Technical report – Assignment03

컴퓨터공학 류다현(2376087)

MeanFilterGray.cpp

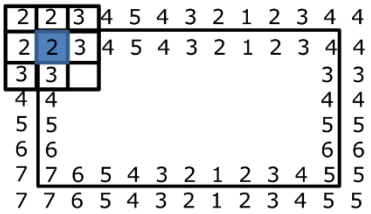
MeanFilterGray.cpp는 이미지에 uniform mean filtering을 적용하는 프로그램이다. uniform mean filtering은 아래와 같이 레퍼런스 픽셀 값을 주변 픽셀들의 산술평균 값으로 설정해 노이즈를 제거하는 작업으로, 간단한 low-pass filter 이며 아래와 같이 수행한다.

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

이때 filtering하고자 하는 픽셀이 이미지의 모서리에 존재한다면 (1)픽셀 값들을 mirroring하거나 (2)zero padding하고, (3)filter kernel을 조정하여 valid한 것만 사용할 수 있다.

1) mirroring

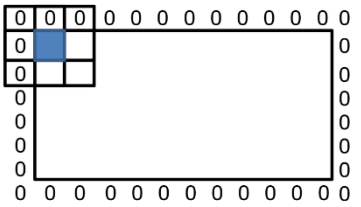
   모서리의 픽셀 값을 이미지 바깥의 filter kernel에 매핑하는 방식이다.



$$O(i,j) = \sum_{s=-1}^{1} \sum_{t=-1}^{1} w(s,t) I'(i+1+s, j+1+t) \ (I' : mirror\ image)$$

2) zero padding

   이미지 범위를 벗어난 픽셀들에 0을 할당한다.



$$O(i,j) = \sum_{s=-1}^{1} \sum_{t=-1}^{1} w(s,t) I'(i+1+s, j+1+t) \ (I' : zero-padded\ image)$$

3) filter kernel 조정

   filter kernel을 조정하여 이미지 범위 내의 것들만 사용하며, 이때 $\sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) = 1$이므로 아래와 같이 계산한다.

$$O(i,j) = \frac{\sum_{s=-1}^{1} \sum_{t=-1}^{1} w'(s,t) I'(i+s, j+t)}{\sum_{s=-1}^{1} \sum_{t=-1}^{1} w'(s,t)} \ (w'(s,t) = (0 \le i+s \le H-1 \& 0 \le j+t \le W-1 \,?\, w(s,t) : 0)$$

uniform mean filter를 적용하면 픽셀들 간의 차이가 줄어들어 이미지가 blurry해지고, 마스크의 크기가 커질수록 결과 이미지가 더욱 흐려진다. 이때 마스크의 크기가 커지면 연산량이 증가하므로, 프로그램 효율을 위해 separable filtering이 중요하다.

associative property는 여러 linear function을 수행할 때 이들을 어떤 순서로 수행하든 결과가 동일한 성질로, 선형 필터인 uniform mean filtering 또한 associative property를 만족한다. n*n 필터를 사용할 경우 한 픽셀 당 n^2번의 연산이 수행되지만, separable filtering을 통해 n*n 필터를 n*1 필터와 1*n 필터로 분리한다면 픽셀 당 2n번의 연산으로 동일한 결과를 얻을 수 있다. 즉, 여러 단계로 필터링할 때 필터의 순서를 변경하거나 필터를 결합 또는 분리할 수 있으며, 이는 runtime 측면에서 매우 유리하다.

1. main()

```
Mat input = imread("lena.jpg", IMREAD_COLOR); // IMREAD_COLOR
Mat input_gray;
Mat output;

cvtColor(input, input_gray, COLOR_RGB2GRAY);
```

imread() 함수와 IMREAD_COLOR flag 를 사용해 "lena.jpg" 이미지를 읽어 와 input 에 저장하고, Mat 형 input_gray 와 output 객체를 선언한 후 input 을 grayscale 로 변환하여 input_gray 에 저장한다.

```
if (!input.data)
{
    cout << "Could not open" << std::endl;
    return -1;
}
```

이미지를 정상적으로 읽어 왔는지 확인하고, 이미지가 invalid하다면 에러 메시지를 출력한 후 프로그램을 종료한다.

```
namedWindow("Grayscale", WINDOW_AUTOSIZE);
imshow("Grayscale", input_gray);

output = meanfilter(input_gray, 3, "mirroring");
//Boundary process: zero-paddle, mirroring, adjustkernel

namedWindow("Mean Filter", WINDOW_AUTOSIZE);
imshow("Mean Filter", output);
```

namedWindow(), imshow() 함수를 사용해 원본 grayscale 이미지와 mean filtering한 결과 이미지를 개별 창에 출력하며, 그 과정에서 meanfilter() 함수를 호출해 mean filtering을 적용한 것을 output에 저장한다. 위 코드에서는 n=3을 전달하고 있으므로, 커널 사이즈는 2*3+1=7, kernel 행렬의 크기는 7*7=49이다.

2. meanfilter(const Mat input, int n, const char* opt)

meanfilter() 함수는 전달받은 인자들로 uniform mean filtering을 수행하며, 픽셀이 이미지 경계에 위치할 경우 mirroring, zero padding, kernel adjusting의 방법으로 처리한다.

2.1 함수 정의

```
Mat kernel;

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

input의 행과 열 크기를 row, col 변수에 저장하고, 커널의 크기 kernel_size는 2*n+1로 설정한다. tempa와 tempb 변수는 이미지 경계에 위치한 픽셀을 처리할 때 인덱스 계산에 사용한다.

```
kernel = Mat::ones(kernel_size, kernel_size, CV_32F) / (float)(kernel_size * kernel_size);

float kernelvalue=kernel.at<float>(0, 0);
```

Mat::ones() 함수를 사용해 모든 원소가 1로 초기화된 커널을 생성한 후, 전체 요소 개수로 나누어 평균값을 적용한다. 이후 kernelvalue 변수를 kernel의 (0, 0) 값으로 초기화한다. uniform mean filtering에서 커널 내의 모든 원소는 동일한 값을 가지므로 kernelvalue 변수를 kernel(0, 0) 값으로 초기화해도 무방하며, 이 값은 이후 커널 내 이미지의 픽셀 값에 곱해진다.

```
Mat output = Mat::zeros(row, col, input.type());
```
결과 이미지를 저장하기 위해 크기가 input과 동일한 Mat형 output 객체를 생성하고 0으로 초기화한다.

```
for (int i = 0; i < row; i++) { //for each pixel in the output
       for (int j = 0; j < col; j++) {
         …
```
이후 이중 for문을 순회하며 이미지의 모든 픽셀 (i, j) 에 대해 uniform mean filtering한다.

2.2 경계 픽셀 처리

zero padding, mirroring, adjust kernel 3개의 옵션을 두어 경계 픽셀을 처리한다.

2.2.1 zero padding

```
if (!strcmp(opt, "zero-paddle")) {
     …
}
```
zero padding은 이미지 경계 바깥의 픽셀 값들을 0으로 처리한다.

```
float sum1 = 0.0;
for (int a = -n; a <= n; a++) { // for each kernel window
     for (int b = -n; b <= n; b++) {
              if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) &&
              (j + b >= 0)) { //if the pixel is not a border pixel
                       sum1 += kernelvalue*(float)(input.at<G>(i + a, j + b));
              }
     }
}
```
윈도우 내 유효한 픽셀들의 합을 저장할 float형 sum1 변수를 선언하고, 2중 for루프를 반복하며 커널 내 유효한 픽셀들에 대해 연산한 결과 합을 저장할 float형 sum1 변수를 선언하고 0으로 초기화한다.

각 for문의 반복 범위를 -n부터 n까지로 설정해 수직 방향과 수평 방향으로 각각 (2n+1) 픽셀만큼 순회하므로 커널 내의 모든 픽셀을 탐색할 수 있으며, if문을 사용해 전체 이미지의 크기 내 범위에서만 반복하도록 하여 valid한 픽셀에 대해서만 (커널 값)*(이미지 픽셀 값) 계산을 수행한다. 이후 sum1 변수에 계산한 결과를 더하고 다음 반복으로 넘어간다.

```
output.at<G>(i, j) = (G)sum1;
```
이중 for문이 종료되면 sum1은 (i, j) 픽셀에 매핑될 커널의 산술 평균 값을 가지게 된다. 결과 이미지의 (i, j) 픽셀에 sum1 값을 저장한 후, 다음 (i, j)에 대해 동일한 과정을 반복한다.

2.2.2 mirroring

```
else if (!strcmp(opt, "mirroring")) {
     float sum1 = 0.0;
     for (int a = -n; a <= n; a++) { // for each kernel window
              for (int b = -n; b <= n; b++) {
                       …
```
mirroring은 커널 내 이미지 범위를 초과한 값이 존재할 때, 해당 픽셀에 (i, j) 주변의 다른 픽셀 값을 "mirroring"해 할당한다. 2.2.1과 동일하게 float sum1 = 0.0으로 선언하고, 수직 및 수평 방향으로 -n부터 n까지 반복하며 커널 내 모든 픽셀들에 대해 연산한다.

2.2.2.1 수직 방향

```
if (i + a > row - 1) {
 tempa = i - a;
}
```

a가 양수이고 현재 픽셀 (i, j)에서 a만큼 내려가면 이미지의 아래 경계를 벗어나는 경우 위쪽으로 대칭된 위치의 픽셀인 (i-a, j)를 사용하므로 이를 tempa에 저장한다.

```
else if (i + a < 0) {
 tempa = -(i + a);
}
```

a가 음수이고 (i, j)에서 a만큼 올라가면 이미지의 위쪽 경계를 벗어나는 경우, 아래쪽으로 대칭된 위치의 픽셀인 (-(i+a), j)를 사용한다.

```
else {
 tempa = i + a;
}
```

픽셀이 이미지 범위 내에 있을 경우 i+a를 그대로 사용하므로 tempa=i+a이다.

2.2.2.2 수평 방향

```
if (j + b > col - 1) {
 tempb = j - b;
}
```

b가 양수이고 (i, j)에서 b만큼 오른쪽으로 이동하면 이미지의 우측 경계를 벗어나는 경우, 왼쪽으로 대칭된 픽셀인 (i, j-b)를 사용하므로 tempb=j-b이다.

```
else if (j + b < 0) {
 tempb = -(j + b);
}
```

b가 음수이고 (i, j)에서 b만큼 왼쪽으로 이동하면 이미지의 좌측 경계를 초과하므로, 오른쪽으로 대칭된 픽셀인 (i, -(j+b))를 사용하고 tempb=-(j+b)이다.

```
else {
 tempb = j + b;
}
```

픽셀이 이미지 범위 내에 있을 경우 j+b를 그대로 사용하므로 tempb=j+b이다.

```
sum1 += kernelvalue*(float)(input.at<G>(tempa, tempb));
```

tempa와 tempb 각각에 i, j값이 저장되므로, (커널 값)*(이미지 픽셀 값) 계산은 kernelvalue*input(tempa, tempb)가 된다. 이를 sum1에 더하고 다음 반복으로 이동한다.

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

이중 for문이 종료되면 sum1은 (i, j) 픽셀에 매핑될 커널의 산술 평균 값을 가지게 된다. 결과 이미지의 (i, j) 픽셀에 sum1 값을 저장한 후, 다음 (i, j)에 대해 동일한 과정을 반복한다.

2.2.3 adjust kernel

```
else if (!strcmp(opt, "adjustkernel")) {
…
}
```

adjustkernel 옵션은 커널에 대해 실제로 존재하는 픽셀만 고려해 평균을 구한다.

```
float sum1 = 0.0;
float sum2 = 0.0;
```

filtering을 거친 픽셀 (i, j)값이 $O(i,j) = \frac{\sum_{s=-1}^{1}\sum_{t=-1}^{1} w'(s,t)I'(i+s,j+t)}{\sum_{s=-1}^{1}\sum_{t=-1}^{1} w'(s,t)}$ 이므로, 실제 픽셀 값의 합을 저장할 sum1, 사용된 커널 값의 합을 저장할 sum2를 선언하고 각각 0으로 초기화한다.

```
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)) {
                    sum1 += kernelvalue*(float)(input.at<G>(i + a, j + b));
                    sum2 += kernelvalue;
            }
    }
}
```

마찬가지로 2중 for문을 통해 -n부터 n까지 수평, 수직 방향으로 탐색하며, 커널 내의 픽셀 (i+a, j+b)가 이미지 범위 내에 있을 경우 sum1에는 (커널 값)*(이미지 픽셀 값)한 결과를, sum2에는 커널 값을 더한다.

```
output.at<G>(i, j) = (G)(sum1/sum2);
```

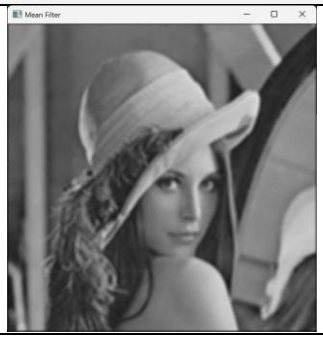
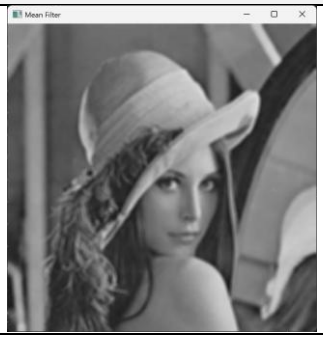
2중 for문이 종료되면 결과 이미지의 (i, j) 픽셀에 sum1/sum2 값을 저장한 후, 다음 픽셀에 대해 동일한 과정을 반복한다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다.

3.1 boundary 처리 방식 별 비교

output = meanfilter(input_gray, 3, "mirroring"), output = meanfilter(input_gray, 3, "zero-paddle"), output=meanfilter(input_gray, 3, "adjustkernel"); 을 각각 호출하여 실행시켰다.

original	zero padding	mirroring	adjust kernel
			

모든 경우 uniform mean filtering을 거쳐 흐릿해진 결과 이미지가 출력되었으며, zero padding의 경우 boundary 바깥의 픽셀을 0으로 채우므로 valid한 픽셀들로만 계산한 것보다 실제 적용되는 평균값이 낮아진다. 따라서 zero padding 결과 이미지의 모서리 부근이 어두워진 것을 확인할 수 있으며, mirroring과 adjust kernel 옵션에서는 이러한 효과가 관찰되지 않는다.

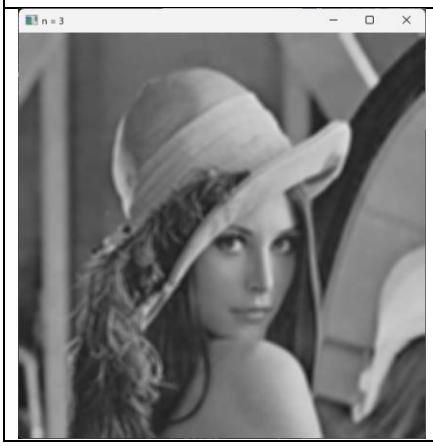
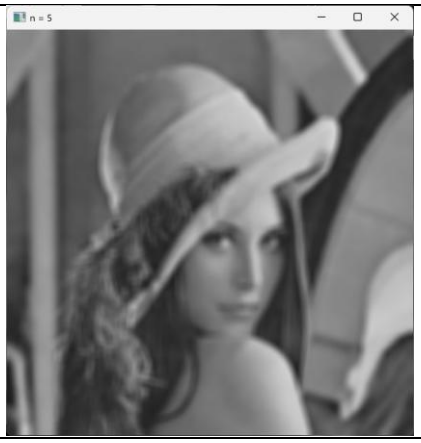
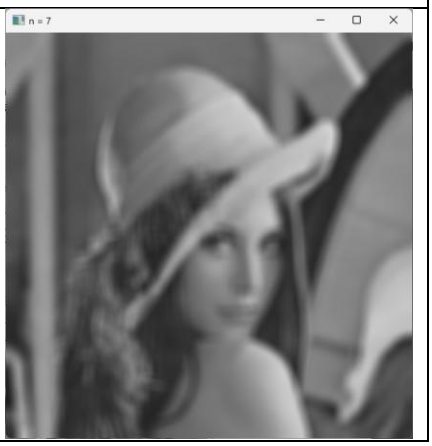
3.2 커널 크기별 결과 비교

동일한 입력 이미지와 동일한 옵션(=mirroring)으로, n을 3, 5, 7로 증가시키며 3번 실행시켜 각각의 실행 시간과 결과 이미지를 비교하였으며, 이를 위해 코드 일부를 아래와 같이 수정하였다.

n = 3	n = 5	n = 7

`double start = (double)getTickCount();`
 	
`Mat output1 = meanfilter(input_gray, 3, "mirroring");`
 	
`double end = (double)getTickCount();`
	`start = (double)getTickCount();`
	
	`Mat output2 = meanfilter(input_gray, 5, "mirroring");`
	
	`end = (double)getTickCount();`
	`time = (end - start) / getTickFrequency();`
		`start = (double)getTickCount();`
		
		`Mat output3 = meanfilter(input_gray, 7, "mirroring");`
		
		`end = (double)getTickCount();`
		`time = (end - start) / getTickFrequency();`

double time = (end - start) / getTickFrequency();
 	
std::cout << "n = 3: " << time << " seconds" << std::endl;
 	
namedWindow("n = 3", WINDOW_AUTOSIZE);
imshow("n = 3", output1);
	std::cout << "n = 5: " << time << " seconds" << std::endl;
	
	namedWindow("n = 5", WINDOW_AUTOSIZE);
	imshow("n = 5", output2);
		std::cout << "n = 7: " << time << " seconds" << std::endl;
		
		namedWindow("n = 7", WINDOW_AUTOSIZE);
		imshow("n = 7", output3);

실행 시간: 0.538554 초	실행 시간: 1.39743초	실행 시간: 2.32752초
		

n이 커질수록 실행 시간이 늘어나며, 결과 이미지가 더욱 blurry해지는 것을 확인할 수 있다. n이 증가하며 커널 크기가 커지면 하나의 픽셀을 계산하기 위해 더 넓은 영역의 픽셀 값들을 평균하게 되고, 이 경우 세부 정보들이 손실되어 이미지가 부드럽고 흐릿하게 보인다.

MeanFilterRGB.cpp

MeanFilterRGB.cpp는 이미지에 uniform mean filtering을 적용하며, 경계값 처리 옵션으로 mirroring, zero padding, adjust kernel을 제공하는 프로그램이다. 전체적인 코드 흐름은 MeanFilterGray.cpp와 유사하나 입력으로 컬러 이미지를 사용한다.

1. main()

```
int main() {

    Mat input = imread("lena.jpg", IMREAD_COLOR);     // IMREAD_COLOR
    Mat output;


    if (!input.data)
    {
            std::cout << "Could not open" << std::endl;
            return -1;
    }

    namedWindow("Original", WINDOW_AUTOSIZE);
    imshow("Original", input);
    output = meanfilter(input, 3, "zero-paddle");

    namedWindow("Mean Filter", WINDOW_AUTOSIZE);
    imshow("Mean Filter", output);

    waitKey(0);

    return 0;
}
```

main함수는 MeanFilterGray.cpp와 동일하다. input을 input_gray로 변환했던 부분은 제거되었다.

2. meanfilter(const Mat input, int n, const char* opt)

meanfilter() 함수는 전달받은 인자들로 uniform mean filtering을 수행하며, 픽셀이 이미지 경계에 위치할 경우 mirroring, zero padding, kernel adjusting의 방법으로 처리한다. MeanFitlerGray.cpp의 meanfilter() 함수와 전반적으로 유사하며, 색상 채널(R, G, B)별로 별도 처리하는 부분이 포함되었다.

2.1 함수 정의

```
Mat kernel;

int row = input.rows;
int col = input.cols;
int kernel_size = (2 * n + 1);
int tempa;
int tempb;
// Initialiazing Kernel Matrix
kernel = Mat::ones(kernel_size, kernel_size, CV_32F) / (float)(kernel_size *
kernel_size);
float kernelvalue = kernel.at<float>(0, 0);

Mat output = Mat::zeros(row, col, input.type());
```

MeanFilterGray.cpp의 2.1.과 동일하다.

2.2 경계 픽셀 처리

zero padding, mirroring, adjust kernel 별 처리 메커니즘은 동일하지만, 각 옵션 안에 R, G, B 채널을 분리해 각각 처리하는 부분이 포함되어 있다.

```
if (!strcmp(opt, "zero-paddle")) {
	float sum1_r = 0.0;
	float sum1_g = 0.0;
	float sum1_b = 0.0;
	for (int a = -n; a <= n; a++) { // for each kernel window
		for (int b = -n; b <= n; b++) {
			…
```

MeanFilterGray.cpp 코드는 (커널 값)*(이미지 픽셀 값) 총합 변수로 sum1 하나만을 사용했지만, MeanFilterRGB.cpp는 컬러 이미지를 다루고 있으므로 R, G, B 각각의 총합을 분리하여 저장하기 위해 sum1_r, sum1_g, sum1_b 3개의 변수를 선언하고 0으로 초기화한다.

```
for (int a = -n; a <= n; a++) { // for each kernel window
	for (int b = -n; b <= n; b++) {
		if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >=
			0)) { //if the pixel is not a border pixel
			sum1_r += kernelvalue*(float)(input.at<C>(i + a, j + b)[0]);
			sum1_g += kernelvalue*(float)(input.at<C>(i + a, j + b)[1]);
			sum1_b += kernelvalue*(float)(input.at<C>(i + a, j + b)[2]);
		}
	}
}
```

-n부터 n까지 수평, 수직 방향으로 탐색하는 것은 동일하지만, input.at<C>(i, j)[0], input.at<C>(i, j)[1], input.at<C>(i, j)[2]와 같이 input 이미지에 접근할 때 채널을 분리한다.

```
output.at<C>(i, j)[0] = (G)sum1_r;
output.at<C>(i, j)[1] = (G)sum1_g;
output.at<C>(i, j)[2] = (G)sum1_b;
```

output 이미지에 접근할 때도 마찬가지로 채널을 분리하여, ouptut.at<C>(i, j)[0]에는 sum1_r값을, ouptut.at<C>(i, j)[1]에는 sum1_g값을, ouptut.at<C>(i, j)[2]에는 sum1_b값을 매핑한다.

zero padding만 설명하였으나, mirroring과 adjustkernel 옵션 또한 동일한 방식으로 커널 총합을 R, G, B에 대해 각각 계산하며 input, output 이미지에 접근할 때는 [0], [1], [2]로 채널 별 할당한다.

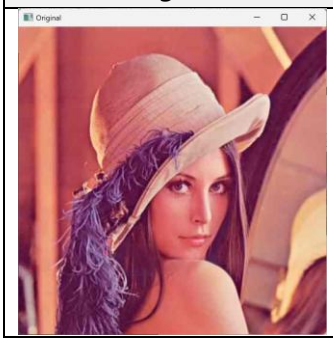
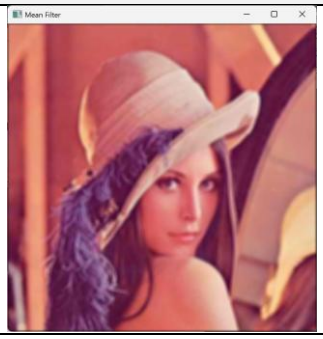
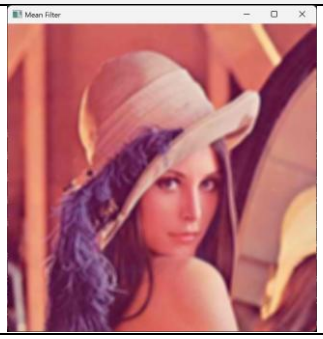
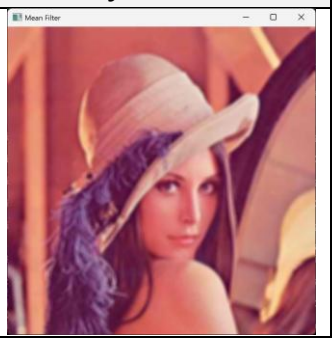
```
float sum1_r = 0.0;
float sum1_g = 0.0;
float sum1_b = 0.0;
float sum2 = 0.0;
for (int a = -n; a <= n; a++) { // for each kernel window
	for (int b = -n; b <= n; b++) {
		if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >=
			0)) {
			sum1_r += kernelvalue*(float)(input.at<C>(i + a, j + b)[0]);
			sum1_g += kernelvalue*(float)(input.at<C>(i + a, j + b)[1]);
			sum1_b += kernelvalue*(float)(input.at<C>(i + a, j + b)[2]);
			sum2 += kernelvalue;
```

다만 adjustkernel 옵션의 경우 sum1은 RGB 별 3개를 사용하지만, sum2는 사용된 픽셀의 수를 저장하는 변수이므로 채널을 분리할 필요 없이 하나로 사용한다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다.

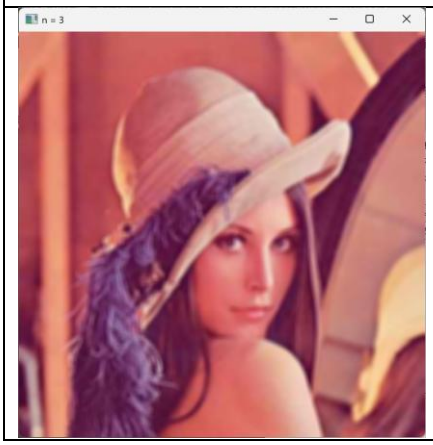
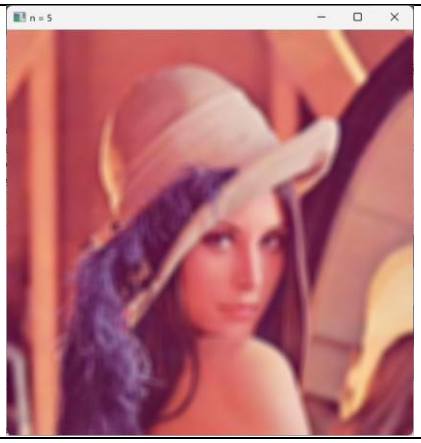
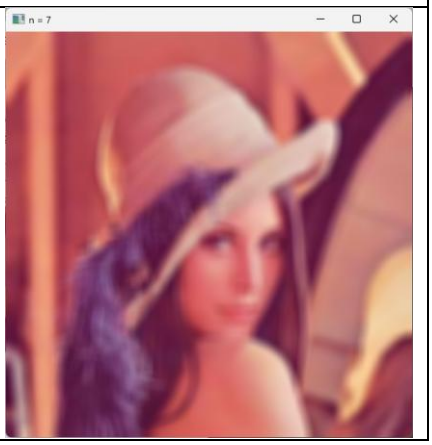
3.1 boundary 처리 방식별 비교

original	zero padding	mirroring	adjust kernel
			

모든 경우 uniform mean filtering을 거쳐 흐릿해진 결과 이미지가 출력되었으며, zero padding의 경우 boundary 바깥의 픽셀을 0으로 채우므로 valid한 픽셀들로만 계산한 것보다 실제 적용되는 평균값이 낮아진다. 따라서 zero padding 결과 이미지의 모서리 부근이 어두워진 것을 확인할 수 있으며, mirroring과 adjust kernel 옵션에서는 이러한 효과가 관찰되지 않는다.

3.2 커널 크기별 비교

동일한 입력 이미지와 동일한 옵션(=mirroring)으로, n을 3, 5, 7로 증가시키며 3번 실행시켜 각각의 실행 시간과 결과 이미지를 비교하였으며, MeanFilterGray.cpp에서 수정한 것과 동일하게 코드 일부를 수정하였다.

n = 3	n = 5	n = 7
실행 시간: 2.1809 초	실행 시간: 4.40485초	실행 시간: 8.33777초
		

MeanFilterGray.cpp와 마찬가지로 n이 커질수록 더 많은 세부 정보가 손실되어 이미지가 더욱 blurry해진다. 또, MeanFilterGray.cpp에서 동일하게 실행시켰을 때와 비교해 전반적인 코드 수행 시간이 증가하였는데, 컬러 이미지 처리 시 R, G, B에 대해 모두 연산하므로 연산량이 증가하여 MeanFilterGray.cpp에서 n=3, n=5, n=7일 때의 실행 시간이 각각 약 0.5초, 약 1.4초, 약 2.3초였던 것에 비해 약 2초, 약 4초, 약 8초가 소요되었다.

GaussianGray.cpp

GaussianGray.cpp는 이미지에 Gaussian filtering을 적용하며, 경계값 처리 옵션으로 mirroring, zero padding, adjust kernel을 제공하는 프로그램이다.

Uniform mean filtering은 커널 내의 픽셀들을 단순 산술 평균한 값을 매핑하므로, 레퍼런스 픽셀 (i, j)와 커널 픽셀의 거리로 인한 연관성을 반영하지 못한다. GaussianGray.cpp는 이미지에 gaussian filtering을 적용하는 프로그램으로, 레퍼런스 픽셀의 값을 주변 픽셀들의 가중 평균 값으로 변경하여, 즉 픽셀 간 거리를 고려해 가중치를 부여함으로써 보다 자연스러운 filtering이 가능하게 한다. Gaussian filtering의 공식은 아래와 같으며

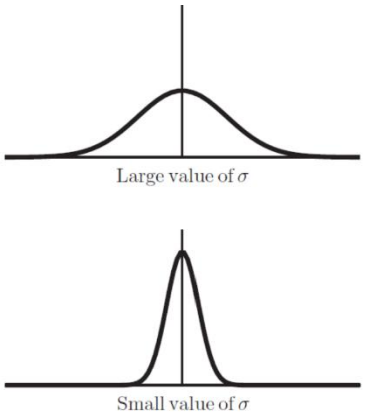


Figure 1

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

2차원 Gaussian 함수 $G(x) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{x^2+y^2}{2\sigma_x\sigma_y}}$이므로

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

이다. Gaussian 함수의 그래프는 (그림1)과 같이 좌우 대칭인 종 모양 곡선으로, |표준편차|가 증가할수록 값이 급격하게 감소하는 형태이다. 중심에서 멀어질수록 함수 값이 감소하므로 레퍼런스 픽셀과 가까운 픽셀에 더 큰 가중치를 부여하며, 표준편차(σ) 값이 커질수록 이미지가 더욱 blurry해진다.

1. main()

main 함수는 MeanFilterGray.cpp의 main 함수와 동일하다. 이미지를 읽어 input에 저장하고, grayscale로 변환해 input_gray에 저장한다. 이후 gaussianfilter() 함수를 호출해 Gaussian filtering을 적용한다.

2. gaussianfilter(const Mat input, int n, float sigmaT, float sigmaS, const char* opt)

gaussianfilter 함수는 입력 이미지 input, 커널 크기 변수 n, 수평 방향 표준편차 sigmaT, 수직 방향 sigmaS, 경계값 처리 옵션 opt를 전달받아 input에 Gaussian filter를 적용한다. 커널 계산 방식이 변경되고 커널 normalization 코드가 추가된 것을 제외하면, 전체적인 흐름과 옵션별 경계 처리 방식은 uniform mean filtering과 동일하다.

2.1 함수 정의

```
Mat kernel;

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

// Initialiazing Kernel Matrix
kernel = Mat::zeros(kernel_size, kernel_size, CV_32F);
```

MeanFilterGray.cpp와 MeanFilterRGB.cpp의 함수 정의와 유사하지만, Gaussian filtering에서는 커널 값이 uniform하지 않으므로 별도의 kernelvalue 변수를 사용하지 않으며, normalization을 위한 float형 denom 변수를 선언한다. $w(s,t) = \frac{1}{\sum_{m=-a}^{a}\sum_{n=-b}^{b}\exp\left(-\frac{m^2}{2\sigma_s^2}-\frac{n^2}{2\sigma_t^2}\right)}\exp\left(-\frac{s^2}{2\sigma_s^2}-\frac{t^2}{2\sigma_t^2}\right)$의 분모가 denom에 저장된다.

2.2 커널 계산 및 normalization

```
denom = 0.0;
for (int a = -n; a <= n; a++) {  // Denominator in m(s,t)
	for (int b = -n; b <= n; b++) {
		float value1 = exp(-(pow(a, 2) / (2 * pow(sigmaS, 2))) -
		(pow(b, 2) / (2 * pow(sigmaT,2))));
		kernel.at<float>(a+n, b+n) = value1;
		denom += value1;
	}
}
```

denom 변수를 0으로 초기화하고, -n부터 n까지 수직, 수평 방향으로 커널 내의 모든 픽셀을 순회하며 커널 값을 계산한다. 전달받은 수평 방향 표준편차 sigmaT와 수직 방향 표준편차 sigmaS를 사용해 각 커널 픽셀에 대해 $\exp\left(-\frac{a^2}{2\sigma_s^2}-\frac{b^2}{2\sigma_t^2}\right)$를 적용하는데, 이때 a와 b는 각각 커널 내의 수직, 수평 오프셋이다. 계산한 커널 원소 값 value1을 kernel에 저장하며, kernel에 접근할 때는 a, b가 아닌 (a+n), (b+n) 인덱스로 접근함에 유의해야 한다. 이후 value1을 denom에 누적하고 다음 반복으로 넘어간다.

```
for (int a = -n; a <= n; a++) {  // Denominator in m(s,t)
	for (int b = -n; b <= n; b++) {
		kernel.at<float>(a+n, b+n) /= denom;
	}
}
```

커널 값의 합이 1이 되어야 하므로 계산한 kernel을 normalize하며, 마찬가지로 -n부터 n까지 수직, 수평 방향으로 커널 내의 모든 픽셀을 탐색하며 (a+n, b+n) 위치의 커널 원소를 denom으로 나누어 계산한다.

2.3 경계 픽셀 처리

옵션 별 인덱싱 등 구현 방법은 동일하지만, MeanFilterGray.cpp 코드에서 레퍼런스 픽셀 계산을 위해 sum1 += kernelvalue * (float)(input.at<G>(i + a, j + b));와 같이 (uniform한 커널 값)*(이미지 픽셀 값)으로 계산한 것과 달리 sum1 += kernel.at<float>(a + n, b + n) * input.at<G>(i + a, j + b);와 같이 각 픽셀의 커널 값을 사용한다.

```
if (!strcmp(opt, "zero-paddle")) {
	float sum1 = 0.0;
	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)) {
				sum1 += kernel.at<float>(a + n, b + n) *
				input.at<G>(i + a, j + b);
			}
		}
	}
	output.at<G>(i, j) = (G)sum1;
}
```

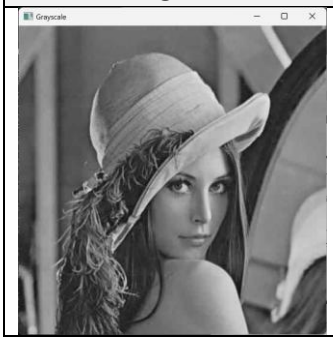
zero padding 옵션의 경우 위와 같으며, sum1에 (커널*이미지 값) 결과를 합하는 코드를 제외하면 uniform mean filtering의 코드와 동일한 것을 확인할 수 있다. 다른 옵션들도 마찬가지로 sum1을 계산하는 부분 외에는 동일한 메커니즘을 가진다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다.

3.1 boundary 처리 방식별 비교

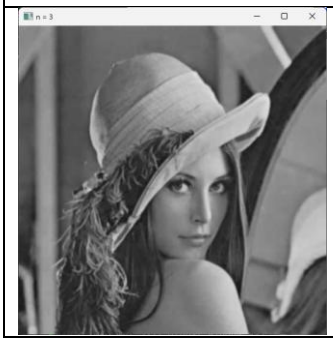
input=input_gray, n=1, sigmaT=1, sigmaS=1로 각각 호출하였다.

original	zero padding	mirroring	adjust kernel
			

모든 옵션에서 원본 이미지와 결과 이미지가 정상적으로 출력되었으며, zero paddle 옵션의 경우 mean filtering에서와 마찬가지로 모서리 부근의 픽셀들이 어두워진 것을 확인할 수 있다.

3.2 커널 크기별 비교

동일한 입력 이미지와 동일한 옵션(=mirroring)으로, n을 3, 5, 7로 증가시키며 3번 실행시켜 각각의 실행 시간과 결과 이미지를 비교하였다. MeanFilterGray.cpp에서 수정한 것과 동일하게 코드 일부를 수정하였다.

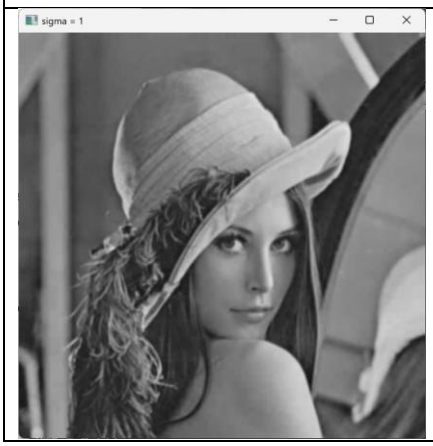
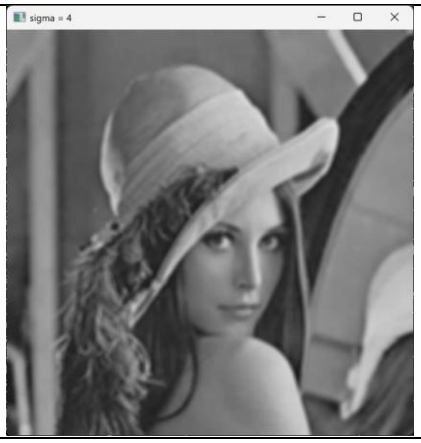
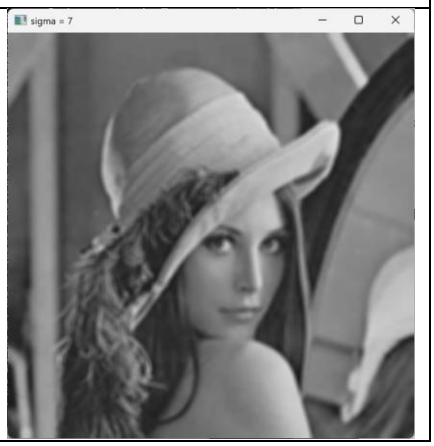
n=3	n=5	n=7	n=20
실행 시간: 1.20483초	실행 시간: 2.56626초	실행 시간: 4.56121초	실행 시간: 38.4847초
			

n이 커질수록 실행 시간이 증가하지만 mean filtering에서 동일하게 실험했을 때보다 커널 크기에 따라 blurry해지는 정도가 적다. n=20으로 극단적으로 증가시켜 실험했을 때도 육안으로 유의미한 차이는 드러나지 않는데, 그 이유는 다음과 같다.

Gaussian 함수는 중심에서 최댓값을 가지며 지수 함수의 특성을 가진다. 따라서 중심에서 멀어질수록, 즉 레퍼런스 픽셀 (i, j)에서 멀리 떨어진 픽셀일수록 filtering 효과에 기여하는 정도가 급격히 줄어든다. n을 증가시켜도 중심 픽셀에 가까운 픽셀들만 blurring에 큰 영향을 미치므로 커진 커널이 모두 평균값 계산에 균등한 정도로 포함되던 mean filtering보다 n의 크기가 blurring에 미치는 영향이 적으며, 이 때문에 n의 증가에 따른 blurring 효과가 미미하다.

3.3 표준편차 크기별 비교

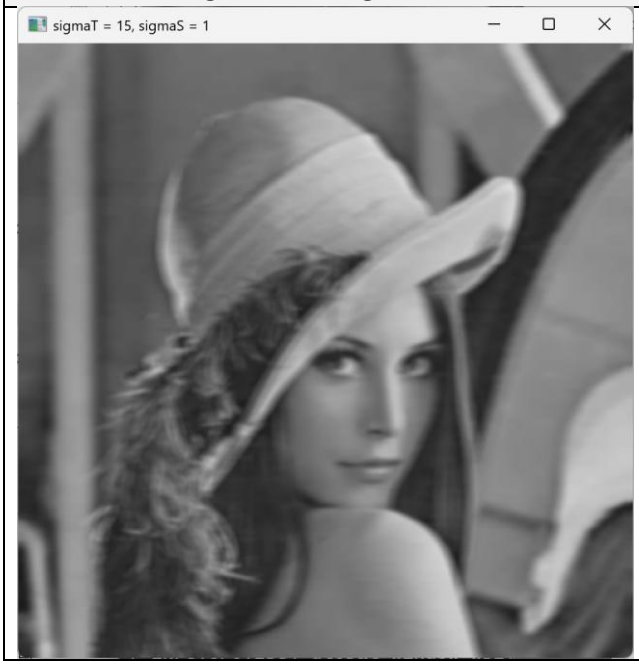
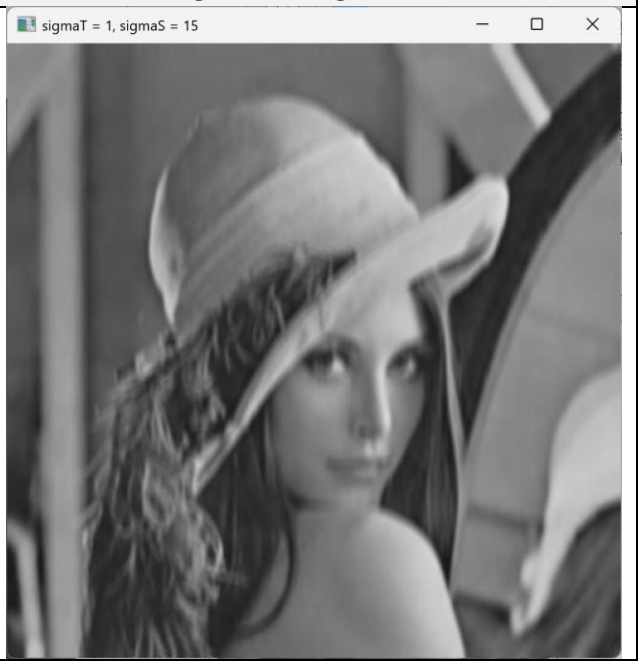
n=3, opt="mirroring"으로 통일하여 sigmaT와 sigmaS를 1, 4, 7로 증가시키며 각각의 실행 시간과 결과 이미지를 비교하였다.

sigma = 1	sigma = 4	sigma = 7
실행 시간: 0.869661 초	실행 시간: 0.985839 초	실행 시간: 0.89725 초
		

실행 시간에는 큰 차이가 없지만, 표준편차가 증가하며 결과 이미지가 blurry해지는 것을 확인할 수 있다. (그림1)과 같이 표준편차가 커질수록 Gaussian 함수 그래프의 경사가 완만해지는데, 이는 레퍼런스 픽셀과 멀리 떨어진 지점의 커널 값이 작아지는 정도가 줄어듦을 뜻한다. 즉, 표준편차가 클수록 레퍼런스 픽셀과 먼 픽셀이 결과 이미지에 영향을 더 많이 미칠 수 있게 되며, 멀리 떨어진 지점까지 blurring된다.

3.4 sigmaT, sigmaS를 anisotropic하게 전달할 경우

수직 방향과 수평 방향 표준편차가 다른 anisotropic한 케이스를 테스트하기 위해 sigmaT=15, sigmaS=1인 경우와 sigmaT=1, sigmaS=15인 경우에 대해 결과 이미지를 비교하였으며, 이때 n=5, opt="mirroring"으로 통일하였다.

sigmaT=15, sigmaS=1	sigmaT=1, sigmaS=15
	

3.3.까지는 sigmaT와 sigmaS에 동일한 인자를 전달해 isotropic한 필터링을 수행했지만, 수직 방향과 수평 방향 표준편차를 다르게 설정하면 blur 효과가 한쪽 방향으로 강하게 나타난다. 수평 방향 표준편차인 sigmaT를 더 크게 하면 가로 방향으로 가중치 분포가 넓게 퍼져 가로로 blur되며, 수직 방향 표준편차인 sigmaS를 더 크게 하면 세로로 blur되는 것을 확인할 수 있다.

GaussianRGB.cpp

GaussianRGB.cpp는 컬러 이미지에 Gaussian filtering을 적용하는 프로그램이다. RGB 각 채널에 대해 따로 접근하고 연산하는 부분을 제외하면 함수 정의, boundary 처리 등 대부분의 코드가 GaussianGray.cpp와 동일하게 작동한다.

1. **main()**

```
Mat input = imread("lena.jpg", IMREAD_COLOR); // IMREAD_COLOR
Mat output;
…
namedWindow("Original", WINDOW_AUTOSIZE);
imshow("Original", input);
output = gaussianfilter(input, 1, 1, 1, "zero-paddle");
```

이미지를 input에 저장하며, grayscale로 바꾸지 않고 바로 사용한다.

2. **gaussianfilter(const Mat input, int n, float sigmaT, float sigmaS, const char* opt)**

함수 정의, normalization은 GaussianGray.cpp와 동일하다. zero padding, mirroring, adjust kernel 옵션은 모두 GaussianGray.cpp와 동일한 흐름으로 수행되며, 각 옵션 안에서 R, G, B 채널을 분리해 각각 처리하는 부분이 추가되었다.

```
if (!strcmp(opt, "zero-paddle")) {
    float sum1_r = 0.0;
    float sum1_g= 0.0;
    float sum1_b = 0.0;

    for (int a = -n; a <= n; a++) {
            for (int b = -n; b <= n; b++) {
                    …
```

GaussianGray.cpp 코드는 sum1 하나만을 사용했지만, GaussianRGB.cpp에서는 각 채널을 분리해 계산하므로 sum1_r, sum1_g, sum1_b 3개의 변수를 선언하고 0으로 초기화한다.

```
if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
    // sum1 += kernel.at<float>(a + n, b + n) * input.at<G>(i + a, j + b);
    sum1_r += kernel.at<float>(a + n, b + n) * (float)(input.at<C>(i + a, j + b)[0]);
    sum1_g += kernel.at<float>(a + n, b + n) * (float)(input.at<C>(i + a, j + b)[1]);
    sum1_b += kernel.at<float>(a + n, b + n) * (float)(input.at<C>(i + a, j + b)[2]);
}
```

```
output.at<C>(i, j)[0] = (G)sum1_r;
output.at<C>(i, j)[1] = (G)sum1_g;
output.at<C>(i, j)[2] = (G)sum1_b;
```

-n부터 n까지 수직, 수평 방향으로 탐색하는 과정은 동일하며, sum1 += kernel.at<float>(a + n, b + n) * input.at<C>(i + a, j + b);으로 작성했던 것을 sum1_r += kernel.at<float>(a + n, b + n) * (float)(input.at<C>(i + a, j + b)[0]);과 같이 채널 별 분리해 input에 접근하는 것으로 수정하였다. zero padding 이외의 다른 옵션들도 마찬가지로 R, G, B에 대해 각각 연산하며, input과 output에 접근할 때는 [0], [1], [2]로 채널 별 할당한다.

```
else if (!strcmp(opt, "adjustkernel")) {
    float sum1_r = 0.0;
    float sum1_g = 0.0;
    float sum1_b = 0.0;
    float sum2 = 0.0;
    …
```

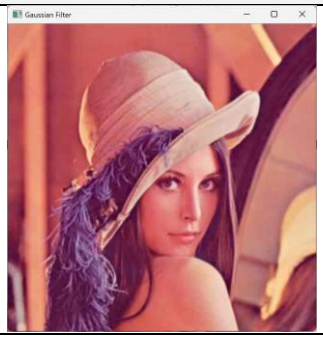
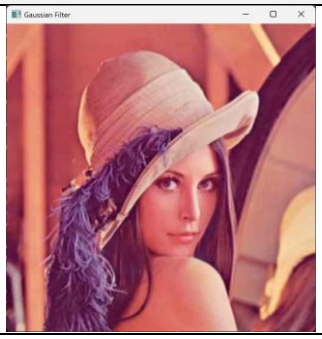
adjustkernel 옵션의 경우 MeanFilterRGB.cpp와 마찬가지로 sum1 변수는 RGB별 3개, sum2 변수는 전체 1개를 사용한다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다. 컬러 이미지를 사용하고 이로 인해 전체적인 실행 시간이 증가한 것을 제외하면 GaussianGray.cpp와 실행 결과가 동일하다.

3.1 boundary 처리 방식별 비교

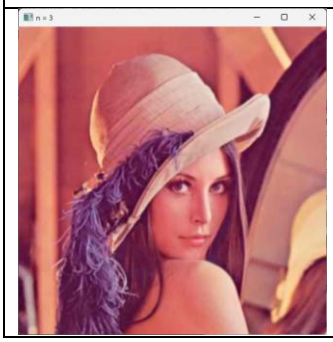
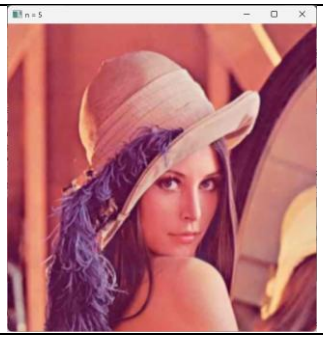
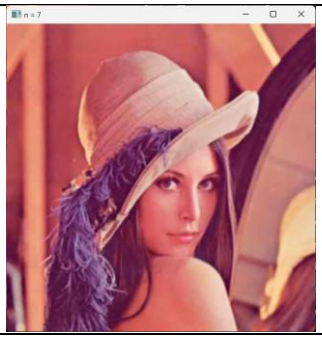
input=input, n=1, sigmaT=1, sigmaS=1로 각각 호출하였다.

original	zero padding	mirroring	adjust kernel
			

모든 옵션에서 원본 이미지와 결과 이미지가 정상적으로 출력되었으며, zero paddle 옵션의 경우 모서리 부근의 픽셀들이 어두워진 것을 확인할 수 있다.

3.2 커널 크기별 비교

동일한 입력 이미지와 동일한 옵션(=mirroring)으로, n을 3, 5, 7로 증가시키며 3번 실행시켜 각각의 실행 시간과 결과 이미지를 비교하였다.

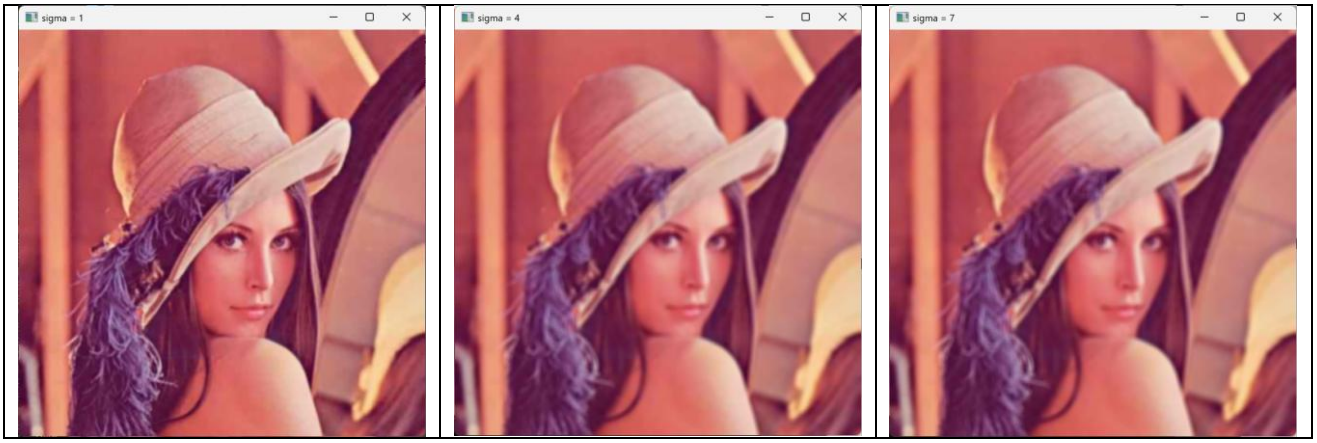
n=3	n=5	n=7	n=20
실행 시간: 7.44536초	실행 시간: 7.44536초	실행 시간: 15.4009초	실행 시간: 121.508초
			

동일하게 n에 따른 blur 정도 차이가 크게 확인되지 않으며, RGB 각각에 대해 연산하므로 grayscale일 때에 비해 연산량과 수행 시간이 증가하였다. n=20일 때도 grayscale의 경우와 동일하게 다른 케이스와 유의미한 차이가 확인되지 않았다.

3.3 표준편차 크기별 비교

n=3, opt="mirroring"으로 통일하여 sigmaT와 sigmaS를 1, 4, 7로 증가시키며 각각의 실행 시간과 결과 이미지를 비교하였다.

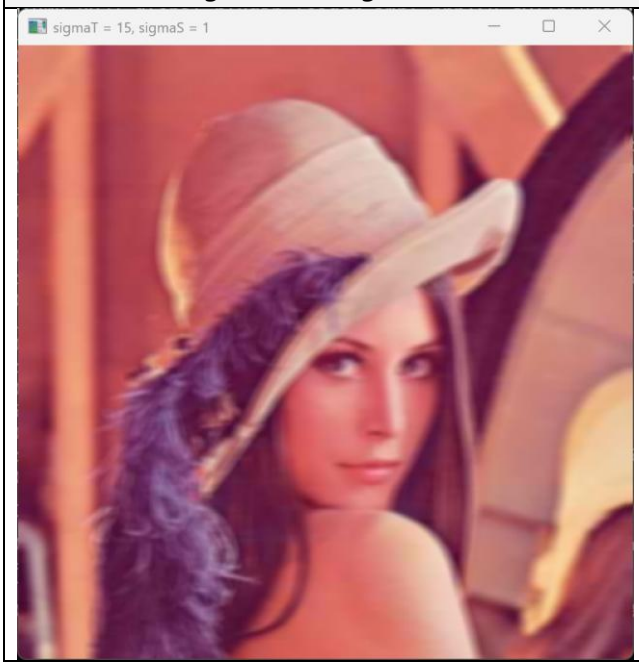
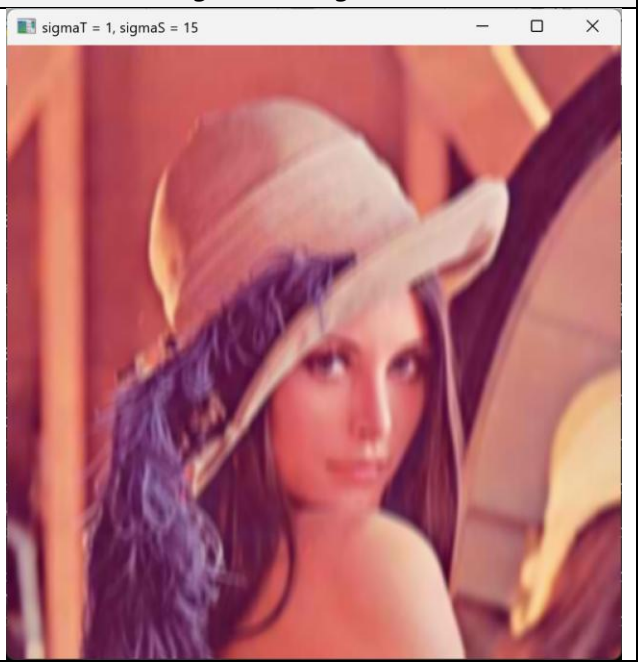
sigma = 1	sigma = 4	sigma = 7
실행 시간: 3.54066 초	실행 시간: 3.18652초	실행 시간: 3.24237초



표준편차가 커질수록 결과 이미지가 더 많이 blurry하다.

3.4 anisotropic한 경우

sigmaT=15, sigmaS=1과 sigmaT=1, sigmaS=15인 경우를 각각 호출해 테스트하며, 이때 n=5, opt="mirroring"으로 통일하였다.

sigmaT=15, sigmaS=1	sigmaT=1, sigmaS=15
	

수평 방향 표준편차인 sigmaT가 클 때에는 blur 효과가 가로로 나타났으며, 수직 방향 표준편차인 sigmaS가 클 때에는 세로로 blurring되었다.

SobelGray.cpp

SobelGray.cpp는 grayscale 이미지에 sobel 필터를 적용해 원본 이미지와 결과 이미지를 출력하며, 경계값 처리 방식으로는 mirroring만을 사용한다.

Sobel 필터는 이미지의 경계(edge)를 검출할 때 사용하는 필터이다. 이미지의 경계는 이미지의 밝기가 급격하게 변화하는 부분으로, 이미지를 함수 f(x, y)로 표현할 때 f의 1차 미분 값이 크게 나타나는 부분으로 정의할 수 있다.

$$\nabla I = (I_x, I_y) = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}) = (|S_x * I|, |S_y * I|)$$

(그림1)rhk 같이 3*3 크기 Sx와 Sy 2개의 마스크를 사용해 수평 방향과 수직 방향으로 미분 근사값을 계산한 다음, 아래와 같이 레퍼런스 픽셀 (i, j)에 대해 gradient magnitude를 구한다.

$$M(x, y) = \sqrt{I_x^2 + I_y^2}$$

S_x

−1	0	1
−2	0	2
−1	0	1

S_y

−1	−2	−1
0	0	0
1	2	1

Figure 1

기울기의 크기를 계산해야 하므로 위와 같이 제곱하여 크기만을 고려한다.

1. main()

```
int main() {
    Mat input = imread("lena.jpg", IMREAD_COLOR);
    Mat input_gray;
    Mat output;

    cvtColor(input, input_gray, COLOR_RGB2GRAY);

    if (!input.data)
    {
            std::cout << "Could not open" << std::endl;
            return -1;
    }

    namedWindow("Grayscale", WINDOW_AUTOSIZE);
    imshow("Grayscale", input_gray);
    output = sobelfilter(input_gray);

    namedWindow("Sobel Filter", WINDOW_AUTOSIZE);
    imshow("Sobel Filter", output);

    waitKey(0);
    return 0;
}
```

위의 설명들과 동일하다. 컬러로 읽어 온 "lena.jpg"에 대해 이미지가 유효한지 검사한 다음 grayscale로 변환해 input_gray에 저장하고, 이를 이용해 sobelfilter() 함수를 호출해 필터링을 수행한다.

2. Mat sobelfilter(const Mat input)

sobelfilter() 함수는 입력 이미지 input을 전달받아 sobel 필터 계산을 수행한다. 위의 프로그램들과 달리 경계처리 옵션은 mirroring으로 통일하며, mirroring 구현 방식은 이전에 서술한 것과 동일하다.

2.1 함수 정의

```
int row = input.rows;
int col = input.cols;
int n = 1; // Sobel Filter Kernel N

// Initialiazing 2 Kernel Matrix with 3x3 size for Sx and Sy
```

```
int Sx[3][3] = { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };
int Sy[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };

Mat output = Mat::zeros(row, col, input.type());

int tempa;
int tempb;
```

input의 행과 열을 각각 col과 row에 저장하고, 해당 크기의 Mat형 결과 이미지 객체 output을 선언한다. 변수 n에는 필터의 커널 크기가 저장되는데, 3*3 커널을 사용하고 있으므로 n=1로 초기화해 추후 -n=-1부터 n=1까지 탐색하며 계산한다. 수평 방향 커널 Sx와 Sy는 (그림1)의 커널과 동일하다.

2.2 경계 처리 및 커널 연산

```
for (int i = 0; i < row; i++) {
     for (int j = 0; j < col; j++) {
          float Xsum = 0.0, Ysum = 0.0;

          for (int a = -n; a <= n; a++) {
                   for (int b = -n; b <= n; b++) {
                            // Use mirroring boundary process
                            …
                            Xsum += Sx[a + 1][b + 1] * input.at<G>(tempa, tempb);
                            Ysum += Sy[a + 1][b + 1] * input.at<G>(tempa, tempb);
                   }
            }// end of for(-n~n)

            int M = sqrt(Xsum * Xsum + Ysum * Ysum);

            // 0~255
            if (M < 0) { M = 0; }
            if (M > 255) { M = 255; }

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

0부터 row까지, 0부터 col까지 2중 for문을 통해 모든 픽셀에 대해 계산하며, 매 내부 반복마다 수평 방향 커널 Sx를 적용한 부분합을 저장할 Xsum, 수직 방향 커널 Sy를 적용한 부분합을 저장할 Ysum을 선언하고 0으로 초기화한다.

레퍼런스 픽셀 (i, j) 에 대해 mirroring 방식으로 tempa와 tempb를 계산하고, 계산한 (tempa, tempb) 픽셀에 커널 값을 곱한 뒤 Sx를 곱한 것은 Xsum에, Sy를 곱한 것은 Ysum에 더한다. 이때 Sx[a][b]와 Sy[a][b]가 아닌 Sx[a+1][b+1]과 Sy[a+1][b+1]를 곱하는 것에 주의해야 하는데, 이는 커널 Sx와 Sy는 3*3 크기로 선언되어 0, 1, 2의 인덱스를 사용하는 반면 반복문에서는 커널 중심을 기준으로 -1, 0, 1의 인덱스를 사용하고 있어 커널 값을 올바르게 매핑하기 위함이다.

(i-1, j-1)	(i, j)	(i+1, j-1)		-1	0	1
(i-1, j)	(i, j)	(i+1, j)	×	-2	0	2
(i-1, j+1)	(i, j)	(i+1, j+1)		-1	0	1

위와 같이 계산함으로써

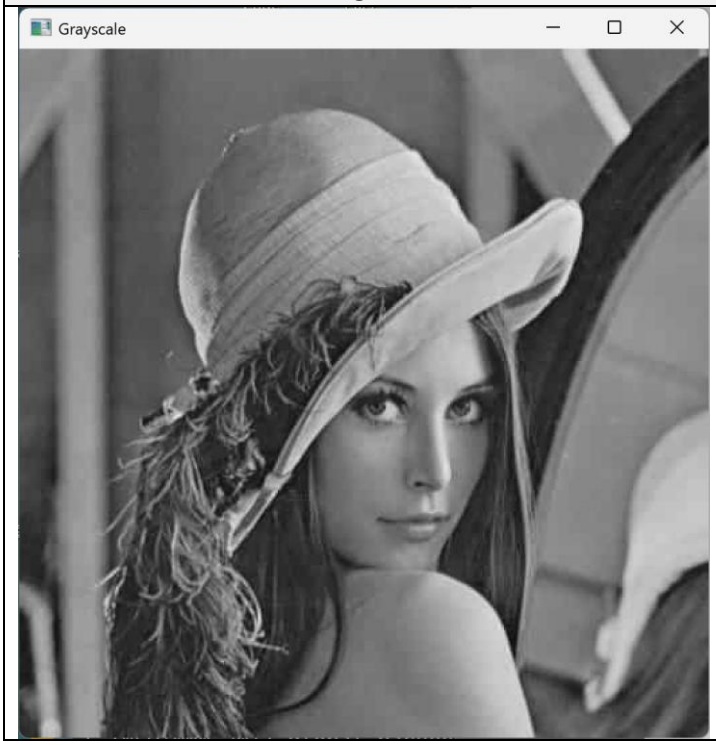
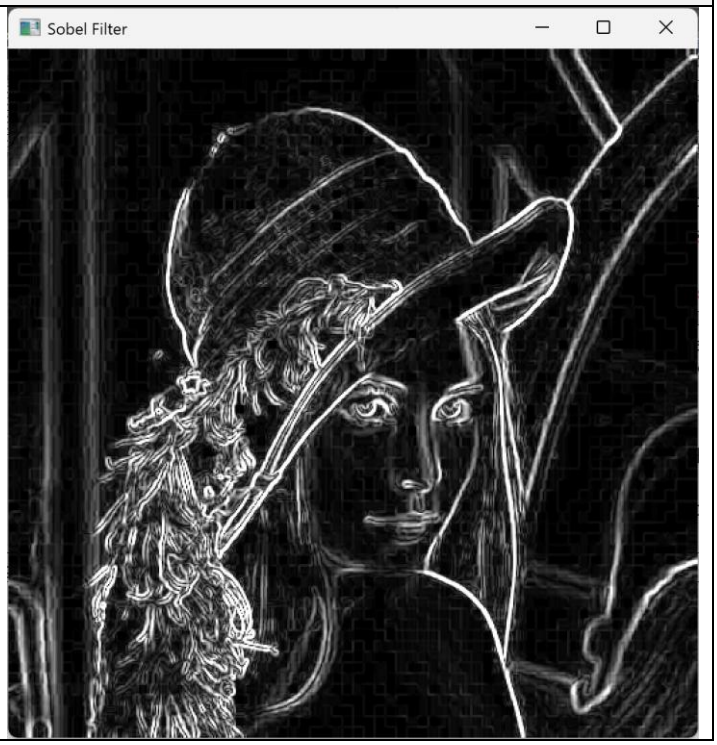
$$Xsum = \big(I(i+1,j-1) - (i-1,j-1)\big) + 2\big(I(i+1,j) - (i-1,j)\big) - (I(i+1,j+1) - I(i-1,j+1))$$

이 된다. Ysum 또한 동일한 방식으로 계산된다.

-n부터 n까지 반복하며 커널 내 픽셀들의 총합을 구한 뒤, 해당 커널의 Xsum과 Ysum을 이용하여 gradient magnitude M을 계산한다. 이후 M이 0~255의 범위를 벗어났을 경우 해당 범위 내로 수정하고 output의 (i, j) 위치에 계산한 M을 매핑한다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다.

Original	Sobel Filter
	

lena.jpg 이미지의 경계가 검출된 것을 확인할 수 있으며, 원본 이미지와 결과 이미지가 별도의 창에 각각 출력되었다.

SobelRGB.cpp

SobelRGB.cpp는 컬러 이미지에 대해 sobel 필터를 적용하는 프로그램이다. 이미지에 접근할 때 R, G, B 채널을 분리하며, 커널 내 총합 또한 채널별로 계산하는 것을 제외하면 SobelGray.cpp와 동일하다.

1. **main()**

```
Mat input = imread("lena.jpg", IMREAD_COLOR); // IMREAD_COLOR
Mat output;
…
output = sobelfilter(input);
```

이미지를 grayscale로 변환하는 과정 없이 컬러 이미지 lena.jpg를 저장한 input을 그대로 사용하여 sobelfilter 함수를 호출한다. 그 외 부분은 SobelGray.cpp의 main 함수와 동일하다.

2. **Mat sobelfilter(const Mat input)**

채널별 계산하는 것을 제외하면 SobelGray.cpp와 동일하다.

```
float Xsum_r = 0.0, Xsum_g = 0.0, Xsum_b = 0.0;
float Ysum_r = 0.0, Ysum_g = 0.0, Ysum_b = 0.0;
```

커널 내 총합 변수 Xsum, Ysum을 커널 내 R, G, B의 총합을 저장할 수 있도록 Xsum_r, g, b와 Ysum_r, g, b로 분리하고, 각각을 0으로 초기화한다.

```
Xsum_r += Sx[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[0]);
Xsum_g += Sx[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[1]);
Xsum_b += Sx[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[2]);

Ysum_r += Sy[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[0]);
Ysum_g += Sy[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[1]);
Ysum_b += Sy[a + 1][b + 1] * float(input.at<C>(tempa, tempb)[2]);
```

이후 input의 (tempa, tempb)값에 커널 값을 곱하여 총합 변수에 저장하는데, 마찬가지로 채널별로 분리하여 Xsum_r에는 (커널 값)*(input(tempa, tempb)의 0채널 값)을, Xsum_g에는 input(tempa, tempb)[1]로 계산한 값을, Xsum_b에는 input(tempa, tempb)[2]로 계산한 것을 저장한다. Sy에 대해서도 동일하게 계산한다. SobelGray.cpp와 동일하게 인덱싱하므로, 마찬가지로 커널 배열에 대해서는 a+1과 b+1을, input에 대해서는 tempa와 tempb를 사용한다.

```
int M_r = sqrt((Xsum_r * Xsum_r + Ysum_r * Ysum_r));
if (M_r < 0) { M_r = 0; }
if (M_r > 255) { M_r = 255; }

int M_g = sqrt((Xsum_g * Xsum_g + Ysum_g * Ysum_g));
if (M_g < 0) { M_g = 0; }
if (M_g > 255) { M_g = 255; }

int M_b = sqrt((Xsum_b * Xsum_b + Ysum_b * Ysum_b));
if (M_b < 0) { M_b = 0; }
if (M_b > 255) { M_b = 255; }
```

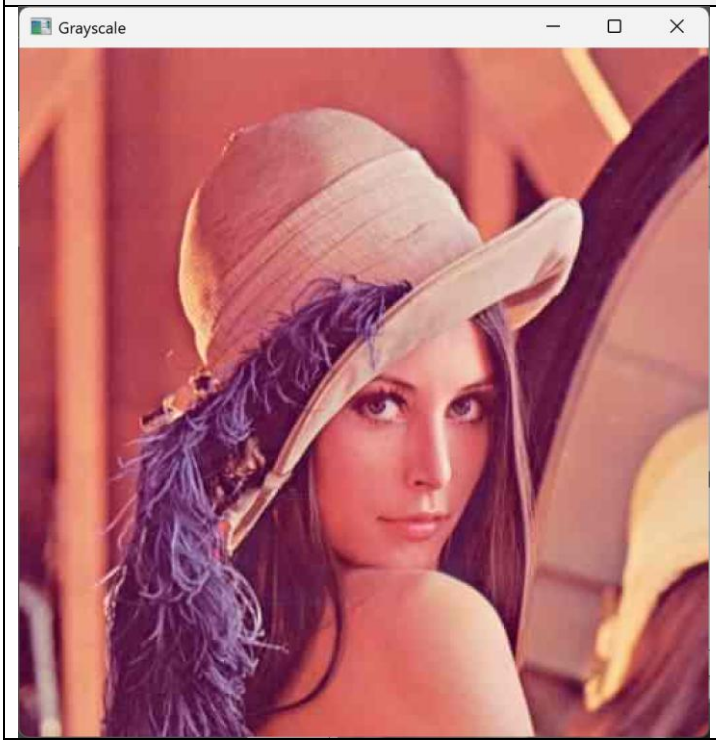
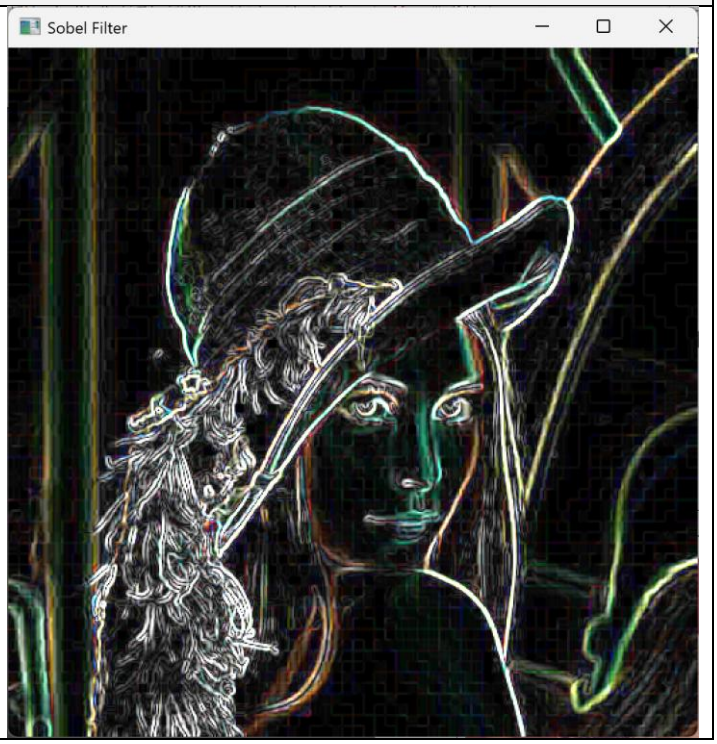
magnitude 변수 M 또한 채널을 분리해 각각 계산한다. Xsum과 Ysum을 계산했던 것과 동일한 원리로 M_r, M_g, M_b를 각각 사용하며, 각각에 대해서 제곱 후 sqrt()한 값을 계산한 다음 0~255 범위로 매핑한다.

```
output.at<C>(i, j)[0] = (G)M_r;
output.at<C>(i, j)[1] = (G)M_g;
output.at<C>(i, j)[2] = (G)M_b;
```

이후 output[0], [1], [2]에 M_r, g, b를 각각 저장한다.

3. 결과 분석

visual studio IDE에서 OpenCV 라이브러리, C++을 사용해 코드를 컴파일 및 실행하였으며, 입력 이미지로는 "lena.jpg"를 사용하였다.

Original	Sobel Filter
	

lena.jpg 이미지의 경계가 검출되었으며, 컬러 이미지를 사용했으므로 결과 이미지에도 색상이 표시되어 보인다.

LaplacianGray.cpp

LaplacianGray.cpp는 grayscale 이미지에 Laplacian 필터를 적용해 이미지 경계를 검출한다.

1차 도함수와 수평, 수직 방향 커널 2개를 사용하는 sobel 필터와 달리 laplacian 필터는 2차 도함수를 계산하며, 커널은 1개만을 사용한다. 이미지 I(x, y)에 대한 1차 도함수는 아래와 같고

$$\nabla I(x,y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right), I_x = \frac{\partial I}{\partial x} = I(x+1,y) - I(x,y), I_y = \frac{\partial I}{\partial x} = I(x,y+1) - I(x,y)$$

I(x, y)의 2차 도함수는 다음과 같으므로

$$\nabla^2 I(x,y) = \left(\frac{\partial^2 I}{\partial x^2}, \frac{\partial^2 I}{\partial y^2}\right),$$

$$\frac{\partial^2 I}{\partial x^2} = I_x(x,y) - I_x(x-1,y) = I(x+1,y) + I(x-1,y) - 2I(x,y),$$

$$\frac{\partial^2 I}{\partial y^2} = I_y(x,y) - I_y(x,y-1) = I(x,y+1) + I(x,y-1) - 2I(x,y)$$

(그림1)을 laplacian 커널 L로 사용할 수 있다. 대각선 방향까지 고려한다면 (그림2)를 사용한다. main함수는 위의 프로그램들과 동일하다.

0	1	0
1	−4	1
0	1	0

Figure 1

1	1	1
1	-8	1
1	1	1

Figure 2

1. Mat laplacianfilter(const Mat input)

```
int row = input.rows;
int col = input.cols;
int n = 1; // Sobel Filter Kernel N

int L[3][3] = { {0, 1, 0}, {1, -4, 1}, {0, 1, 0} };
```

커널 배열 L에 (그림1)의 커널을 저장한다.

```
float sum1 = 0.0;

for (int a = -n; a <= n; a++) {
    for (int b = -n; b <= n; b++) {
            if (i + a > row - 1)
                    tempa = i + a;
            }
            // mirroring
            else {
                    tempb = j + b;
            }

            sum1 += L[a + n][b + n] * input.at<G>(tempa, tempb);
    }
}
```

mirroring 과정은 동일하며, 커널 L 값과 이미지의 (tempa, tempb)를 곱한 것을 sum1에 더한다. 이때 커널 L은 0, 1, 2의 인덱스를 사용하나 반복문에서는 -1, 0, 1의 인덱스를 사용하고 있으므로 L[a+n][b+n]과 input(tempa, tempb)로 계산한다.

```
int M = sum1;
// 0~255
if (M < 0) { M = 0; }
if (M > 255) { M = 255; }
```

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

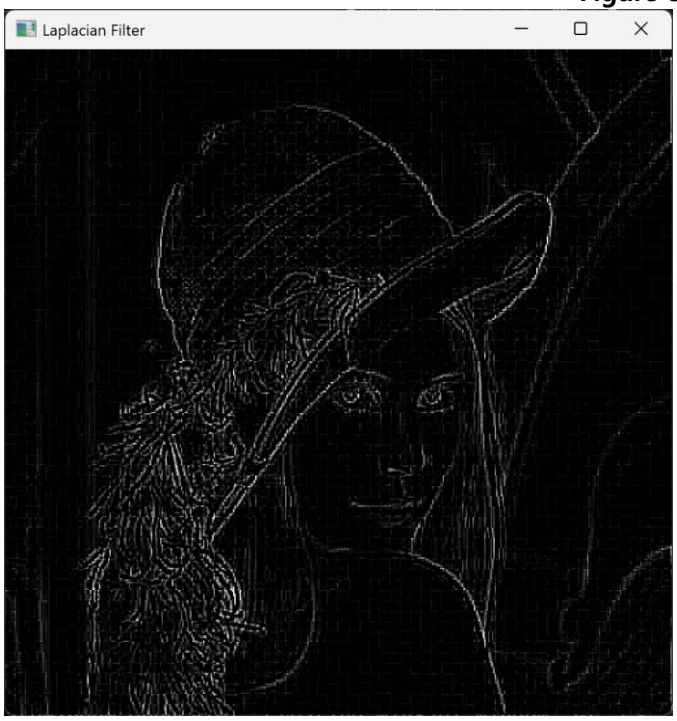
gradient magnitude M에 계산한 sum1을 저장하고, M이 0~255의 범위에 존재하도록 매핑한 뒤 output의 (i, j) 픽셀에 M을 할당한다. 이때 $O = |L * I|$이므로 abs(M)으로 저장한다.

2. 결과 분석

Original	Laplacian FIlter
	

실행 결과 laplacian 필터가 적용되어 경계가 검출된 것을 확인할 수 있다. 커널 L을 L[3][3] = { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} };으로 변경하면 (그림3)과 같이 경계가 더 선명하게 검출되는데, { {0, 1, 0}, {1, -4, 1}, {0, 1, 0} } 커널은 상하좌우 4개의 인접 픽셀만을 고려하지만 { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} } 커널은 대각선 방향을 포함해 8개의 인접 픽셀을 고려하므로 중심 픽셀에 대한 주변 정보를 더 많이 고려하여 경계의 변화가 더 두드러지게 나타나기 때문이다.v

Figure 3



LaplacianRGB.cpp

LaplacianRGB.cpp는 컬러 이미지에 laplacian 필터를 적용한다. input을 grayscale로 변환하는 과정 없이 laplacianfilter() 함수에 컬러 input을 그대로 전달하며, 이를 제외하면 main함수는 LaplacianGray.cpp와 동일하다.

1. Mat laplacianfilter(const Mat input)

함수 정의, 커널 배열 L, mirroring 처리 방식 구현은 LaplacianGray.cpp와 동일하다.

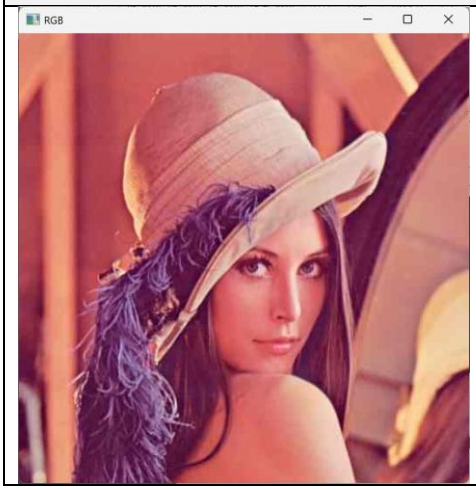
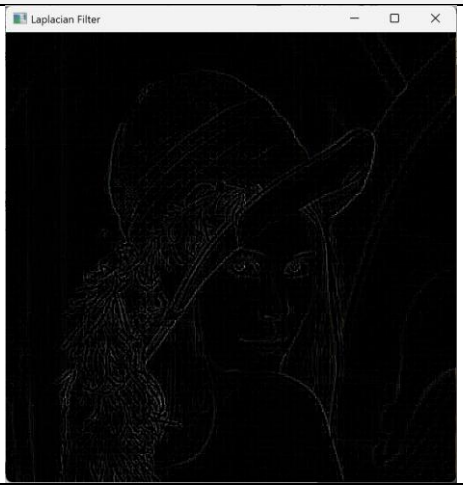
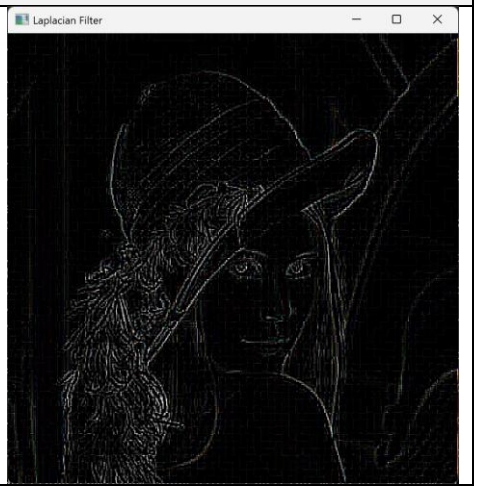
```
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++)
            float sum1_r = 0.0;
            float sum1_g = 0.0;
            float sum1_b = 0.0;
            // mirroring …
            sum1_r += L[a + n][b + n] * (float)(input.at<C>(tempa, tempb)[0]);
            sum1_g += L[a + n][b + n] * (float)(input.at<C>(tempa, tempb)[1]);
            sum1_b += L[a + n][b + n] * (float)(input.at<C>(tempa, tempb)[2]);
```

이전의 RGB 프로그램들과 마찬가지로 커널 내 총합 변수 sum1을 sum1_r, sum1_g, sum1_b로 분리해 선언하며, (커널 값)*(픽셀 값) 계산 또한 input(tempa, tempb)[0], [1], [2]와 같이 각각 계산한다.

```
int M_r = sum1_r;
if (M_r < 0) { M_r = 0; }
if (M_r > 255) { M_r = 255; }

int M_g = sum1_g;
if (M_g < 0) { M_g = 0; }
if (M_g > 255) { M_g = 255; }

int M_b = sum1_b;
if (M_b < 0) { M_b = 0; }
if (M_b > 255) { M_b = 255; }

output.at<C>(i, j)[0] = abs((G)M_r);
output.at<C>(i, j)[1] = abs((G)M_g);
output.at<C>(i, j)[2] = abs((G)M_b);
```

M도 R, G, B 채널을 분리하며, 각각을 0~255로 매핑하고 각 output 채널에 절댓값 연산한 결과를 할당한다.

2. 결과 분석

Original	Laplacian	L = { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} }
		

laplacian 필터링된 결과 이미지가 출력되며, 마찬가지로 L = { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} }을 사용했을 때 경계가 더 선명하게 검출된다.

GaussianGray_sep.cpp

GaussianGray_sep.cpp는 grayscale 이미지에 대해 gaussian 필터를 적용하되, GaussianGrap.cpp에서 n*n 커널을 적용했던 것과 달리 커널을 분리하여 n*1 커널과 1*n 커널을 각각 적용한다.

associative property는 여러 linear 연산을 수행할 때 각 연산을 어떤 순서로 수행하든 같은 결과가 도출되는 성질이다. gaussian 필터 함수 또한 linear 연산이므로, 이미지에 n*n 커널을 1번 적용한 결과와 이를 분해한 n*1, 1*n 커널을 각각 1번씩 적용한 결과는 동일하다.

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

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

gaussianfilter() 함수 내에서 (1)N*1 커널과 (2)1*N 커널을 순차적으로 적용하며, (1)에서는 수직 방향 인덱스가 0 하나뿐이지만 (2)에서는 수평 방향 인덱스가 0 하나뿐이므로 각 단계에서 별도의 경계 처리 프로세스를 구현하였다.

1. **main()**
   main 함수는 GaussianGray.cpp와 동일하다.
2. **Mat gaussianfilter(const Mat input, int n, float sigmaT, float sigmaS, const char* opt)**
   GaussianGray.cpp에서 N*N 커널을 1번 적용하는 것과 달리 N*1 커널과 1*N커널을 각각 적용하며, 각 적용 당 내부 구현은 인덱스 사용을 제외하면 GaussianGray.cpp와 동일하다.

   2.1 함수 정의 및 normalization

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

// Initialiazing Kernel Matrix
Mat kernelT = Mat::zeros(kernel_size, 1, CV_32F);        // N*1
Mat kernelS = Mat::zeros(1, kernel_size, CV_32F);        // 1*N
```

GaussianGray.cpp와 유사하다. row와 col에 input의 세로와 가로 크기를 저장하고 전달받은 n으로 (2n+1) 크기의 커널 크기 변수를 생성해 kernel_size에 저장하며, mirroring 구현에 사용할 tempa, tempb 변수와 커널 각각에 사용할 normalization 용 denomT, denomS 변수를 선언한다. 이후 kernel = Mat::zeros(kernel_size, kernel_size, CV_32F);로 커널을 하나 생성했던 것과 달리 n*n 커널을 수평, 수직 방향으로 분리한 kernelT와 kernelS를 생성한다. 이때 kernelT는 N*1 크기의 가로로 긴 수평 커널이고 kernelS는 1*N 크기의 세로로 긴 수직 커널이다.

```
denomT = 0.0;
for (int i = -n; i <= n; i++) {
	float value1 = exp(-(pow(i, 2) / (2 * pow(sigmaT, 2))));
	kernelT.at<float>(i + n, 0) = value1;
	denomT += value1;
}
denomS = 0.0;
for (int i = -n; i <= n; i++) {
	float value1 = exp(-(pow(i, 2) / (2 * pow(sigmaS, 2))));
	kernelS.at<float>(0, i + n) = value1;
```

```
        denomS += value1;
}
for (int i = -n; i <= n; i++) {
        kernelT.at<float>(i + n, 0) /= denomT;
        kernelS.at<float>(0, i + n) /= denomS;
}
```

커널 값 생성과 normalization 원리는 GaussianGray.cpp와 동일하지만, 기존에는 1개의 커널 객체로 -n부터 n까지 이중 for문을 순회하며 한 번 초기화한 것과 달리 각 커널 객체 kernelT와 kernelS에 대해 -n부터 n까지 수평 또는 수직 방향으로 각각 초기화한다.

2.2 N*1 커널 적용

```
Mat temp = Mat::zeros(row, col, input.type());
```

input에 N*1 커널 kernelT를 적용한 것을 저장할 temp 객체를 선언한다.

```
for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
                if (!strcmp(opt, "zero-paddle")) {
                        float sum1 = 0.0;
                        for (int b = -n; b <= n; b++) {
                                if ((j + b <= col - 1) && (j + b >= 0)) {
                                        sum1 += kernelT.at<float>(b + n, 0) *
                                        input.at<G>(i, j + b);
                                }
                        }
                        temp.at<G>(i, j) = (G)sum1;
                }

                else if (!strcmp(opt, "mirroring")) {
                        float sum1 = 0.0;
                        for (int b = -n; b <= n; b++) {
                                ...
                        temp.at<G>(i, j) = (G)sum1;
                }
                else if (!strcmp(opt, "adjustkernel")) {
                        float sum1 = 0.0;
                        float sum2 = 0.0;
                        for (int b = -n; b <= n; b++) {
                                ...
                        }
                        temp.at<G>(i, j) = (G)(sum1 / sum2);
                }
        }
}
```

옵션별 구현은 위의 프로그램들과 동일하지만 수평 필터 kernelT가 N*1 크기의 가로로 긴 모양이므로 수직 인덱스를 0으로 고정하는 것에 주의해야 한다. 외부 반복문은 동일하게 0에서 (row-1)까지(행), 0에서 (col-1)까지(열) 순회하지만, 각 옵션에서 내부 for문 for(int b=-1; b<=n; b++)을 순회한다. kernelT가 N*1이므로 2중 for문을 사용하지 않고 중심 픽셀 j에서 좌우로 오프셋 b만큼 이동하며, 내부 조건문 또한 기존에 a와 b에 대해 모두 검사하던 것을 (j+b)만 이미지의 열 범위 내에 있는지 확인한다.

input 객체에 접근할 때는 수직 방향 인덱스를 i로 고정해 (i, j+b)를 사용한다(mirroring 옵션의 경우 (i, tempb) 사용). kernelT 객체에 접근할 때도 마찬가지로 수직 방향 인덱스를 0으로 고정해 (b+n, 0)을 사용한다.

kernelT(b+n, 0)과 input(i, j+b)(또는 input(i, tempb))를 곱한 것을 누적한 뒤, 반복문이 종료되면 temp의 동일한 위치에 저장한다.

2.3 1*N 커널 적용

```
Mat output = Mat::zeros(row, col, input.type());
```

2.2.의 temp 객체에 1*N 커널 kernelS를 적용한 것을 저장할 output 객체를 선언한다.

```
for (int i = 0; i < row; i++) {
	for (int j = 0; j < col; j++) {
		if (!strcmp(opt, "zero-paddle")) {
			float sum1 = 0.0;
			for (int a = -n; a <= n; a++) {
				if ((i + a <= row - 1) && (i + a >= 0)) {
					sum1 += kernelS.at<float>(0, a + n) *
					temp.at<G>(i + a, j);
				}
			}
			output.at<G>(i, j) = (G)sum1;
		}
		// 동일
		}
	}
}
```

2.1.과 동일한 메커니즘을 사용한다. 수직 필터 kernelS가 1*N 크기의 세로로 긴 모양이므로 수평 인덱스를 0으로 고정하며, 외부 반복문은 행과 열 전체를 순회하지만 내부 for문은 for(int a=-n; a<=n; a++)를 사용한다. kernelS가 1*n이므로 중심 픽셀 i에서 위아래로 오프셋 a만큼 이동하며, 내부 조건문은 (i+a)만 이미지의 행 범위 내에 있는지 확인한다.

마찬가지로 input 객체에 접근할 때는 수평 방향 인덱스를 j로 고정해 (i+a, j)를 사용한다(mirroring 옵션의 경우 (tempa, j) 사용). kernelS 객체에 접근할 때도 마찬가지로 수평 방향 인덱스를 0으로 고정해 (0, a+n)을 사용한다. kernelS(0, a+n)과 input(i+a, j)를 곱한 것을 누적한 뒤, 반복문이 종료되면 output의 (i, j)에 저장한다.

3. 결과 분석

3.1 GaussianGray_sep.cpp

input=input_gray, n=1, sigmaT=1, sigmaS=1로 경계 처리 옵션을 달리하여 호출하였다.

original	zero padding	mirroring	adjust kernel
			

모든 옵션에서 gaussian 필터가 적용되어 노이즈가 사라지고 blurry해진 이미지가 별도의 창에 출력되었다.

3.2 GaussianGray.cpp와 비교

MeanFilterGray.cpp의 서론에서 서술했든, associative property를 이용해 separable filtering을 수행하면 동일한 n을 사용했을 때 커널이 separable하지 않은 경우보다 실행 시간이 감소한다. gaussianfilter() 함수 내에서 kernel_size=2*n+1로 선언하는데, n*n 커널을 하나 사용할 때는 -n부터 n까지 2중 for문을 통해 한

픽셀당 (2n+1)*(2n+1)번 연산하므로 $O(n^2)$의 시간복잡도를 가지지만, n*1 커널과 1*n 커널을 분리했을 때는 각 커널 당 -n부터 n까지 단일 for문을 사용하여 한 픽셀 당 (2n+1)번 연산하고 이를 2번 반복하므로 2*(2n+1), 즉 O(n)의 시간복잡도를 가진다.

n=10, sigmaT=1, sigmaS=1, opt="zero-paddle"로 통일하여 GaussianGray.cpp와 GaussianGray_sep에서 output = gaussianfilter(input_gray, 10, 1, 1, "zero-paddle");을 각각 호출하고 그 결과를 관찰하면 다음과 같다.

실행 결과 GaussianGray_sep.cpp에서 n*1, 1*n으로 커널을 분리했을 때의 수행 시간이 훨씬 짧음을 확인할 수 있으며, 이 차이는 n이 커질수록 극대화된다. n=20으로 바꾸고 실행했을 때의 결과는 아래와 같다.

GaussianRGB_sep.cpp

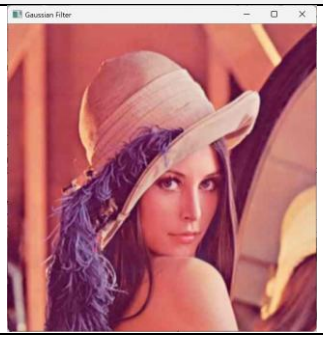
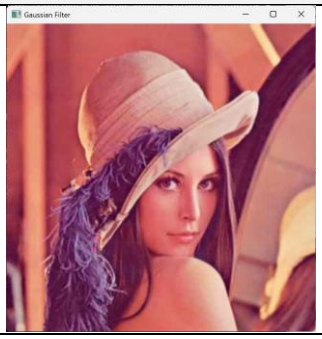
GaussianGray_sep.cpp와 같이 n*1, 1*n 크기로 분리한 2개의 커널로 gaussian blurring을 수행하되, 입력 이미지로 컬러 이미지를 사용한다.

```
float sum1_r = 0.0;
float sum1_g = 0.0;
float sum1_b = 0.0;
…
sum1_r += kernelT.at<float>(b + n, 0) * float(input.at<C>(i, j + b)[0]);
sum1_g += kernelT.at<float>(b + n, 0) * float(input.at<C>(i, j + b)[1]);
sum1_b += kernelT.at<float>(b + n, 0) * float(input.at<C>(i, j + b)[2]);
```

GaussianGray_sep.cpp의 sum1을 모두 sum1_r, sum1_g, sum1_b로 분리해 각 채널에 대해 각각 계산하고, sum1에 kernel 값과 input 값을 곱한 것을 누적하는 부분 또한 sum_r, g, b에 input[0], [1], [2]를 사용하는 것으로 수정한다. 채널을 분리하는 것을 제외하면 코드를 동일하다.

1. **GaussianRGB_sep.cpp**

   input=input, n=1, sigmaT=1, sigmaS=1로 경계 처리 옵션을 달리하여 호출하였다.

original	zero padding	mirroring	adjust kernel
			

   모든 옵션에서 gaussian 필터가 적용되어 노이즈가 사라지고 blurry해진 이미지가 별도의 창에 출력되었다.

2. **GaussianRGB.cpp와 비교**

   GaussianRGB 또한 커널을 n*1, 1*n으로 분리했을 때 분리하지 않은 것보다 수행 시간이 짧다. n=10, sigmaT=1, sigmaS=1, opt="zero-paddle"로 통일하여 GaussianRGB.cpp와 GaussianRGB_sep에서 output = gaussianfilter(input, 10, 1, 1, "zero-paddle");을 각각 호출하고 그 결과를 관찰하면 다음과 같다.

```
not separated: 33.3304 seconds
```

```
separable filter: 2.82572 seconds
```

   RGB 3개의 채널에 대해 모두 계산하므로 grayscale 이미지를 사용할 때보다 수행 시간이 증가했지만, 여전히 커널을 분리했을 때가 분리하지 않았을 때보다 월등히 빠르게 작동한다.

   n=20으로 증가했을 때 두 프로그램의 수행 시간 차이가 더 커진다.

```
not separated: 130.41 seconds
```

```
separable filter: 5.37228 seconds
```

UnsharpGray.cpp

UnsharpGray.cpp는 grayscale 이미지에 대해 unsharp masking을 적용한다.

Unsharp masking은 이미지에 low-pass 필터를 적용해 blur한 뒤, low-frequency 성분을 가진 blurry한 이미지와 high-frequency 요소를 포함한 원본 이미지를 비교해 강조하고 싶은 요소(edge)를 추출하며, 추출한 edge를 조절해 보다 선명해진(sharp) 결과 이미지를 생성한다.

$$output = \frac{I - kL}{1 - k} (I = image, L = low - pass\ image, k = scaling\ factor)$$

위의 공식을 사용해 unsharp masking을 수행하며, 이때 스케일 계수 k는 unsharp mask를 얼마나 강하게 적용할지 조절하는 scaling factor이다. k가 커질수록 I-kL에서 kL, 즉 low-frequency 성분이 더 많이 제거되므로 남겨진 high-frequency 부분이 상대적으로 강조되며 결과 이미지가 더욱 선명해진다. 분자가 1-k이므로 k는 1보다 작은 값으로 설정해야 한다.

1. **main()**

   main() 함수는 위의 프로그램들과 동일하다. input 변수에 "lena.jpg" 이미지를 읽어들여 grayscale 변환한 것을 input_gray에 저장하고, input_gray로 unsharpmask() 함수를 호출해 unsharp masking을 수행한다.

```
float k = 0.5f;
output = unsharpmask(input_gray, 3, 3, 3, k, “mirroring”);
```

   k=0.5로 초기화하지만 다른 값을 사용해도 무방하다.

2. **Mat unsharpmask(const Mat input, int n, float sigmaT, float sigmaS, float k, const char* opt)**

```
// low-pass filter로 gaussian 사용
Mat temp = gaussianfilter(input, n, sigmaT, sigmaS, "mirroring");
namedWindow("Gaussian Filter", WINDOW_AUTOSIZE);
imshow("Gaussian Filter", temp);
```

   low-pass 이미지를 만들기 위해 gaussian 필터를 적용하며, GaussianGray.cpp에서 구현한 gaussianfilter() 함수를 그대로 사용한다. gaussianfilter() 에는 main()에서 unsharpmask() 호출 시 사용한 n, sigmaT, sigmaS, opt를 전달한다. gaussian 필터링한 이미지를 temp 객체에 저장하므로 temp는 위 공식의 L을, input은 I를 의미한다. 위 코드에서는 커널 크기 변수 n, 표준편차 sigmaT, sigmaS 모두 3으로 호출하고 있으나 다른 값을 전달할 수 있다. 이후 input에 gaussian 필터가 적용된 temp를 결과 창으로 출력한다.

```
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
            // (I - kL) / (1 - k)
            // 원본 이미지 I = input.at<G>(i, j);
            // low-pass 이미지 L = temp.at<G>(i, j);
            float o = ((float)input.at<G>(i, j) - k * (float)temp.at<G>(i, j)) / (1.0f - k);

            if (o < 0) { o = 0; }
            if (o > 255) { o = 255; }

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

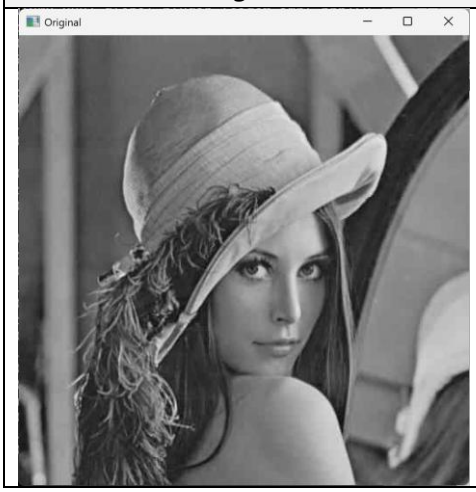
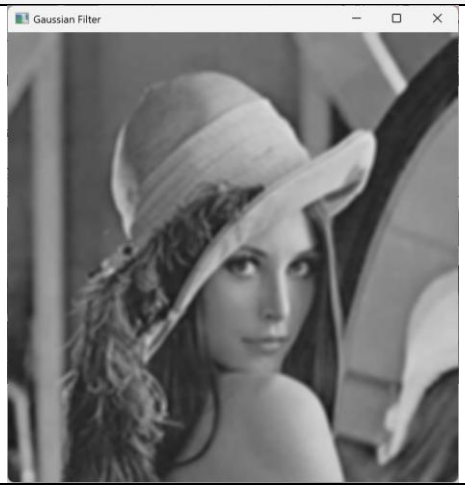
   2중 for문을 통해 이미지의 모든 픽셀에 대해 계산한다. input 객체는 원본 이미지 I이고 temp 객체는 low-pass 이미지 L이므로 $output = \frac{I-kL}{1-k}$공식을 그대로 적용해 input(i, j)에 k*temp(i, j)한 것을 (1-k)로 나누어 output 변수

o에 저장한다.

이후 o를 0~255 범위에 매핑하고, output의 (i, j) 위치에 할당한다.

3. 결과 분석

output = unsharpmask(input, 3, 3, 3, k, "zero-paddle")로 호출한다. 입력 이미지로는 "lena.jpg"를 사용했다. gaussian 필터 또한 커널 크기 3, 수직 및 수평 표준편차 3으로 호출된다.

Original	Low-pass(gaussian)	Unsharp mask
		

원본 이미지, gaussian 필터링한 low-pass 이미지, unsharp masking한 이미지가 별도의 창에 각각 출력되었으며 결과 이미지가 선명해진 것을 확인할 수 있다.

이후 동일한 k=0.5, n=sigmaT=sigmaS=3으로 경계 처리 옵션을 달리하여 3번 실행하였다.

Zero padding	mirroring	Adjust kernel
		

각 옵션별로 unsharp mask가 적용된 grayscale 이미지가 출력되며, zero paddle 옵션에서는 이미지 boundary 부근이 밝게 표현된다.

uniform mean filter나 gaussian filter에서 zero padding하면 이미지 경계 값이 어두워지지만 unsharp filtering에서는 반대로 밝아지는데, 이는 gaussianfilter() 함수에서 zero paddle 옵션으로 모서리 값을 zero padding했을 때 unsharp masking 공식 $output = \frac{I-kL}{1-k}$에서 k*L의 값이 작아지므로 boundary 픽셀들의 k*L이 주변 픽셀들보다 상대적으로 높은 값을 갖게 되어 결과 이미지에서 밝게 표시되기 때문이다.

이후 k값에 따른 sharpening 정도를 비교하기 위해 k=0.1, k=0.5, k=0.8로 각각 호출하였고, 그 결과는 아래와 같다.

k=0.2	k=0.5	k=0.8

k가 커질수록 k*L, 즉 k*(low-frequency 성분)이 커져 더 많이 제거되고 I의 high-frequency 성분이 강조되므로 sharp한 이미지가 생성된다.

UnsharpRGB.cpp

UnsharpRGB.cpp는 컬러 이미지에 대해 unsharp masking을 수행한다. 동일하게 $output = \frac{I-kL}{1-k}$ 공식을 사용하며 R, G, B 채널을 분리하고 컬러 이미지용 gaussianfilter() 함수를 사용하는 것을 제외하면 UnsharpGray.cpp와 동일하다.

1. **main()**

```
float k = 0.5f;
output = unsharpmask(input, 3, 3, 3, k, “mirroring”);
```

input을 grayscale input_gray로 변환하지 않고 unsharpmask()에 그대로 전달한다. 이를 제외하면 UnsharpGray.cpp의 main()과 동일하다. k=0.5로 초기화하였으나 다른 값으로 수정해도 무방하다.

2. **Mat unsharpmask(const Mat input, int n, float sigmaT, float sigmaS, float k, const char* opt)**

```
Mat temp = gaussianfilter(input, n, sigmaT, sigmaS, "mirroring");
```

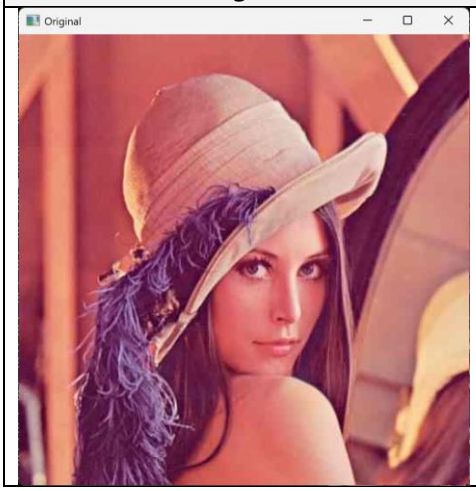
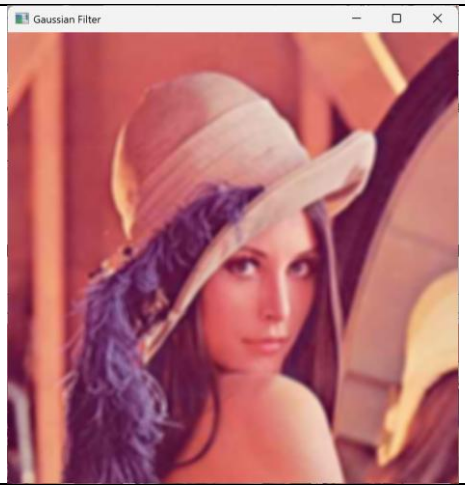
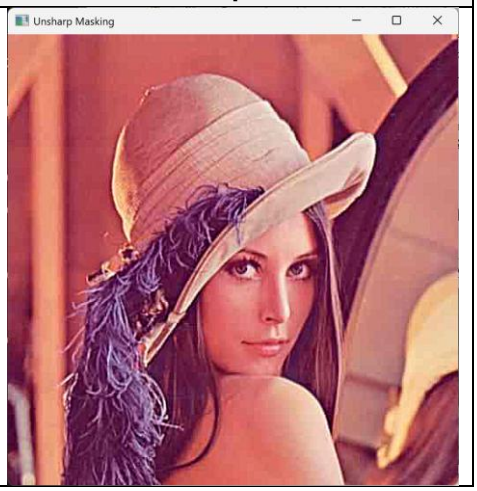
UnsharpGray.cpp와 동일하게 gaussian 필터를 적용해 low-frequency 이미지를 생성하되, 이때 컬러 이미지용 gaussianfilter() 함수를 호출한다. GaussianRGB.cpp에서 구현한 것과 동일하다. 마찬가지로 gaussian 필터를 적용한 이미지를 temp에 저장하므로 temp는 공식의 L, input은 공식의 I를 의미한다. gaussianfilter() 에는 main()에서 unsharpmask() 호출 시 사용한 n, sigmaT, sigmaS, opt를 전달한다. 이후 input에 gaussian 필터가 적용된 temp를 결과 창으로 출력한다.

```
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
            float o_r = ((float)input.at<C>(i, j)[0] - k * (float)temp.at<C>(i, j)[0]) /
            (1.0f - k);
            if (o_r < 0) { o_r = 0; }
            if (o_r > 255) { o_r = 255; }

            output.at<C>(i, j)[0] = (G)(o_r);

            float o_g = ((float)input.at<C>(i, j)[1] - k * (float)temp.at<C>(i, j)[1]) /
            (1.0f - k);
            if (o_g < 0) { o_g = 0; }
            if (o_g > 255) { o_g = 255; }

            output.at<C>(i, j)[1] = (G)(o_g);

            float o_b = ((float)input.at<C>(i, j)[2] - k * (float)temp.at<C>(i, j)[2]) /
            (1.0f - k);
            if (o_b < 0) { o_b = 0; }
            if (o_b > 255) { o_b = 255; }

            output.at<C>(i, j)[2] = (G)(o_b);
    }
}
```

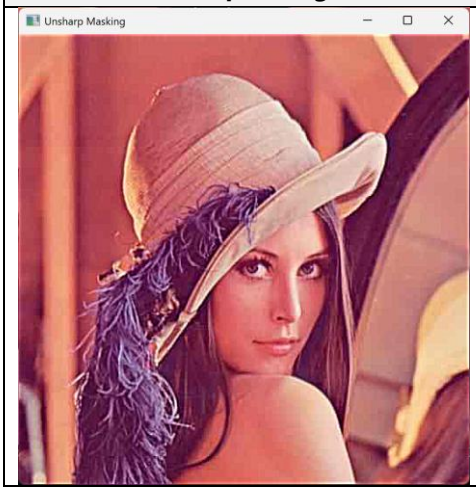
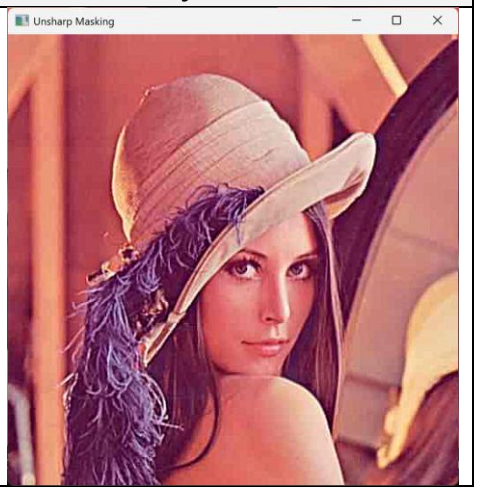
2중 for문으로 이미지 전체 픽셀을 순회하며 각 레퍼런스 픽셀 (i, j)에 대해 계산하는 것은 UnsharpGray.cpp와 동일하나, 컬러 이미지를 사용하고 있으므로 output 변수 o를 RGB 채널을 분리한 o_r, o_g, o_b로 사용하고 input, temp, output에 접근할 때도 output.at<C>(i, j)[0] = (G)(o_r); 과 같이 [0], [1], [2] 채널을 구분해 채널별로 각각 계산한다.

3. **결과 분석**

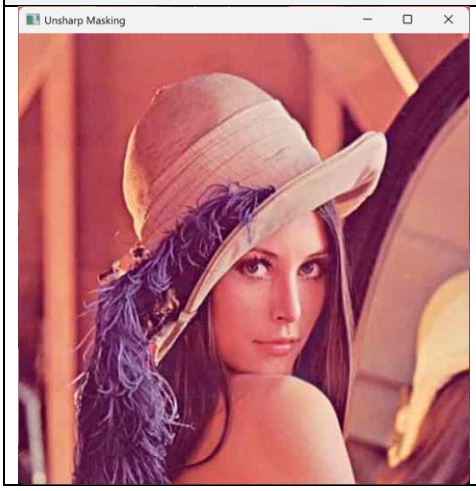
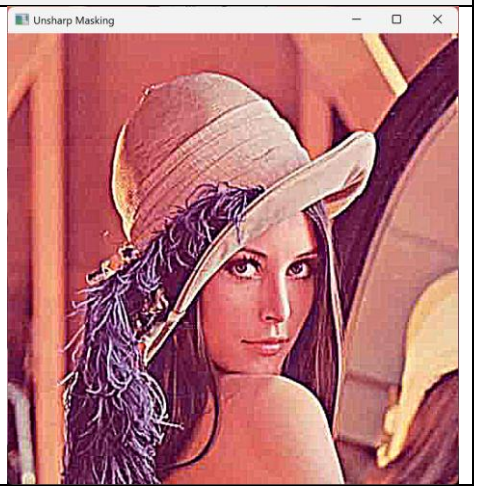
output = unsharpmask(input, 3, 3, 3, k, "mirroring")으로 호출한다. 입력 이미지로는 "lena.jpg"를 사용했다. gaussian 필터 또한 커널 크기 3, 수직 및 수평 표준편차 3으로 호출된다.

Original	Low-pass(gaussian)	Unsharp mask
		

원본 이미지, gaussian 필터링한 low-pass 이미지, unsharp masking한 이미지가 별도의 창에 각각 출력되었으며 결과 이미지가 선명해진 것을 확인할 수 있다. 이후 동일한 k=0.5, n=sigmaT=sigmaS=3으로 경계 처리 옵션을 달리하여 3번 실행하였다.

Zero padding	mirroring	Adjust kernel
		

마찬가지로 zero paddle 옵션에서 이미지 모서리 픽셀들이 밝게 표시되었다. 이후 k값에 따른 sharpening 정도를 비교하기 위해 k=0.1, k=0.5, k=0.8로 각각 호출하였고, 그 결과는 아래와 같다.

k=0.2	k=0.5	k=0.8
		

UnsharpGray.cpp와 동일하게 k가 클수록 결과 이미지가 sharp하다.