# Technical report-Assignment05

컴퓨터공학 류다현(2376087)

**otsu.cpp**

othu's method는 히스토그램이 bimodal함을 가정하고 2개의 클래스 $C_1$, $C_2$를 설정한 뒤 완전탐색을 통해 클래스 내 분산을 최소화하거나 클래스 간 분산을 최대화하는 threshold값 t를 구하는 방식이다. 클래스 내 분산 $\sigma_W^2(t)$, 클래스 간 분산 $\sigma_B^2(t)$에서 $\sigma_W^2(t) = q_1\sigma_1^2 + q_2\sigma_2^2$, $\sigma_B^2(t) = q_1q_2(m_1 - m_2)^2$로 계산하며 이 프로그램에서는 클래스 간 분산을 최대화하는 t를 찾는다.

1. **int main()**

   main() 함수에서는 입력 이미지를 읽어 grayscale로 변환하고, grayscale 이미지에 대해 otsu's method로 threshold k를 찾아 이미지 segmentation을 수행한다. 작업 전, 후 이미지를 저장할 Mat 객체는 아래와 같이 사용한다.

   - input: 입력 컬러 이미지
   - input_gray: input을 grayscale로 변환한 이미지
   - output: segmentation한 결과 이미지

```
tie(t, output) = otsu_gray_seg(input_gray);
```

   에서 otsu_gray_seg() 함수는 (T, output) 형태의 튜플을 반환하는데, tie() 함수를 사용하면 반환된 튜플의 첫 번째 값 T가 tie의 첫 번째 인자 t에, 두 번째 값 output이 tie의 두 번째 인자 output에 저장된다.

2. **tuple<float, Mat> otsu_gray_seg(const Mat input)**

   otsu_gray_seg() 함수는 클래스 간 분산을 최대화하는 t를 찾고, 이를 기준으로 픽셀들을 0 또는 255로 변경한다.

   **2.1 변수 정의**

```
int row = input.rows;
int col = input.cols;
Mat output = Mat::zeros(row, col, input.type());
int n = row*col;
float T = 0, var = 0, var_max = 0, sum = 0, sumB = 0, q1 = 0, q2 = 0, sigma1 = 0, sigma2 = 0;
int count = 0;   //추가 변수
int histogram[L] = { 0 };  // initializing histogram values
```

   input의 세로, 가로 크기를 row와 col에 저장하고, 결과를 저장할 Mat형 output 객체를 생성해 0으로 초기화한다. n은 이미지 전체 픽셀 수인 row*col이며, T는 계산 결과 최종 threshold 값이다. q1, m1, sigma1은 클래스 1의 변수, q2, m2, sigma2는 클래스 2의 변수이고 이들은 추후 계산 과정에서 사용된다. histogram 배열은 히스토그램을 저장하고, histogram[i]는 intensity 값을 i로 가지는 픽셀의 수를 의미한다.

   sum은 0부터 L까지의 i에 대해 i*histogram[i]의 총합이며, sumB는 클래스1(이하 C1)에서 0부터 t 범위 i의 i*histogram[i]의 총합이다. count 변수에는 C1에서 탐색 중인 t까지의 픽셀 개수, 즉 0부터 t까지 i의 histogram[i] 총합이 저장된다. 이 값들은 조건문 내부에서 q1, q2, m1, m2를 구하기

위해 사용되며, 계산한 q1, q2, m1, m2로 클래스 간 분산 var을 계산하고 이들 중 가장 큰 것이 var_max에 저장된다. 이후 return make_tuple(T, output);을 통해 계산한 threshold 값 T와 T를 사용해 계산한 결과 이미지를 반환한다.

### 2.2 히스토그램 계산

```
for (int i = 0; i < input.rows; i++) {
      for (int j = 0; j < input.cols; j++) {   // finding histogram of the image
            histogram[input.at<G>(i, j)]++;
        }
}

for (int i = 0; i < L; i++) {     //auxiliary value for computing mean value
      sum += i * histogram[i];
}
```

histogram[i]는 intensity 값이 i인 픽셀의 개수이므로 2중 for문으로 rows, cols를 모두 순회하며 전체 픽셀에 대해 histogram 배열을 계산한다. $\sigma_B^2(t) = q_1 q_2 (m_1 - m_2)^2$에서 intensity i의 PDF를 $p(i)$라 했을 때 $q_1(t) = \sum_{i=0}^{t} p(i)$, $m_1(t) = \frac{1}{q_1(t)}(\sum_{i=0}^{t} ip(i))$이므로 추후 계산을 위해 값을 미리 계산해 sum에 저장한다.

### 2.3 최대 분산 계산

```
for (int t = 0; t < L; t++) {  //update q
        …
}
```

전체 intensity 크기 L에 대해 for문을 순회하며 계산한다.

```
sumB += t * histogram[t];
count += histogram[t];
```

매 반복마다 sumB와 count를 갱신하여 sumB에 0부터 t까지 $\sum_{i=0}^{t} i * hist(i)$ 를, count에 지금까지 사용된 픽셀 수를 저장한다.

```
if (count != 0 && count != n) {
      float q1 = (float)count / n;
      float q2 = 1.0f - q1;
      float m1 = sumB / count;
      float m2 = (sum - sumB) / (n - count);

      var = q1 * q2 * (m1 - m2) * (m1 - m2);
      if (var > var_max) {
            T = t; //threshold
            var_max = var;
      }
}
```

divide by zero 오류를 방지하기 위해 count가 0과 n이 아닐 때에만 계산을 수행하도록 조건문을 설정한다. $q_1(t) = \sum_{i=0}^{t} p(i)$ 에서 q1은 PDF의 합이므로 (현재까지 사용된 C1 픽셀 수)/(전체 픽셀 수)로 계산하며, 이는 count/n이다. 또, q1+q2=1이므로 q2=1-q1로 업데이트한다.
$m_1(t) = \frac{1}{q_1(t)}(\sum_{i=0}^{t} ip(i))$ 에서 i*PDF[i]는 i*histogram[i]/n으로 바꿀 수 있고 이는 sumB/n이며, q1은 count/n이므로 m1을 sumB/count로 계산할 수 있다. m2 또한 (C2의 i*PDF[i] 총합)/q2이므로 (sum-sumB)/(n-count)로 계산한다.

이후 $q_1 q_2 (m_1 - m_2)^2$로 클래스 간 분산 var값을 갱신하며, 업데이트된 var이 기존의 var 최댓값보다 더 크다면 var_max를 var로 변경하고 threshold T를 현재 탐색된 값인 t로 변경한다.

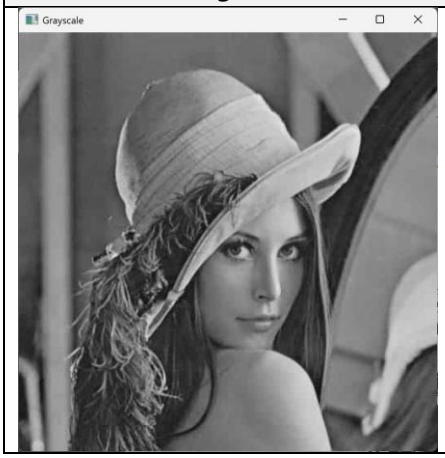
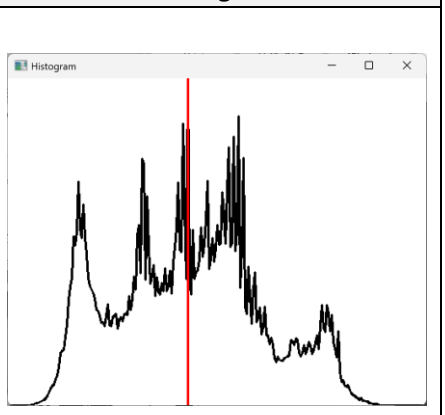
### 2.4 0, 255로 매핑

```
for (int i = 0; i < row; ++i) {
      for (int j = 0; j < col; ++j) {
            if (input.at<G>(i, j) > T) { output.at<G>(i, j) = 255; }
            else { output.at<G>(i, j) = 0; }
      }
}
```

2중 for문으로 전체 픽셀을 순회하며 픽셀 값이 threshold보다 클 경우 255, 그 이하일 경우 0으로 매핑해 segmentation한다.

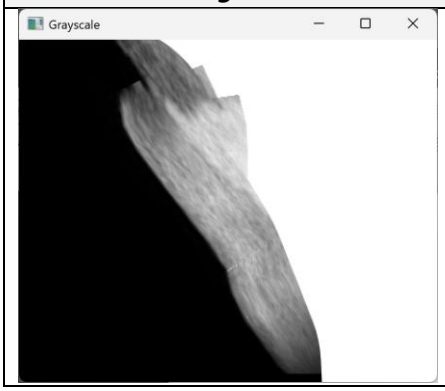
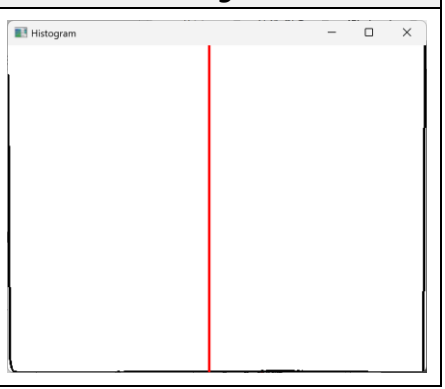
## 3. 결과 분석

input으로 "lena.jpg"를 사용해 실행하며, threshold가 어디에서 형성되었는지 시각적으로 확인할 수 있도록 히스토그램을 계산해 plot하는 코드를 추가하였다. 아래 히스토그램에서 빨간색으로 표시된 부분이 threshold이다.

| Original | Segmentation | Histogram |
|---|---|---|
|  |  |  |

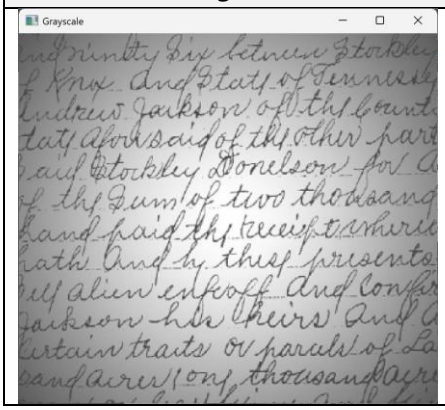
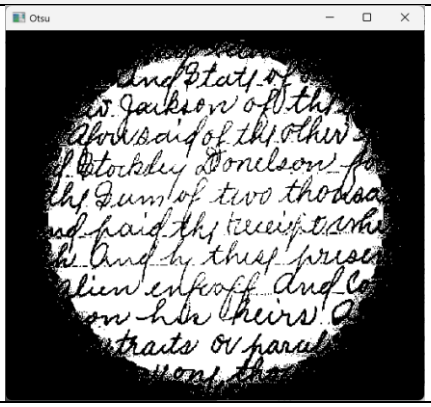
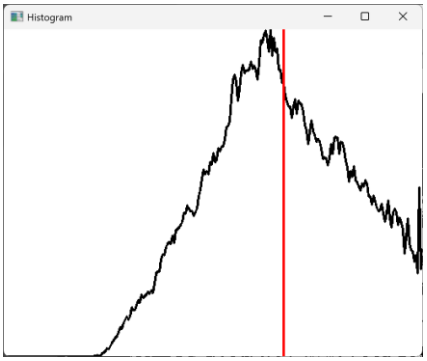


원본 이미지와 segmentation한 결과 이미지가 별도의 창에 출력되었으며, threshold T는 123에서 형성된다. 결과 이미지에서 threshold 값을 기준으로 픽셀 값이 0 또는 255로 분리된 것을 확인할 수 있다. 히스토그램이 보다 bimodal한 이미지로 테스트해 T가 형성되는 지점을 관찰하면 다음과 같다.

| Original | Segmentation | Histogram |
|---|---|---|
|  |  |  |

하지만 위 방식은 이미지 특성에 따라 적합하지 않을 수 있다. 단일 threshold를 사용하면 이미지의 영역별 특성이 충분히 반영되지 않을 수 있으며, otsu's method를 사용하면 히스토그램이 bimodal하게 분리된 경우에는 두 클래스를 적절히 나눌 수 있지만, 클래스를 뚜렷하게 분리하기 어려운 경우 단일 threshold만으로 픽셀들을 분류하기 어려워 결과 이미지에 의도하지 않은 결과를 출력하게 된다.

히스토그램이 bimodal하지 않아 배경과 물체의 구분이 뚜렷하지 않은 이미지를 사용했을 경우, 영역별 특성을 반영하지 않고 모든 픽셀에 대해 하나의 threshold만을 적용했을 때 아래와 같은 효과가 발생한다.

| Original | Segmentation | Histogram |
| --- | --- | --- |
|  |  |  |

**kmeans.cpp**

otsu's method는 히스토그램이 bimodal함을 전제로 하나의 threshold를 계산하므로, 클래스가 뚜렷하게 분리되지 않는 경우 불리하다. k-means clustering은 각 클러스터의 센터를 알고 있다면 이와 인접한 픽셀들을 탐색해 클러스터 멤버들을 파악할 수 있고, 클러스터 멤버들을 알고 있다면 클러스터 내 mean값을 계산하여 각 클러스터의 센터를 찾을 수 있음을 이용해 여러 클래스마다 센터를 찾는다.

k개의 랜덤한 값을 설정해 이들이 각각 클러스터의 센터라 가정하고 (1)이들과 인접한 픽셀들을 찾아 클러스터 멤버에 포함시킨다. 이후 (2)(1)에서 찾은 멤버 값들로 각 클러스터의 센터를 계산하며, 새롭게 구한 센터가 기존 센터와 같다면(=수렴했다면) 센터를 결정한 것이고 그렇지 않다면 (1)로 돌아가 센터가 고정될 때까지 반복한다.

1. **변수 정의**

   Mat input, input_gray, output 객체는 동일하며, clusterCount와 attempts 변수는 각각 생성할 클러스터 개수와 k-means clustering을 수행할 횟수이다.

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

   opencv에서 제공하는 kmeans() 함수는 위와 같고, 아래와 같이 호출해 사용한다.

```
kmeans(samples, clusterCount, labels, TermCriteria(TermCriteria::COUNT | TermCriteria::EPS,
10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

   samples 배열은 kmeans 함수에 전달할 입력값으로 Mat samples(input.rows * input.cols, 3, CV_32F); 구조이다. 행의 개수는 이미지 전체 픽셀 수인 (rows*cols)이며 열의 개수는 3으로, 이는 feature space가 3차원임을 의미한다. 따라서 kmeans() 함수를 호출해 클러스터링하기 전에 이미지 특성에 따라 sample 배열 3개의 열 모두에 적절한 값을 채워야 한다.

   labels 배열은 이미지의 각 픽셀이 어떤 클러스터에 포함되어 있는지 저장하며, labels.at<int>(i, 0)=3이라면 전체 픽셀 중 i번째 픽셀이 3번 클러스터에 속함을 의미한다. centers 배열은 각 클러스터의 중심점들을 저장한다.

   TermCriteria는 종료 조건을 설정하며 현재 TermCriteria(TermCriteria::COUNT | TermCriteria::EPS, 10000, 0.0001)로 호출한다. COUNT 모드는 지정된 반복 횟수만큼 수행되면 종료하고 EPS 모드는 오차, 즉 새로 계산한 센터와 이전 센터의 차이가 일정 이하라면 종료하는데, 이 둘을 함께 사용하고 있으므로 둘 중 하나의 조건만 만족되면 종료한다. 10000번 반복하거나 오차가 0.0001 이하라면 자동 종료하되, kmeans()에 별도의 attemps=5를 전달하고 있으므로 5번 반복 후 종료된다. TermCriteria의 구조는 아래와 같다.

```
cv::TermCriteria::TermCriteria(int        type,
                               int        maxCount,
                               double     epsilon
```

```
)
```

## 2. Intensity only

픽셀들을 클러스터링할 때 intensity값만을 고려한다. 따라서 kmeans() 호출 전 samples 배열을 초기화할 때 입력 이미지(input 또는 input_gray)의 intensity만을 사용해야 한다.

### 2.1 Grayscale

```
for (int y = 0; y < rows; y++) {
	for (int x = 0; x < cols; x++) {
		samples.at<float>(y + x * rows, 0) = input_gray.at<G>(y, x);
		samples.at<float>(y + x * rows, 1) = 0;
		samples.at<float>(y + x * rows, 2) = 0;
	}
}
```

input_gray는 단일 채널의 grayscale 이미지이므로 samples에 한 채널의 값만 전달할 수 있다. 픽셀 간 유사도를 판단할 때 (y, x) 픽셀의 intensity 값 이외의 다른 값은 고려하지 않으므로 samples 배열의 1, 2번째 열에는 0을 전달한다.

```
kmeans(samples, clusterCount, labels, TermCriteria(TermCriteria::COUNT |
TermCriteria::EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

이후 값을 채운 samples를 전달해 kmeans() 함수를 호출한다. 실행 결과 labels 배열에는 픽셀별 클러스터 정보, centers 배열에는 클러스터별 센터 정보가 저장된다.

```
Mat gray_intensity(input.size(), input_gray.type());
for (int y = 0; y < rows; y++) {
	for (int x = 0; x < cols; x++) {
		gray_intensity.at<G>(y, x) = (G)(centers.at<float>(labels.at<int>(y + x *
		rows, 0), 0));
	}
}
```

이후 labels, centers 배열대로 결과 이미지에 값을 할당한다. intensity만을 고려하는 grayscale 결과 이미지를 저장하기 위해 Mat형 gray_intensity 객체를 생성하고 2중 for문을 통해 rows, cols 전체 범위에 대해 반복하며 각 픽셀당 자신이 속한 클러스터의 intensity 값을 매핑한다.

단일 채널 grayscale 이미지를 사용하므로 centers의 (y+x*rows, 0)에만 접근하는데, 이때 labels는 열 벡터 형태이므로 rows*cols 구조인 gray_intensity와 연결할 때 인덱스를 적절히 사용해야 한다. labels(y+x*rows, 0)는 해당 픽셀의 클러스터 번호를 가지므로 이를 centers 배열에 전달하면 해당 클러스터 번호가 가지는 intensity 값을 얻을 수 있고, labels의 (y+x*rows, 0) 지점은 gray_intensity의 (y, x)에 대응하므로 이를 gray_intensity(y, x)에 매핑한다.

반복문이 종료되면 각 픽셀이 자신이 속한 클러스터의 intensity로 채워진 결과 이미지 gray_intensity를 얻을 수 있다.

### 2.2 RGB

```
for (int y = 0; y < input.rows; y++)
	for (int x = 0; x < input.cols; x++)
		for (int z = 0; z < 3; z++)
			samples.at<float>(y + x*input.rows, z) = input.at<Vec3b>(y,
			x)[z];
```

이 경우에도 마찬가지로 intensity만을 사용하지만, input은 3채널의 컬러 이미지이므로 sample의 각 열에 R, G, B 채널 intensity 값을 각각 전달한다. 3중 for문을 통해 samples의 z열에 z채널 값을 매핑한다.

```
labels = Mat();
centers = Mat();
kmeans(samples, clusterCount, labels, TermCriteria(TermCriteria::COUNT |
TermCriteria::EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
```

이후 동일하게 값을 채운 samples를 전달해 kmeans() 함수를 호출하되, labels와 centers 배열을 초기화한다. 실행 결과 labels 배열에는 픽셀별 클러스터 정보, centers 배열에는 클러스터별 센터 정보가 저장된다.

```
Mat new_image(input.size(), input.type());
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));
        }
}
```

2.2의 경우와 마찬가지로 intensity만을 사용하는 RGB 결과 이미지를 저장할 new_image 배열을 생성하며, 동일하게 2중 for문으로 전체 픽셀에 대해 계산한다.
(y+x*rows, 0) 인덱스로 labels에 접근해 소속된 클러스터 값을 추출하고 다시 이 값으로 centers에 접근해 (y, x) 픽셀의 클러스터 intensity를 매핑하는 과정은 동일하나, 이 경우 3채널의 컬러 이미지를 사용하므로 (cluster_idx, 0), (cluster_idx, 1), (cluster_idx, 2)와 같이 centers의 각 채널에서 new_image의 대응하는 채널로 값을 할당한다.

## 3. Intensity & Position

픽셀들을 클러스터링할 때 intensity와 픽셀 위치를 모두 고려하므로 kmeans() 호출 전 samples() 배열을 초기화할 때 픽셀의 intensity와 위치 값을 모두 사용해야 한다. 이때 intensity는 0에서 255까지의 값을 사용하고 픽셀 위치는 입력 이미지의 크기에 따라 달라지므로 이들을 동일한 규격으로 맞추는 정규화 작업이 필요하다.

### 3.1 Grayscale

```
for (int y = 0; y < rows; ++y) {
        for (int x = 0; x < cols; ++x) {
                int idx = y + x * rows;
                samples.at<float>(idx, 0) = input_gray.at<G>(y, x) / (float) (L - 1)
                samples.at<float>(idx, 1) = float(x) / sigma;
                samples.at<float>(idx, 2) = float(y) / sigma;
        }
}
```

samples의 0번째 열에 intensity를, 2, 3번째 열에는 x, y 좌표를 전달해 클러스터링 과정에서 intensity와 픽셀 위치를 모두 고려할 수 있도록 한다. 모든 픽셀은 0부터 255까지의 intensity를 가질 수 있으며, L=256으로 선언했으므로 intensity 값은 0에서 1 범위로 정규화된다. 마찬가지로

sigma=512로 선언했으므로 x, y 좌표 또한 0에서 1 범위로 정규화된다.

```
kmeans(samples, clusterCount, labels, TermCriteria(TermCriteria::COUNT |
TermCriteria::EPS, 10000, 0.0001), 5, KMEANS_PP_CENTERS, centers);
Mat gray_intensity_position(input_gray.size(), input_gray.type());
for (int y = 0; y < rows; ++y) {
        for (int x = 0; x < cols; ++x) {
                gray_intensity_position.at<G>(y, x) = (float)((L - 1) *
                centers.at<float>(labels.at<int>(y + x * rows, 0), 0));
        }
}
```

이후 kmeans() 함수롤 호출해 labels와 centers를 계산하고, intensity와 position을 모두 고려한 grayscale 결과 이미지 gray_intensity_positon을 생성해 2중 for문을 순회하며 픽셀별 소속 클러스터 값을 매핑한다. 단일 채널 이미지를 사용하므로 2.1과 동일하게 centers(index, 0)에 대해서만 할당한다. 이때 위에서 intensity를 0~1로 정규화하여 centers에 0~1 범위의 값이 저장되어 있으므로 centers 원소를 grat_intensity_position에 저장할 때 (L-1)을 다시 곱해 기존 0~255 범위로 복원해야 결과 이미지에 올바른 픽셀 값을 표시할 수 있다.

### 3.2 RGB

3차원 samples로 충분했던 앞의 경우들과 달리 RGB 이미지에서 intensity와 위치를 모두 사용하려면 RGB(3D)+(x, y)(2D) 총 5차원 samples가 필요하다. 따라서 Mat(rows * cols, 3, CV_32F)로 선언했던 samples 배열을 samples = Mat(rows * cols, 5, CV_32F);로 다시 생성하여 사용한다.

```
samples = Mat(rows * cols, 5, CV_32F);
for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
                int idx = y + x * rows;
                samples.at<float>(idx, 0) = input.at<C>(y, x)[0] / (float)(L - 1);
                samples.at<float>(idx, 1) = input.at<C>(y, x)[1] / (float)(L - 1);
                samples.at<float>(idx, 2) = input.at<C>(y, x)[2] / (float)(L - 1);
                samples.at<float>(idx, 3) = float(x) / sigma;
                samples.at<float>(idx, 4) = float(y) / sigma;
        }
}
```

따라서 samples의 0~2열에는 각 R, G, B 채널의 intensity값을 저장하고 3, 4열에는 동일하게 정규화한 좌표 위치를 저장한다.

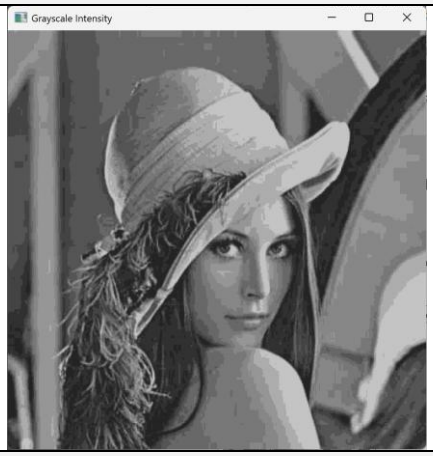
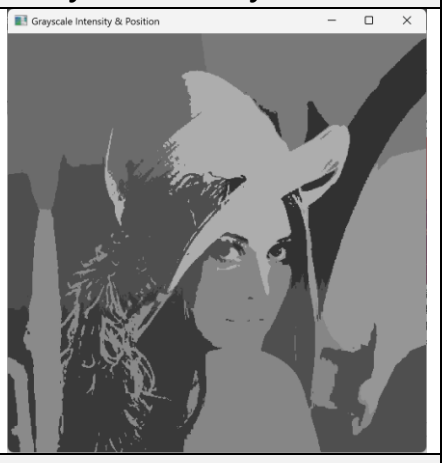
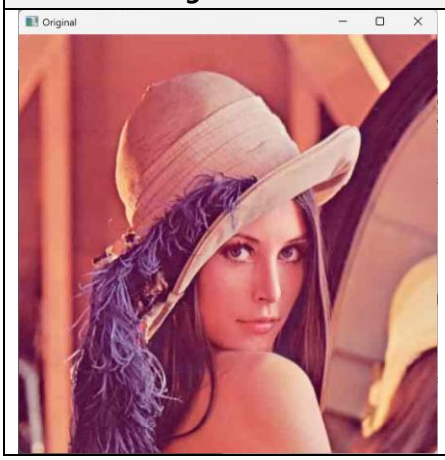
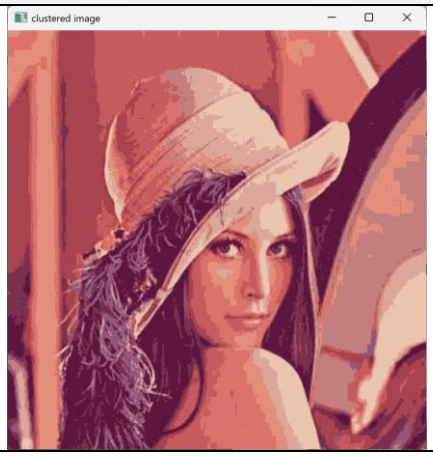
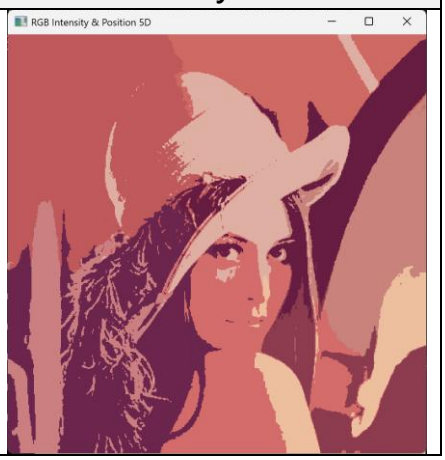
```
kmeans(samples, clusterCount, labels, TermCriteria(TermCriteria::COUNT |
TermCriteria::EPS, 10000, 0.0001), attempts, KMEANS_PP_CENTERS, centers);
Mat RGB_intensity_position_5D(input.size(), input.type());
for (int y = 0; y < input.rows; y++) {
        for (int x = 0; x < input.cols; x++)
        {
                int cluster_idx = labels.at<int>(y + x * input.rows, 0);
                RGB_intensity_position_5D.at<C>(y, x)[0] = (float)((L - 1) *
                centers.at<float>(cluster_idx, 0));
                RGB_intensity_position_5D.at<C>(y, x)[1] = (float)((L - 1) *
                centers.at<float>(cluster_idx, 1));
                RGB_intensity_position_5D.at<C>(y, x)[2] = (float)((L - 1) *
                centers.at<float>(cluster_idx, 2));
        }
}
```

kmeans() 함수를 호출해 계산하면, labels와 centers 배열에는 픽셀 intensity와 픽셀 위치를 모두 고려해 clustering한 결과가 저장된다. intensity와 position을 모두 사용하는 RGB 결과 이미지를 저장할 RGB_intensity_position_5D 배열을 생성하고, 동일하게 2중 for문을 통해 픽셀을 매핑한다. 2.2와 마찬가지로 centers 각 열의 값에 (L-1)을 곱한 것을 결과 이미지의 [0], [1], [2] 채널에 저장해 채널별 intensity를 할당한다.

## 4. 결과 분석

```
#define sigma 512.0f;
```

sigma를 512로 설정하고 lena.jpg 이미지로 실행하였다.

| Original Gray | Grayscale only Intensity | Grayscale Intensity & Position |
|---|---|---|
|  |  |  |
| **Original RGB** | **RGB only Intensity** | **RGB Intensity & Position** |
|  |  |  |

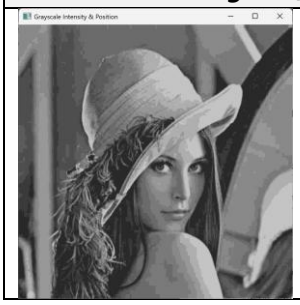
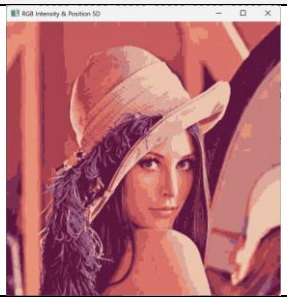
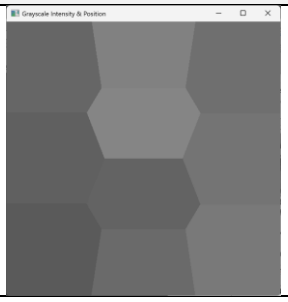
모든 경우 segmentation한 결과 이미지가 별개의 창에 출력되었다. 원본 이미지에서 유사한 값을 가지던 픽셀들이 픽셀 위치와 관계없이 결과 이미지에서는 동일한 값을 가지게 된 것과 달리, intensity와 픽셀 위치를 모두 고려했을 때 segmentation 과정에 위치 정보가 개입해 같은 intensity를 가지던 픽셀이라도 멀리 떨어져 있다면 다른 클러스터에 포함된다. 즉, 픽셀 위치까지 고려한 경우 비슷한 위치의 픽셀들끼리 같은 클러스터로 뭉치는 경향이 더 큰 것을 확인할 수 있다.

### 4.1 sigma별 비교

sigma를 512로 두고 512*512 크기인 "lena.jpg"를 사용하면 픽셀 좌표 값이 0에서 1 범위로 정규

화된다. 따라서 2.2, 3.2와 같이 구현했을 때 intensity와 픽셀 위치가 모두 0에서 1 사이 값을 가지므로 클러스터를 형성할 때 동일한 영향을 가진다.

하지만 이미지 크기 대비 sigma를 지나치게 크게 설정하면 픽셀 위치 정보가 [0, 1] 범위의 intensity에 비해 작은 값을 가지므로 세 feature 중 intensity만이 유의미한 분산을 가지게 되어 클러스터 계산 과정에서 intensity를 더 많이 고려하게 된다. 반대로 sigma가 지나치게 작다면 픽셀 위치 정보가 더욱 유의미하므로 intensity보다는 위치에 따라 클러스터가 형성된다.

| #define sigma (float)pow(2, 32); | | #define sigma 1.0f; | |
|---|---|---|---|
|  |  |  |  |

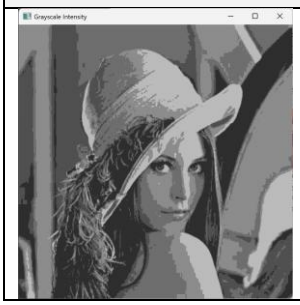
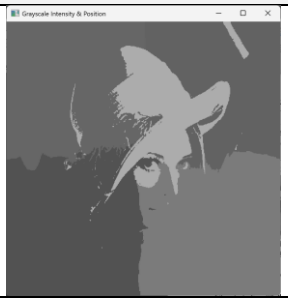
실행 결과 sigma를 크게 설정한 경우 정규화된 x, y 값이 거의 0에 수렴해 결과 이미지가 intensity만 고려한 결과 이미지와 유사해졌으며, sigma가 작은 경우 좌표 정보 대비 intensity의 중요도가 줄어 픽셀 값은 거의 무시되고 픽셀 위치에 따라 10개의 클러스터가 형성되었다.

위 결과를 통해 이미지 특성에 따라 정규화를 적절하게 수행해야 최적의 결과를 얻을 수 있음을 확인할 수 있다.

### 4.2 clusterCount별 비교

```
int clusterCount = 5;
```

위와 같이 클러스터 개수를 5개로 줄여 실행한 후 결과를 관찰하였다.

| Intensity only | | Intensity & Position | |
|---|---|---|---|
|  |  |  |  |

clusterCount를 줄이면 전체 이미지가 더 적은 단계로 묶이며 덜 세밀하게 분할된 것을 확인할 수 있다. 클러스터가 10개인 경우보다 보다 큰 덩어리로 분리되었으며, 결과 이미지가 상대적으로 단순해졌다.

**adaptivethreshold.cpp**

adaptive threshold 방식은 픽셀별로 다른 threshold 값을 사용해 segmentation한다. (i, j) 픽셀의 threshold T(i, j)는 $T(i,j) = b \times m(i,j)$로 계산하며, 이때 b는 상수, m(i, j)는 (i, j) 픽셀의 윈도우 내 평균 값이다. 아래 설명에서는 윈도우 내 픽셀들로 uniform mean filtering한 뒤 상수 bnumber를 곱한 값을 threshold로 사용한다.

이때 bnumber는 threshold의 민감도를 조정하기 위한 값이다. b<1일 때 $T(i,j) = b \times m(i,j) < m(i,j)$이므로 threshold가 평균보다 낮아져 더 많은 픽셀들이 255로 매핑되며, b>1일 때 $T(i,j) = b \times m(i,j) > m(i,j)$이므로 더 많은 픽셀들이 0으로 선택된다.

1. **int main()**

   input에 "writing.jpg" 컬러 이미지를 읽어 grayscale로 변환한 것은 input_gray에 저장하고, 결과 이미지용 Mat형 output 배열을 선언한다.

```
output = adaptive_thres(input_gray, 2, 0.9);
```

   이후 adaptive_thres() 함수를 호출해 픽셀별로 threshold를 적용한 결과 이미지를 생성하는데, 이때 위와 같이 커널 크기는 2, threshold 계산에 쓰일 상수 b(=bnumber)는 0.9로 설정한다.

2. **Mat adaptive_thres(const Mat input, int n, float bnumber)**

   **2.1 변수 정의**

```
Mat kernel;

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

   (2n+1)*(2n+1) 크기의 커널 kernel을 사용하며, kernel의 각 원소는 모두 1/(kernel_size^2)로 통일하고 이 값을 kernelvalue 변수에 저장해 사용한다. 이후 2중 for문으로 row, col 범위 내 모든 픽셀들을 순회하며 (i, j) 픽셀의 threshold를 계산한다.

   **2.2 Uniform mean filtering**

```
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));
		}
	}
}
```

   -n부터 n까지 커널 내 픽셀들에 대해 w*I 계산한 결과를 sum1에 누적하며, 조건문을 통해 이미지 경계를 벗어나지 않는 픽셀들만 계산하고 경계를 벗어나는 좌표는 sum1에 포함하지 않고 무시함으로써 zero padding 옵션을 구현한다. 반복문이 종료되면 sum1에 m(i, j)가 저장된다.

   **2.3 Threshold값 생성 및 매핑**

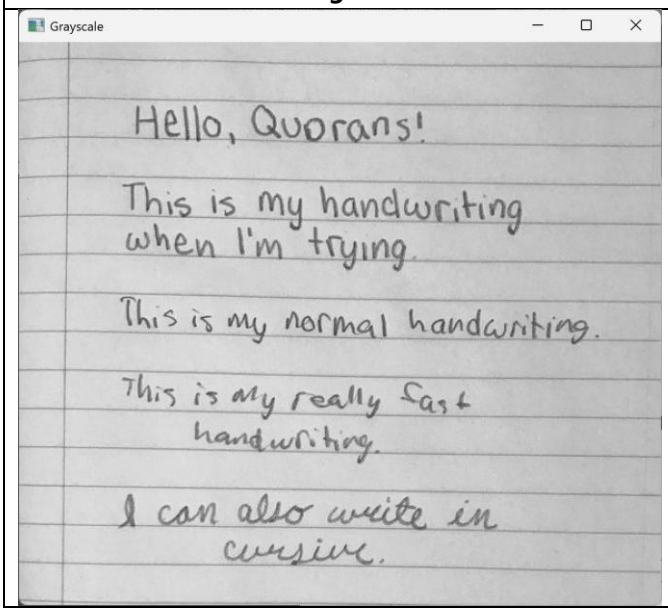
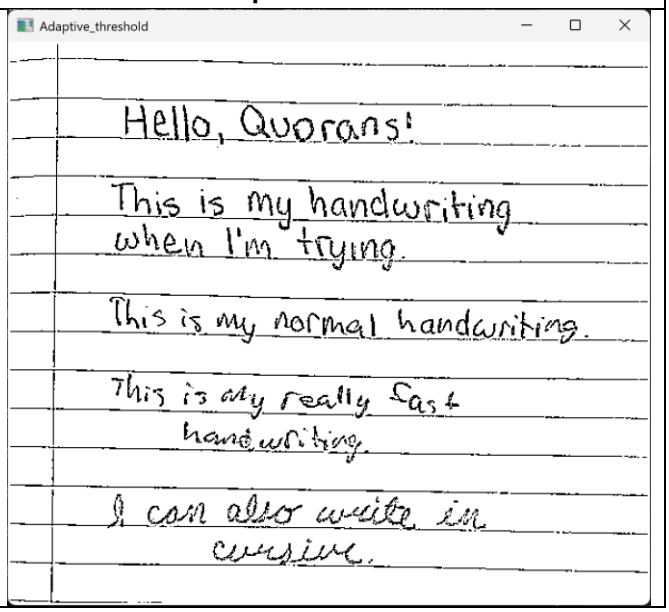
```
float T = bnumber*(G)sum1;

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

2.2에서 계산한 m(i, j)로 (i, j) 픽셀의 threshold를 계산한다. 에 따라 인자로 전달받은 bnumber과 계산한 sum1을 곱한 값 T가 input(i, j)의 threshold이며, 이 T를 기준으로 픽셀 값이 T 초과일 때 255, 이하일 때 0으로 매핑하여 이진화한다.

## 3. 결과 분석

```
output = adaptive_thres(input_gray, 2, 0.9);
```

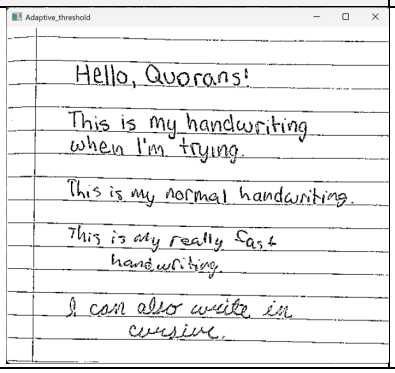
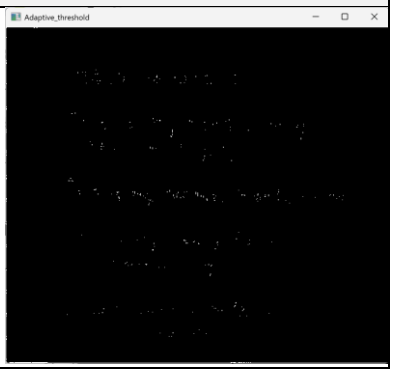
5*5 크기의 윈도우를 사용하며, bnumber=0.9로 호출하면 아래와 같이 배경(255)과 글자(0)이 분리된다.

| Original | Adaptive thrershold |
|---|---|
|  |  |



### 3.1 bnumber별 비교

bnumber를 0.5, 0.9, 1.1로 변화시키며 결과를 비교한다.

| bnumber=0.5 | bnumber=0.9 | bnumber=1.3 |
|---|---|---|
|  |  |  |



bnumber를 0.5로 설정하면 T=0.5m(i, j), 즉 픽셀 값이 평균의 절반만 넘어도 255로 표시되므로 이미지의 대부분이 255 값을 가진다. 반면 bnumber가 1.3일 경우 T=1.3m(i, j)이므로 픽셀 값이 평균의 1.3배보다 작은 픽셀들은 모두 0으로 매핑되어 이미지의 대부분이 0 값을 가진다.

**meanshift.cpp**

k-means clustering에서 k는 kmeans.cpp의 clusterCount와 같이 수동으로 지정해야 하는 hyperparameter이다. 이와 달리 mean shift segmentation은 히스토그램에서 반복을 통해 mode를 계산하므로 k를 자동으로 구할 수 있다. k를 구하는 과정은 아래와 같다.

1. random seed 설정
2. 히스토그램에 윈도우 W를 적용해 윈도우 내 center of gravity 계산
3. (2)에서 계산한 센터로 윈도우를 이동
4. 계산한 센터가 이전 반복의 센터와 일치할(=수렴) 때까지 (2), (3)을 반복

이때 센터가 수렴하기까지 윈도우가 거쳐 온 경로의 모든 픽셀을 같은 클러스터에 포함시킨다. (2)에서 히스토그램 H와 윈도우 W에 대해 center of gravity를 계산하는 공식은 다음과 같다.

$$\frac{\sum_{x \in W} xH(x)}{\sum_{x \in W} H(x)}$$

**1. int main()**

```
pyrMeanShiftFiltering(input_gray, output_gray, 31, 20, 1);  // Grayscale
pyrMeanShiftFiltering(input, output, 31, 20, 3);   // RGB
```

읽어 온 컬러 입력 이미지 input을 grayscale로 변환해 input_gray에 저장하고, 결과 이미지를 저장할 output을 생성하는 것은 동일하다. 별도의 mean shift filtering 구현 없이 opencv에서 제공하는 pyrMeanShiftFiltering() 함수를 호출해 사용한다.

**2. pyrMeanShiftFiltering()**

```
void cv::pyrMeanShiftFiltering ( InputArray         src,
                                 OutputArray        dst,
                                 double             sp,
                                 double             sr,
                                 int                maxLevel = 1,
                                 TermCriteria       termcrit = TermCriteria(TermCriteria::MAX_ITER
                                                      + TermCriteria::EPS, 5, 1) )
```

함수 선언은 위와 같으며, 파라미터는 다음과 같다.

- src: 사용할 8비트, 3채널의 입력 이미지
- dst: src와 타입 및 크기가 동일한 결과 이미지
- sp: spatial 윈도우 반지름
- sr: 컬러 윈도우 반지름
- maxLevel: 가우시안 피라미드의 최대 레벨
- termcrit: 반복 종료 조건

pyrMeanShiftFiltering()은 픽셀의 intensity(R, G, B)와 위치(X, Y)를 모두 고려해 mean shift 필터링을 수행한다. sp와 sr에 따라 (X, y) 픽셀에서의 윈도우를 설정하고, 그 안에서 윈도우 내 픽셀들의 평균 위치 정보 (X', Y')와 평균 색상 벡터 (R', G', B')를 계산한다. 이 평균값을 새로운 센터로 삼아 다시 계산하는 과정을 종료 조건까지 반복하는데, 반복이 종료되면 반복이 시작된 픽셀 값들은 최종 평균값 (R', G', B')로

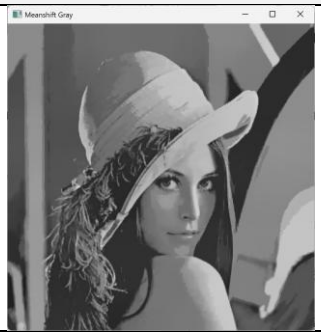
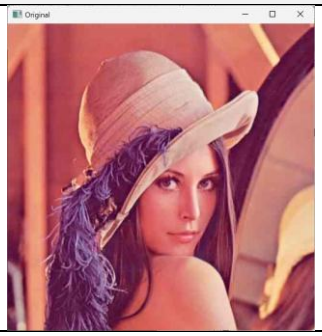
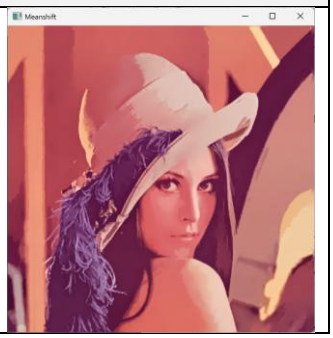
설정되므로 클러스터 내 픽셀들은 같은 값을 가진다.

이 과정에서 가우시안 피라미드를 사용한다. 인자로 전달받은 maxLevel이 0보다 크면 (maxLevel+1) 크기의 가우시안 피라미드가 형성되며, 작은 해상도의 이미지부터 위의 mean shift 과정을 수행한다. 작은 이미지의 계산 결과는 더 큰 이미지에 전파되고 해당 이미지에서 다시 동일한 과정을 반복하는데, 이때 피라미드 하위 레이어와 픽셀 값 차이가 sr 이상인 픽셀에 대해서만 연산이 수행되게 함으로써 경계를 더욱 선명하게 한다.

### 3. 결과 분석

"lena.jpg"를 입력 이미지로 사용해 다음과 같이 호출하였다.

```
pyrMeanShiftFiltering(input_gray, output_gray, 31, 20, 1);   // Grayscale
pyrMeanShiftFiltering(input, output, 31, 20, 3);    // RGB
```

| Original Gray | Meanshift Gray | Original RGB | Meanshift RGB |
|---|---|---|---|
|  |  |  |  |