1-1. MeanFilterGray.cpp

코드 목적:

흑백 이미지에 대해 simplest low-pass filter 인 Uniform Mean filter 를 구현한다.

kernel 크기와 image boundary 의 픽셀 처리 방법에 따라 다른 결과 이미지를 출력한다.

코드 흐름:

1) 이미지 파일 불러와 흑백으로 전환한다.
2) meanfilter 함수를 통해 Mean filtering한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명: meanfilter(const Mat input, int n, const char* opt)

매개변수:

input: input 흑백 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

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

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

함수 목적:

input 을 opt 적용해 mean filtering 한 결과를 반환(Mat)

설명:

1) Filter kernel을 만든다. 각 pixel에 적용될 가중치이다. Mean Filter의 모든 kernel의 값의 합은 1이다.

```
Mat kernel;
kernel = Mat::ones(kernel_size, kernel_size, CV_32F) / (float)(kernel_size * kernel_size);
float kernelvalue=kernel.at<float>(0, 0);
```

kernel 행렬에 kernel 을 저장한다. kernel 을 kernel_size(N=2*n+1)의 1 로 된 행렬로 초기화 한 뒤, kernel 의 픽셀 수로 나눠준다. kernel 의 N*N 개의 픽셀에 모두 같은 가중치가 지정되었다. kernel 의 요소를 모두 더하면 1 이다. mean filter 에서는 이 kernel 값이 모두 같으므로, 매번 행렬에 접근하지 않고, kernelvalue 에 1/(N^2)의 값을 저장해 가져다 쓰자

2) type of boundary processing
   a. "zero-paddle"

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

zero paddling 은 이미지의 범위에서 벗어난 pixel 의 값은 0 으로 반영한다. 따라서 이미지의 경계쪽이 어둡게 보인다는 특징이 있다. (i , j) 를 필터링 할 때, 이미지의 범위에 포함된다면 kernel 안의 픽셀의 값에 가중치를 곱해서 더해준다. 이미지의 범위에서 벗어난 pixel 은 더해지지 않아 값이 0 으로 반영되는 것이다.

b. "mirroring"

```
else if (!strcmp(opt, "mirroring")) {
        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;
                        else if (i + a < 0)         tempa = -(i + a);
                        else                        tempa = i + a;

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

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

mirroring 은 image boundary 를 벗어나면 image 안의 pixel 의 값을 mirroring 하여 가져온다. kernel 의 y 좌표 i+a, x 좌표 j+b 가 0 보다 작거나 이미지크기-1 보다 크면 (i,j)를 기준으로 미러링한 좌표를 temp 변수에 저장하고, 그 좌표의 값을 가져와 가중치를 곱한다.

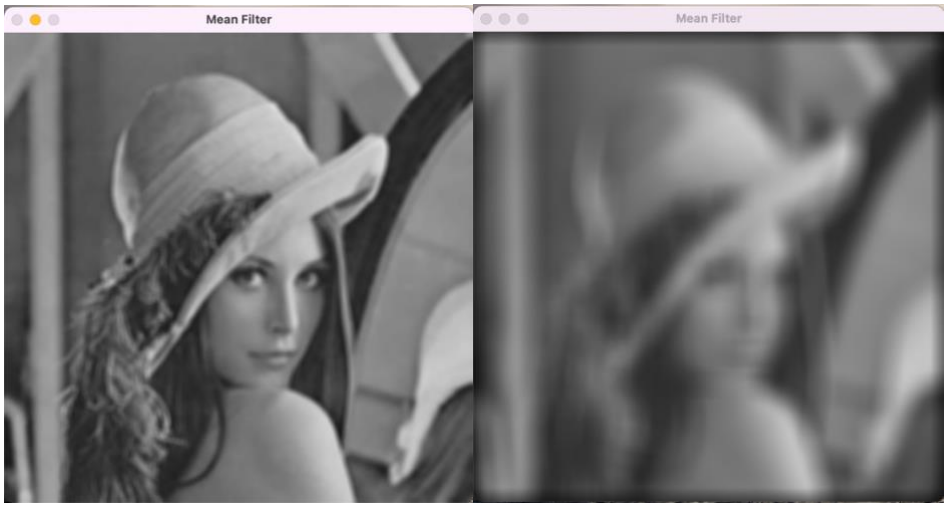
c. "adjustkernel"

```
else if (!strcmp(opt, "adjustkernel")) {
    float sum1 = 0.0;
    float sum2 = 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 += kernelvalue*(float)(input.at<G>(i + a, j + b));
                sum2 += kernelvalue;
            }
        }
    }
    output.at<G>(i, j) = (G)(sum1/sum2);
}
```

"adjustkernel"은 이미지 범위를 벗어나는 픽셀은 사용하지 않고, 나머지 픽셀들을 한번 더 정규화한다. kernel 을 돌며 가중치와 색 값을 곱해 sum1 에 더하고, 가중치만은 sum2 에 더한다. kernel 을 모두 거친 후에 sum1 을 계산에 반영된 pixel 들의 가중치의 총 합인 sum2 로 나눠주면 이미지 안에 포함되는 kernel 안의 pixel 들만의 평균을 구할 수 있다.

실행 결과(Original 이미지 생략)

1) meanfilter(input_gray,3, "mirroring");
2) meanfilter(input_gray,15, "mirroring");



1-2. MeanFilterRGB.cpp

코드 목적:

컬러 이미지에 대해 simplest low-pass filter 인 Uniform Mean filter 를 구현한다.
kernel 크기와 image boundary 의 픽셀 처리 방법에 따라 다른 결과 이미지를 출력한다.

코드 흐름:

1) 이미지 파일을 가져온다
2) meanfilter 함수를 통해 Mean filtering한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명: meanfilter(const Mat input, int n, const char* opt)

매개변수:

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

함수 목적:

input 을 opt 적용해 mean filtering 한 결과를 반환(Mat)

설명: 1-1 과 똑같다. 컬러 이미지이기에, 3 개의 채널에 대해 각각 계산을 진행해주면 된다.

1) zero-paddle만 예시로 보자

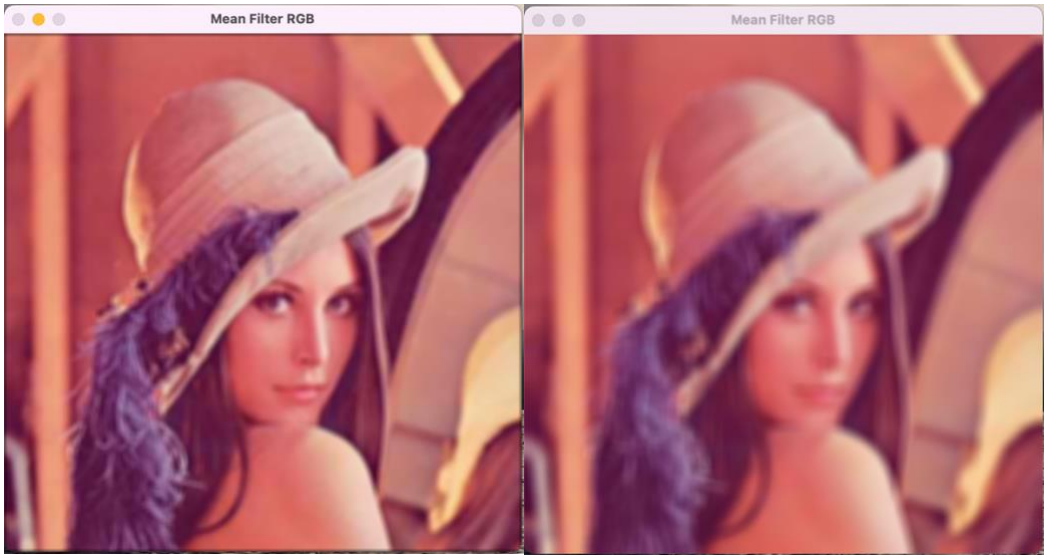
```
if (!strcmp(opt, "zero-paddle")) {
    float sum1_r = 0.0;
    float sum1_g = 0.0;
    float sum1_b = 0.0;

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

각 R, G, B 를 위한 변수 sum1_r, sum1_g, sum1_b 를 선언하고, pixel 들에 접근하여 해당 색의 인덱스에 접근해 데이터를 가져오고, output 의 각 채널에 계산한 값을 저장한다.

실행 결과

1) meanfilter(input, 3, "zero-paddle");
2) meanfilter(input, 6, "adjustkernel");



2-1. GaussianGray.cpp

코드 목적:

흑백 이미지에 대해 Gaussian filter 를 구현한다.

kernel 크기와 x 축에 대한 표준편차, y 축에 대한 표준편차, image boundary 의 픽셀 처리 방법에 따라 다른 결과 이미지를 출력한다.

코드 흐름:

1) 이미지 파일 불러와 흑백으로 전환한다.
2) gaussianfilter 함수를 통해 Gaussian filtering한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

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

매개변수:

input: input 흑백 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

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

함수 목적:

input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

설명:

1) Zero mean Gaussian Filter

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

kernel의 픽셀에 (a,b) 모두 접근해 $\exp\left(-\frac{a^2}{2{\sigma_s}^2}-\frac{b^2}{2{\sigma_t}^2}\right)$를 계산하고, value1에 더해준다.

value는 $\sum_{m=-a}^{a}\sum_{n=-b}^{b}\exp\left(-\frac{m^2}{2{\sigma_s}^2}-\frac{n^2}{2{\sigma_t}^2}\right)$가 되므로, value1으로 kernel을 다시 나눠주면 $w(s,t)$를 구할 수 있다. 따라서 행렬 kernel에 $w(s,t)$가 저장되어 있다.

```
denom = 0.0;
for (int a = -n; a <= n; a++) {
    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;
        }
}

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

2)

```
sum1 += kernel.at<float>(a+n, b+n)*(float)(input.at<G>(i + a, j + b));
```

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

for문을 통해 (i,j) 픽셀의 filtering을 위한 kernel의 (a+n,b+n)픽셀에 접근한다. input에 접근해 가져온 데이터를 해당하는 kernel의 가중치에 곱해 sum1에 더해준다. kernel의 모든 픽셀에 대한 가중치*input데이터를 더해준 값을 output 이미지의 해당 비트에 넣어준다. 가우시안 필터에서 값을 정해주고 계산하는 과정은 위와 같고, boundary processing에 따라 위의 코드를 어디에 쓰는지가 다르다.

type of boundary processing

a. "zero-paddle"

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

zero paddling 이므로, 이미지의 범위에 포함될 때만 값을 더한다.

b. "mirroring"

```
else if (!strcmp(opt, "mirroring")) {
    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;
            else if (i + a < 0)     tempa = -(i + a);
            else                    tempa = i + a;

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

            sum1 += kernel.at<float>(a+n, b+n)*(float)(input.at<G>(tempa, tempb));
        }
    }
    output.at<G>(i, j) = (G)sum1;
}
```

mirroring 은 image boundary 를 벗어나면 (i,j)를 기준으로 미러링한 좌표를 temp 변수에 저장하고, 그 좌표의 값을 가져와 가중치를 곱한다.

c. "adjustkernel"

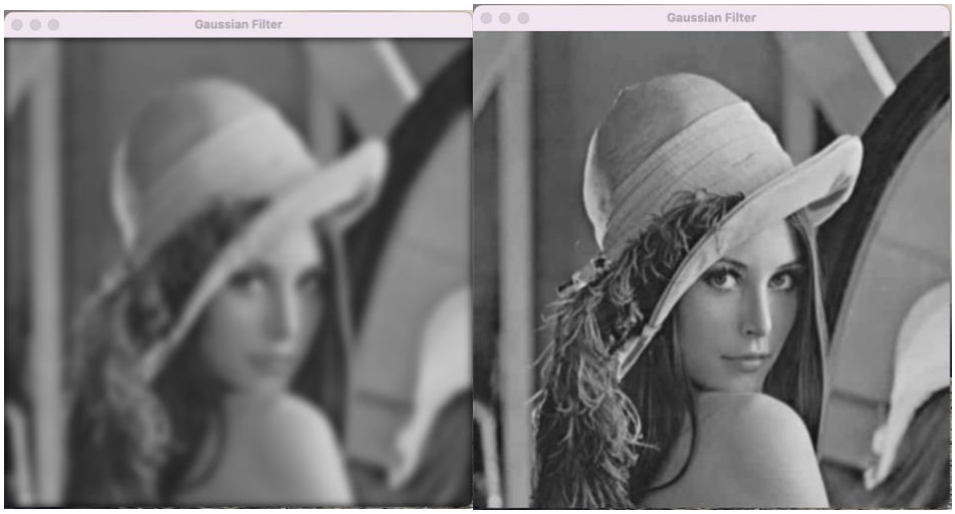
```
else if (!strcmp(opt, "adjustkernel")) {
        float sum1 = 0.0;
        float sum2 = 0.0;

        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)*(float)(input.at<G>(i + a, j + b));
                            sum2 += kernel.at<float>(a+n, b+n);
                    }
            }
        }
      output.at<G>(i, j) = (G)(sum1/sum2);
}
```

"adjustkernel"은 이미지 범위에 해당하는 픽셀의 가중치만을 따로 더하고, 마지막에 이로 나누어 이미지 안에 포함되는 kernel 안의 pixel 들만의 평균을 구한다.

실행 결과

1) gaussianfilter(input_gray, 11,5,5, "zero-paddle");
2) gaussianfilter(input_gray, 1,2,2, "mirroring");



2-2. GaussianRGB.cpp

코드 목적:

컬러 이미지에 대해 Gaussian filter 를 구현한다.

코드 흐름:

1) 이미지 파일 불러와 저장한다.
2) gaussianfilter 함수를 통해 Gaussian filtering한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

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

매개변수:

input: input 흑백 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

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

함수 목적:

input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

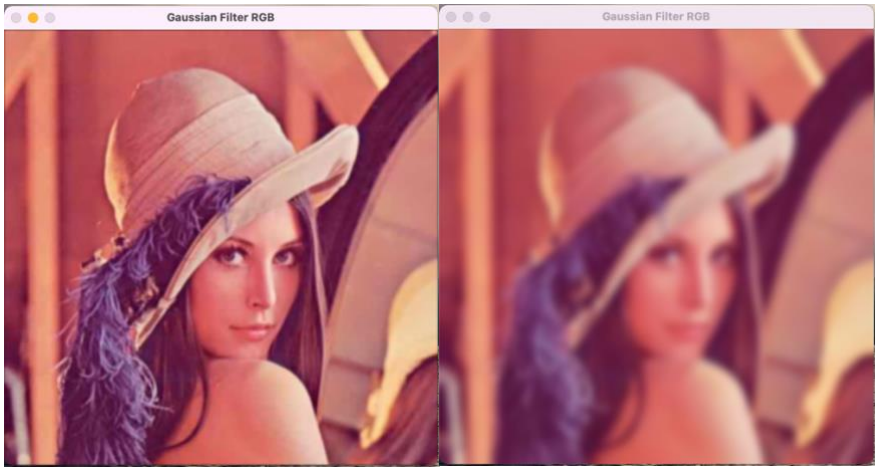
설명:

2-1와 같이, Zero mean Gaussian Filter인 $w(s,t)$를 계산해 kernel행렬에 저장한다.

1-2에서, 1-1의 모든 과정을 똑같이 하되, rgb를 고려해 세 채널 각각 계산을 해줬듯이 2-1의 모든 과정을 rgb 세 채널에 대해 따로 계산을 진행한다.

실행 결과

1) gaussianfilter(input, 9,1,1, "zero-paddle");
2) gaussianfilter(input, 20,5,5, "mirroring");



3-1. SobelGray.cpp

코드 목적:

흑백 이미지에 대해 Sobel filter 를 구현한다. boundary processing 은 mirroring 을 사용한다.

코드 흐름:

1) 이미지 읽어와 흑백으로 전환
2) sobelfilter 함수를 통해 흑백 이미지를 sobel filtering
3) 결과 이미지 새로운 창으로 띄운다

함수 설명: sobelfilter (const Mat input)

매개변수:

input: 입력 이미지 행렬

함수 목적:

input 이미지를 sobel filtering 한다.

설명:

1)

```
Mat kernel_Sx = *(Mat_<float>(3, 3) << -1,0,1, -2,0,2, -1,0,1);
Mat kernel_Sy = *(Mat_<float>(3, 3) << -1,-2,-1, 0,0,0, 1,2,1);
```

Sobel filtering 은 x 축, y 축으로의 1 차 편미분을 이용한다. 이를 근사화 한 Sx, Sy 에 대해(아래그림 참고), $I_x = |S_x * I|$, $I_y = |S_y * I|$이고, output 이미지 M(x,y)는$\sqrt{{I_x}^2 + {I_y}^2}$이다.

S_x

−1	0	1
−2	0	2
−1	0	1

S_y

−1	−2	−1
0	0	0
1	2	1

행렬 kernel_Sx 에 S_x를, kernel_Sy 에 S_y를 직접 할당해 선언해준다.

2)

(i,j)픽셀의 값을 정할 때, kernel 을 모두 접근하기 위해 이중 for 문을 돌린다. mirroring 이기 때문에, 이미지 범위를 벗어난 경우 tempa, tempb 에 mirroring 으로 값을 가져올 픽셀의 인덱스를 저장한다.

$S_x * I$ 를 계산해 sumX 에, $S_y * I$ 를 계산해 sumY 에 더해준다.

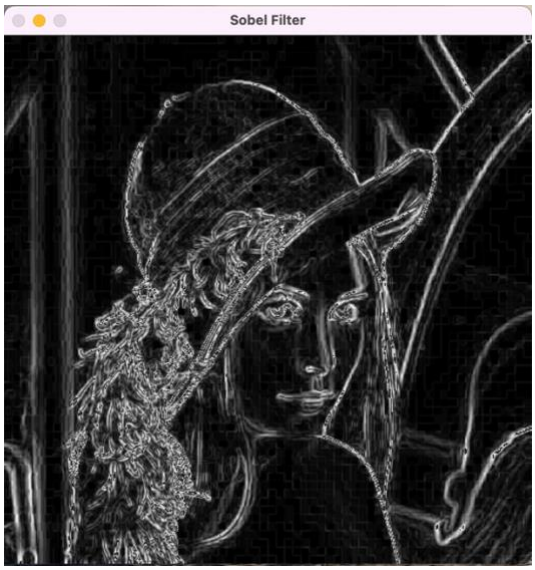
M(x,y)=$\sqrt{{I_x}^2 + {I_y}^2}$이므로, kernel 안의 모든 픽셀을 돌아 I_x와 I_y를 구한 후 계산하여 output 이미지의 색 데이터에 저장한다.

```
for (int a = -n; a <= n; a++) {
        for (int b = -n; b <= n; b++) {
            if (i + a > row - 1)      tempa = i - a;
            else if (i + a < 0)       tempa = -(i + a);
            else                      tempa = i + a;

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

            sumX += kernel_Sx.at<float>(a+n, b+n)*(float)(input.at<G>(tempa, tempb));
            sumY += kernel_Sy.at<float>(a+n, b+n)*(float)(input.at<G>(tempa, tempb));
        }
}
            output.at<G>(i, j) = sqrt( sumX*sumX + sumY*sumY );
```

실행 결과



3-2. SobelRGB.cpp

코드 목적:

컬러 이미지에 대해 Sobel filter 를 구현한다. boundary processing 은 mirroring 을 사용한다.

코드 흐름:

1) 이미지 읽어 행렬에 저장
2) sobelfilter 함수를 통해 흑백 이미지를 sobel filtering
3) 결과 이미지 새로운 창으로 띄운다

함수 설명: sobelfilter (const Mat input)

매개변수:

input: 입력 이미지 행렬. 컬러이기 때문에 3 개의 채널을 가지고 있다.

함수 목적:

input 이미지를 sobel filtering 한다.

설명:

1) 3-1의 과정을, r g b 세 채널로 나눠 각각 계산해준다

```
        sumX_r += kernel_Sx.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[0]);
        sumY_r += kernel_Sy.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[0]);

        sumX_g += kernel_Sx.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[1]);
        sumY_g += kernel_Sy.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[1]);

        sumX_b += kernel_Sx.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[2]);
        sumY_b += kernel_Sy.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[2]);
    }
}

float Mr=sqrt( sumX_r*sumX_r + sumY_r*sumY_r );
float Mg=sqrt( sumX_g*sumX_g + sumY_g*sumY_g );
float Mb=sqrt( sumX_b*sumX_b + sumY_b*sumY_b );
```

2) RGB 이미지에 대해, sobel filter의 결과는 세 채널의 결과값의 평균으로, 흑백이다.

For color image,

$$M(x,y) = \left(M_R(x,y) + M_G(x,y) + M_B(x,y)\right)/3$$

```
output.at<G>(i, j) = (Mr+Mb+Mg)/3;
```

실행 결과

4-1. LaplacianGray.cpp

코드 목적:

흑백 이미지에 대해 Laplacian filter 를 구현한다. boundary processing 은 mirroring 을 사용한다.

코드 흐름:

1) 이미지 읽어와 흑백으로 전환
2) Laplacian함수를 통해 흑백 이미지를 Laplacian filtering
3) 결과 이미지 새로운 창으로 띄운다

함수 설명: laplacianfilter (const Mat input)

매개변수:

input: 입력 이미지 행렬. 흑백

함수 목적:

input 이미지를 laplacianfilter 하여 결과 이미지(흑백)을 리턴한다.

설명:

1)

```
Mat kernel = *(Mat_<float>(3, 3) << 0,1,0, 1,-4,1, 0,1,0);
```

Laplacian filter 는 2 차 미분을 근사화 한 filter 인 L 을 사용하고(아래 그림 참고), 결과 이미지 O 는 $O = |L * I|$이다. kernel 에 L 을 직접적으로 값을 넣어주어 정의한다.

0	1	0
1	-4	1
0	1	0

2)

(i,j)픽셀의 값을 정할 때, kernel 을 모두 접근하기 위해 이중 for 문을 돌린다. mirroring 이기 때문에, 이미지 범위를 벗어난 경우 tempa, tempb 에 mirroring 으로 값을 가져올 픽셀의 인덱스를 저장한다.

$L * I$를 계산해 sum 에 더해준다. abs()함수로 절댓값을 계산해주고, 계산된 값을 Output 이미지의 해당 픽셀에 넣어준다

```
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
        float sum = 0.0;
        for (int a = -n; a <= n; a++) {
            for (int b = -n; b <= n; b++) {
                    if (i + a > row - 1)        tempa = i - a;
                else if (i + a < 0)  tempa = -(i + a);
                else                            tempa = i + a;

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

                    sum += kernel.at<float>(a+n, b+n)*(float)(input.at<G>(tempa, tempb));}
            }
            sum=abs(sum);
            sum*=5;
            if(sum>255)sum=255;
            output.at<G>(i, j) = (G)sum;
        }
    }
```

3) 색 데이터 보정

```
sum*=5;
if(sum>255)sum=255;
```

실행해보면 아래 사진과 같이, 데이터들이 대체로 작은 값을 가져 결과가 눈으로 잘 보이지 않는 것을 알 수 있다.

따라서 연산이 끝나고 output 에 저장하기 전, sum 의 값을 5 씩 곱해준다. 하지만 별도의 변수로 가장 큰 색상 값을 구해보면 248 이 나온다. 따라서 5 를 곱했을 때, 색이 0~255 의 범위를 벗어나는 것이 있을 수 있으므로, 255 보다 큰 경우 255 로 고정시켜준다.

실행 결과

4-2. LaplacianRGB.cpp

코드 목적:

컬러 이미지에 대해 Laplacian filter 를 구현한다. boundary processing 은 mirroring 을 사용한다.

코드 흐름:

1) 이미지 읽어와 input 변수에 저장
2) Laplacian함수를 통해 컬러 이미지를 Laplacian filtering
3) 결과 이미지 새로운 창으로 띄운다

함수 설명: laplacianfilter (const Mat input)

매개변수:

input: 입력 이미지 행렬. 흑백

함수 목적:

input 이미지를 Laplacian filtering 하여 결과 이미지(흑백)을 리턴한다.

설명

1)

```
Mat input_gray;
cvtColor(input, input_gray, CV_RGB2GRAY);
Mat output = Mat::zeros(row, col, input_gray.type() );
```

laplacianfilter 의 반환 이미지는 흑백이므로, output 행렬을 선언하고 초기화 할 때, type 으로 흑백 이미지의 타입을 넣는다.

2) 컬러 이미지이므로, 4-1의 모든 연산을 똑같이 하되, rgb 세 채널에 대해 각각 진행한다.

```
            sum_r += kernel.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[0]);
            sum_g += kernel.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[1]);
            sum_b += kernel.at<float>(a+n, b+n)*(float)(input.at<C>(tempa, tempb)[2]);
        }
    }
    sum_r = abs(sum_r)*5;
    sum_g= abs(sum_g)*5;
    sum_b = abs(sum_b)*5;
```

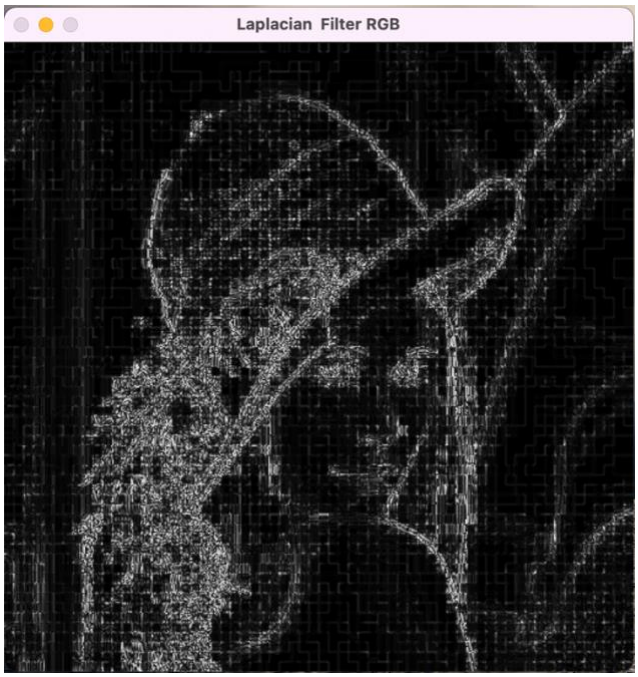
3) RGB 이미지에 대해, Laplacian filter의 결과는 세 채널의 결과값의 평균이다.

For color image,

$$O(x,y) = \left(O_R(x,y) + O_G(x,y) + O_B(x,y)\right)/3$$

```
float out=(G)((sum_r+sum_g+sum_b)/3);
```

실행 결과



5-1. Gaussian_sep_Gray.cpp

코드 목적:

2 번의 Gaussian Filtering 에서, 연산량을 줄이기 위해 $w(s,t)$를 두개의 filter 로 나눠 계산한다. kernel 크기와 x 축에 대한 표준편차, y 축에 대한 표준편차, image boundary 의 픽셀 처리 방법에 따라 다른 결과 이미지를 출력한다.

코드 흐름:

1) 이미지 파일 불러와 흑백으로 전환한다.
2) gaussianfilter 함수를 통해 Gaussian filtering한 결과 이미지를 구한다
3) 가장 큰 색 데이터를 이용해 색을 맞춰준다
4) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

Gaussianfilter_sep (const Mat input, int n, float sigmaT, float sigmaS, const char* opt)

매개변수:

input: input 흑백 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

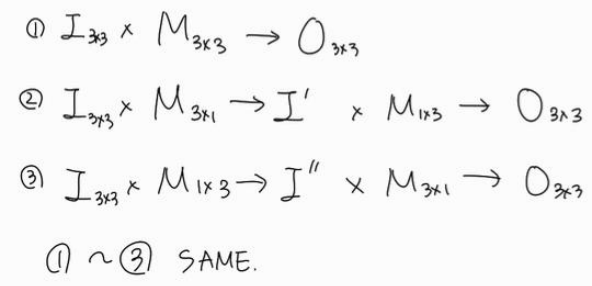
"zero-paddle" "mirroring" "adjustkernel" 중 하나

함수 목적:

input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

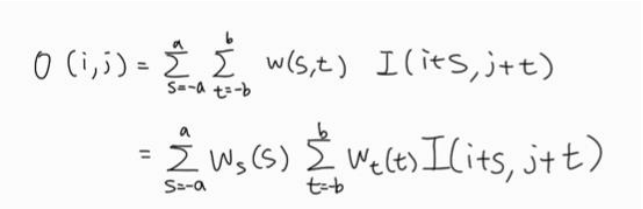
설명:

I 를 filtering 하여 O 를 만든다고 하자.

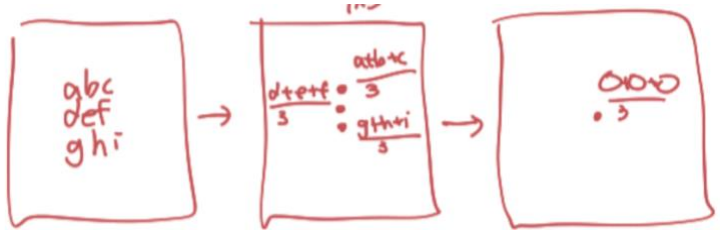


다음과 같이, 1 번~3 번은 모두 같은 결과 O 를 나타낸다. 하지만 1 번은 3*3 행렬과 3*3 행렬의 행렬곱으로 복잡한 연산이 필요하다. 하지만 이 M 을 1*3 행렬과 3*1 행렬로 나눠 I 와 곱한다면, 간단한 연산 두번으로 같은 결과를 만들어낼 수 있다. 따라서 이 코드에서는 2,3 번과 같이 계산을 두번으로 나눌 것이다.

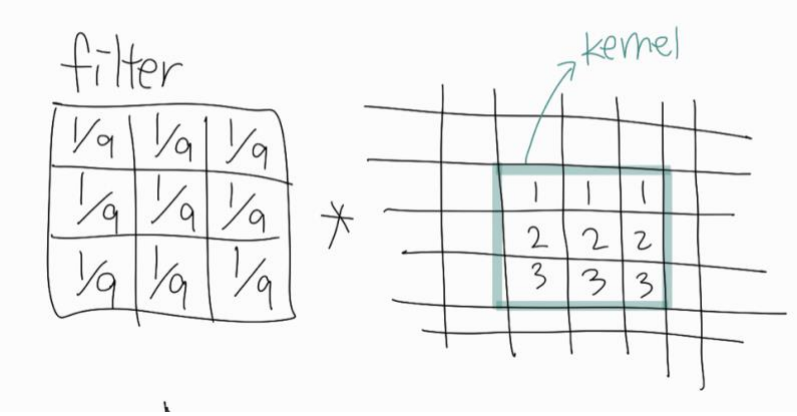
Gaussian Filter 의 식을 가져와보면 다음과 같다. 첫번째 제시된 식이 기존의 계산 식이고, 아래의 식으로 나누어 연산이 가능하다.



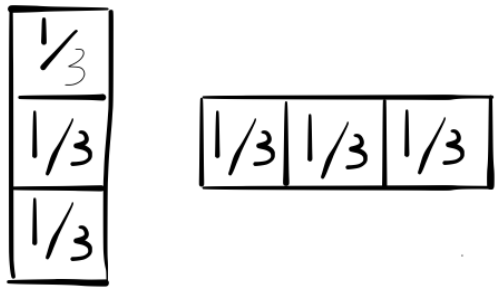
즉 1*(2n+1)크기의 $w_t(t)$로 x 축에 대한 계산을 먼저 하고, 이를 다시 $w_s(s)$를 이용해 y 축에 대한 계산을 진행한다.



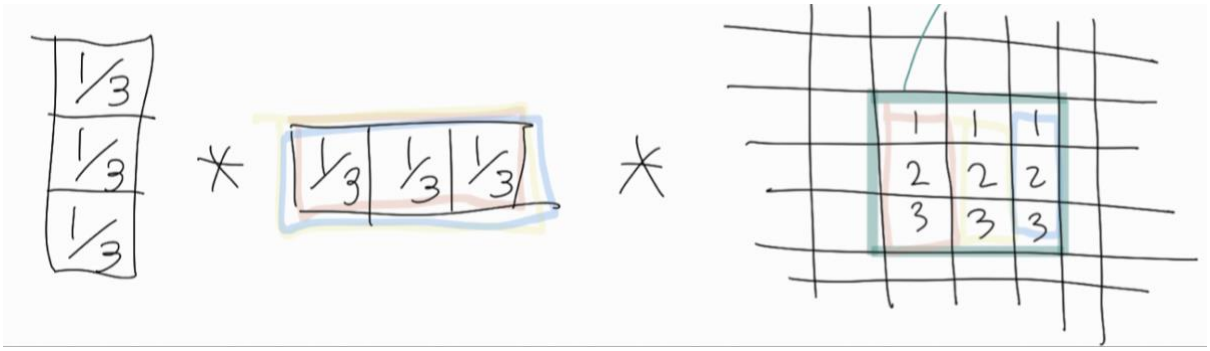
이때, $w_t(t)$까지 연산 완료한 (2n+1)*1 크기의 데이터를 temp 행렬에 잠시 저장할 것이다.
Gaussian filter 의 kernel 의 값의 양상과는 전혀 다르지만 임의의 예시를 들어 $w(s,t)$를 $w_t(t)$와 $w_s(s)$로 나누어 계산하는 과정을 이해해보자.



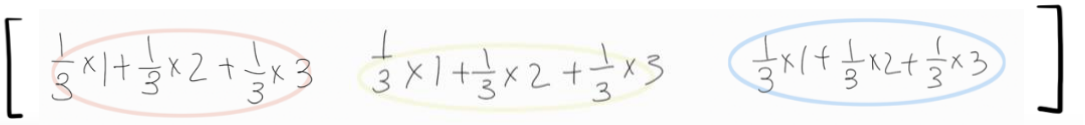
왼쪽에 n=1 인 filter 가 정해져 있다. 오른쪽은 이미지의 일부로, kernel 크기에 해당하는 부분만 데이터를 적어놓았다. 두개의 (2*n+1)* (2*n+1)크기의 행렬의 곱은 연산의 부담이 크다.
왼쪽의 필터는 아래와 같이 분리가 가능하다.

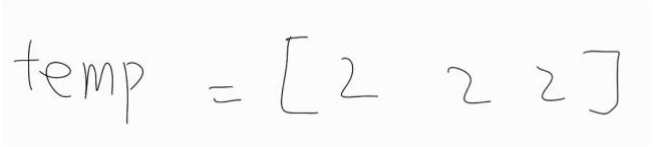


따라서 전체 식은 다음과 같다.

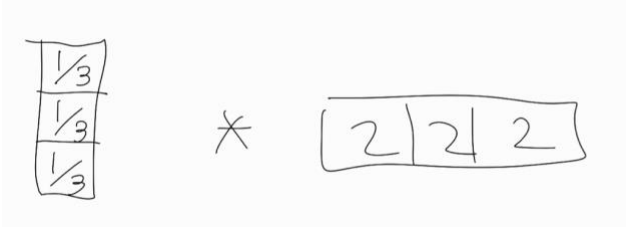


여기서, 앞에서부터가 아닌 뒤의 두 1*3 행렬과 3*3 행렬을 곱한다.

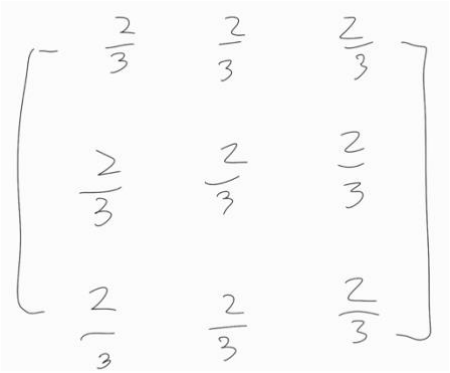




이제 이 temp 를 맨 앞의 3*1 행렬과 곱해준다.



=



맨 처음 두 3*3 행렬을 계산해도 위와 같은 결과가 나온다.

이제 이 과정을 코드로 표현해보자.

1)

```
Mat kernel_s = Mat::zeros(kernel_size,1,CV_32F);
Mat kernel_t = Mat::zeros(1,kernel_size,CV_32F);

denom = 0.0;      //분모
for (int a = -n; a <= n; a++) {
    float value1 = exp(-(pow(a, 2) / (2 * pow(sigmaS, 2))));
    kernel_s.at<float>(a+n, 0) = value1;
    denom += value1;
}
for (int a = -n; a <= n; a++) {
    kernel_s.at<float>(a+n, 0) /= denom;
}
denom = 0.0;      //분모
for (int b = -n; b <= n; b++) {
    float value1 = exp(-(pow(b, 2) / (2 * pow(sigmaT, 2))));
    kernel_t.at<float>(0, b+n) = value1;
    denom += value1;
}
for (int b = -n; b <= n; b++) {
    kernel_t.at<float>(0, b+n) /= denom;
}
```

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

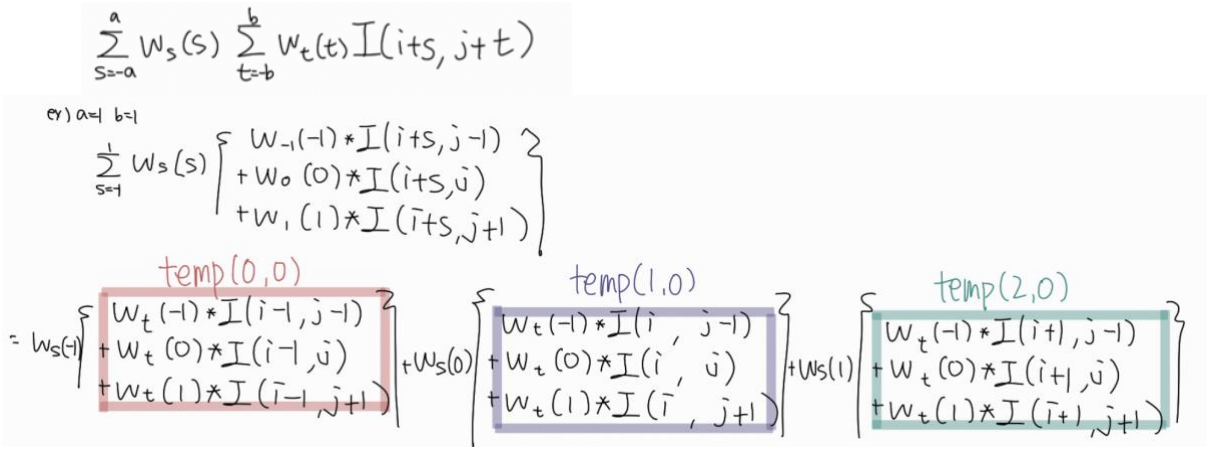
$w_t(t)$와 $w_s(s)$를 계산해 kernel_s 와 kernel_t 에 저장한다. value1 을 계산하고 계속해서 demon 에 더해준 뒤 마지막에 demon 으로 나눈다.

2) 1번에서 세 boundary processing방식을 비교하여 정리하였으니, 여기에서는 zero-padding의 코드만 보겠다.

```
if (!strcmp(opt, "zero-paddle")) {
    float sum1 = 0.0;
    Mat temp =Mat::zeros(kernel_size,1,CV_32F);

    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)) {
                temp.at<float>(a+n, 0)+=kernel_t.at<float>(0, b+n)*(float)(input.at<G>(i + a, j + b));
            }
        }
    }
    for (int a = -n; a <= n; a++) {
        sum1+=kernel_s.at<float>(a+n, 0)*temp.at<float>(a+n,0);
    }
    output.at<G>(i, j) = (G)(sum1);
    if(maxN<output.at<G>(i, j)) maxN=output.at<G>(i, j);
```

kernel 을 돌며 우선 kernel_t 로 x 축에 대한 연산을 먼저 하고, temp 에 넣어준다.
이후 이 temp 와 kernel_s 를 곱해 output 을 구한다. temp 가 저장하는 것은 다음과 같다



3)

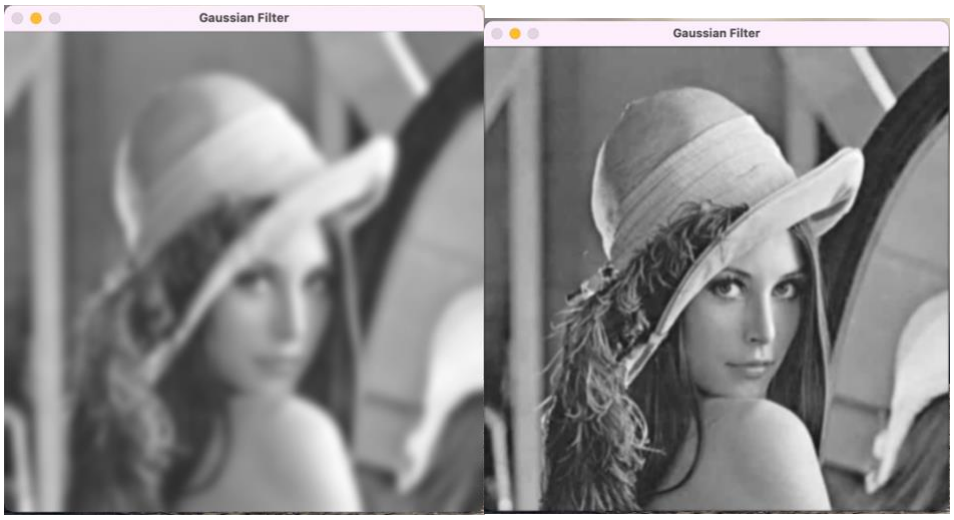
```
if(maxN<output.at<G>(i, j)) maxN=output.at<G>(i, j);
```

```
//main 함수 안:
for (int i = 0; i < input_gray.rows; i++) {
        for (int j = 0; j < input_gray.cols; j++) {
                output.at<G>(i, j) = (G)(output.at<G>(i, j)*255/maxN);
            }
      }
```

함수의 매개변수에 따라 몇몇 경우 전체적으로 데이터가 줄어들어 어두워진다. 이를 방지하기 위해 output 을 구할 때 가장 큰 색 데이터의 값 maxN 을 구해두고, 마지막 출력 전에 모든 픽셀의 값에 대해 maxN 으로 나누고 255 를 곱해 색 데이터가 0~255 에 골고루 퍼지게 만든다.

실행 결과

1) Gaussianfilter_sep(input_gray, 15,5,5, "mirroring")
2) Gaussianfilter_sep(input_gray, 10,1,1, "zero-paddle");



5-2. Gaussian_sep_RGB.cpp

코드 목적:

2 번의 Gaussian Filtering 에서, 연산량을 줄이기 위해 $w(s,t)$를 두개의 filter 로 나눠 계산한다. kernel 크기와 x 축에 대한 표준편차, y 축에 대한 표준편차, image boundary 의 픽셀 처리 방법에 따라 다른 결과 이미지를 출력한다. 이미지는 컬러 이미지이다.

코드 흐름:

1) 이미지 파일 불러온다
2) gaussianfilter 함수를 통해 Gaussian filtering한 결과 이미지를 구한다
3) 가장 큰 색 데이터를 이용해 색을 맞춰준다
4) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

Gaussianfilter_sep (const Mat input, int n, float sigmaT, float sigmaS, const char* opt)

매개변수:

input: input 컬러 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

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

함수 목적:

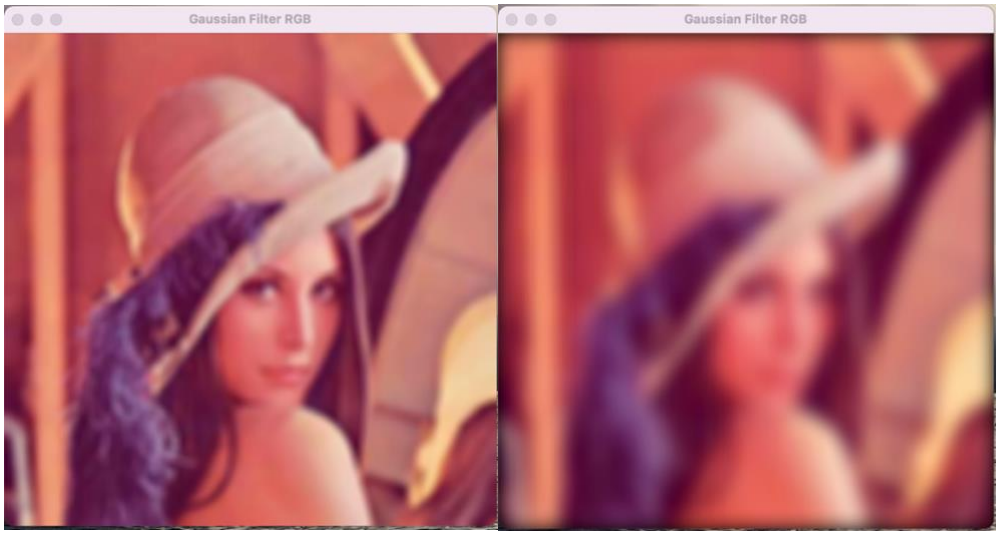
input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

설명:

5-1 과 같은 연산을 수행한다. 컬러이미지이기 때문에, r g b 세 채널에 대해 각각 연산해준다.

실행 결과

1) Gaussianfilter_sep(input, 6,5,5, "adjustkernel");
2) Gaussianfilter_sep(input, 20,10,10, "zero-paddle");



6-1. UnsharpMaskGray.cpp

코드 목적:

흑백 이미지에 대해 Unsharp Masking 를 구현한다.

kernel 크기와 표준편차, image boundary 의 픽셀 처리 방법에 따라 Gaussian filtering 을 진행하고, 이를 low-pass filtering 으로 하는 Unsharp Masking 을 진행한다.

코드 흐름:

1) 이미지 파일 불러와 흑백으로 전환한다.
2) UnsharpMaskfilter 함수를 통해 Unsharp Masking한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

UnsharpMaskfilter (const Mat input, int n, float sigmaT, float sigmaS, const char* opt, float k);

매개변수:

input: 흑백 input 이미지

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT: x 축에 대한 표준편차

sigmaS: y 축에 대한 표준편차

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

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

k: losspass filter 반영 비율

함수 목적:

input 을 opt 적용해 Unsharp Masking 한 결과를 반환(Mat)

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

매개변수:

input: input 흑백 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

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

함수 목적:

input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

설명:

Unsharp Masking 은 input 이미지 I 에서 low-pass filtering 한 L 을 k 의 비율로 반영해 빼준 뒤, 전체 상태 유지를 위해(1-k)로 나눈다.

이 코드에선 low-pass filter 로 gaussianfilter 를 사용한다. UnsharpMaskfilter 함수 안에서 gaussianfilter 함수를 호출해 low-pass filtering 한 이미지를 lowpassfilter 행렬에 저장한다

```
Mat lowpassfilter;
    lowpassfilter = gaussianfilter(input, n,sigmaT,sigmaS, opt);
```

모든 픽셀에 대해 $\boldsymbol{output} = (\mathbf{1} - \boldsymbol{kL})/(\mathbf{1} - \boldsymbol{k})$를 계산한다. 계산한 결과가 색의 범위 0~255 를 벗어나는 경우 0 또는 255 로 고정한다.

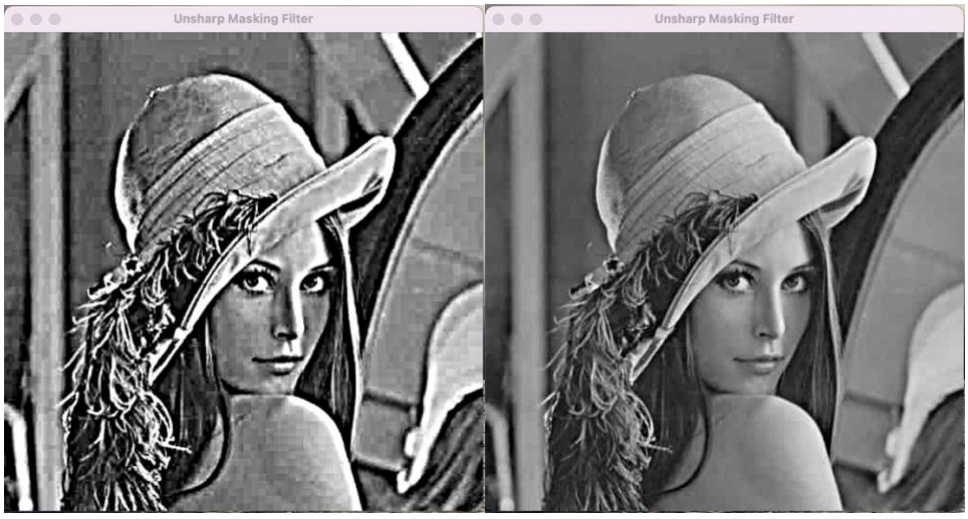
```
float temp;
for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            for(int l=0;l<3;l++){
                temp=(input.at<G>(i, j) -   lowpassfilter.at<G>(i, j) * k)/(1 - k);

                if (temp < 0) temp = 0;
                else if (temp > 255) temp = 255;

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

실행 결과

1) UnsharpMaskfilter(input_gray, 10, 7, 7, "mirroring", 0.8);
2) UnsharpMaskfilter(input_gray, 10, 7, 7, "mirroring", 0.3);



6-2. UnsharpMaskRGB.cpp

코드 목적:

흑백 이미지에 대해 Unsharp Masking 를 구현한다.

kernel 크기와 표준편차, image boundary 의 픽셀 처리 방법에 따라 Gaussian filtering 을 진행하고, 이를 low-pass filtering 으로 하는 Unsharp Masking 을 진행한다.

코드 흐름:

1) 이미지 파일 불러온다
2) UnsharpMaskfilter 함수를 통해 Unsharp Masking한 결과 이미지를 구한다
3) 새로운 창에 input이미지와 결과 이미지를 띄운다

함수 설명:

UnsharpMaskfilter (const Mat input, int n, float sigmaT, float sigmaS, const char* opt, float k);

매개변수:

input: 컬러 input 이미지

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT: x 축에 대한 표준편차

sigmaS: y 축에 대한 표준편차

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

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

k: losspass filter 반영 비율

함수 목적:

input 을 opt 적용해 Unsharp Masking 한 결과를 반환(Mat)

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

매개변수:

input: input 컬러 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

sigmaT, sigmaS: σt σs 표준편차

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

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

함수 목적:

input 을 opt 적용해 Gaussian filtering 한 결과를 반환(Mat)

설명:

6-1 과 같은 과정을 진행한다. 컬러 이미지이기 때문에 세 채널에 대해 각각 연산을 처리한다.

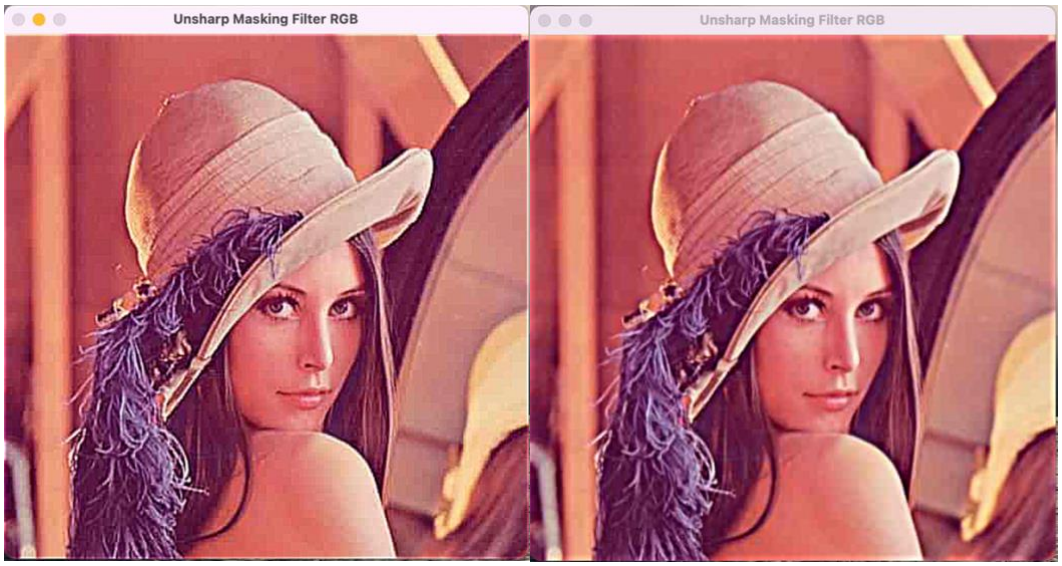
```
for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            for(int l=0;l<3;l++){
                temp=(input.at<G>(i, j) - lowpassfilter.at<G>(i, j) * k)/(1 - k);

                if (temp < 0) temp = 0;
                else if (temp > 255) temp = 255;

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

실행 결과

1) UnsharpMaskfilter(input, 2, 3, 3, "zero-paddle", 0.7);
2) UnsharpMaskfilter(input, 6, 8, 8, "zero-paddle", 0.4);



참고자료:

오픈 SW 프로젝트 Lec04 수업자료