Open Software Project

Assignment 03 Technical Report

1976235 오진솔

1. MeanFilterGray

`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 paddle>
	
	픽셀이 타겟 픽셀로부터 가로세로 -n ~ n 위치에 있고 이미지의 바운더리 안쪽에 있다면
	
	sum1 에 kernelvalue * 해당 픽셀의 색상값을 더한다.
	
	Kernelvalue는 1 / (2n+1)(2n+1)의 값을 가지고 있다.
	
	따라서 sum1의 값은 범위 내에 있는 픽셀들의 색상 값의 평균이 되고, 바운더리 바깥쪽의 픽셀은 더하지 않으므로 0으로 취급한 것과 같다.

`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));`
	<mirroring>
	
	커널 범위 내의 픽셀이 이미지의 바운더리를 벗어날 경우, 이미지의 경계를 기준으로 대칭 위치에 있는 점의 값을 대신 사용한다.
	
	코드에서는 (픽셀의 위치)<0일 경우 바르게 구현되지만 (픽셀의 위치)>이미지가 끝나는 지점일 경우 이미지의 경계가 아닌 타겟 픽셀을 기준으로 미러링을 수행함.

```
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);
```

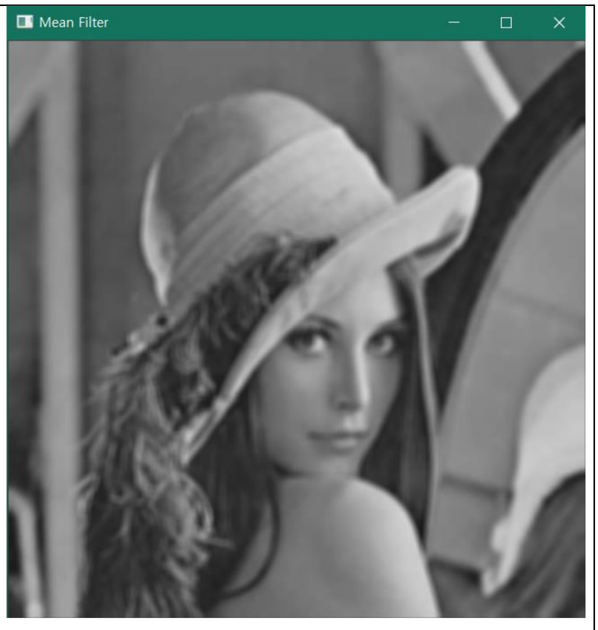
<adjustkernel>

zero paddle과 같은 작업을 수행한 후, 결과값을 kernelvalue*유효한 픽셀 개수로 나누어준다.

만약 커널 픽셀들이 모두 이미지 바운더리 안쪽에 있다면 sum2 값은 1이 되어 결과는 sum1의 값이 된다.

이미지 바운더리 바깥쪽에 있는 픽셀이 있다면 sum2 값은 1보다 작아지기 때문에 결과값은 커진다.(색상이 밝아진다)

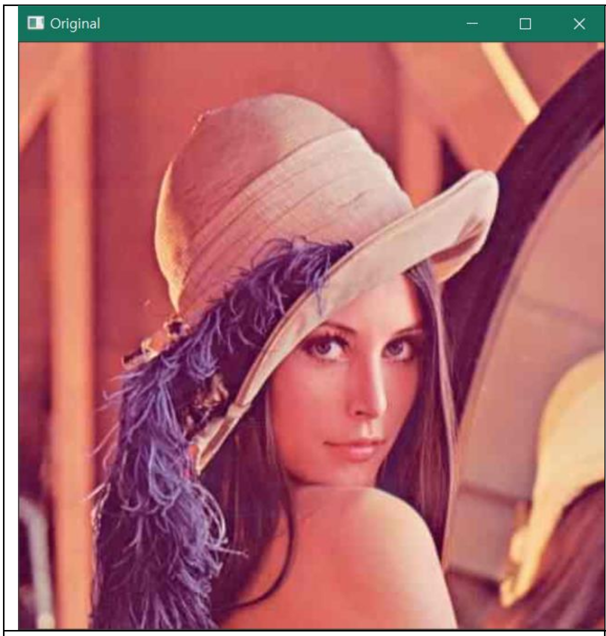
original



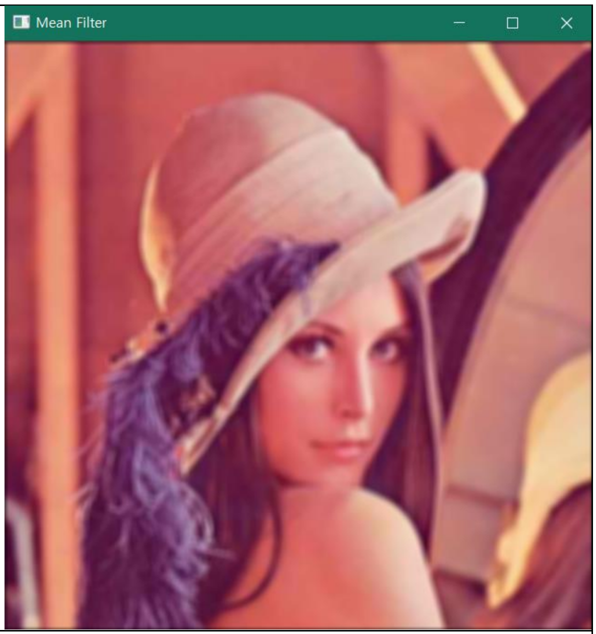
mean filter (7*7, mirroring)

2. MeanFilterRGB

sum1_r += kernelvalue*(float)(input.at<C>(tempa, tempb)[0]);
sum1_g += kernelvalue*(float)(input.at<C>(tempa, tempb)[1]);
sum1_b += kernelvalue*(float)(input.at<C>(tempa, tempb)[2]);
	MeanfilterGray와 같은 작업을 R,G,B 각각에 대해 수행한다.
	이 외의 작업은 모두 같다.



original



mean filter (7*7, zero-paddle)

3. Gaussian

```
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;}
}
```

범위 내의 픽셀에 대해

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

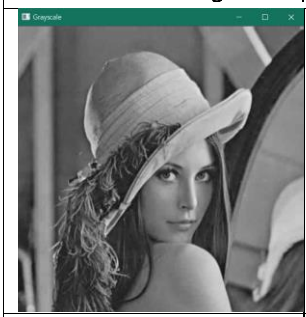
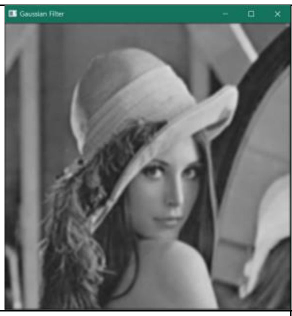
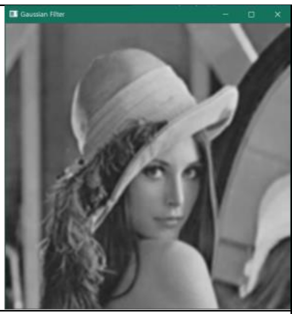
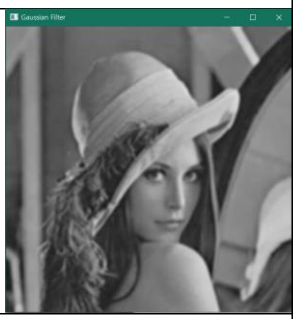
를 계산해서 kernel 에 저장한다.₩

kernel 의 값을 각각 denom 으로 나누어서 normalize 한다.

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

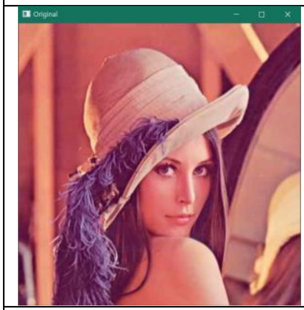
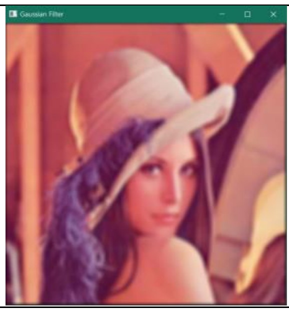
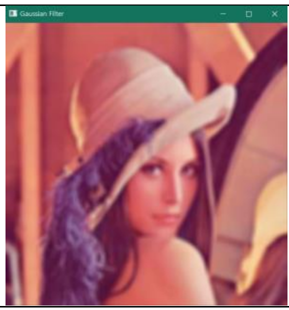
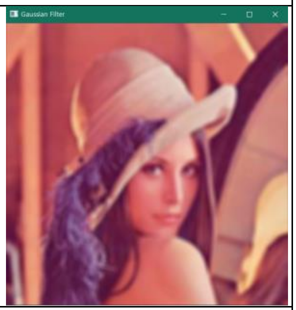
모든 kernelvalue 의 값이 같았던 uniform mean filtering 과 달리 kernelvalue 가 각각 다른 값을 가지므로 위치에 맞게 kernelvalue 를 설정한 뒤 같은 작업을 수행한다.

Gaussian filtering examples (7*7, $\sigma_s = 3$, σ_t=3)

			
original	zero-paddle	mirroring	adjustkernel

4. GaussianRGB

GaussianGray와 같은 작업을 R,G,B 각각에 대해 수행한다. 이 외의 작업은 모두 같다.

Gaussian filtering examples (11*11, $\sigma_s = 5$, σ_t=5)			
			
original	zero-paddle	mirroring	adjustkernel

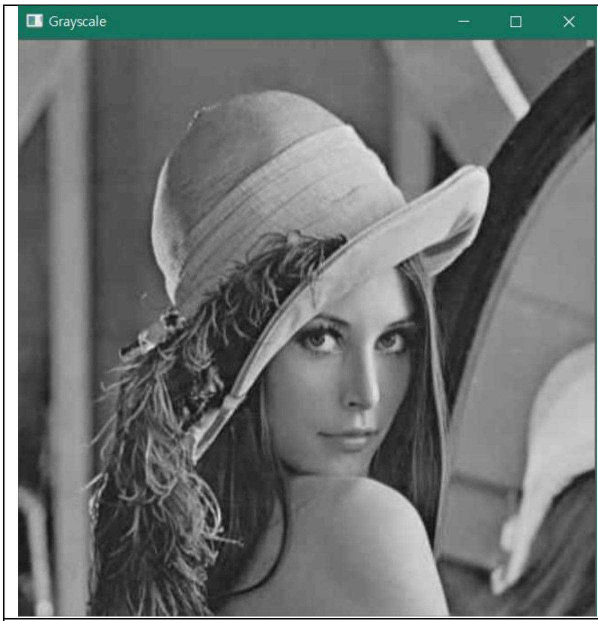
5. SobelGray

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 };	Sx, Sy 초기화
(중간생략)	

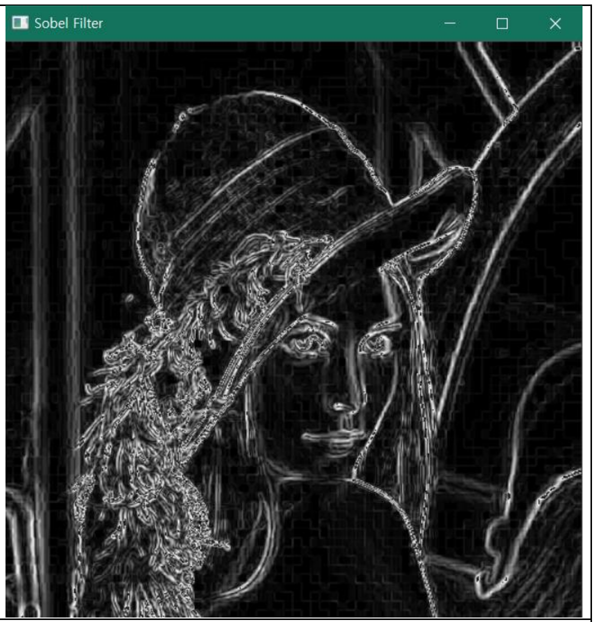
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
        float sumX = 0.0;
        float sumY = 0.0;
        for (int a = -n; a <= n; a++) {
            for (int b = -n; b <= n; b++) {
                if (i + a > row - 1)   tempa = i - a;
                else if (i + a < 0)   tempa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

sumX += Sx[a + n][b + n] * (float)(input.at<G>(tempa, tempb));	$\sum Sx * I$
sumY += Sy[a + n][b + n] * (float)(input.at<G>(tempa, tempb));	$\sum Sy * I$
} }	

result = (G)sqrt(pow(sumX, 2) + pow(sumY, 2));
if (result < 0) result = 0;
if (result > 255) result = 255;
output.at<G>(i, j) = result;
	$M(x,y) = \sqrt{{I_x}^2 + {I_y}^2}$
	값이 0~255 사이의 값을 가지게끔 조정



original



sobel filtering

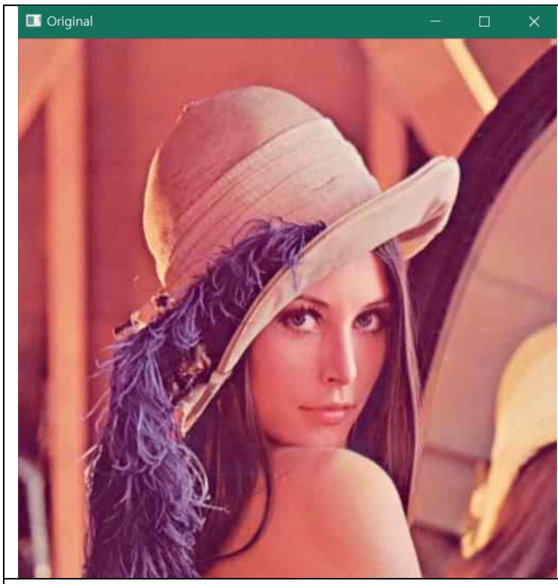
6. SobelRGB

```
sumX_r += Sx[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[0]);
sumX_g += Sx[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[1]);
sumX_b += Sx[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[2]);
sumY_r += Sy[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[0]);
sumY_g += Sy[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[1]);
sumY_b += Sy[a + n][b + n] *
      (float)(input.at<C>(tempa, tempb)[2]);
```

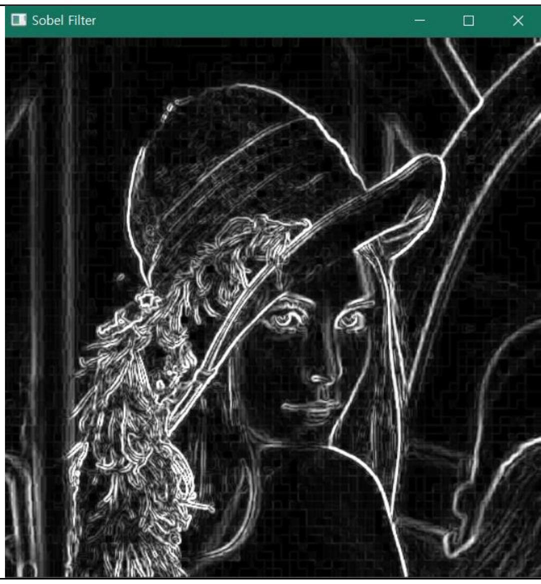
r, g, b, 각각에 대해 같은 작업을 수행한다.

```
result = (result_r + result_g + result_b)/3;
   output.at<C>(i, j)[0] = result;
   output.at<C>(i, j)[1] = result;
   output.at<C>(i, j)[2] = result;
```

r, g, b 결과를 모두 더해 평균을 구하고
output의 r, g, b 값에 대입한다.



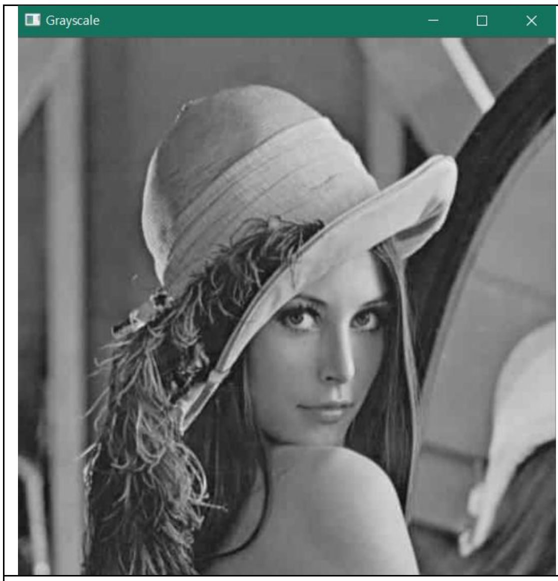
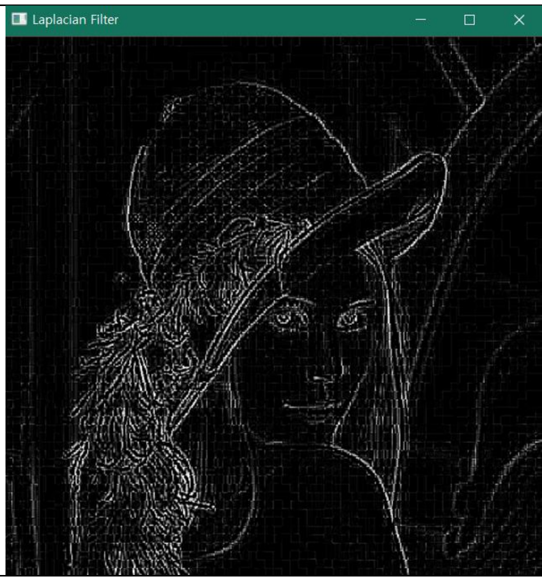
Original



Sobel Filter

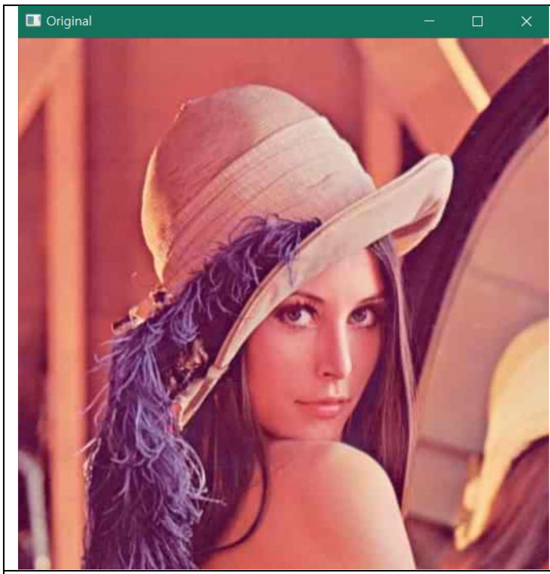
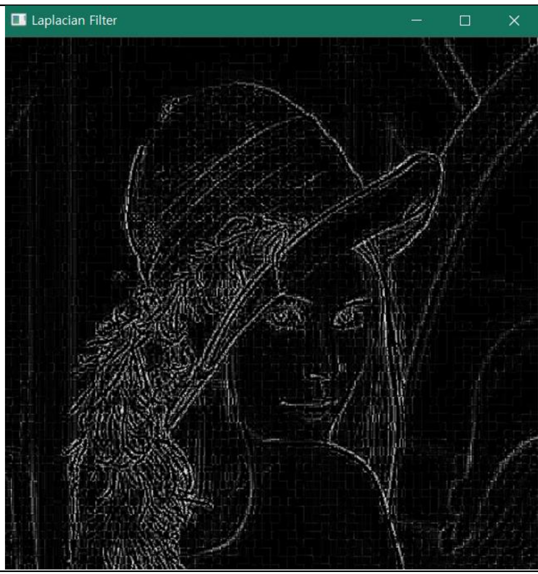
7. Laplacian Gray

int L[3][3] = {0, 1, 0, 1, -4, 1, 0, 1, 0};
 	
(중간 생략)
 	
sum += L[a + n][b + n]
        *(float)(input.at<G>(tempa, tempb));
 	
(중간 생략)
 	
sum = sum * 5;
if (sum < 0) sum = 0;
if (sum > 255) sum = 255;
output.at<G>(i, j) = (G)sum;
	Laplacian kernel matrix L 초기화
	
	Laplacian filter 결과값을 계산한다.
	
	선명한 결과를 얻기 위해 상수를 곱한다.
	색상 범위를 보정한다.

	
Original	Laplacian Filter(x5)

8. Laplacian RGB

sum_r += L[a + n][b + n] *
(float)(input.at<C>(tempa, tempb)[0]);
 	
sum_g += L[a + n][b + n] *
(float)(input.at<C>(tempa, tempb)[1]);
 	
sum_b += L[a + n][b + n] *
(float)(input.at<C>(tempa, tempb)[2]);
 	
sum = (sum_r + sum_g + sum_b)/3;
sum *= 5;
output.at<C>(i, j)[0] = sum;
output.at<C>(i, j)[1] = sum;
output.at<C>(i, j)[2] = sum;
	r, g, b 각각에 대해 같은 작업을 수행한다.
	
	모두 더해서 평균을 구하고
	상수 5를 곱한다.
	output에 대입한다.

	
Original	Laplacian Filter (x5)

9. GaussianGray_sep,

```
for (int a = -n; a <= n; a++) {
  float value_s = exp(-(pow(a, 2) /
                        (2 * pow(sigmaS, 2))));
  kernel_s.at<float>(a + n, 0) = value_s;
  denom_s += value_s;
}
for (int b = -n; b <= n; b++) {
   float value_t = exp(-(pow(b, 2) /
                        (2 * pow(sigmaT, 2))));
   kernel_t.at<float>(0, b + n) = value_t;
   denom_t += value_t;
}
for (int a = -n; a <= n; a++) {
   kernel_s.at<float>(a + n, 0) /= denom_s;}
for (int b = -n; b <= n; b++) {
   kernel_t.at<float>(0, b + n) /= denom_t;}
Mat output = Mat::zeros(row, col, input.type());
Mat temp = input.clone();
if (!strcmp(opt, "zero-paddle")) {
   for (int i = 0; i < row; i++) {
      for (int j = 0; j < col; j++) {
         float sum1 = 0.0;
         for (int a = -n; a <= n; a++) {
            if ((i+a <= row-1) &&(i+a >= 0)
               sum1 += kernel_s.at<float>(a + n, 0)
                       * (float)(input.at<G>(i + a, j));
         }
         temp.at<G>(i, j) = sum1;
      }
   }
   for (int i = 0; i < row; i++) {
      for (int j = 0; j < col; j++) {
         float sum1 = 0.0;
          for (int b = -n; b <= n; b++) {
            if ((j+b <= col-1) && (j+b >= 0))
               sum1 += kernel_t.at<float>(0, b + n)
                       * (float)(temp.at<G>(i, j + b));
         }
         output.at<G>(i, j) = sum1;
      }
   }
}
```

기존의 방식이 row * col * kernelSize * kernelSize 만큼의 복잡도를 가지는 데 비해 이 방식은 row * col * (kernelSize + kernelSize)의 복잡도를 가지므로 효율적이다.

결과는 일반 Gaussian filter와 같고,
체감할 수 있을 만큼 속도가 빨라진 것을 확인하였다.

s, t kernel matrix를 각각 생성한다.

$$w_s(s) = \frac{1}{\sum_{m=-a}^{a} \exp\left(-\frac{m^2}{2\sigma_s{}^2}\right)} \exp\left(-\frac{s^2}{2\sigma_s{}^2}\right)$$

$$w_t(t) = \frac{1}{\sum_{n=-b}^{b} \exp\left(-\frac{n^2}{2\sigma_t{}^2}\right)} \exp\left(-\frac{t^2}{2\sigma_t{}^2}\right)$$

(예시)zero paddle
input에 w(s)를 적용하고 temp에 저장한다.

temp에 w(t)를 적용하고 output에 저장한다.

mirroring과 adjustkernel 방식도 같은 방법으로 kernel을 분리해서 적용함. (큰 차이가 없으므로 생략합니다)

10.GaussianRGB_sep

sum_r += kernel_s.at<float>(a + n, 0) * (float)(input.at<C>(i + a, j)[0]);
sum_g += kernel_s.at<float>(a + n, 0) * (float)(input.at<C>(i + a, j)[1]);
sum_b += kernel_s.at<float>(a + n, 0) * (float)(input.at<C>(i + a, j)[2]);
 	
(중략)
 	
temp.at<C>(i, j)[0] = sum_r;
temp.at<C>(i, j)[1] = sum_g;
temp.at<C>(i, j)[2] = sum_b;
 	
(중략)
 	
sum_r += kernel_t.at<float>(0, b + n) * (float)(temp.at<C>(i, j + b)[0]);
sum_g += kernel_t.at<float>(0, b + n) * (float)(temp.at<C>(i, j + b)[1]);
sum_b += kernel_t.at<float>(0, b + n) * (float)(temp.at<C>(i, j + b)[2]);
 	
(중략)
 	
output.at<C>(i, j)[0] = sum_r;
output.at<C>(i, j)[1] = sum_g;
output.at<C>(i, j)[2] = sum_b;
	GaussianGray_sep를 r,g,b 각각에 대해 수행한다.
	
	(기존에 sum1이었던 부분을 sum_r, sum_g, sum_b로 나누어 각각 수행)
	
	결과는 일반 Gaussian filter(RGB)와 같고, 체감할 수 있을 만큼 속도가 빨라진 것을 확인하였다.

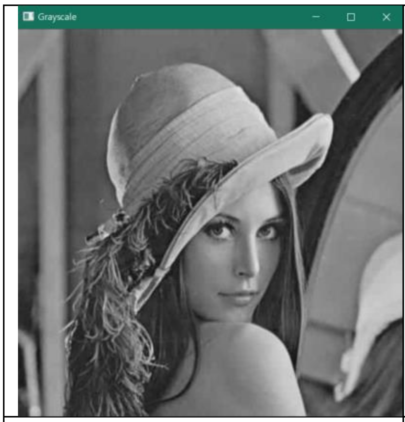
11. unsharpGray

```
Mat unsharpMask(const Mat input, int n, float sigmaT, float sigmaS, const char* opt, float k) {

    Mat L = gaussianfilter(input, n, sigmaT, sigmaS, opt);
    Mat output(input.rows, input.cols, input.type());
    float temp;

    for (int i = 0; i < input.rows; i++) {
        for (int j = 0; j < input.cols; j++) {
            temp = abs(input.at<G>(i, j) - (k * L.at<G>(i, j)))
                                                / (1 - k);
            if (temp > 255) temp = 255;
            output.at<G>(i, j) = temp;
        }
    }
    return output;
}
```

가우시안 필터를 적용한다.

output = abs(I - kL) / (1 - k)

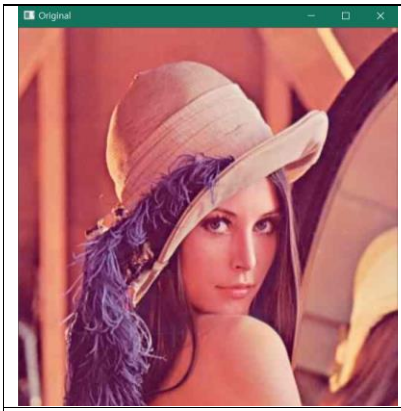
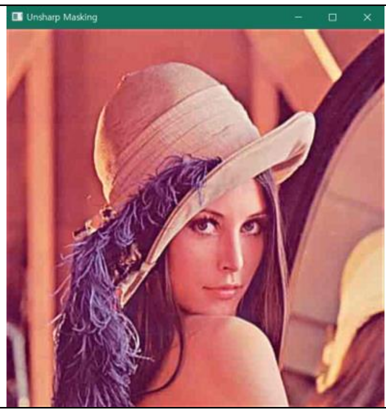
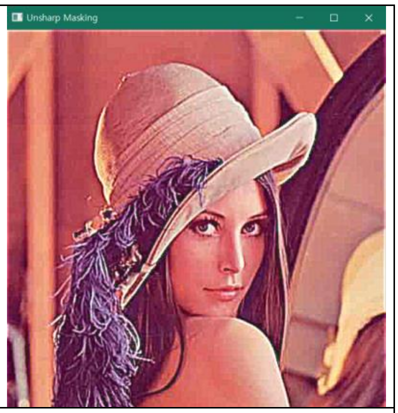
범위를 벗어난 값을 조정한다.

		
original	k = 0.4	k = 0.7

12. unsharpRGB

```
Mat L = gaussianfilter(input, n, sigmaT, sigmaS, opt);
Mat output(input.rows, input.cols, input.type());
float temp[3];

for (int i = 0; i < input.rows; i++) {
    for (int j = 0; j < input.cols; j++) {
        for (int c = 0; c < 3; c++) {
            temp[c] = abs(input.at<C>(i, j)[c]
                          - (k * L.at<C>(i, j)[c])) / (1 - k);
             if (temp[c] > 255) temp[c] = 255;
             output.at<C>(i, j)[c] = temp[c];
        }
    }
}
return output;
```

r,g,b 각각에 대해 같은 작업을 수행한다.

		
original	k = 0.4	k = 0.7