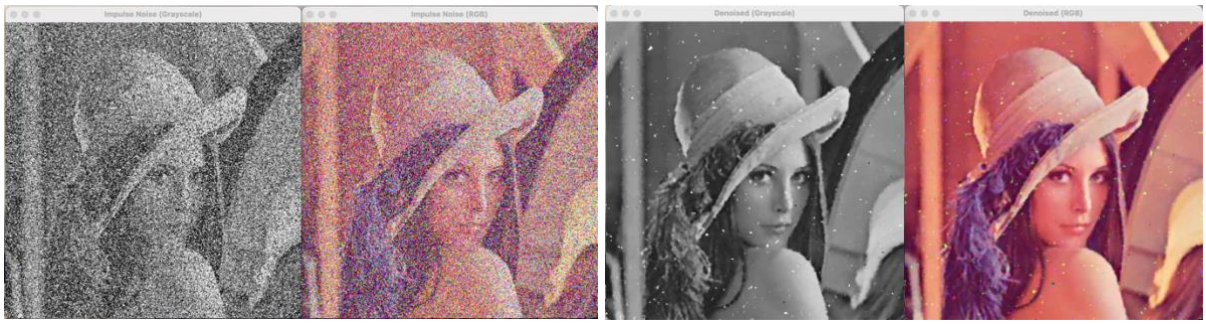
Salt_pepper_noise_removal_Gray(noise_Gray, window_radius, "mirroring");

Salt_pepper_noise_removal_RGB(noise_RGB, window_radius, "mirroring");



3) (original 이미지 생략)

```
Add_salt_pepper_Noise(input_gray, 0.3f, 0.3f);
Add_salt_pepper_Noise(input, 0.3f, 0.3f);
Salt_pepper_noise_removal_Gray(noise_Gray, window_radius, "adjustkernel");
Salt_pepper_noise_removal_RGB(noise_RGB, window_radius, "adjustkernel");
```



2. Gaussian.cpp

코드 목적:

Gaussian noise 를 만들고, 이를 Gaussian filter 를 사용해 제거한다.

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지를 만든다.
2) 컬러 이미지와 흑백 이미지를 각각 Add_Gaussian_noise ()함수에 넣어 noise가 생긴 이미지를 return 받는다.
3) (2)의 흑백 이미지는 Gaussianfilter_Gray() 함수에, 컬러이미지는 Gaussianfilter_RGB() 에 넣어 noise를 제거한다.
4) 모든 과정에서의 이미지를 새로운 창에 띄운다. (원본이미지, 원본이미지 흑백, Gaussian noise가진 흑백 이미지, Gaussian noise가진 컬러 이미지 , Gaussian filtering한 흑백 이미지, Gaussian filtering한 컬러 이미지)

함수 설명: Add_Gaussian_noise(const Mat input, double mean, double sigma)

매개변수:

input: input 이미지 행렬

mean: noise 를 생성할 가우스 함수의 평균

sigma: noise 를 생성할 가우스 함수의 표준편차

함수 목적:

input 이미지에 평균을 mean, 표준편차를 sigma 로 하는 가우스 함수에서 랜덤으로 가져온 수들이 값에 더해져 Gaussian Noise 를 가지게 된 output 이미지 행렬을 반환

Gaussianfilter_Gray(const Mat input, int n, double sigma_t, double sigma_s, const char *opt)

매개변수:

input: input 이미지 행렬(흑백)

n: kernel 의 크기 (2n+1)x(2n+1)

sigma_t: x 축에 대한 표준편차

sigma_s: y 축에 대한 표준편차

opt: type of boundary processing. image boundary 의 픽셀 처리 방법

"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: Gaussian filter를 이용해 Gaussian noise를 제거한 이미지를 반환 (흑백)

Gaussianfilter_RGB(const Mat input, int n, double sigma_t, double sigma_s, const char *opt)

매개변수:

input: input 이미지 행렬(흑백)

n: kernel 의 크기 (2n+1)x(2n+1)

sigma_t: x 축에 대한 표준편차

sigma_s: y 축에 대한 표준편차

opt: type of boundary processing. image boundary 의 픽셀 처리 방법

"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: Gaussian filter를 이용해 Gaussian noise를 제거한 이미지를 반환 (컬러)

코드 설명:

1) Add_Gaussian_noise

```
Mat NoiseArr = Mat::zeros(input.rows, input.cols, input.type());
RNG rng;

rng.fill(NoiseArr, RNG::NORMAL, mean, sigma);

add(input, NoiseArr, NoiseArr);

return NoiseArr;
```

fill 함수를 사용하여 NoiseArr 행렬을 mean 을 평균, sigma 를 표준편차로 가지는 Gaussian random variable(가우스 랜덤 변수)로 채운다. RNG::NORMAL 은 openCV 에서 정의한 변수로, enum 으로 UNIFORM 은 0, NORMAL 은 1 이다.

Gaussian Noise 는 $I_G(x,y) = I(x,y) + N(x,y)$이다. NoiseArr 에 저장된 값이 $N(x,y)$에 해당하므로, input 에 NoiseArr 를 더한 값을 NoiseArr 행렬에 저장해 리턴한다.

add(a, b, c); 는 c=a+b(a 와 b 를 더해 c 에 저장)의 의미이다.

2) Gaussianfilter_Gray

Lecture4 의 Gaussian filter 와 같다.

Gaussian filter 에서 kernel 은 w(s,t)로 가우스 함수의 분포에 따른 가중치를 가진다.

$$w(s,t) = \frac{1}{\sum_{m=-a}^{a}\sum_{n=-b}^{b}\exp\left(-\frac{m^2}{2{\sigma_s}^2}-\frac{n^2}{2{\sigma_t}^2}\right)}\exp\left(-\frac{s^2}{2{\sigma_s}^2}-\frac{t^2}{2{\sigma_t}^2}\right)$$

kernel 의 픽셀에 (a,b) 모두 접근해 $\exp(-\frac{a^2}{2{\sigma_s}^2}-\frac{b^2}{2{\sigma_t}^2})$를 계산하고, value1 에 더해준다.

value 는 $\sum_{m=-a}^{a}\sum_{n=-b}^{b}\exp(-\frac{m^2}{2{\sigma_s}^2}-\frac{n^2}{2{\sigma_t}^2})$가 되므로, value1 으로 kernel 을 다시 나눠주면 $w(s,t)$를 구할 수 있다. 따라서 행렬 kernel 에 $w(s,t)$가 저장되어 있다.

```
Mat kernel;
kernel = Mat::zeros(kernel_size, kernel_size, CV_32F);
float denom = 0.0;
for (int x = -n; x <= n; x++) {
    for (int y = -n; y <= n; y++) {
        float value = exp(-(pow(x, 2) / (2 * pow(sigma_s, 2))) - (pow(y, 2) / (2 * pow(sigma_t, 2))));
        kernel.at<float>(x+n, y+n) = value;
        denom += value;
    }
}
for (int x = -n; x <= n; x++) {
    for (int y = -n; y <= n; y++) {
        kernel.at<float>(x+n, y+n) /= denom;
    }
}
```

```
sum += kernel.at<float>(a+n, b+n)*(float)(input.at<G>(i + a, j + b));
```

```
output.at<G>(i, j) = (G)sum1;
```

for문을 통해 (i,j) 픽셀의 filtering을 위한 kernel의 (a+n,b+n)픽셀에 접근한다. input에 접근해 가져온 데이터를 해당하는 kernel의 가중치에 곱해 sum에 더해준다. kernel의 모든 픽셀에 대한 가중치*input데이터를 더해준 값을 output 이미지의 해당 비트에 넣어준다.

가우시안 필터에서 값을 정해주고 계산하는 과정은 위와 같고, boundary processing에 따라 위의 코드를 어디에 쓰는지가 다르다. 이는 Lecture4에 자세히 설명되어 있으니 생략하겠다.

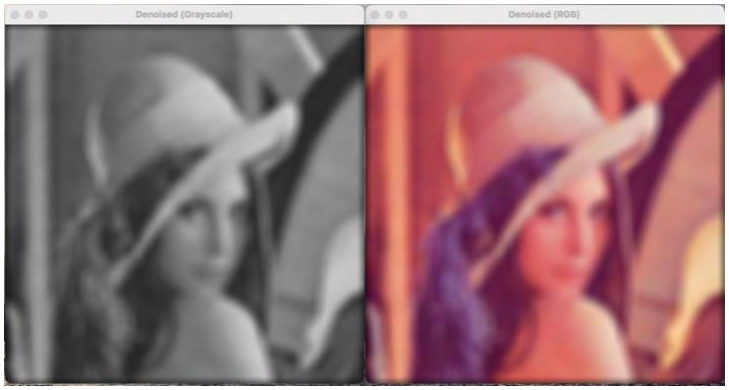
3) Gaussianfilter_RGB ()

컬러 이미지에 대한 noise removal 이다. (2)와 동일한 작업을 수행하지만, RGB 각 세 채널에 대해 따로따로 연산한다.

실행 결과 (Original Image 생략)

1)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.1);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.1);
Mat Denoised_Gray = Gaussianfilter_Gray(noise_Gray, 10, 10, 10, "zero-padding");
Mat Denoised_RGB = Gaussianfilter_RGB(noise_RGB, 10, 10, 10, "zero-padding");
```

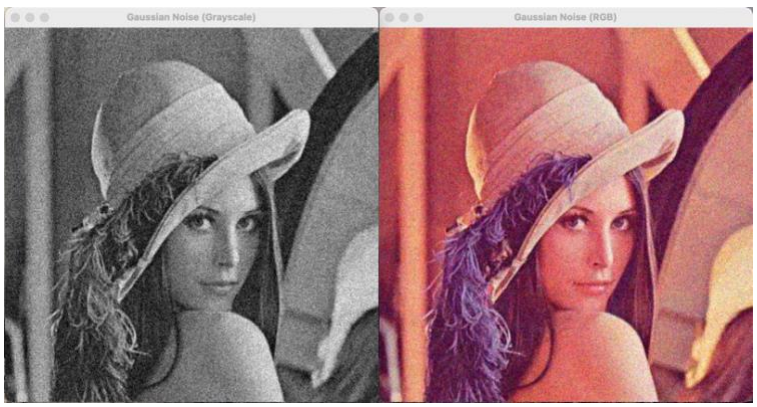


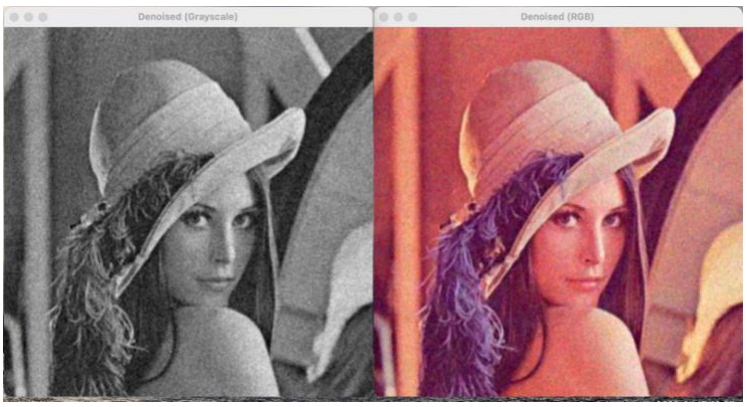
2)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.3);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.3);
Mat Denoised_Gray = Gaussianfilter_Gray(noise_Gray, 3, 10, 10, "mirroring");
Mat Denoised_RGB = Gaussianfilter_RGB(noise_RGB, 3, 10, 10, "mirroring");
```

3)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.1);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.1);
Mat Denoised_Gray = Gaussianfilter_Gray(noise_Gray, 1, 5, 5, "adjustkernel");
Mat Denoised_RGB = Gaussianfilter_RGB(noise_RGB, 1, 5, 5, "adjustkernel");
```





3. Bilateral.cpp

코드 목적:

Gaussian noise 를 만들고, 이를 Bilateral filtering 를 사용해 제거한다.

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지를 만든다.
2) 컬러 이미지와 흑백 이미지를 각각 Add_Gaussian_noise ()함수에 넣어 noise가 생긴 이미지를 return 받는다.
3) (2)의 흑백 이미지는 Bilateralfilter_Gray() 함수에, 컬러이미지는 Bilateralfilter_RGB()에 넣어 noise를 제거한다.
4) 모든 과정에서의 이미지를 새로운 창에 띄운다. (원본이미지, 원본이미지 흑백, Gaussian noise가진 흑백 이미지, Gaussian noise가진 컬러 이미지 , Bilateral filtering한 흑백 이미지, Bilateral filtering한 컬러 이미지)

함수 설명: Add_Gaussian_noise(const Mat input, double mean, double sigma)

매개변수:

input: input 이미지 행렬
mean: noise 를 생성할 가우스 함수의 평균
sigma: noise 를 생성할 가우스 함수의 표준편차

함수 목적:

input 이미지에 평균을 mean, 표준편차를 sigma 로 하는 가우스 함수에서 랜덤으로 가져온 수들이 값에 더해져 Gaussian Noise 를 가지게 된 output 이미지 행렬을 반환

Bilateralfilter_Gray
(const Mat input, int n, double sigma_t, double sigma_s, double sigma_r, const char *opt)

매개변수:

input: input 이미지 행렬(흑백)
n: kernel 의 크기 (2n+1)x(2n+1)
sigma_t: x 축에 대한 표준편차
sigma_s: y 축에 대한 표준편차
sigma_r: 명도에 대한 표준편차
opt: type of boundary processing. image boundary 의 픽셀 처리 방법
"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: Bilateral filter를 이용해 Gaussian noise를 제거한 이미지를 반환 (흑백)

Bilateralfilter_RGB
(const Mat input, int n, double sigma_t, double sigma_s, double sigma_r, const char *opt);

매개변수:

input: input 이미지 행렬(흑백)
n: kernel 의 크기 (2n+1)x(2n+1)

sigma_t: x 축에 대한 표준편차
sigma_s: y 축에 대한 표준편차
sigma_r: 명도에 대한 표준편차
opt: type of boundary processing. image boundary 의 픽셀 처리 방법
"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적: Bilateral filter를 이용해 Gaussian noise를 제거한 이미지를 반환 (컬러)

코드 설명:

Gaussian Noise 를 만드는 함수는 2 번 코드와 동일하다

1) Bilateralfilter_Gray()

Bilateral filter 는 spatial 뿐만 아니라 intensity distance 까지 고려한다. 기존의 Gaussian Filter 는 (i,j)를 중심으로 하는 kernel 에서, 가우스 함수에 따라 (i,j)에 가까울 수록 더 높은 가중치를 부여하였다. Bilateral filter 는 이 거리 뿐만 아니라, (i,j)와 색이 비슷할수록 더 높은 가중치를 가지게 된다.

$$O(i,j) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) I(i+s, j+t)$$

input 데이터를 필터링 할 kernel 은 w(s,t)의 가중치를 가지고, w(s,t)는 다음과 같은 식을 가진다.

$$w(s,t) = \frac{1}{W(i,j)} \exp\left(-\frac{s^2}{2{\sigma_s}^2} - \frac{t^2}{2{\sigma_t}^2}\right) \exp\left(-\frac{(I(i,j) - I(i+s, j+t))^2}{2{\sigma_r}^2}\right)$$

여기서 분모에 있는 W(i,j)는 1/ W(i,j) 를 제외한 부분들의 전체 합이다.

$$W(i,j) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} \exp\left(-\frac{m^2}{2{\sigma_s}^2} - \frac{n^2}{2{\sigma_t}^2}\right) \exp\left(-\frac{(I(i,j) - I(i+m, j+n))^2}{2{\sigma_r}^2}\right)$$

따라서 w(s,t)를 구하기 위해서는, 모든 픽셀에 대하여 $\exp\left(-\frac{s^2}{2{\sigma_s}^2} - \frac{t^2}{2{\sigma_t}^2}\right) \exp\left(-\frac{(I(i,j) - I(i+s, j+t))^2}{2{\sigma_r}^2}\right)$ 를 계산하고 값을 하나씩 저장해두는 동시에 모두를 더한 값을 저장한다. 모든 픽셀에 대한 계산이 끝나면 각 픽셀의 값을 전체 합으로 나눠주면 된다.

우선, input 값과 관계없이 고정된 $\exp\left(-\frac{s^2}{2{\sigma_s}^2} - \frac{t^2}{2{\sigma_t}^2}\right)$ 를 계산해준다.

```
Mat kernel1;
kernel1 = Mat::zeros(kernel_size, kernel_size, CV_32F);
for (int x = -n; x <= n; x++) {
    for (int y = -n; y <= n; y++) {
        kernel1.at<float>(x+n, y+n) =
                    exp(-(pow(x, 2) / (2 * pow(sigma_s, 2))) - (pow(y, 2) / (2 * pow(sigma_t, 2))));
    }
}
```

$\exp\left(-\frac{s^2}{2{\sigma_s}^2}-\frac{t^2}{2{\sigma_t}^2}\right)\exp\left(-\frac{(I(i,j)-I(i+s,j+t))^2}{2{\sigma_r}^2}\right)$ 는 kernel2 에 저장해둘 것이다.

"zero-padding"의 경우를 살펴보자.

kernel2 에 저장될 값 하나하나를 모두 더해 denom 에 저장할 것이다.

```
float denom=0.0;
    for (int x = -n; x <= n; x++) { // for each kernel window
         for (int y = -n; y <= n; y++) {
            if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
                float value=exp(-(pow( (float)(input.at<G>(i, j))-(float)(input.at<G>(i+x, j+y)) ,2))/ (2*pow(sigma_r,2)));
                float value2=kernel1.at<float>(x+n, y+n)*value;
                kernel2.at<float>(x+n, y+n)=value2;
                denom += value2;
            }
        }
}
```

제로 패딩이므로, 이미지 범위안에 들어가는 픽셀만 신경쓴다.

kernel 안의 픽셀에 대하여 $\exp\left(-\frac{(I(i,j)-I(i+s,j+t))^2}{2{\sigma_r}^2}\right)$ 를 계산해준다.

이 값에 kernel1 을 곱해준다.

$\exp\left(-\frac{s^2}{2{\sigma_s}^2}-\frac{t^2}{2{\sigma_t}^2}\right)\exp\left(-\frac{(I(i,j)-I(i+s,j+t))^2}{2{\sigma_r}^2}\right)$ 이 완성되었으므로, kernel2 행렬에 저장하고, denom 에 이 값을 더해준다.

kernel2 에 아직 W(i,j)로 나누지 않은 w(s,t)값이 저장되어 있고, denom 에 W(i,j)가 저장되어 있다.
따라서 kernel 의 모든 픽셀을 돌며 denom 으로 kernel2 의 값을 나눠주면 w(s,t)가 완성된다.
그리고 그 w(s,t)를 input(i+x,j+y)값에 곱하면, 가중치를 반영한 값이 완성되고, 이 값들의 합은 output 값이 된다.

```
for (int x = -n; x <= n; x++) {
    for (int y = -n; y <= n; y++) {
        if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
            kernel2.at<float>(x+n, y+n)/=denom;
            sum+=kernel2.at<float>(x+n, y+n)*(float)(input.at<G>(i+x, j+y));
        }
    }
}
output.at<G>(i, j) = (G)sum;
```

다른 코드와 마찬가지로, "mirroring"은 반사되는 좌표 temp 를 고려하고, "adjustkernel"은 가중치 w(s,t)의 총합을 계산해 마지막에 나누어주면 된다.

2) Bilateralfilter_RGB()

RGB 데이터를 가질 때, 비록 R 값과 G 값이 같아도, B 의 값이 전혀 다르다면 다른 색이 된다. 따라서, Bilateral Filtering 에서 고려하는 color 의 distance 는 각 r, g, b 가 아니라 RGB 모든 것을 포함한 거리를 의미한다. 따라서, 흑백에 해당하는 아래의 식에서,

$$w(s,t) = \frac{1}{W(i,j)} \exp\left(-\frac{s^2}{2{\sigma_s}^2} - \frac{t^2}{2{\sigma_t}^2}\right) \exp\left(-\frac{(I(i,j) - I(i+s,j+t))^2}{2{\sigma_r}^2}\right)$$

$(I(i,j) - I(i+s,j+t))^2$를 R,G,B 를 반영하여
$\sqrt{(R(i,j) - R(i+s,j+t))^2 + (G(i,j) - G(i+s,j+t))^2 + (B(i,j) - B(i+s,j+t))^2}$로 계산한다.

"zero-padding"을 보자

```
float denom=0.0;
for (int x = -n; x <= n; x++) {
      for (int y = -n; y <= n; y++) {
            if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
                  float Cp=0.0;
                  Cp+=pow((float)input.at<C>(i,j)[0]-(float)input.at<C>(i+x, j+y)[0],2);
                  Cp+=pow((float)input.at<C>(i,j)[1]-(float)input.at<C>(i+x, j+y)[1],2);
                  Cp+=pow((float)input.at<C>(i,j)[2]-(float)input.at<C>(i+x, j+y)[2],2);
                  Cp=sqrt(Cp);

                  float value=exp(-1*Cp/ (2*pow(sigma_r,2))) ;
                  float value2=kernel1.at<float>(x+n, y+n)*value;
                  kernel2.at<float>(x+n, y+n)=value2;
                  denom += value2;
             }
      }
}
```

Cp 에 순차적으로 R 사이의 거리의 제곱, G 의 거리의 제곱, B 의 거리의 제곱을 더한 뒤 루트를 씌워 Color distance 를 계산해준다.
위 코드를 마치면 흑백일 때와 마찬가지로, W(i,j)로 나누지 않은 w(s,t)값이 kernel2 에 저장되어 있고, denom 에 W(i,j)가 저장되어 있다.

따라서 kernel 의 모든 픽셀을 돌며 denom 으로 kernel2 의 값을 나눠주면 w(s,t)가 완성된다. 그리고 그 w(s,t)를 input(i+x,j+y)값에 곱하면, 가중치를 반영한 값이 완성되고, 이 값들의 합은 output 값이 된다.

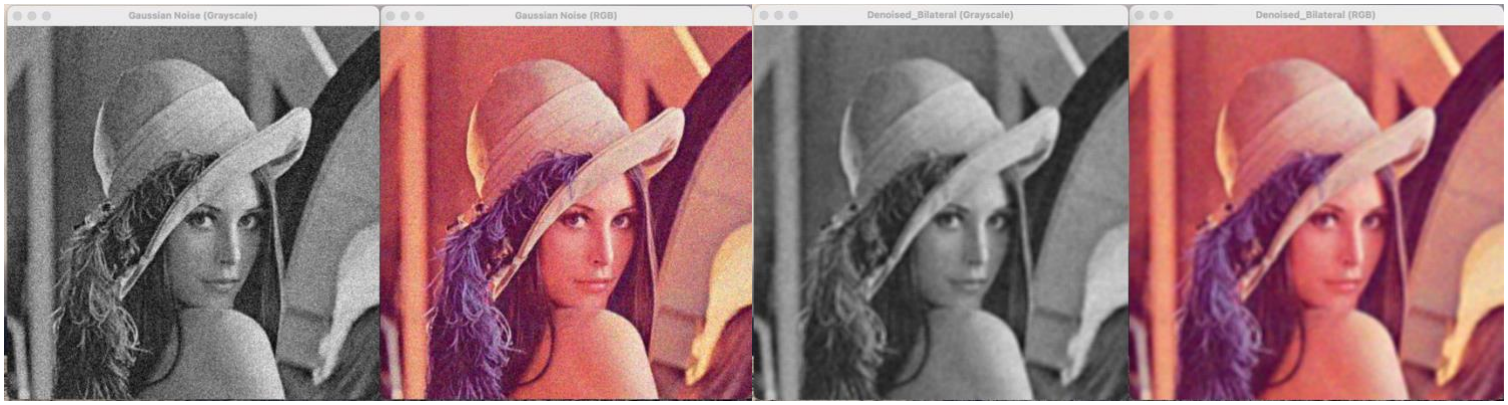
```
for (int x = -n; x <= n; x++) {
        for (int y = -n; y <= n; y++) {
            if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
                kernel2.at<float>(x+n, y+n)/=denom;
                sum_r+=kernel2.at<float>(x+n, y+n)*(float)(input.at<C>(x+i, y+j)[0]);
                sum_g+=kernel2.at<float>(x+n, y+n)*(float)(input.at<C>(x+i, y+j)[1]);
                sum_b+=kernel2.at<float>(x+n, y+n)*(float)(input.at<C>(x+i, y+j)[2]);
            }
        }
}
output.at<C>(i, j)[0] = (G)sum_r;
output.at<C>(i, j)[1] = (G)sum_g;
output.at<C>(i, j)[2] = (G)sum_b;
```

이는 흑백의 경우와 같다. R, G, B 에 대해서만 따로따로 계산해주면 된다.

실행 결과 (Original 이미지 생략)

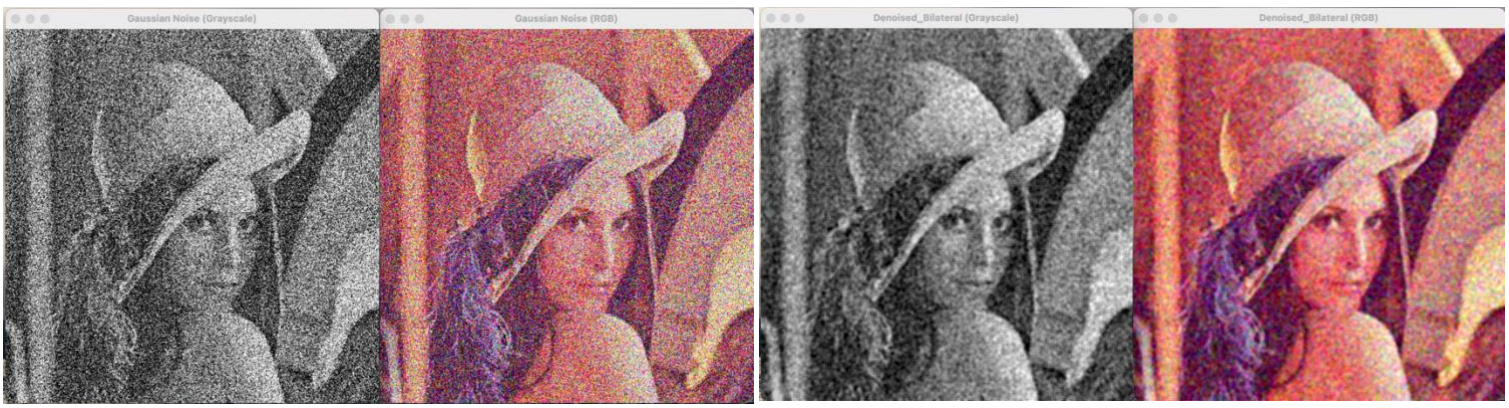
1)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.1);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.1);
Mat Denoised_Gray = Bilateralfilter_Gray(noise_Gray, 3, 10, 10, 10, "zero-padding");
Mat Denoised_RGB = Bilateralfilter_RGB(noise_RGB, 3, 10, 10, 10, "zero-padding");
```



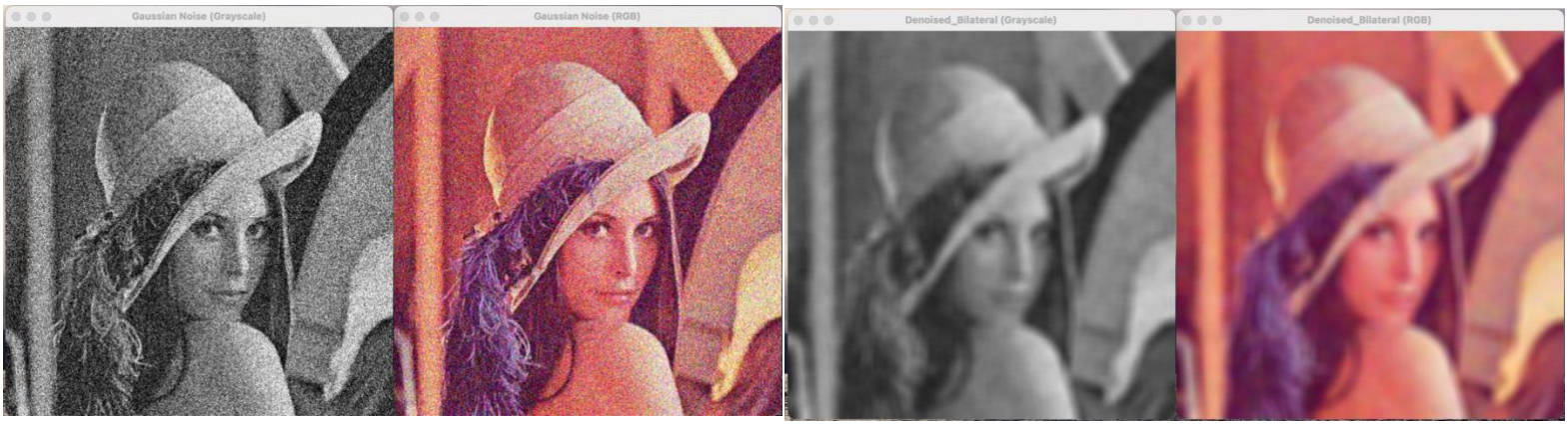
2)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.4);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.4);
Mat Denoised_Gray = Bilateralfilter_Gray(noise_Gray, 2, 8, 8, 8, "mirroring");
Mat Denoised_RGB = Bilateralfilter_RGB(noise_RGB, 2, 8, 8, 8, "mirroring");
```



3)

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.2);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.2);
Mat Denoised_Gray = Bilateralfilter_Gray(noise_Gray, 8, 10, 10, 10, "adjustkernel");
Mat Denoised_RGB = Bilateralfilter_RGB(noise_RGB, 8, 10, 10, 10, "adjustkernel");
```



참고자료:

오픈 SW 프로젝트 Lec05 수업자료