1. adaptivethreshold.cpp

코드 목적:

Moving Average 를 이용한 Adaptive Thresholding 을 적용한다

코드 흐름:

1) writing.jpg를 불러와 흑백 이미지를 만든다.
2) adaptive_thres()함수를 호출한다.
3) 각 pixel에 대해 kernel의(주변 pixel의) 평균을 구하고, 평균에 b를 곱한 값을 Threshold로 한다.
4) Adaptive Thresholding을 완료한 이미지를 새로운 창에 출력한다.

함수 설명: adaptive_thres(const Mat input, int n, float b)

매개변수:

input: input 이미지 행렬

n: kernel 의 크기 (2n+1)x(2n+1)

b: weight

함수 목적: kernel 의 mean 에 b 의 가중치를 준 것을 Threshold 로 하여 input 이미지를 Adaptive Thresholding 하여 반환

내용 설명:

Adaptive Thresholding 의 계산 과정은 다음과 같다.

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

픽셀 (i ,j)에 대해, kernel 에 해당하는 모든 pixel 의 평균을 구한다

$$T(i,j) = b \times m(i.j)$$

구한 평균에 가중치 b 를 곱한 값을 기준점(Threshold)로 정한다

$$g(i,j) = \begin{cases} 1 & if\ I(i,j) > T(i,j) \\ 0 & otherwise \end{cases}$$

값이 Threshold 보다 크면 1, 작으면 0 으로 만든다.

즉, 매 픽셀마다 다른 Threshold 를 가지게 되어, 주변보다 상대적으로 밝은 부분은 1, 상대적으로 많이 어두운 부분은 0 으로 적용되는 것이다.

코드 설명:

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

주변값들의 평균을 구하는 meanfilter 의 가중치 w(s,j)는 N(=2*n+1)에 대해 kernel 크기 N x N 을 가지고, 모든 값이 1/(N*N)이다. 이를 kernel 행렬에 저장한다. 하지만 모든 kernel 에 대해 같은 가중치를 가지기 때문에, 간편하게 kernelvalue 에 하나의 값을 넣어 사용하자.

```
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++) {
                        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));
                                }
                         }
                }
```

이미지의 모든 pixel (i,j)에 접근한다.
kernel 안의 pixel 들에 접근하여 계산을 진행한다. zero-padding 을 사용하므로 이미지의 범위 안에 들어가는 kernel 에 대해서만, 색 데이터에 가중치를 곱한 값을 sum1 에 더한다.
이미지 범위에 들어가지 않는 부분은 더해지지 않아 0 으로 반영된다

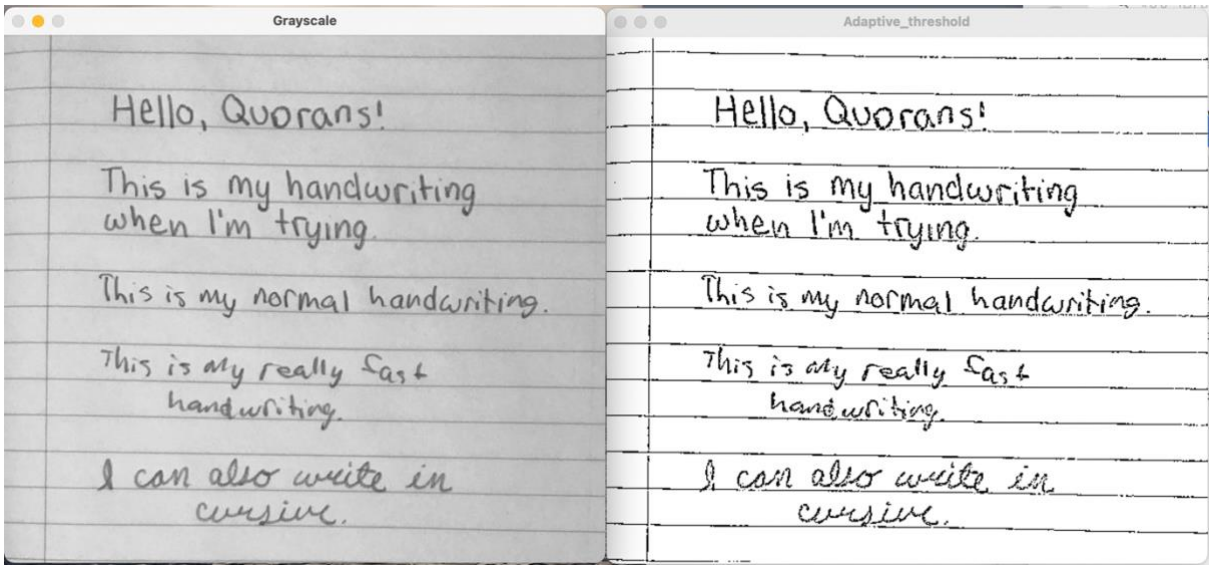
```
float temp = bnumber*(G)sum1;

if(input.at<G>(i, j)>temp) output.at<G>(i, j)=(G)255;
else output.at<G>(i, j)=(G)0;
```

계산한 mean 값에 b 를 곱한 값을 temp 에 저장한다. 이 temp 가 데이터를 1 또는 0 으로 보내는 기준점이 된다. 이미지의 색 데이터가 temp 보다 크다면 255(흰색), 작다면 0(검은색)의 값을 가지게 된다.

실행 결과

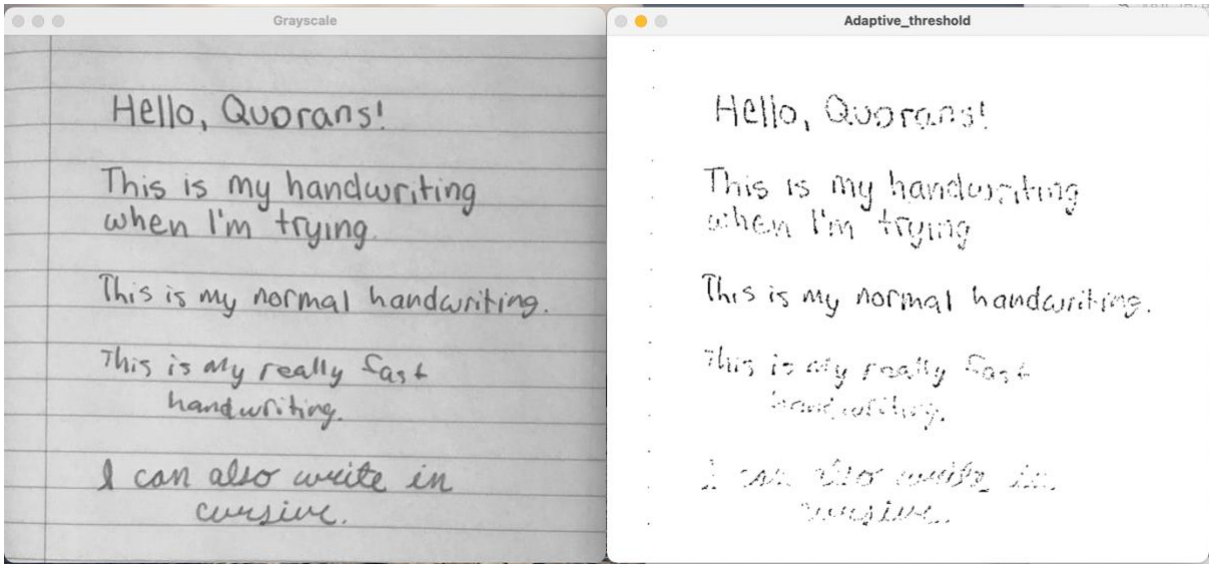
1)

adaptive_thres(input_gray, 2, 0.9)



2)

adaptive_thres(input_gray, 3, 0.7)



실행 결과, 원하지 않는 정보인 종이 일부분의 어두운 부분이 제거되고, 주변보다 뚜렷하게 진하고 중요한 정보인 글씨만 남았다. (2)와 같이 b 를 작게 설정하면 255(흰색)으로 판단하기 위한 기준이 높아져, (1)의 실행결과에 비해 더욱 적은 글씨만이 남은 것을 확인할 수 있다. 입력 이미지에 따라 적절히 b 를 결정하는 것이 중요하다

2. kmeans.cpp

코드 목적:

흑백 이미지에 대한 K-means Clustering 을 진행한다. intensity 만을 고려한 Clustering 과 position 까지 고려한 Clustering 에 대해 각각 결과를 도출한다.

코드 흐름:

1) lena.jpg를 불러와 흑백 이미지를 만든다.
2) Kmeans()함수를 호출해 intensity similarity를 고려해 pixel을 grouping
3) KmeansPosition()함수를 호출해 intensity similarity를 고려해 pixel을 grouping
4) 결과 이미지를 새로운 창으로 출력

함수 설명: Kmeans(const Mat input,int clusterCount, int attempts)

매개변수:

input: input 이미지 행렬

clusterCount: 군집화할 개수(k)

attempts: 실행되는 알고리즘의 실행 횟수

함수 목적:

input 이미지의 데이터를 clustering 하고, 결과값에 맞게 output 이미지를 만들어 반환한다.

KmeansPosition(const Mat input,int clusterCount, int attempts ,float sigmaX, float sigmaY)

매개변수:

input: input 이미지 행렬

clusterCount: 군집화할 개수(k)

attempts: 실행되는 알고리즘의 실행 횟수

sigma: intensity 와 position 의 다른 ratio 를 맞춰 주기 위한 상수

함수 목적:

input 이미지의 데이터를 intensity 와 position 에 따라 clustering 하고, 결과값에 맞게 output 이미지를 만들어 반환한다.

double cv::kmeans(InputArray data, int K, InputOutputArray bestLabels,
TermCriteria criteria, int attempts, int flags, OutputArray centers = noArray())

활용 예: kmeans(samples, clusterCount, labels, TermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);

매개 변수:

samples : input 이미지

clusterCount (K) : 군집화할 개수

(반환값) labels : 라벨에 대한 배열. 가장 가까운 center 의 인덱스

criteria : 반복을 종료할 조건 (type, max_iter 최대반복횟수, epsilon 정확도)

attempts : 다른 초기 라벨링을 사용하면서 실행되는 알고리즘의 실행 횟수를 지정하는 플래그

flags : 초기값을 잡을 중심에 대한 플래그로써 cv2.KMEANS_PP_CENTERS 와 cv2.KMEANS_RANDOM_CENTERS 중 하나

(반환값) centers : 클러스터의 중심이 저장된 배열

함수 목적:

samples 를 k 개의 group 으로 cluster 하여 해당하는 그룹의 center 인덱스와 center 의 색을 반환한다.

설명:

1) Intensity 만을 고려하는 Clustering 인 Kmeans()

```
Mat samples(input.rows * input.cols, 1, CV_32F);
for (int y = 0; y < input.rows; y++)
        for (int x = 0; x < input.cols; x++)
                samples.at<float>(y + x*input.rows, 0) = (float)(input.at<uchar>(y, x));1
```

kmeans 함수를 활용하기 위해 samples 행렬에 데이터들을 일차원 배열로 저장한다.

```
kmeans(samples, clusterCount, labels, TermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000,
0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

labels 행렬에 각 인덱스에 해당하는 pixel 이 어떤 cluster center 의 group 에 해당되는지, 그 center 의 인덱스가 저장되어 반환된다.

centers 행렬에 각 cluster center 들의 값이 저장되어 반환된다.

```
for (int y = 0; y < input.rows; y++){
            for (int x = 0; x < input.cols; x++){
                int cluster_idx = labels.at<int>(y + x*input.rows, 0);
                new_image.at <uchar>(y, x) = (uchar)centers.at<float>(cluster_idx, 0);
            }
    }
```

모든 픽셀(i,j)에 접근한다. labels 행렬에 접근해 (i,j)가 포함되는 group 의 center 의 인덱스를 가져온다. centers 행렬에 접근해 그 center 의 색을 알아내, output 이미지의 (i,j)픽셀에 저장한다.

즉, 모든 픽셀이 자신이 포함된 cluster center 의 값으로 바뀌어, 총 k 가지의 색만으로 구성된 output 이미지가 완성되었다.

2) Intensity 와 Positions 을 고려하는 Clustering 인 KmeansPositions()

```
Mat samples(input.rows * input.cols, 3, CV_32F);
for (int y = 0; y < input.rows; y++){
            for (int x = 0; x < input.cols; x++){
                    samples.at<float>(y * input.cols+x, 0) = (float)(input.at<G>(y, x));
                    samples.at<float>(y * input.cols+x, 1) =(float)(y)/sigmaY;
                    samples.at<float>(y * input.cols+x, 2) = (float)(x)/sigmaX;
            }
}
```

position 까지 고려하는 clustering 은 3D 의 Feature space 를 가진다. 이는 intensity 와 x 축 위치, y 축 위치로 구성된다. 이 때, intensity 는 MAX 가 255 지만, x,y 는 이미지의 크기가 MAX 이다. 이 요소들을 공평하게 반영하기 위해, sigma 를 계산에 포함시킨다.

```
float sigmaX=(float)input.cols/255.0;
float sigmaY=(float)input.rows/255.0;
```

sigma 는 고정된 상수로, 이미지 크기를 255(intensity 의 최대값)으로 나눈 값이다. sigma 로 x 좌표, y 좌표를 나눠주면, 본인의 최대값으로 나뉘어 normalized 되고, 255 가 곱해지며 intensity 와 같은 비율로 계산에 반영되게 된다.

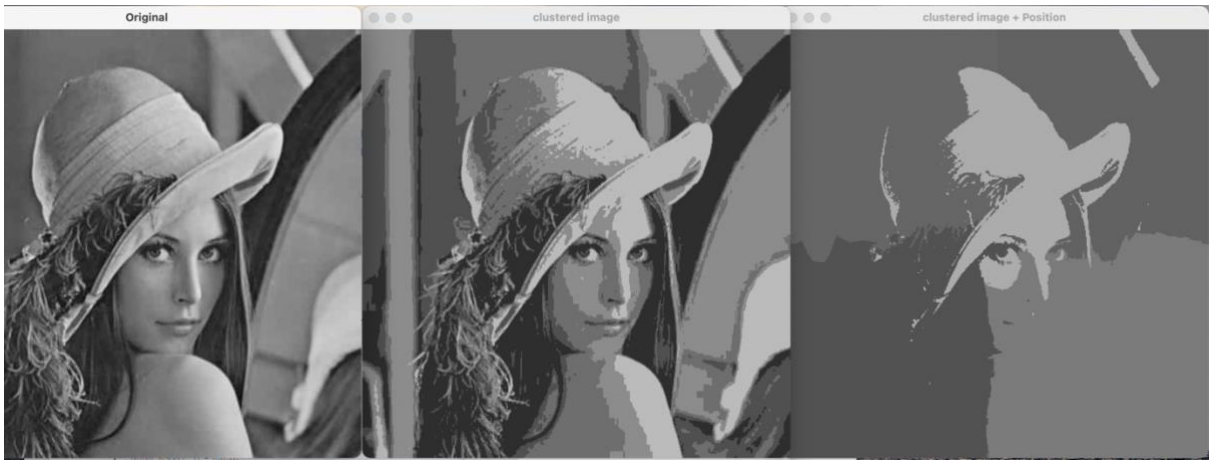
```
kmeans(samples, clusterCount, labels, TermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

```
for (int y = 0; y < input.rows; y++){
        for (int x = 0; x < input.cols; x++){
            int cluster_idx = labels.at<int>(y * input.cols+x, 0);
            new_image.at<G>(y, x) = (G)((centers.at<float>(cluster_idx, 0)));
        }
}
```

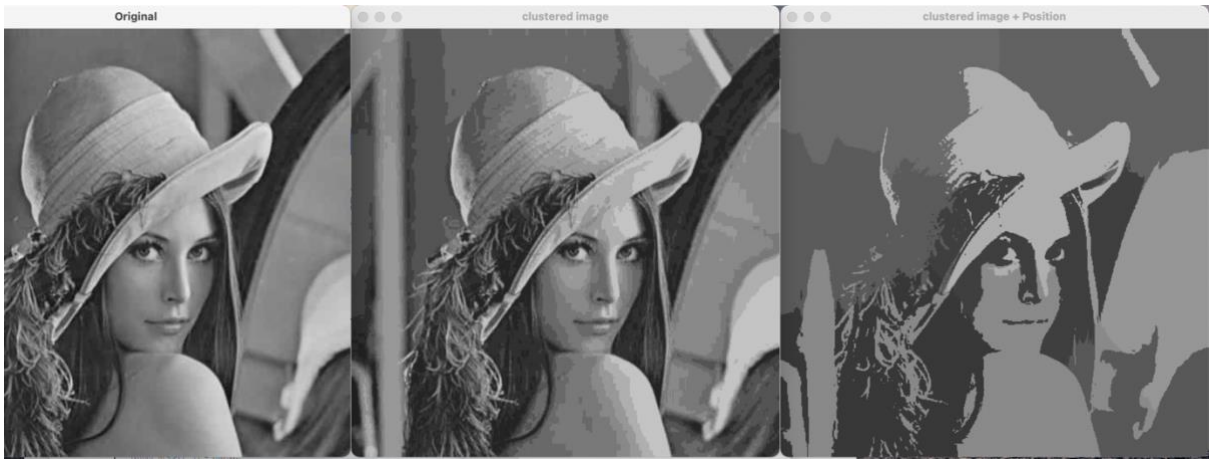
위와 마찬가지로, kmeans 함수를 통해 labels 행렬엔 해당 center 의 인덱스가, centers 행렬엔 각 cluster center 들의 값이 반환된다. 이를 통해 clustering 을 진행해 ouput 이미지의 값을 넣어준다.

실행 결과:

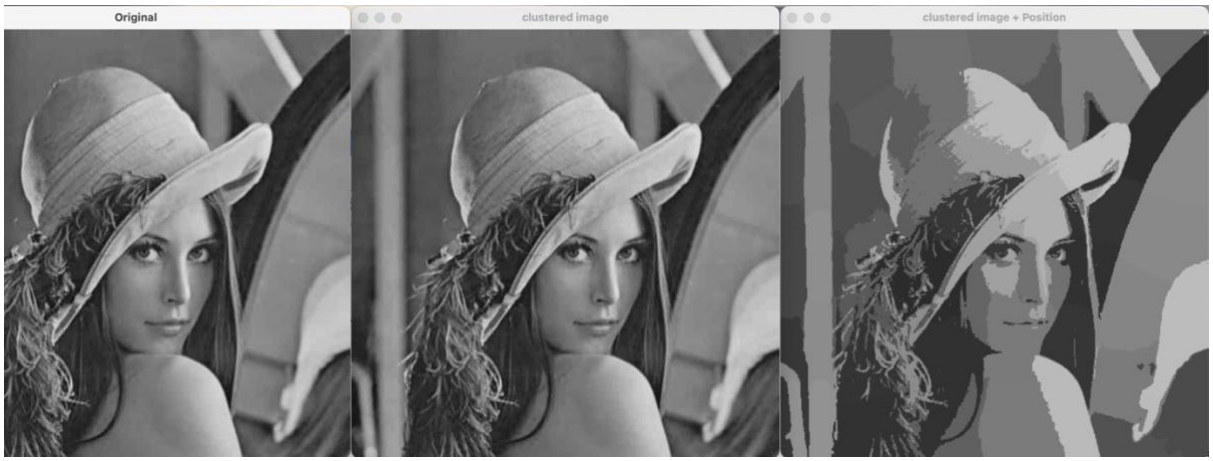
int clusterCount = 4;



int clusterCount = 10;



int clusterCount = 50;



intensity 만 고려한 이미지와 다르게, position+intensity clustering 은 가까이 있는 비슷한 색들이 같은 그룹에 속함을 확인 가능하다

3. kmeansRGB.cpp

코드 목적:

RGB 이미지에 대한 K-means Clustering 을 진행한다. intensity 만을 고려한 Clustering 과 position 까지 고려한 Clustering 에 대해 각각 결과를 도출한다.

코드 흐름:

1) lena.jpg를 불러온다.
2) Kmeans()함수를 호출해 intensity similarity를 고려해 pixel을 grouping
3) KmeansPosition()함수를 호출해 intensity similarity를 고려해 pixel을 grouping

4) 결과 이미지를 새로운 창으로 출력

설명:함수의 기능과 연산, 원리는 2번 코드와 비슷하다. 흑백과 컬러의 차이점을 위주로 설명한다
설명:

1) Intensity 만을 고려하는 Clustering 인 Kmeans()

```
for (int y = 0; y < input.rows; y++){
        for (int x = 0; x < input.cols; x++){
                samples.at<float>(y + x*input.rows, 0) = (float)(input.at<C>(y, x)[0]);
                samples.at<float>(y + x*input.rows, 1) = (float)(input.at<C>(y, x)[1]);
                samples.at<float>(y + x*input.rows, 2) = (float)(input.at<C>(y, x)[2]);
        }
    }
    kmeans(samples, clusterCount, labels, TermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 10000, 0.0001), attempts,
KMEANS_PP_CENTERS, centers);
    for (int y = 0; y < input.rows; y++){
        for (int x = 0; x < input.cols; x++)
        {
            int cluster_idx = labels.at<int>(y + x*input.rows, 0);
                new_image.at <C>(y, x)[0] = (G)centers.at<float>(cluster_idx, 0);
                new_image.at <C>(y, x)[1] = (G)centers.at<float>(cluster_idx, 1);
                new_image.at <C>(y, x)[2] = (G)centers.at<float>(cluster_idx, 2);
        }
    }
```

R, G, B 에 대해 kmean 연산을 각각 진행한다. 각 R G B 에 대한 각각의 clustering 결과가 이차원 배열에 저장된다.

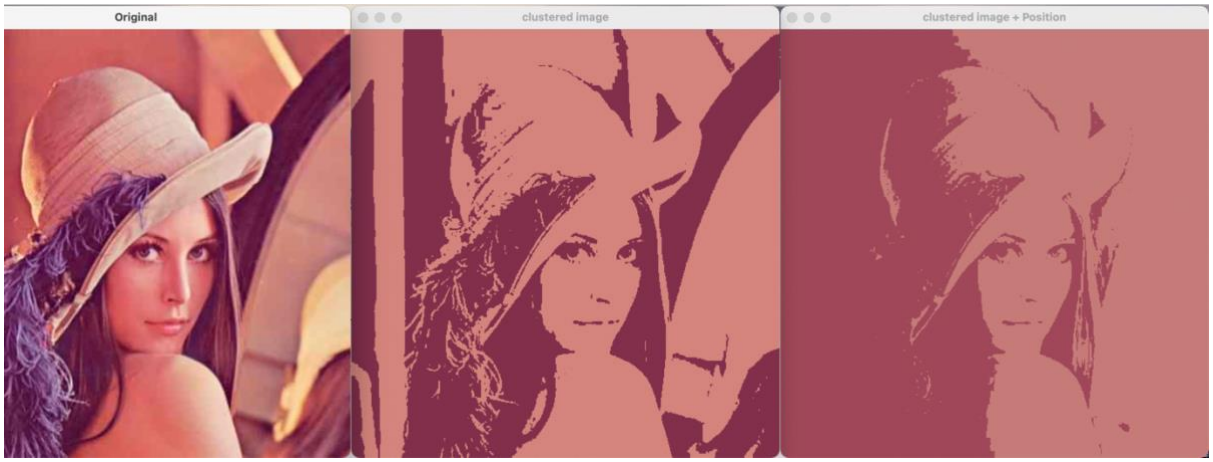
2) Intensity 와 Positions 을 고려하는 Clustering 인 KmeansPositions()

```
Mat samples(input.rows * input.cols, 5, CV_32F);
    for (int y = 0; y < input.rows; y++){
            for (int x = 0; x < input.cols; x++){
                samples.at<float>(y * input.cols+x, 0) = (float)(input.at<C>(y, x)[0]);
                samples.at<float>(y * input.cols+x, 1) = (float)(input.at<C>(y, x)[1]);
                samples.at<float>(y * input.cols+x, 2) = (float)(input.at<C>(y, x)[2]);

                samples.at<float>(y * input.cols+x, 3) =(float)(y) / sigmaY;
                samples.at<float>(y * input.cols+x, 4) = (float)(x)/sigmaX;
            }
    }
```

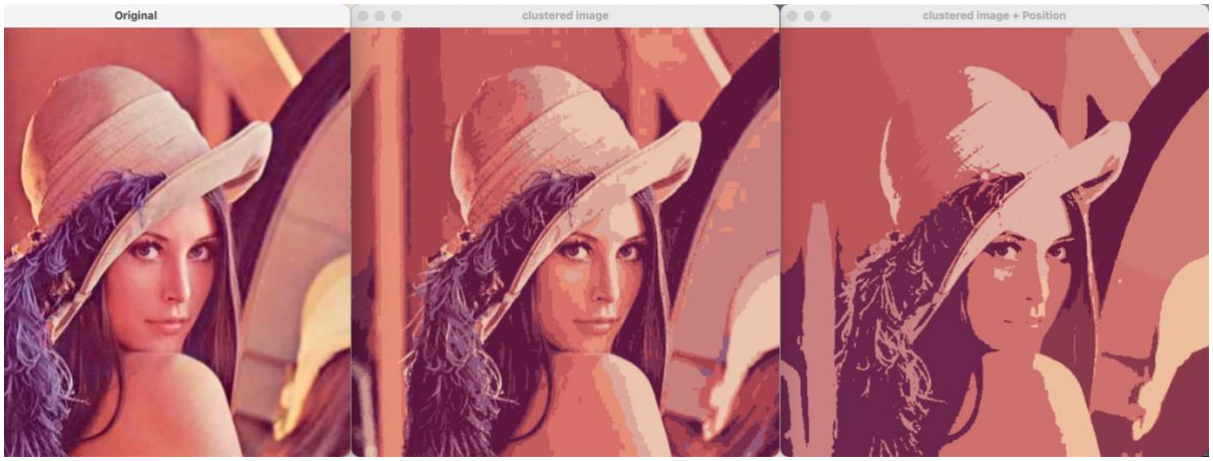
컬러 이미지에 대해 intensity 와 position 모두 고려하기 위해서는 5 차원의 배열이 필요하다. 각 r, g, b, x 좌표, y 좌표에 대하여 clustering 을 따로 따로 진행한다.

실행 결과:

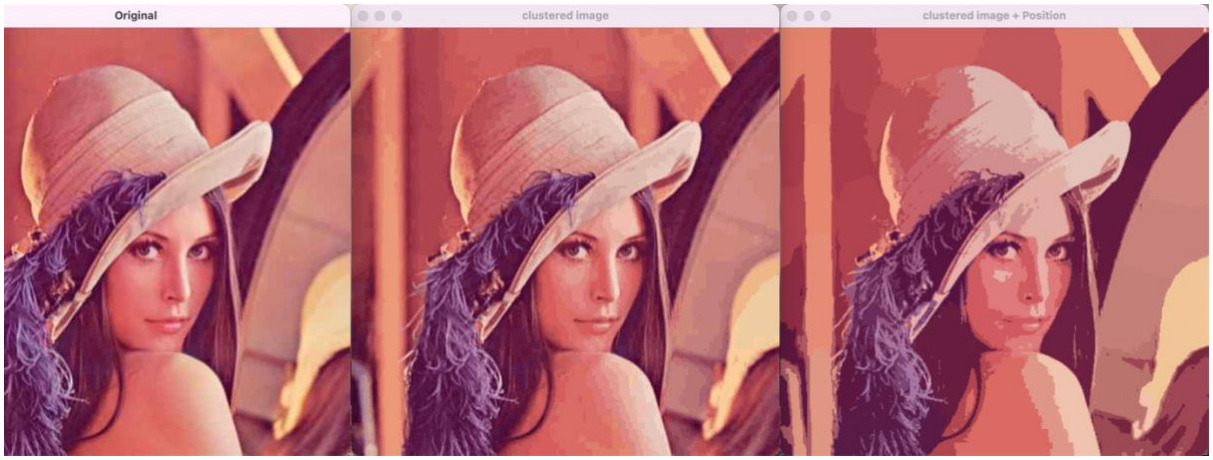
int clusterCount = 2;



int clusterCount = 10;

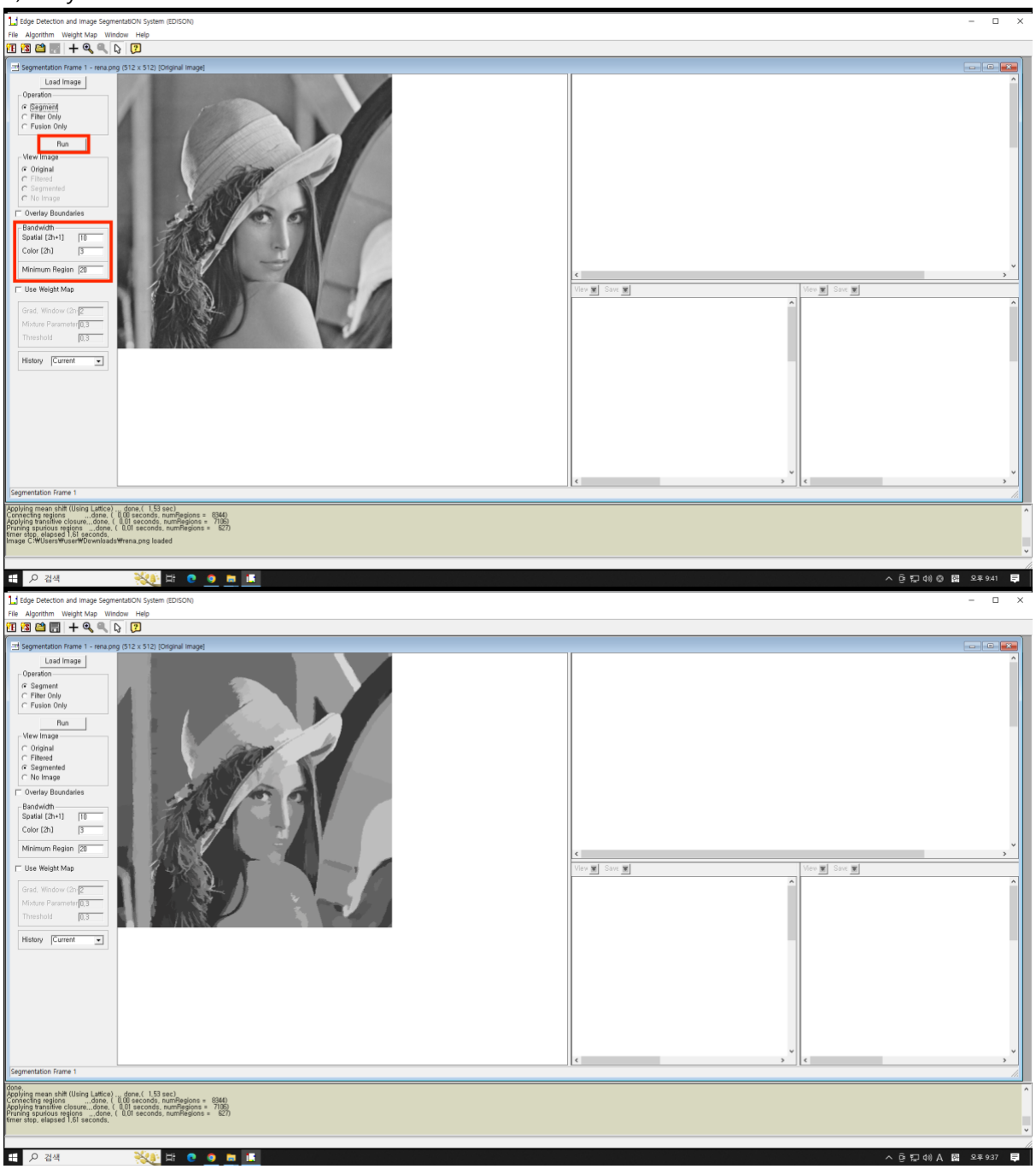


int clusterCount = 50;

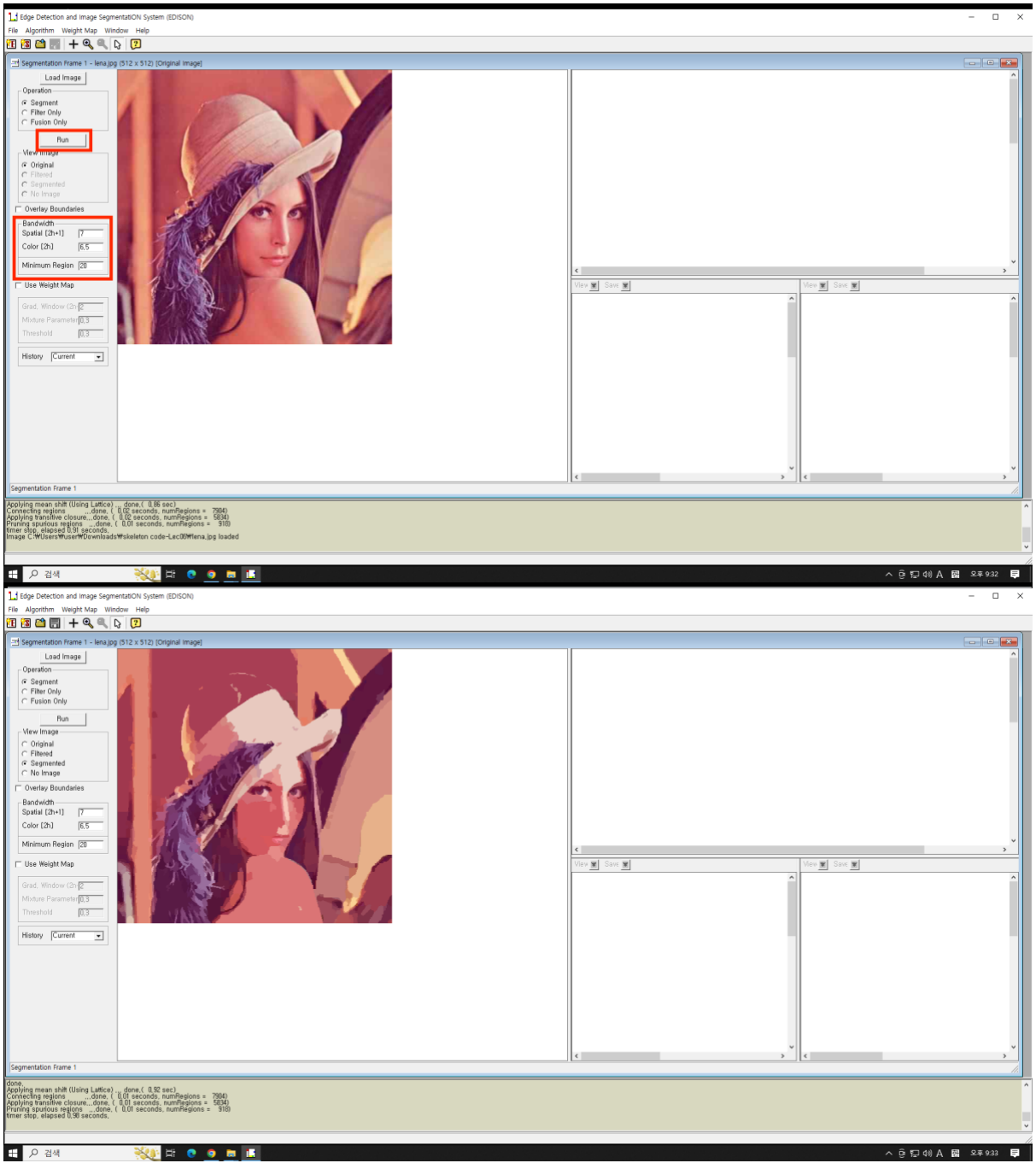


4. Mean Shift Segmentation

1) Gray



2) RGB



참고자료:

오픈 SW 프로젝트 Lec06 수업자료