Open Software Project

Assignment 04 Technical Report

1976235 오진솔

1. Add_salt_pepper_Noise

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

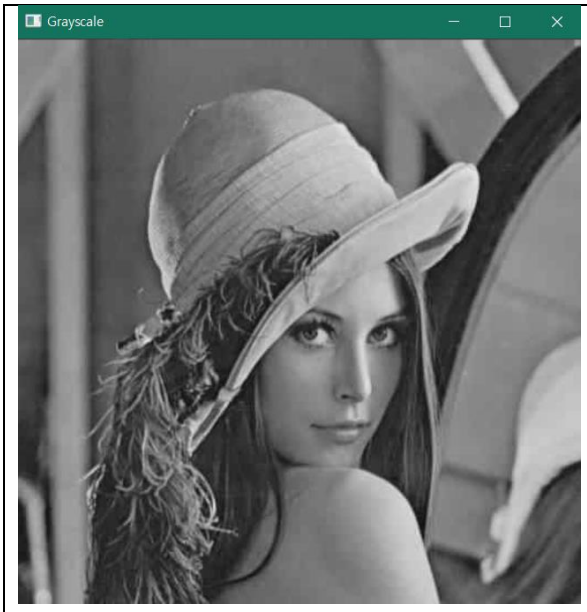
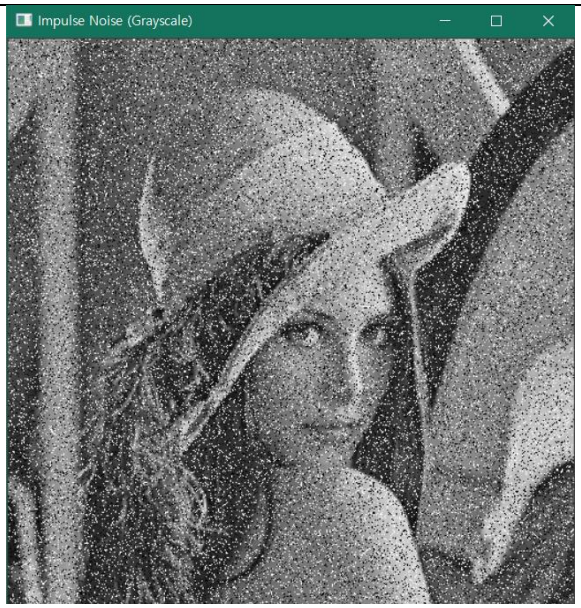
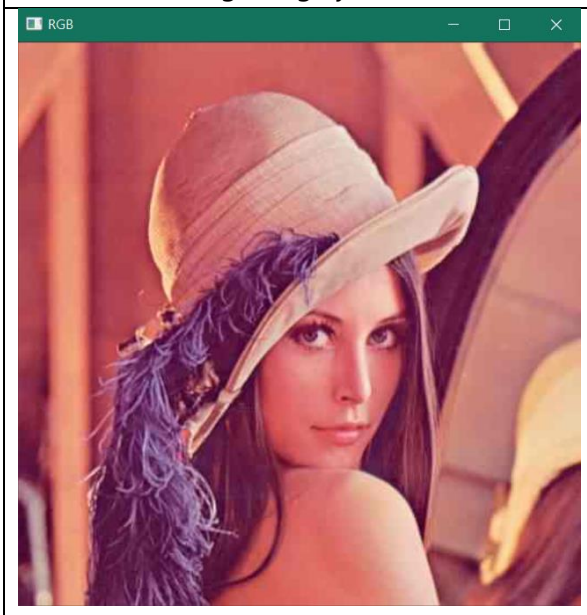
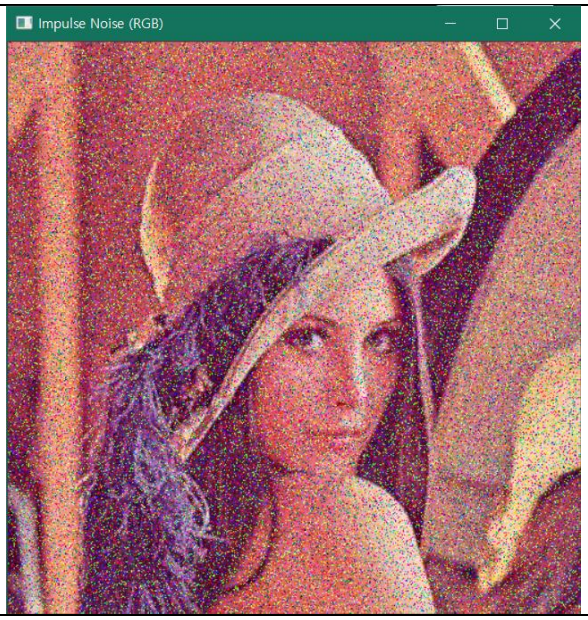
입력받은 pp, ps를 기반으로 노이즈의 양을 계산한다

```
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;
```

랜덤으로 intensity가 0(pepper)과 1(salt)인 노이즈를 생성한다. (grayscale 이미지일때)

```
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;
        }
}
```

랜덤으로 R, G, B 값 중 하나 이상이 0또는 255인 노이즈를 생성한다. (RGB 이미지일때)

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>original (grayscale)</td><td>salt and pepper noise (grayscale)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>original(RGB)</td><td>salt and pepper noise (RGB)</td></tr>
</table>

2. Salt_pepper_noise_removal_Gray

```
if (!strcmp(opt, "zero-padding")) {
    median = ceil(kernel_size*kernel_size / 2);
    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))
                kernel.at<G>((x + n)*kernel_size + (y + n), 0) = input.at<G>(i + x, j + y);
        }
    }
}
```

<zero-padding>
바운더리 안쪽에 있는 값에 한해 kernel에 대응되는 input 값을 그대로 옮긴다.
kernel이 이미 0으로 초기화되어 있으므로 바운더리를 벗어나는 점은 자동으로 0이 된다.

kernel를 나중에 정렬할 예정이므로 median에는 kernel의 중간점의 인덱스를 저장한다.

```
else if (!strcmp(opt, "mirroring")) {
    median = ceil(kernel_size*kernel_size / 2);
    int tempa;
    int tempb;
    for (int x = -n; x <= n; x++) { // for each kernel window
        for (int y = -n; y <= n; y++) {
            if (i + x > row - 1)    tempa = i - x;
            else if (i + x < 0)     tempa = -(i + x);
            else                    tempa = i + x;

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

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

<mirroring>
바운더리를 벗어나는 픽셀들에 대해 mirroring 기법을 사용해 주변의 픽셀의 값을 복사한다.

kernel를 나중에 정렬할 예정이므로 median에는 kernel의 중간점의 인덱스를 저장한다.

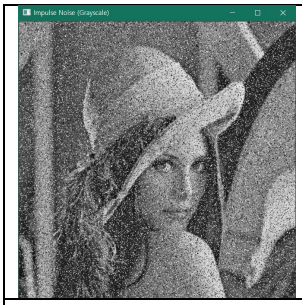
```
else if (!strcmp(opt, "adjustkernel")) {
    median = kernel_size * kernel_size;
    for (int x = -n; x <= n; x++) { // for each kernel window
        for (int y = -n; y <= n; y++) {
            if ((i + x <= row - 1) && (i + x >= 0) && (j + y <= col - 1) && (j + y >= 0)) {
                kernel.at<G>((x + n)*kernel_size + (y + n), 0) = input.at<G>(i + x, j + y);
                if ((x + y) % 2) median--;
            }
        }
    }
}
```

<adjustkernel>
바운더리를 벗어나는 값은 취급하지 않는다. 바운더리 안쪽에 있는 값에 대해서만 중간값을 구한다.

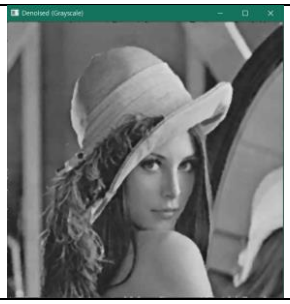
median을 kernel size * kernel size 로 초기화해두고, x+y가 홀수일 때마다 -1한다.
바운더리 안쪽의 픽셀 개수는 항상 4개 혹은 6개로 짝수이며 x+y가 홀수인 경우와 짝수인 경우가 균등하므로 위와 같은 방법을 사용하면 중간값 중 큰 값을 구할 수 있음.

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

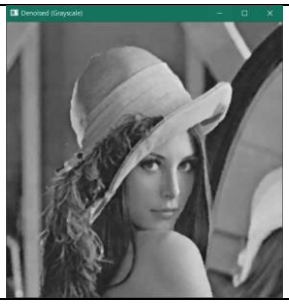
kernel을 정렬하고 output에 중간값을 저장한다.



noised



zero-padding



mirroring

adjustkernel

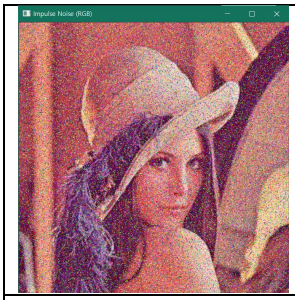
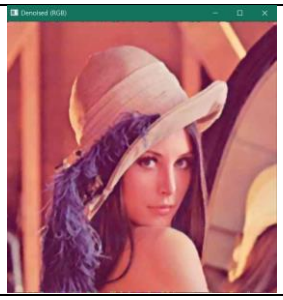
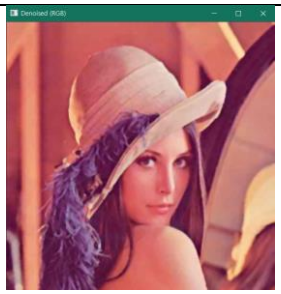
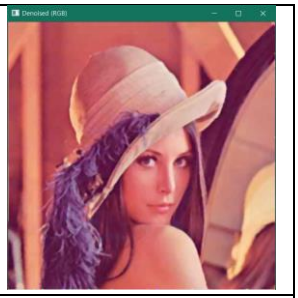
2. Salt_pepper_noise_removal_RGB

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

Gray에서 위와 같았던 부분을

```
kernel.at<G>((x + n)*kernel_size + (y + n), 0) = input.at<C>(i + x, j + y)[0];
kernel.at<G>((x + n)*kernel_size + (y + n), 1) = input.at<C>(i + x, j + y)[1];
kernel.at<G>((x + n)*kernel_size + (y + n), 2) = input.at<C>(i + x, j + y)[2];
```

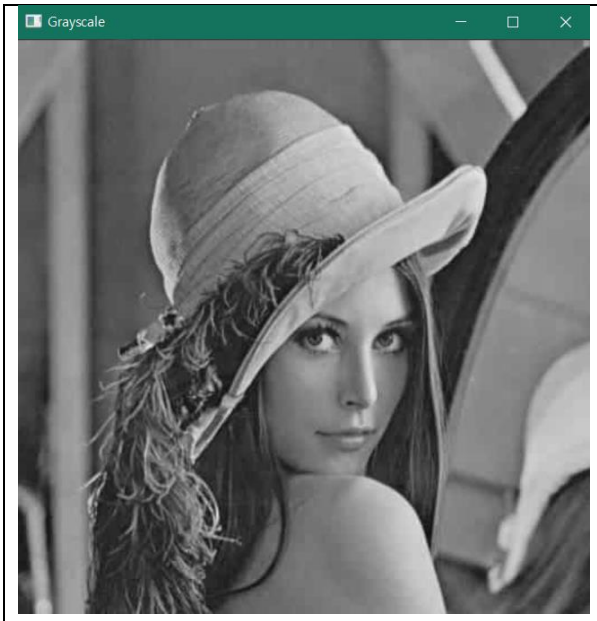
다음과 같이 R,G,B 각각의 채널에 대해 반복한다.

			
noised	zero-padding	mirroring	adjustkernel

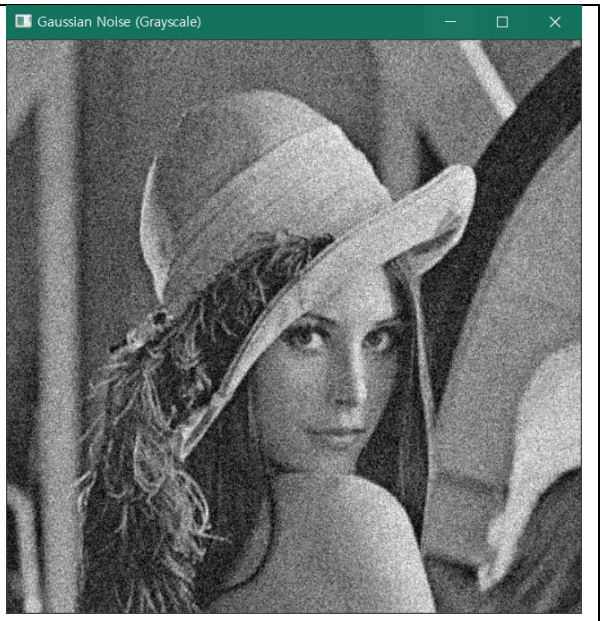
4. Add_Gaussian_noise

```
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;
```

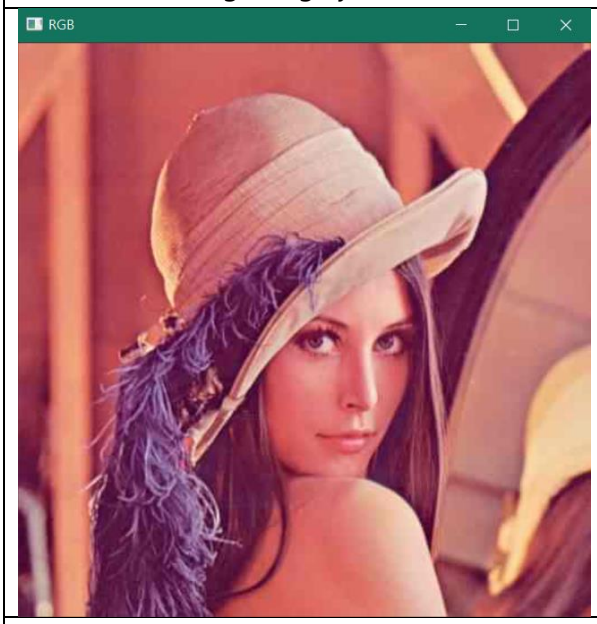
NoiseArr에 랜덤으로 노이즈를 생성하고 input에 노이즈를 더한다.



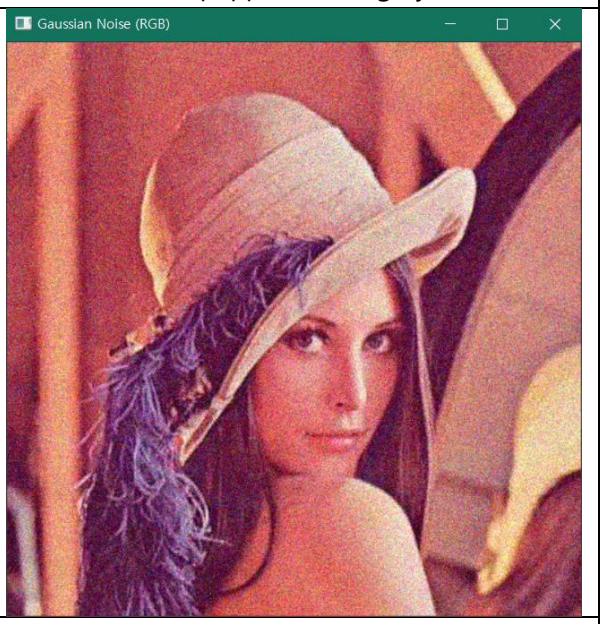
original (grayscale)



salt and pepper noise (grayscale)



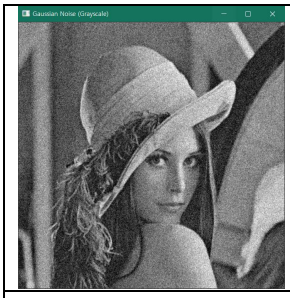
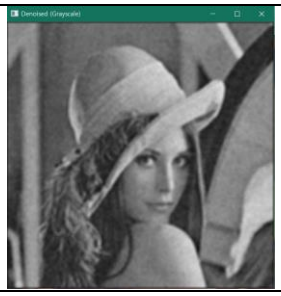
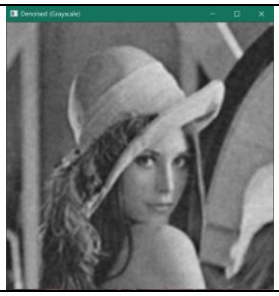
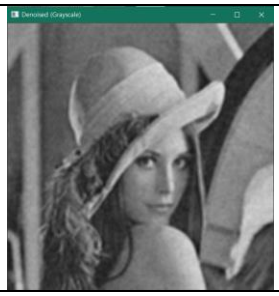
original(RGB)

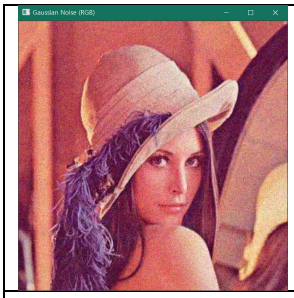
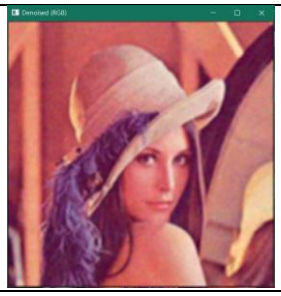
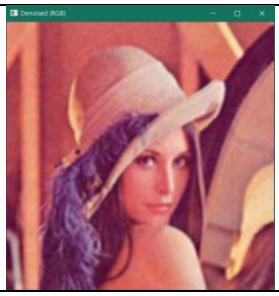
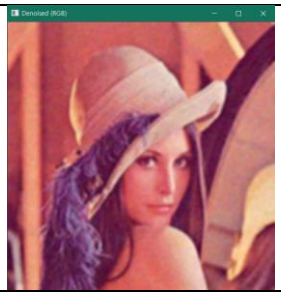


salt and pepper noise (RGB)

5. Gaussian noise remove (using gaussian filter)

Gaussian filter 를 적용하여 노이즈를 제거한다. (자세한 내용은 과제 3 과 같으므로 생략합니다)

			
noised	zero-padding	mirroring	adjustkernel

			
noised	zero-padding	mirroring	adjustkernel

6. bilateral filtering

```
if (!strcmp(opt, "zero-padding")) {
    float sum1 = 0.0;
    float denom = 0.0;
    float I_sub;

    for (int a = -n; a <= n; a++) {
        for (int b = -n; b <= n; b++) {
            if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                I_sub = input.at<G>(i, j) - input.at<G>(i + a, j + b);
            }
            else {
                I_sub = input.at<G>(i, j);
            }
            I_sub *= Bilateral_amount;
            kernel_s.at<float>(a + n, b + n) = exp(-(pow(a, 2) / (2 * pow(sigma_s, 2))) - (pow(b, 2) / (2 *
                                                                                    pow(sigma_t, 2))));
            kernel_r.at<float>(a + n, b + n) = exp(-(pow(I_sub, 2) / (2 * pow(sigma_r, 2))));
    }
}

for (int a = -n; a <= n; a++) {
    for (int b = -n; b <= n; b++) {
        kernelvalue_s = kernel_s.at<float>(a + n, b + n);
        kernelvalue_r = kernel_r.at<float>(a + n, b + n);

        denom += kernelvalue_s * kernelvalue_r;

        if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
            sum1 += kernelvalue_s * kernelvalue_r * input.at<G>(i + a, j + b);
        }
    }
}
        output.at<G>(i, j) = (G)sum1 / denom;
```

<Gray, zero-padding>
I_sub : 두 점 사이의 intensity 차이
I_sub 에 Bilateral amount (20으로 define 되어있음) 을 곱한다.
(이 값을 곱하지 않을 경우 거리 차이가 intensity 차이보다 훨씬 큰 영향을 끼쳐서 gaussian filter를 사용했을 때와 큰 차이가 없었습니다. 원래 이런 것인지, 제 코드가 잘못된 것인지 잘 모르겠지만 상수를 곱할 경우 앞서 말한 문제가 완화되는 것 같아서 여러 번 실험 끝에 적절한 값을 찾았습니다.)

kernel_s 거리 차이에 따른 Gaussian filter kernel
kernel_r intensity 차이에 따른 Gaussian filter kernel

kernel_s * kernel_r 가 새로운 하나의 kernel처럼 작용한다.
kernel_s * kernel_r * input(I, j) 로 각 픽셀의 값을 구하고 kernel_s * kernel_r 의 총 크기로 나누어서 normalize한다.

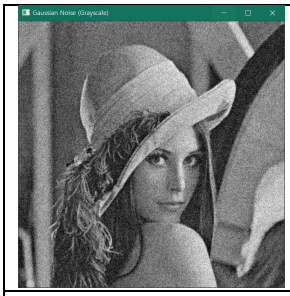
```
else if (!strcmp(opt, "mirroring")) {
    float sum1 = 0.0;
    float denom = 0.0;
    float I_sub;
    for (int a = -n; a <= n; a++) {
        for (int b = -n; b <= n; b++) {
            if (i + a > row - 1)      tempa = i – a;
            else if (i + a < 0)       tempa = -(i + a);
            else                      tempa = i + a;
            if (j + b > col - 1)      tempb = j - b;
            else if (j + b < 0)       tempb = -(j + b);
            else                      tempb = j + b;
            I_sub = input.at<G>(i, j) - input.at<G>(tempa, tempb);
            I_sub *= Bilateral_amount;

            kernel_s.at<float>(a + n, b + n) = exp(-(pow(a, 2) / (2 * pow(sigma_s, 2))) - (pow(b, 2) /
                                                                                    (2 * pow(sigma_t, 2))));
            kernel_r.at<float>(a + n, b + n) = exp(-(pow(I_sub, 2) / (2 * pow(sigma_r, 2))));
        }
    }
    for (int a = -n; a <= n; a++) {
        for (int b = -n; b <= n; b++) {
            kernelvalue_s = kernel_s.at<float>(a + n, b + n);
            kernelvalue_r = kernel_r.at<float>(a + n, b + n);
            denom += kernelvalue_s * kernelvalue_r;
            if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                sum1 += kernelvalue_s * kernelvalue_r * input.at<G>(i + a, j + b);}
        }
    }
```

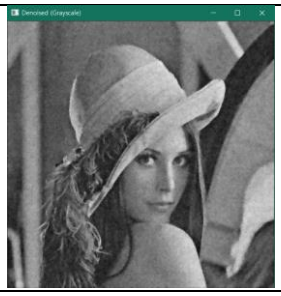
```
    output.at<G>(i, j) = (G)sum1 / denom;
}
```

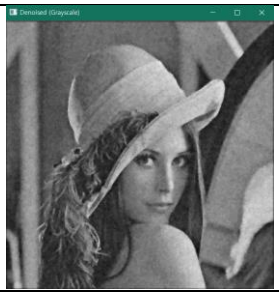
<mirroring>
비슷한 방식으로 수행함.

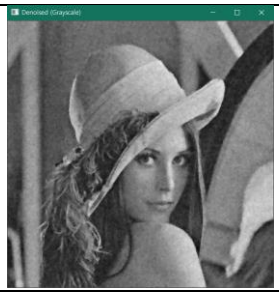
```
else if (!strcmp(opt, "adjustkernel")) {
   float sum1 = 0.0;
   float denom = 0.0;
   float I_sub;

   for (int a = -n; a <= n; a++) {
      for (int b = -n; b <= n; b++) {
         if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
            I_sub = input.at<G>(i, j) - input.at<G>(i + a, j + b);
            I_sub *= Bilateral_amount;
            kernel_s.at<float>(a + n, b + n) = exp(-(pow(a, 2) / (2 * pow(sigma_s, 2))) -
                                                   (pow(b, 2) / (2 * pow(sigma_t, 2))));
            kernel_r.at<float>(a + n, b + n) = exp(-(pow(I_sub, 2) / (2 * pow(sigma_r, 2))));
         }
      }
   }

   for (int a = -n; a <= n; a++) {
      for (int b = -n; b <= n; b++) {
         if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
            kernelvalue_s = kernel_s.at<float>(a + n, b + n);
            kernelvalue_r = kernel_r.at<float>(a + n, b + n);

            denom += kernelvalue_s * kernelvalue_r;

            sum1 += kernelvalue_s * kernelvalue_r * input.at<G>(i + a, j + b);
         }
      }
   }
   output.at<G>(i, j) = (G)sum1 / denom;
}
```

<adjustkernel>
비슷한 방식으로 수행함.


noised


zero-padding

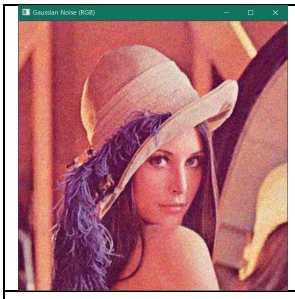
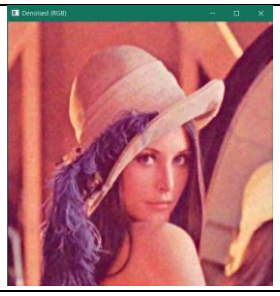
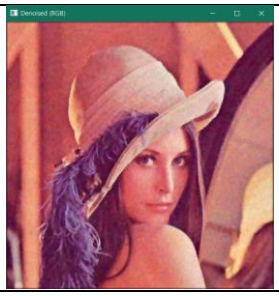
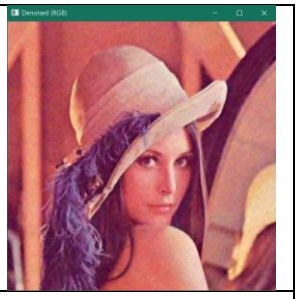

mirroring


adjustkernel

```
if (!strcmp(opt, "zero-padding")) {
    float sum1[3] = { 0, };
    float denom[3] = { 0, };
    float rgb_sub[3] = { 0, };

    for (int a = -n; a <= n; a++) {
        for (int b = -n; b <= n; b++) {
            if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                rgb_sub[0] = input.at<C>(i, j)[0] - input.at<C>(i + a, j + b)[0];
                rgb_sub[1] = input.at<C>(i, j)[1] - input.at<C>(i + a, j + b)[1];
                rgb_sub[2] = input.at<C>(i, j)[2] - input.at<C>(i + a, j + b)[2];
            }
            else {
                rgb_sub[0] = input.at<C>(i, j)[0];
                rgb_sub[1] = input.at<C>(i, j)[1];
                rgb_sub[2] = input.at<C>(i, j)[2];
            }
            rgb_sub[0] *= Bilateral_amount;
            rgb_sub[1] *= Bilateral_amount;
            rgb_sub[2] += Bilateral_amount;

            kernel_s.at<float>(a + n, b + n) = exp(-(pow(a, 2) / (2 * pow(sigma_s, 2))) - (pow(b, 2) /
                                                                                    (2 * pow(sigma_t, 2))));
            kernel_r.at<Vec3f>(a + n, b + n)[0] = exp(-(pow(rgb_sub[0], 2) / (2 * pow(sigma_r, 2))));
            kernel_r.at<Vec3f>(a + n, b + n)[1] = exp(-(pow(rgb_sub[1], 2) / (2 * pow(sigma_r, 2))));
            kernel_r.at<Vec3f>(a + n, b + n)[2] = exp(-(pow(rgb_sub[2], 2) / (2 * pow(sigma_r, 2))));
    }
}

for (int a = -n; a <= n; a++) {
    for (int b = -n; b <= n; b++) {
        for (int k = 0; k < 3; k++) {
            kernelvalue_s = kernel_s.at<float>(a + n, b + n);
            kernelvalue_r = kernel_r.at<Vec3f>(a + n, b + n)[k];

            denom[k] += kernelvalue_s * kernelvalue_r;

            if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                sum1[k] += kernelvalue_s * kernelvalue_r * input.at<C>(i + a, j + b)[k];
            }
        }
    }
}
```

```
    output.at<C>(i, j)[0] = sum1[0] / denom[0];
    output.at<C>(i, j)[1] = sum1[1] / denom[1];
    output.at<C>(i, j)[2] = sum1[2] / denom[2];
}
```

<RGB>
같은 작업을 R, G, B 각각에 대해 수행함.
mirroring, adjustkernel도 Gray와 똑같은 작업을 RGB 각각에 대해 수행함

			
noised	zero-padding	mirroring	adjustkernel