Technical report-Assignment04

컴퓨터공학 류다현(2376087)

이미지 복원(restoration)은 이미지에 노이즈, out-of-focus blur, motion blur 등의 degradation이 있을 때 이를 제거하는 작업으로, 손상되지 않은 목표 이미지 f(x, y)에 blurring 커널 h(x, y), 노이즈 n(x, y)가 작용한 손상 이미지 g(x, y)가 주어졌을 때 g(x, y)로부터 f(x, y)를 찾아낸다. 이를 수식으로 나타내면 아래와 같다.

$$g(x,y) = h(x,y) * f(x,y) + n(x,y)$$

이때 노이즈란 외부 요인으로 인해 이미지에 나타나는 손상을 통칭하며, salt and pepper 노이즈, gaussian 노이즈, speckle 노이즈, periodic 노이즈 등이 있다.

salt_and_pepper.cpp

salt and pepper 노이즈는 이미지에 흰색 또는 검은색 점이 무작위로 나타나는 것으로, 값이 0인 픽셀을 pepper 노이즈, 값이 255인 픽셀을 salt 노이즈라 한다. low-pass 필터나 median 필터를 적용해 이를 개선할 수 있다. 이때 uniform mean filter와 같은 low-pass 필터를 적용하는 것보다 median 필터를 적용하는 것이 노이즈 개선 효과가 더 뛰어난데, 이는 각 필터의 작동 방식 차이 때문이다.

uniform mean 필터는 중앙의 레퍼런스 픽셀 (i, j)에 대해 그 주변 픽셀에 일정 크기의 커널을 적용하고, 커널 내 모든 픽셀 값을 단순 산술 평균한 값을 레퍼런스 픽셀에 적용한다. 커널 내 모든 픽셀에 균등한 가중치를 적용해 평균하므로, 커널 안에 극단적인 intensity를 가지는 노이즈 픽셀이 존재한다면 이들이 평균에 큰 영향을 미쳐 커널 평균이 왜곡된다. 또, 산술 평균 값을 적용하는 과정에서 edge 등 세부 정보가 희석되어 이미지가 blurry해지는 효과 또한 발생한다.

이와 달리 median 필터는 커널 내의 픽셀 값을 크기 순으로 정렬한 다음, 그 중간에 위치하는 픽셀 값을 레퍼런스 픽셀에 할당한다. 정렬된 값의 중앙값을 선택하여 극단적인 값을 배제하는 방식이므로 outlier 값을 가지는 픽셀, 즉 노이즈 픽셀을 제거하는 데 효과적이다.

1. **main()**

   main() 함수에서는 입력 이미지를 읽어 grayscale로 변환하고, 노이즈 생성 함수로 grayscale 이미지와 컬러 이미지에 salt and pepper 노이즈를 생성한다. 이후 노이즈 제거 함수를 호출해 노이즈를 제거하고 원본 이미지와 결과 이미지를 별도의 결과 창에 출력한다.

   작업 전, 후 이미지를 저장할 Mat 객체는 아래와 같이 사용한다.

   - input: 입력 컬러 이미지
   - input_gray: input을 grayscale로 변환한 이미지
   - noise_Gray: input_gray에 salt and pepper 노이즈를 생성한 이미지
   - noise_RGB: input에 salt and pepper 노이즈를 생성한 이미지
   - Denoised_Gray: noise_gray에서 노이즈를 제거한 grayscale 이미지
   - Denoised_RGB: noise_RGB에서 노이즈를 제거한 컬러 이미지

2. **Mat Add_salt_pepper_Noise(const Mat input, float ps, float pp)**

   전달받은 input 이미지에 salt and pepper 노이즈를 생성해 노이즈가 적용된 output을 반환한다. input은 원본 이미지를 저장하는 Mat형 객체, float형 ps는 salt 노이즈의 비율, pp는 pepper 노이즈의 비율이다.

   Add_salt_pepper_Noise(input, 0.1f, 0.1f)으로 호출한다면 input의 전체 픽셀에서 각각 10%씩 총 픽셀의 20%를 salt 노이즈와 pepper 노이즈로 변경한다.

2.1 함수 정의

```
Mat output = input.clone();
RNG rng;
```

input을 복제해 output 객체를 생성한다. RNG형 rng 객체는 난수를 생성과 인덱스 지정에 사용된다.

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

amount1, amount2 변수는 각각 pepper 노이즈와 salt 노이즈로 변경할 픽셀 개수를 저장한다.

```
int x, y;
```

x, y 변수는 행과 열에서 무작위 픽셀을 저장하며, 아래에서 수행되는 각 반복 당 무작위로 선택된 픽셀의 위치를 저장한다.

2.2 grayscale 이미지에 노이즈 생성

```
if (output.channels() == 1) {
	…
}
```

이미지의 채널이 1개라면, 즉 grayscale 이미지라면 단일 채널에 대해 salt and pepper 노이즈를 생성한다.

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

counter가 0부터 (amount1-1)까지 반복한다. rng.uniform(0, output.rows), rng.uniform(0, output.cols)을 통해 이미지 전체 픽셀 중에서 랜덤한 행과 열을 선택하며, output 객체에서 선택한 위치의 픽셀 값을 0으로 설정해 검은색 pepper 노이즈를 생성한다.

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

counter가 0부터 (amount2-1)까지 반복한다. 마찬가지로 랜덤한 행과 열을 선택하고, output 객체에서 선택한 픽셀 값에 255를 할당해 흰색 salt 노이즈를 생성한다.

2.3 컬러 이미지에 노이즈 생성

```
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;
        }
}
```

이미지가 채널이 3개인 컬러 이미지라면 각 RGB 채널에 대해 계산한다. 2.2.와 노이즈 생성 방식은 동일하지만, 랜덤 픽셀 값에 0 또는 255를 할당하는 작업을 RGB 채널에 대해 각각 수행한다. 즉, grayscale 이미지에서 난수 생성된 위치의 픽셀이 완전한 0 또는 255 값이 될 수 있는 것과 달리 컬러 이미지는 세 번의 랜덤 픽셀 선정 시 모두 동일한 픽셀이 선정되어 해당 픽셀의 R, G, B 값이 모두 0으로 바뀌거나 모두 255로 바뀌는 경우가 아니라면 최종 이미지에서 노이즈가 흰색 또는 검은색으로 표시되지 않을 수 있다.

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

Salt_pepper_noise_removal_Gray() 함수는 grayscale 이미지 inpunt에 대해 median 필터를 적용해 salt and pepper 노이즈를 제거한다. input은 원본 이미지를 저장하는 Mat형 객체, n은 커널의 반지름, opt는 경계 처리 방식이다.

3.1 변수 정의

```
int row = input.rows;
int col = input.cols;
int kernel_size = (2 * n + 1);
int median;

Mat kernel = Mat::zeros(kernel_size * kernel_size, 1, input.type());
Mat output = Mat::zeros(row, col, input.type());
```

row와 col에 이미지의 세로, 가로 크기를 저장한다. kernel_size는 커널의 한 변의 크기로, (2n+1)로 설정해 레퍼런스 픽셀을 기준으로 좌우, 상하 2n개의 픽셀들을 포함한다. median은 커널 내 값들의 중앙값을 저장할 변수이다.

(kernel_size*kernel_size)*1 크기의 Mat형 객체 kernel을 생성하고 0으로 초기화하여 커널을 생성한다. output 객체는 결과 이미지를 저장하는 데 사용한다. kernel 객체가 (kernel_size*kernel_size)*1의 1열짜리 행렬이므로 convolution 과정에서 커널 내에서 인덱스를 증가시키며 커널 행렬에 값을 저장할 kenel_index 변수를 선언해 사용한다.

3.2 Convolution

```
for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
                std::vector<G> kernel_vector = {}; // adjustkernel 용
                  …
```

2중 for문을 반복하며 이미지 내 전체 픽셀에 대해 커널을 적용하고 레퍼런스 픽셀 (i, j)에 median 필터를 적용한다. 경계 처리 옵션으로 zero-padding, mirroring, adjustkernel을 구현하며, adjustkernel 옵션에서 유효한 픽셀들을 저장하기 위해 kernel_vector 벡터 객체를 선언한다. 레퍼런스 픽셀이 어디에 위치하는지에 따라 커널 내 유효한 픽셀의 개수가 달라질 수 있으므로 배열 크기를 동적으로 조절하기 위해 백터 객체를 사용한다.

3.2.1 Zero-padding

```
if (!strcmp(opt, "zero-padding")) {
      int kernel_index = 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)) {
                        kernel.at<G>(kernel_index, 0) = input.at<G>(i + x, j + y);
                  }
                  kernel_index++;
            }
   }
}
```

opt가 "zero-padding"이면 이미지 범위 밖의 픽셀을 0으로 처리한다. 커널 좌표 (i+x, j+y)가 입력 이미지의 범위 내에 존재하면 해당 값을 kernel의 (kernel_index, 0) 위치에 저장한다. 범위 밖일 경우 아무런 구현 없이 지나가는데, kernel을 선언할 때 0으로 초기화했으므로 zero padding 효과를 구현할 수 있다. 매 반복마다 kernel_index를 1 증가시켜 다음 인덱스에 저장할 수 있도록 한다.

3.2.2 Mirroring

```
else if (!strcmp(opt, "mirroring")) {
   int tempa, tempb;
   int kernel_index = 0;

   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>(kernel_index, 0) = input.at<G>(tempa, tempb);
                  kernel_index++;
            }
      }
}
```

opt가 "mirroring"일 때, 처리할 픽셀이 이미지 범위를 벗어났을 경우 픽셀을 대칭시켜 처리한다. (i+x)가 row-1보다 클 경우 tempa를 i-x로 계산해 대칭된 인덱스를 사용하며, (i+x)가 음수 범위일 경우 -(i+x)와 같이 양수로 대칭시켜 사용한다. (i+x)가 정상 범위 내에 존재할 경우 이를 그대로 사용한다. 수직 방향 픽셀 (j+y)에 대해서도 동일한 방식으로 계산하며, 마찬가지로 kernel의 (kernel_index, 0)위치에 (tempa, tempb) 픽셀 값을 매핑하고 매 반복마다 kernel_index를 1 증가시켜 kernel의 다음 위치에 저장할 수 있게 한다.

3.2.3 Adjustkernel

```
else if (!strcmp(opt, "adjustkernel")) {
   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_vector.push_back(input.at<G>(i + x, j + y));

                    }
            }
    }
}
```

adjustkernel 옵션으로 경계를 처리한다면 커널 내 값들 중 이미지 범위 내의 유효한 픽셀들만을 커널에 저장한다. 이때 다른 옵션들은 유효하지 않은 위치에 0이나 mirroring한 값을 매핑해 (kernel_size*kernel_size) 고정 크기의 커널 전체를 사용하였지만, adjustkernel 옵션은 유효한 픽셀들만 고려하고 유효하지 않은 픽셀은 채워 넣지 않으므로 커널 크기가 다를 수 있다. 따라서 3.2.에서 생성한 kernel_vector 객체를 사용해 각 반복의 커널 크기에 따라 배열 크기를 동적으로 조절하며, 유효한 픽셀에 대해서만 kernel_vector에 push한다.

3.2.4 커널 정렬 및 중앙값 계산

경계 처리 옵션이 adjustkernel일 때와 아닐 때를 구분해 처리한다. 두 경우 모두 커널 내에 저장된 값들을 오름차순 정렬해 중앙값을 계산하지만, 정렬과 계산 방식에 차이가 있다.

```
if (!strcmp(opt, "adjustkernel")) {
    sort(kernel_vector.begin(), kernel_vector.end());
    median = floor(kernel_vector.size() / 2);
    output.at<G>(i, j) = kernel_vector.at(median);
}
```

adjustkernel 옵션의 경우 kernel_vector에 저장된 값들을 std::sort를 이용해 오름차순 정렬하고 kernel_vector.size()의 절반 위치에 있는 원소를 중앙값으로 결정하여 이를 결과 이미지 output의 (i, j)에 할당한다.

```
else {
    // Sort the kernels in ascending order
    sort(kernel, kernel, SORT_EVERY_COLUMN + SORT_ASCENDING);
    median = floor(kernel_size * kernel_size / 2);
    output.at<G>(i, j) = kernel.at<G>(median, 0);
}
```

zero padding과 mirroring 옵션의 경우 kernel 객체를 대상으로 정렬하며, 정렬 후 kernel의 크기인 (kernel_size*kernel_size)의 절반 위치에 있는 원소를 중앙값으로 결정하여 이를 결과 이미지 output의 (i, j) 위치에 저장한다.

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

3에서 설명한 salt pepper 노이즈 제거 작업을 동일하게 수행하되 컬러 이미지를 사용한다. 3채널 RGB 이미지를 사용하고 있으므로 kernel 객체를 생성하는 방식과 채널별로 분리하여 계산하는 것을 제외하면 동일한 메커니즘을 가진다.

row, col, kernel_size 등 함수 초반의 변수 선언은 유사하며, 사용하는 경계 처리 옵션이나 반복문을 통해 커널 내부 값들을 계산하고 해당 값들로 중앙값을 계산하는 전체적인 구조 또한 동일하다.

4.1 채널 처리 및 커널 생성

```
int channel = input.channels();
// initialize ( (TypeX with 3 channel) - (TypeX with 1 channel) = 16 )
// ex) CV_8UC3 - CV_8U = 16
Mat kernel = Mat::zeros(kernel_size * kernel_size, channel, input.type() - 16);
```

3채널 RGB 이미지를 사용하므로 channel 변수에는 3이 저장된다. Salt_pepper_noise_removal_Gray() 함수에서는 단일 채널 이미지에 대해 1열짜리 커널을 사용했으나, 컬러 이미지의 경우 채널이 3개이므로 커널을(kernel_size*kernel_size)*3 크기로 생성한다.

이때 type에 input.type()-16을 전달하는데, 이는 3채널(CV_8UC3)과 단일 채널(CV_8U)의 타입 차이가 16이므로 RGB 각 채널 별 단일 채널 타입으로 커널 객체를 생성하기 위함이다.

4.2 Convolution

2중 for문을 반복하며 이미지 내 전체 픽셀에 대해 커널을 적용하고 커널 내 값들을 오름차순 정렬해 얻은 중앙값을 결과 이미지의 (i, j) 위치에 매핑하는 구조는 동일하다.

```
std::vector<G> kernel_vector_r = {};
std::vector<G> kernel_vector_g = {};
std::vector<G> kernel_vector_b = {};
```

adjustkernel 옵션에서 커널로 사용할 kernel_vector의 경우 각 채널당 하나의 배열을 지정해 채널별로 커널 값을 계산하므로 kernel_vector_r, g, b로 분리하여 3개의 커널 벡터를 사용한다.

이후 옵션별로 경계 픽셀을 처리하는 원리는 동일한데, 이때 kernel 객체에 접근할 때 아래와 같이 각 채널별로 분리해 계산하는 것에 유의한다.

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

(kernel_size*kernel_size)*3 크기의 3열짜리 kernel 배열 하나를 사용하므로 kernel에 접근할 때 (kernel_index, k) (이때 k=채널 인덱스)를 사용하고 input 또한 (i+x, j+j)[0]과 같이 채널별로 접근한다. 매 반복마다 kernel_index 변수를 1 증가시켜 다음 인덱스로 넘어갈 수 있도록 하는 것은 위와 동일하다.

```
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_vector_r.push_back(input.at<C>(i + x, j + y)[0]);
             kernel_vector_g.push_back(input.at<C>(i + x, j + y)[1]);
             kernel_vector_b.push_back(input.at<C>(i + x, j + y)[2]);
        }
   }
}
```

adjustkernel 옵션으로 경계를 처리할 때도 분리된 kernel_vector_r, g, b 배열에 각각 [0], [1], [2] 채널의 픽셀 값을 저장한다.

4.3 커널 정렬 및 중앙값 계산

채널별 커널 값들을 각각 오름차순 정렬한 뒤, 커널 종류(adjustkernel 옵션이면 kernel_vector, 아니라면 kernel)에 따라 중앙 인덱스를 계산해 중앙값을 결과 이미지에 할당한다.

```
if (!strcmp(opt, "adjustkernel")) {
       sort(kernel_vector_r.begin(), kernel_vector_r.end());
       median = floor(kernel_vector_r.size() / 2);       // rgb 커널 사이즈 동일하므로 r 만
       sort(kernel_vector_g.begin(), kernel_vector_g.end());
       sort(kernel_vector_b.begin(), kernel_vector_b.end());

       output.at<C>(i, j)[0] = kernel_vector_r.at(median);
       output.at<C>(i, j)[1] = kernel_vector_g.at(median);
       output.at<C>(i, j)[2] = kernel_vector_b.at(median);
}
```

adjustkernel 옵션일 때 각 채널 커널별로 오름차순 정렬하고 kernel_vector_r의 절반 크기 인덱스를

median에 저장한다. kernel_vector_r, g, b의 크기는 모두 동일하므로 kernel_vector_r에 대해서만 인덱스를 계산하고 이를 모든 채널 커널 벡터에 사용해도 무방하다.이후 결과 이미지 output의 (i, j)픽셀 0, 1, 2 채널에 각각 R채널 커널의 median번째 원소, G채널 커널의 median번째 원소, B채널 커널의 median번째 원소를 저장한다.

```
else {
	// Sort the kernels in ascending order
	sort(kernel, kernel, SORT_EVERY_COLUMN + SORT_ASCENDING);

	median = floor(kernel_size * kernel_size / 2);

	output.at<C>(i, j)[0] = kernel.at<G>(median, 0);
	output.at<C>(i, j)[1] = kernel.at<G>(median, 1);
	output.at<C>(i, j)[2] = kernel.at<G>(median, 2);
}
```

다른 옵션을 사용할 경우, 각 채널별 커널을 분리해 총 3개의 커널을 사용하는 adjustkernel 옵션과 달리 3열짜리 커널 하나를 사용한다. 따라서 SORT_EVERY_COLUMN+SORT_ASCENDING 플래그로 kernel 객체를 채널별로 오름차순 정렬하며, (kernel_size*kernel_size)의 절반 인덱스를 median으로 지정한다. 이후 output 객체의 각 채널에 kernel의 (median, channel)픽셀 값을 저장한다.

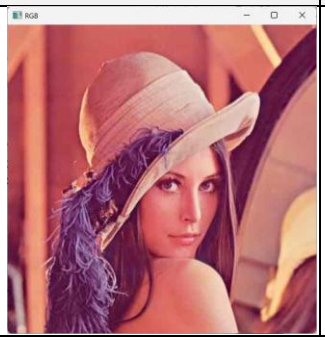
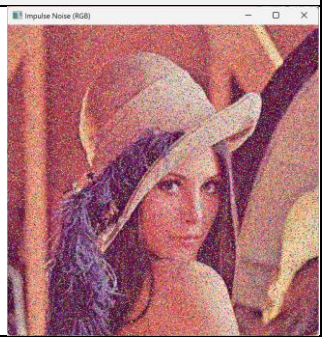
5. 결과 분석

입력 이미지로는 "lena.jpg"를 사용한다.

5.1 Salt and pepper noise 추가

```
Mat noise_Gray = Add_salt_pepper_Noise(input_gray, 0.1f, 0.1f);
Mat noise_RGB = Add_salt_pepper_Noise(input, 0.1f, 0.1f);
```

위와 같이 호출해 salt 노이즈와 pepper 노이즈가 각각 전체 픽셀의 10%를 차지하도록 하였다.

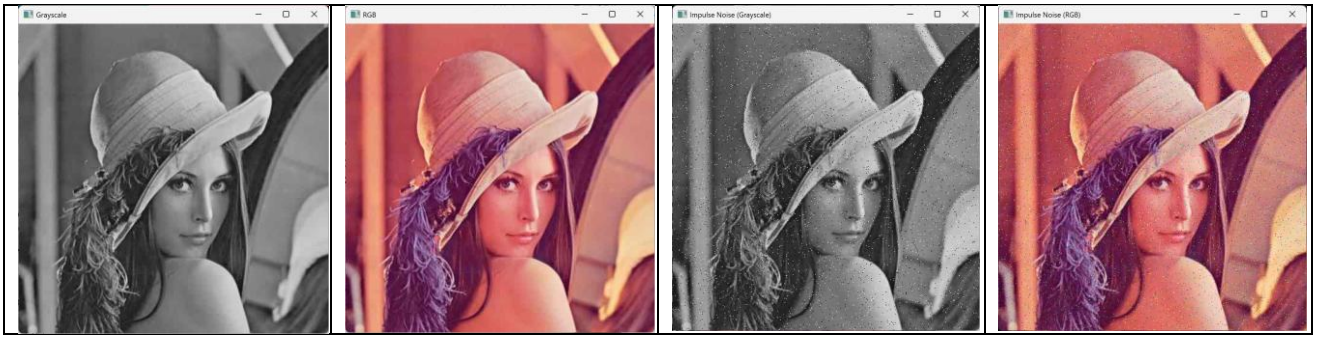
Grayscale	RGB	Impulse Noise (Grayscale)	Impulse Noise (RGB)
			

원본 이미지와 salt and pepper 노이즈가 생성된 이미지가 별도의 창에 출력되었다. 2.2에 서술한 바와 같이 grayscale 노이즈 이미지에는 흰색과 검은색의 노이즈가 표시되고 RGB 노이즈 이미지에는 색깔이 있는 노이즈가 표시되는데, 이는 Add_salt_pepper_Noise() 함수에서 컬러 이미지에 노이즈를 생성할 때 한 채널만을 변경하여 전체 채널이 0 또는 255가 되지 않는 경우가 존재하기 때문이다.

```
Mat noise_Gray = Add_salt_pepper_Noise(input_gray, 0.01f, 0.01f);
Mat noise_RGB = Add_salt_pepper_Noise(input, 0.01f, 0.01f);
```

위와 같이 노이즈 비율을 0.01로 설정하면 salt 노이즈와 pepper 노이즈가 각각 전체 픽셀의 1%를 차지하게 되어 0.1로 호출했을 때보다 노이즈 픽셀이 드물게 나타나는 것을 확인할 수 있다.

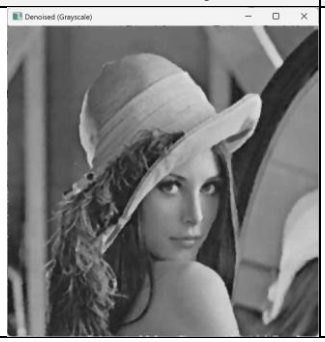
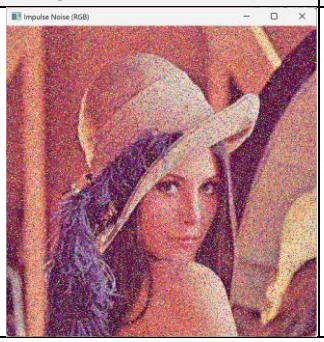
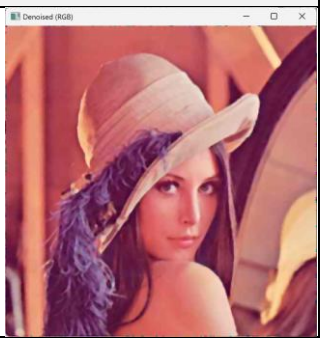
Grayscale	RGB	Impulse Noise (Grayscale)	Impulse Noise (RGB)



5.2 Salt and pepper noise 제거

```
int window_radius = 2;
Mat Denoised_Gray = Salt_pepper_noise_removal_Gray(noise_Gray, window_radius, "zero-
padding");
Mat Denoised_RGB = Salt_pepper_noise_removal_RGB(noise_RGB, window_radius, "zero-
padding");
```

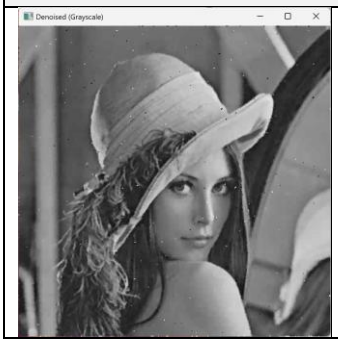
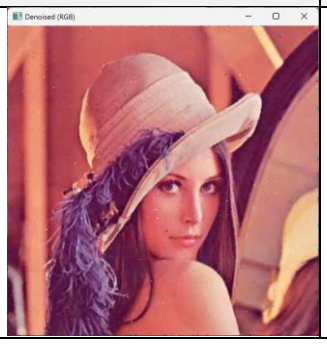
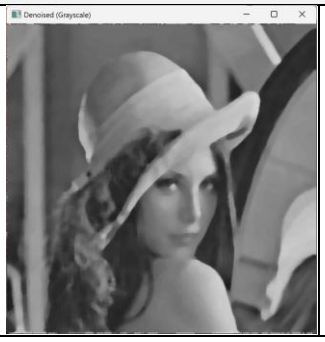
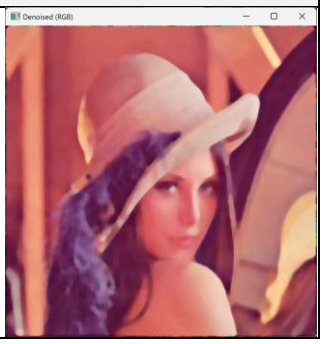
grayscale, 컬러 이미지 모두 커널 반지름을 2로 "zero padding" 옵션을 전달해 노이즈 제거 함수를 호출, 크기가 (2*2+1=5)인 커널을 사용해 zero padding 방식으로 경계 처리를 수행하여 salt and pepper 노이즈를 제거했다.

Impulse Noise (Grayscale)	Denoised (Grayscale)	Impulse Noise (RGB)	Denoised (RGB)
			

에지가 다소 흐려졌으나 노이즈가 제거된 것을 확인할 수 있다.

5.3 커널 크기별 비교

커널 반지름 window_radius를 1과 5로 설정해 노이즈 제거 함수를 각각 호출하고 결과를 비교한다.

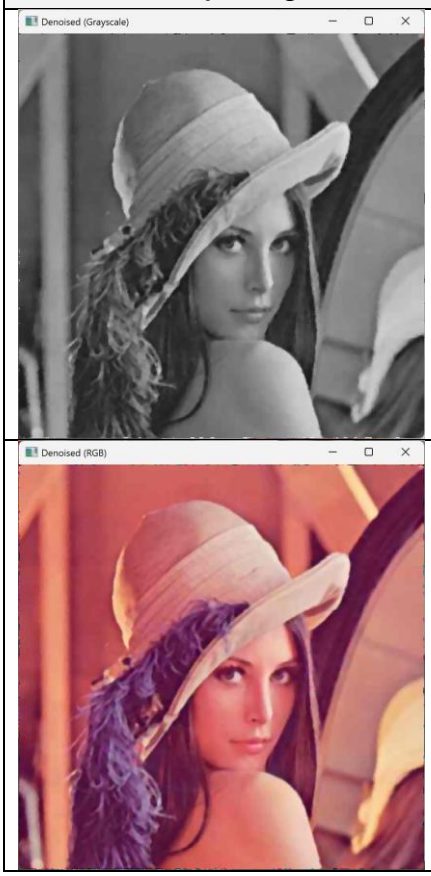
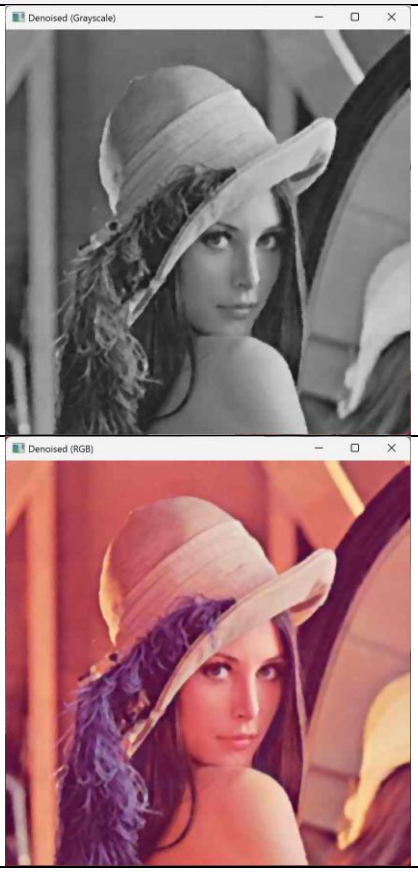
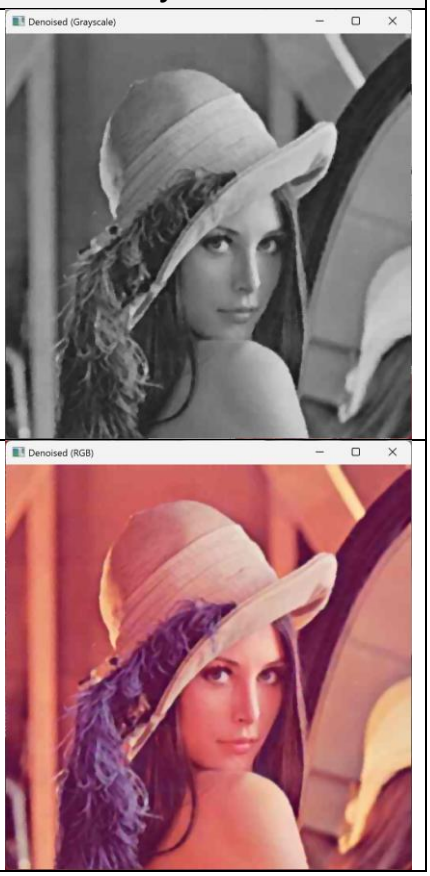
Window_radius=1	Window_radius=5
 	 

window_radius=1일 때 더 좁은 커널 영역에서 중앙값을 선택하므로 5.2의 경우보다 에지가 선명하고, window_radius=5일 경우 더 넓은 커널 영역에서 중앙값을 선택하므로 세부 정보가 더욱 희석된다.
또, window_radius=1일 때 일부 노이즈가 제거되지 않아 결과 이미지에서 여전히 표시되고 있는 것을 확인할 수 있는데, 이는 작은 커널 크기로 인해 중앙값 계산 시 충분히 많은 주변 픽셀들을 고려하지 못하므로 노이즈가 있는 픽셀이 중앙값으로 채택되었기 때문이다.
이를 통해 커널이 지나치게 작을 경우 노이즈 제거 성능이 나빠지고, 지나치게 클 경우 세부 정보가 희석

되어 에지가 불분명하게 나타나는 문제가 발생하므로 커널 크기를 적절히 설정해야 함을 알 수 있다.

5.4 경계 처리 옵션별 비교

window_radius=2로 통일하고 "zero-padding", "mirroring", "adjustkernel"의 모든 opt로 실행하였다.

Zero padding	mirroring	adjustkernel
		

모든 옵션에 대해 노이즈가 제거된 결과 이미지가 출력되었다.

Gaussian.cpp

가우시안 노이즈는 노이즈가 가우시안 분포에 따라 형성된 것을 말하며, white 노이즈는 주파수 대역 전체에 골고루 분포한다. 이때 노이즈가 있는 이미지 I_G는 원본 이미지 I에 백색 가우시안 노이즈 N이 더해진(additive) 형태이며, Additive White Gaussian Noise(AWGN) N의 평균은 0이다. 이를 수식으로 나타내면 아래와 같다.

$$I_G(x,y) = I(x,y) + N(x,y)$$

따라서 n개의 I_G에 대해 이들을 평균하면 $\frac{1}{n}\sum_{i=1}^{n} I_G^i(x,y) = I(x,y) + \frac{1}{n}\sum_{i=1}^{n} N^i(x,y)$이다. n개의 노이즈 이미지를 평균할 경우 원본 이미지 I는 모든 i에서 동일하므로 그대로 유지되고, $\frac{1}{n}\sum_{i=1}^{n} N^i(x,y)$가 0이므로 사라져 $\frac{1}{n}\sum_{i=1}^{n} I_G^i(x,y) = I(x,y) + \frac{1}{n}$가 된다.

인접 픽셀들을 평균하는 방식으로 이를 구현할 수 있는데, 커널 내 값들을 단순 산술 평균하는 uniform mean filter 보다는 레퍼런스 픽셀과의 거리를 고려하는 가우시안 필터를 사용하는 것이 필터링 성능을 향상시킬 수 있다.

1. **main()**

   main() 함수에서는 입력 이미지를 읽어 grayscale로 변환하고, 노이즈 생성 함수로 grayscale 이미지와 컬러 이미지에 가우시안 노이즈를 생성한다. 이후 노이즈 제거 함수를 호출해 가우시안 필터를 적용함으로써 노이즈를 제거하고 원본 이미지와 결과 이미지를 별도의 결과 창에 출력한다.

   작업 전, 후 이미지를 저장할 Mat 객체는 아래와 같이 사용한다.

   - input: 입력 컬러 이미지
   - input_gray: input을 grayscale로 변환한 이미지
   - noise_Gray: input_gray에 가우시안 노이즈를 생성한 이미지
   - noise_RGB: input에 가우시안 노이즈를 생성한 이미지
   - Denoised_Gray: noise_gray에서 노이즈를 제거한 grayscale 이미지
   - Denoised_RGB: noise_RGB에서 노이즈를 제거한 컬러 이미지

2. **Mat Add_Gaussian_noise(const Mat input, double mean, double sigma)**

   Add_Gaussian_noise() 함수는 입력 이미지 input에 가우시안 노이즈를 생성하는데, 이때 생성된 가우시안 노이즈는 mean만큼의 평균과 sigma만큼의 표준편차를 가진다.

```
Mat Add_Gaussian_noise(const Mat input, double mean, double sigma) {

    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;
}
```

   크기와 타입이 input과 동일한 NoiseArr 배열을 선언해 0으로 초기화하고, 난수 생성기 RNG형 rng로 rng.fill() 함수를 사용하여 NoiseArr에 가우시안 분포를 가지는 노이즈를 생성한다. 이후 add() 함수를 사용, 원본 이미지 input에 NoiseArr 배열을 더한 것을 다시 NoiseArr 배열에 저장하고 이를 return한다.

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

   Gaussianfilter_Gray() 함수와 Gaussianfilter_RGB() 함수는 이전에 구현한 것과 동일하다. Gaussianfilter_Gray() 함수는 grayscale 이미지에, Gaussianfilter_RGB() 함수는 컬러 이미지에 가우시안 필터를 적용하는데, input에 대해

sigmaT, sigmaS 2개의 표준편차를 사용해 노이즈를 제거하고 경계 처리 옵션으로 mirroring, zero-padding, adjustkernel의 3가지를 제공한다. 노이즈가 있는 input에 가우시안 필터를 적용하면 픽셀 값들이 평균되어 노이즈가 개선된다.

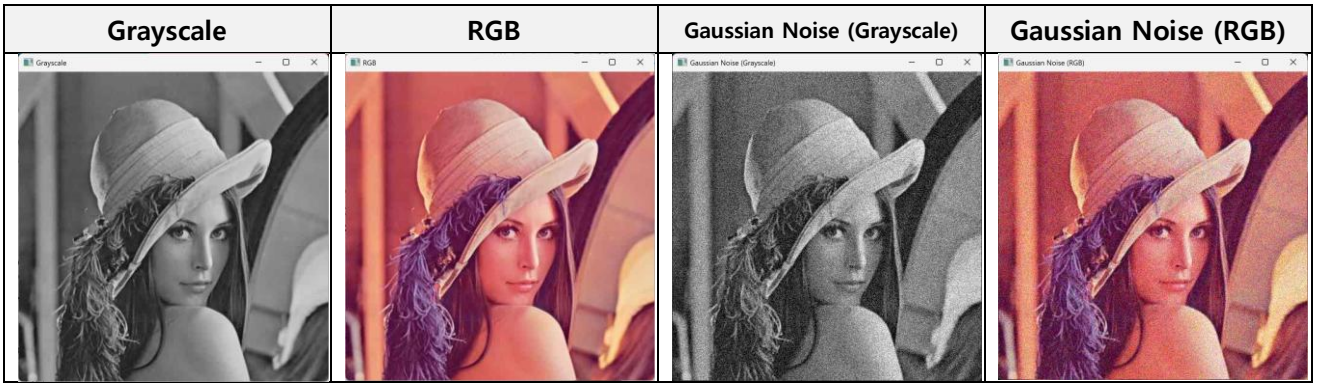
4. 결과 분석

입력 이미지로는 "lena.jpg"를 사용한다.

4.1 가우시안 노이즈 추가

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.1);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.1);
```

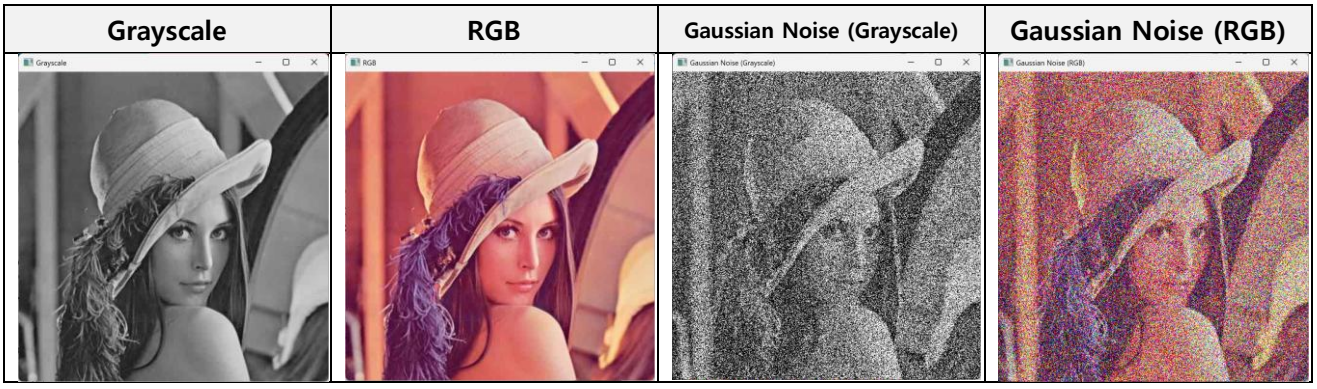
위와 같이 호출해 가우시안 노이즈의 평균을 0, 표준편차를 0.1로 설정했다.



실행 결과 원본 이미지와 가우시안 노이즈가 생성된 이미지가 별도의 창에 출력되었다.

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.5);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.5);
```

위와 같이 표준편차를 0.5로 설정해 실행하였다.



가우시안 분포에서 표준편차는 가우시안 함수 그래프의 폭을 결정한다. 표준편차가 클 경우 그래프의 폭이 넓어지므로 노이즈 분포가 넓어지고 양 극단의 값이 자주 나타나며, 작을 경우 그래프의 폭이 좁아져 대부분의 노이즈가 0에 가까운 값을 가진다. 따라서 sigma=0.5로 설정했을 때 각 픽셀에 진폭이 더 큰 값이 더해지게 되어 결과 이미지에 더 강한 노이즈가 생성된다.

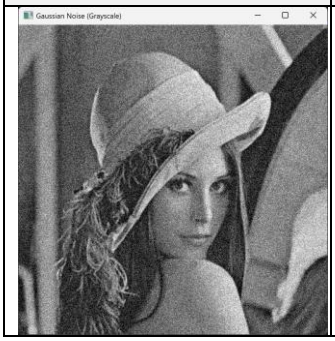
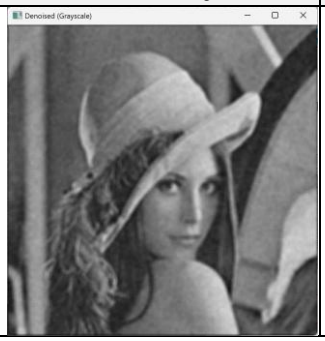
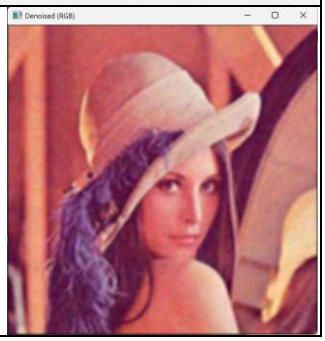
모든 노이즈 픽셀이 0 또는 255의 값을 가지는 salt and pepper 노이즈와 달리, 노이즈 생성 과정에서 가우시안 함수를 사용해 가우시안 분포에 따른 랜덤 값을 원본 이미지 픽셀에 더하므로 다양한 intensity의 노이즈 픽셀이 표시되는 것을 확인할 수 있다.

4.2 가우시안 필터링으로 노이즈 제거

```
Mat Denoised_Gray = Gaussianfilter_Gray(noise_Gray, 3, 10, 10, "zero-padding");
Mat Denoised_RGB = Gaussianfilter_RGB(noise_RGB, 3, 10, 10, "zero-padding");
```

위와 같이 호출해 커널 크기는 2*3+1=7로, 수평 및 수직 방향으로 각각 10만큼의 표준편차를 적용하였

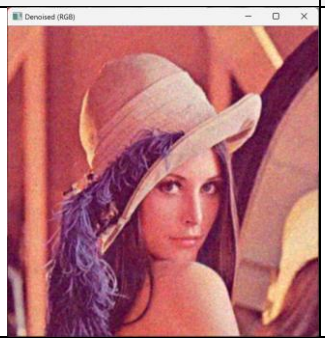
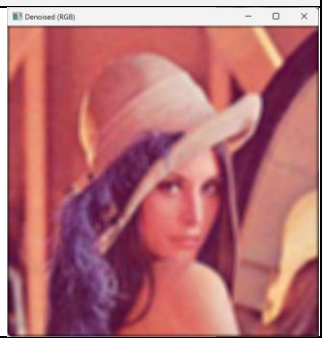
다. 경계 처리 옵션은 zero padding으로 한다.

Gaussian Noise (Grayscale)	Denoised (Grayscale)	Gaussian Noise (RGB)	Denoised (RGB)
			

실행 결과 노이즈 이미지에 가우시안 필터가 적용되어 노이즈가 개선되었다. 하지만 가우시안 필터는 커널 내 값들에 가우시안 함수를 적용하여 평균하므로, 주변 픽셀의 값이 희석되어 이미지가 blurry해지는 것을 확인할 수 있다. 따라서 median 필터를 적용해 salt and pepper 노이즈를 제거한 결과와 비교하면 에지가 상대적으로 덜 보존된다.

4.3 커널 크기별 비교

커널 반지름을 1, 5로 설정하여 각 실행 결과를 비교한다. 표준편차와 경계 처리 옵션은 10과 zero padding으로 통일한다.

n=1		n=5	
			

4.2에서 n=3으로 실행한 것에 비해 n=1일 때 노이즈 제거 효과가 미미하고 에지는 상대적으로 잘 보존된다. n=5로 증가시켰을 때 노이즈가 잘 제거되었으나, 결과 이미지가 더욱 blurry해져 세부 정보가 소실되었다.

Bilateral.cpp

가우시안 필터는 커널 픽셀이 레퍼런스 픽셀로부터 얼마나 떨어졌는지를 고려하여, 즉 공간적인 요소만을 고려해 가중치를 계산한다. 따라서 레퍼런스 픽셀과의 거리가 비슷한 픽셀에 비슷한 가중치가 적용되므로 에지 근처의 픽셀이 희석되어 blurry해진다. 반면 bilateral 필터는 공간적인 요소(spatial difference)와 픽셀 값 차이(range difference)를 모두 고려하는데, 레퍼런스 픽셀과 intensity가 다른 픽셀은 멀리 떨어진 것으로 고려하여 가중치를 적게 적용하므로 에지가 덜 blurry해진다.

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

위 bilateral 필터 공식에서 커널 함수 $w(s,t)$는 $w(s,t) = \frac{1}{W(i,j)} \exp\left(-\frac{s^2}{2\sigma_s^2} - \frac{t^2}{2\sigma_t^2}\right) \exp\left(-\frac{(I(i,j)-I(i+s,j+t))^2}{2\sigma_r^2}\right)$이고, 이때 $W(i,j) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} \exp\left(-\frac{m^2}{2\sigma_s^2} - \frac{n^2}{2\sigma_t^2}\right) \exp\left(-\frac{(I(i,j)-I(i+m,j+n))^2}{2\sigma_r^2}\right)$ 이므로 레퍼런스 픽셀 p(i, j)와 커널 내 픽셀 q(i+s, j+t)에 대해 bilateral 커널 공식은 다음과 같다.

$$O_p = \frac{1}{W_p} \sum_{q \in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|) I_q, W_p = \sum_{q \in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|)$$

공간 커널 G_s는 레퍼런스 픽셀과의 거리에 따라 가중치를 부여하고, 거리가 가까울수록 더 큰 가중치가 적용된다. G_r은 intensity 커널로 레퍼런스 픽셀과의 밝기 차이를 고려하며, 밝기가 비슷할수록 큰 가중치를 부여해 에지를 보존한다.

1. **main()**
   main() 함수에서는 입력 이미지를 읽어 grayscale로 변환하고, 노이즈 생성 함수로 grayscale 이미지와 컬러 이미지에 salt and pepper 노이즈를 생성한다. 이후 노이즈 제거 함수를 호출해 노이즈를 제거하고 원본 이미지와 결과 이미지를 별도의 결과 창에 출력한다.
   작업 전, 후 이미지를 저장할 Mat 객체는 아래와 같이 사용한다.
   - input: 입력 컬러 이미지
   - input_gray: input을 grayscale로 변환한 이미지
   - noise_Gray: input_gray에 가우시안 노이즈를 생성한 이미지
   - noise_RGB: input에 가우시안 노이즈를 생성한 이미지
   - Denoised_Gray_bilateral: noise_gray에서 노이즈를 제거한 grayscale 이미지
   - Denoised_RGB_bilateral: noise_RGB에서 노이즈를 제거한 컬러 이미지

2. **Mat Add_Gaussian_noise(const Mat input, double mean, double sigma)**
   Gaussian.cpp의 가우시안 노이즈 생성 함수와 동일하다. 전달받은 평균 mean과 표준편차 sigma를 이용하여 input에 가우시안 노이즈를 생성하고, noisy한 이미지를 반환한다.

3. **Mat bilateralfilter_Gray(const Mat input, int n, float sigmaT, float sigmaS, float sigmaR, const char* opt)**
   bilateralfilter_Gray() 함수는 noisy한 이미지 input에 bilateral 필터를 적용하여 가우시안 노이즈를 제거한다.

   3.1 변수 정의 및 spatial 커널 G_s 계산

```
int row = input.rows;
int col = input.cols;
int kernel_size = (2 * n + 1);
int tempa;
int tempb;
float denom;
```

```
// Gs = exp(-s^2/2sigmaS^2-t^2/2sigmaT^2)
Mat Gs = Mat::zeros(kernel_size, kernel_size, CV_32F);
denom = 0.0;
```

Gaussian.cpp 코드의 가우시안 필터 함수 Gaussianfilter_Gray()와 유사하다. Gaussianfilter_Gray() 함수에서 가우시안 필터링에 사용될 단일 커널 kernel 객체를 생성하고 초기화했었는데, 이를 동일하게 사용하되 spatial 커널과 intensity 커널의 구별이 용이하도록 변수명을 Gs로 변경하였다. Gaussianfilter_Gray() 함수의 가우시안 커널은 Gs와 동일하게 공간 커널이므로 이와 같이 수정해도 무방하다.

3.2 Convolution

위 코드들과 마찬가지로 zero padding, mirroring, adjustkernel의 방식을 사용해 경계 값을 처리한다. 각 옵션을 구현하는 방식은 동일하지만 intensity 커널을 한번 더 적용하므로 Gaussianfilter_Gray() 함수와는 for문 내부 구현에 차이가 있다.

```
float Gr = exp(-(pow(input.at<G>(i, j) - input.at<G>(i + a, j + b), 2) / (2 *
pow(sigmaR, 2))));
float GsGr = Gs.at<float>(a + n, b + n) * Gr;  // Wp = Gs*Gr
sum1 += GsGr * input.at<G>(i + a, j + b);      // Gs*Gr*I
W += GsGr;
```

Gaussianfilter_Gray() 함수에서는 sum1 += kernel.at<float>(a + n, b + n) * input.at<G>(i + a, j + b);과 같이 매 반복마다 sum 변수에 (가중치)*(이미지 픽셀)한 값을 합하기만 하였으나 bilateral 함수에서는 intensity 커널 Gr을 추가로 생성한다.

Gr은 $G_r = \exp\left(-\frac{(I(i,j)-I(i+m,j+n))^2}{2\sigma_r^2}\right)$이므로, 이를 동일하게 코드로 변환한 것이 float Gr = exp(-(pow(input.at<G>(i, j) - input.at<G>(i + a, j + b), 2) / (2 * pow(sigmaR, 2))));이다. 이후 spatial 커널과 intensity 커널을 곱한 값을 변수 GsGr에 저장하면 매 반복 GsGr의 총합으로 수식에서의 W(i, j)를 만들 수 있고, GsGr에 이미지 픽셀 값을 곱한것을 sum1에 더해 $\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|)I_q$를 구할 수 있으므로 위와 같이 계산한다. 이때 W는 옵션별 구현 초반에 sum1 변수와 함께 선언한 것이며, 0으로 초기화되어 있다.

```
if (W != 0) {
    output.at<G>(i, j) = (G)(sum1 / W);
}
else {
    output.at<G>(i, j) = (G)sum1;
}
```

커널에 대해 모두 연산하고 2중 for문을 벗어나면 sum1에는 $\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|)I_q$가, W에는 $\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|)$가 저장되어 있으므로 $O_p = \frac{1}{W_p}\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|I_p - I_q|)I_q$에서 최종 픽셀 값 O=output.at<G>(i, j)= sum1/W이다. 이때 divide by zero exception을 방지하기 위해 W != 0인 경우에만 sum/W를 사용하였으며, W=0이라면 sum1을 그대로 할당하게끔 구현하였다. 위 내용을 골자로 옵션마다 조금씩 수정하여 적용하면 3가지 경계 처리 방식을 사용해 bilateral 필터를 적용할 수 있다.

3.2.1 zero padding

```
if (!strcmp(opt, "zero-padding")) {
    float sum1 = 0.0f;
    float W = 0.0f;
    float Gr = 0.0f;
    float GsGr = 0.0f;
    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)) {
```

```
                Gr = exp(-(pow(input.at<G>(i, j) - input.at<G>(i + a, j + b), 2) / (2
                    * pow(sigmaR, 2))));
                GsGr = Gs.at<float>(a + n, b + n) * Gr;
                sum1 += GsGr * input.at<G>(i + a, j + b);      // Gs*Gr*
            }

            W += GsGr;
        }
    }
    // Op = sum(Gs*Gr*I)/W
    if (W != 0) {
        output.at<G>(i, j) = (G)(sum1 / W);
    }
    else {
        output.at<G>(i, j) = (G)sum1;
    }
}
```

기존 가우시안 필터 구현에서는 커널 단위로 사용하는 변수가 누적합 변수 sum1뿐이었고, 커널 범위 밖 픽셀은 zero padding하여 무시하였으므로(zero padding되어 0이므로 sum1에 더하든 더하지 않든 동일하다) sum1은 valid한 영역, 즉 if 조건문 내부에서만 업데이트되었다. 반면 bilateral 필터 구현에서는 커널 단위 변수가 sum1과 W로 2개이다. 이때 W는 모든 커널 위치에 대해 계산한 가중치를 포함해야 하므로, 0을 더하기 때문에 valid한 픽셀에서만 변경해도 무방한 sum1과 달리 zero padding될 범위 밖 픽셀에 대해서도 GsGr 값이 W에 누적되어야 한다. 따라서 Gr 및 GsGr 계산, W를 업데이트하는 부분을 픽셀 유효성 여부와 관계없이 공통적으로 수행할 수 있도록 조건문 바깥으로 이동하였다.

3.2.2 Mirroring, adjustkernel

zero padding을 제외한 나머지 옵션은 크게 변형하지 않는다. mirroring 옵션은 (i+a, j+b)가 아닌 (tempa, tempb) 인덱스를 사용하도록 수정하며, adjustkernel 옵션은 기존 Gaussianfilter 함수의 adjustkernel 구현에서 Gr, GsGr, W를 계산하는 것으로 변경하기만 하면 된다.

4. Mat bilateralfilter_RGB(const Mat input, int n, float sigmaT, float sigmaS, float sigmaR, const char* opt)

bilateralfilter_RGB() 함수에서는 input을 컬러로 사용하며, 커널과 누적합 sum1 변수 계산에서만 bilateralfilter_Gray() 함수와 차이가 있다. 컬러 이미지에 대해 bilateral 필터를 사용하려면 다음과 같은 공식을 사용한다.

$$R_p = \frac{1}{W_p}\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\,G_{\sigma_r}(|C_p - C_q|)R_q$$

$$G_p = \frac{1}{W_p}\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\,G_{\sigma_r}(|C_p - C_q|)G_q$$

$$B_p = \frac{1}{W_p}\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\,G_{\sigma_r}(|C_p - C_q|)B_q$$

$$W_p = \sum_{q\in N_p} G_{\sigma_s}(|p-q|)\,G_{\sigma_r}(|C_p - C_q|)$$

이때 $C_p = (R_p, G_p, B_p)^T$로, 각 픽셀 p의 컬러 값을 1*3 열 벡터로 나타낸 것이다.

4.1 Convolution

```
Mat Cp = Mat::zeros(3, 1, CV_32F);
Cp.at<float>(0, 0) = input.at<C>(i, j)[0];
Cp.at<float>(1, 0) = input.at<C>(i, j)[1];
Cp.at<float>(2, 0) = input.at<C>(i, j)[2];
```

레퍼런스 픽셀 p에 대해 3*1 벡터 Cp 객체를 선언해 채널 값을 저장한다. 이후 커널 반복문 내에서 인접

픽셀 q에 대해서도 동일한 방식으로 Cq 객체를 선언해 input(i+a, j+b)의 [0], [1], [2] 채널 값을 저장한다 (mirroring 옵션의 경우 input(tempa, tempb) 사용). $R_p = \frac{1}{W_p}\sum_{q\in N_p} G_{\sigma_s}(|p-q|)\, G_{\sigma_r}(|C_p - C_q|) R_q$이므로 Cp-Cq 값을 별도의 Mat형 객체 d에, d를 제곱한 것을 powd 변수에 저장하고 이들로 Gr을 계산한다. 3.2와 동일하게 Gr, GsGr, sum1, W를 계산한다. 이때 sum1을 업데이트하기 위해 input이 아닌 Cq를 사용함에 주의해야 한다.

```
Mat Op;

if (W != 0) {
    Mat Op = sum1 / W;

    C o;
    o[0] = Op.at<float>(0, 0);
    o[1] = Op.at<float>(1, 0);
    o[2] = Op.at<float>(2, 0);

    output.at<C>(i, j) = o;
}
else {
    output.at<C>(i, j) = input.at<C>(i, j);
}
```

커널 반복문을 빠져나오면 W에는 GsGr의 누적 합이, sum1에는 GsGr*Cq의 누적합이 저장되어 있으며, 이들 값으로 레퍼런스 픽셀 값을 계산하되 Cp와 Cq가 float형이 아닌 3*1 벡터이므로 결과 또한 Mat형 Op에 저장한다. 이후 Op의 값을 C형(=Vec3d)로 저장하기 위해 C o;로 선언하고, Op의 각 열을 o에 저장한 후 이를 output(i, j)에 매핑한다.

위 방식에 기초해 픽셀별로 가중치를 계산하며, zero padding 옵션 구현 방식 등 기존 가우시안 필터 구현과 달라져야 하는 요소는 bilateralfilter_Gray() 코드와 같은 방식으로 작성한다.

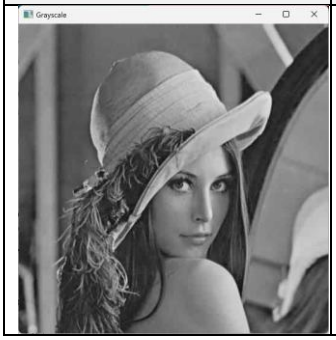
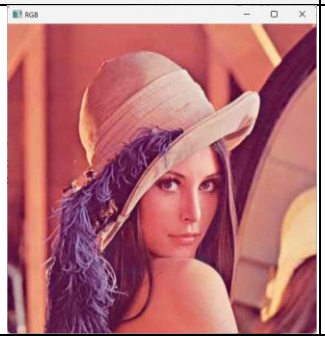
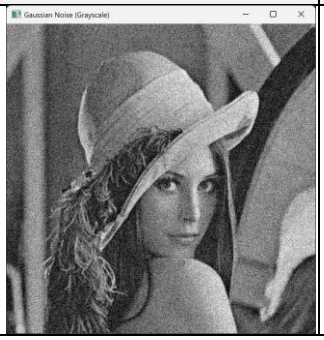
5. 결과 분석

입력 이미지로는 "lena.jpg"를 사용한다.

5.1. 가우시안 노이즈 추가

```
Mat noise_Gray = Add_Gaussian_noise(input_gray, 0, 0.1);
Mat noise_RGB = Add_Gaussian_noise(input, 0, 0.1);
```

위와 같이 호출해 가우시안 노이즈의 평균을 0, 표준편차를 0.1로 설정했다.

Grayscale	RGB	Gaussian Noise (Grayscale)	Gaussian Noise (RGB)
			

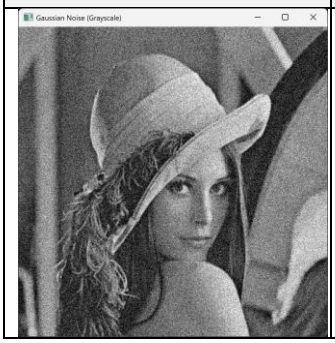
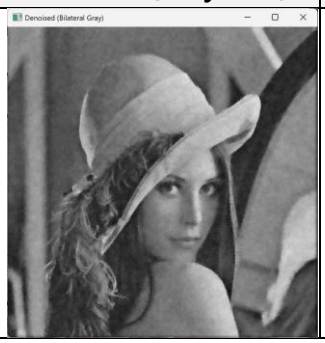
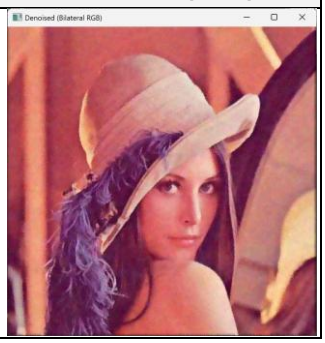
실행 결과 원본 이미지와 가우시안 노이즈가 생성된 이미지가 별도의 창에 출력되었다.

5.2. bilateral 필터링으로 노이즈 제거

```
Mat Denoised_Gray_bilateral = bilateralfilter_Gray(noise_Gray, 5, 3, 3, 0.2, "zero-
padding");
```

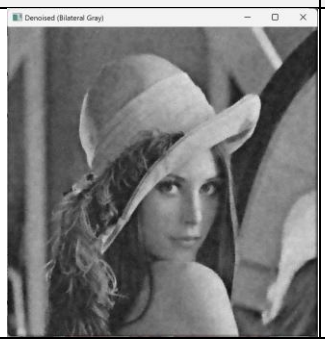
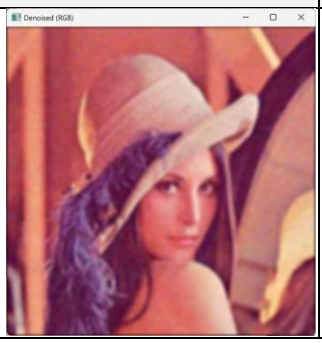
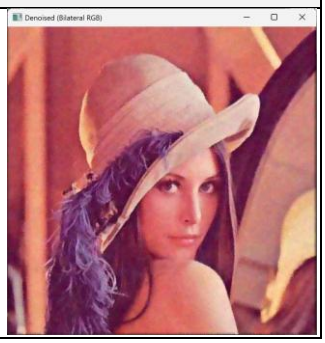
```
Mat Denoised_RGB_bilateral = bilateralfilter_RGB(noise_RGB, 5, 3, 3, 0.2, "zero-padding");
```

위와 같이 호출해 커널 크기는 2*5+1=11로, 수평 및 수직 표준편차는 3으로, intensity 표준편차는 0.2로 설정하였다. 경계 처리 옵션은 zero padding으로 한다.

Gaussian Noise (Grayscale)	Denoised (Grayscale)	Gaussian Noise (RGB)	Denoised (RGB)
			

실행 결과 노이즈가 개선된 이미지가 출력되었으며, 가우시안 필터링으로 노이즈를 제거했을 때보다 결과 이미지의 에지가 보존되어 덜 blurry한 것을 확인할 수 있다.

가우시안 필터와 bilateral 필터에 대해 동일하게 mean=0, sigma=0.1로 가우시안 노이즈를 생성하고 n=5, sigmaT=sigmaS=3으로 노이즈 제거를 수행한 결과를 비교하면 아래와 같다.

Gaussian filter (Grayscale)	Bilateral filter (Grayscale)	Gaussian filter (RGB)	Bilateral filter (RGB)
			

가우시안 필터를 적용했을 때와 비교하면 bilateral 필터를 적용했을 때 에지가 선명하다. 위에서 서술했듯, bilateral 필터는 레퍼런스 픽셀과의 거리뿐 아니라 intensity 차이도 고려한 가중치를 부여하므로 결과 이미지가 덜 blurry하다. intensity가 비슷한 픽셀은 큰 가중치를, 많이 차이나는 픽셀일수록, 즉 에지에 작은 가중치를 부여해 커널 내 값들이 희석되지 않으므로 에지가 선명하게 보존된다.