

exVision: AlgoSegment

Kareem Nouredine, Youssef A. Shawki, Nouran M. Khattab, Nada Omran, Mohamed Samy

Supervised by **Dr. Ahmed Badawy** as part of the Classical Computer Vision course, Systems and Biomedical Engineering Faculty, Cairo University (2023/2024)

Image Segmentation

The image Segmentation task is the splitting of the image into meaningful regions to serve for other tasks like image understanding and object detection, knowing the two basic properties of image intensity values: discontinuity and similarity. In the first category, the approach is to partition an image into regions based on abrupt changes in intensity, such as edges. Approaches in the second category are based on partitioning an image into regions that are similar according to a set of predefined criteria. Thresholding, region growing, and region splitting and merging are examples of methods in this category.

Thresholding

Thresholding is an image processing method that is based on splitting the parts in the image based on pixel values. It relies on the concept that any part of the image has pixels of similar values belonging to the same class. There are multiple thresholding algorithms, they can be performed in either a global or local manner. When it's **Global Thresholding**, it applies the algorithm on the total image all at once and when it's **Local Thresholding**, it applies the algorithm on equal-sized windows of the original image. There's also the concept of **Binary and Multi-Modal Thresholding**. When it's Binary Thresholding the histogram/grey levels values are split into 2 regions only while in Multi Modal Thresholding, there can be more than 2 regions. In practice, using Multi-Modal Thresholding is considered a viable approach when there is reason to believe that the problem can be solved effectively with two thresholds. Although this result is applicable to an arbitrary number of classes, it begins to lose meaning as the number of classes increases because we are dealing with only one variable (intensity). We will discuss **Optimal Thresholding** (Binary) and **Otsu Thresholding** (Binary and Multi-Modal) for both global and local thresholding.

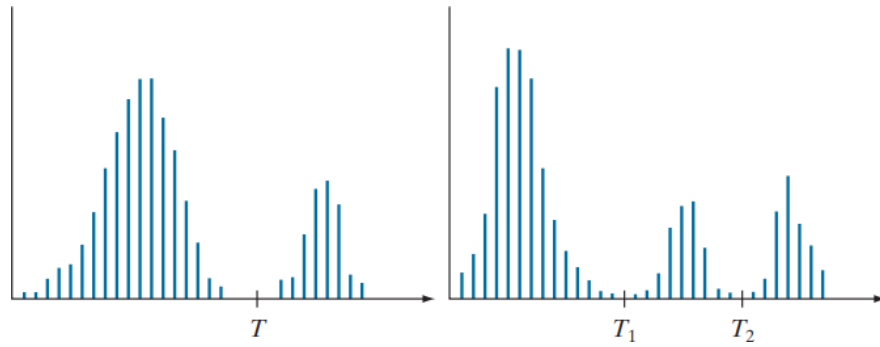


Figure 1, Binary Thresholding (on the left) and Multi-Modal Thresholding (on the right).

Important Considerations

Thresholding algorithms are highly sensitive to illumination gradients and noise. It also provides no way to provide spatial understanding.

Optimal Thresholding (Binary Thresholding)

Brief

When the intensity distributions of objects and background pixels are sufficiently distinct, it is possible to use a single (global) threshold applicable over the entire image. In most applications, there is usually enough variability between images that, even if global thresholding is a suitable approach, an algorithm capable of estimating the threshold value for each image is required. Optimal Thresholding provides automatic BinaryThresholding results. This is done through successively thresholding the input image and calculating the means at each step. It works well in situations where there is a reasonably clear valley between the modes of the histogram related to objects and background.

Steps

1. Select an initial estimate for the global threshold, T as the average of the 4 corner pixels in the image.

$$T_1 = \frac{1}{4} (f(0,0) + f(0, -1(\text{last row})) + f(-1(\text{last column}), 0) + f(-1(\text{last column}), -1(\text{last row})))$$

2. Segment the image using T in the above equation. This will produce two groups of pixels:
 - G_1 which consists of pixels with intensity values $> T$
 - G_2 which consists of pixels with intensity values $\leq T$

$$g(x) = \begin{cases} 1 & \text{if } f(x, y) > T \\ 0 & \text{if } f(x, y) < T \end{cases} \quad \text{Equation 01}$$

3. Compute the average (mean) intensity values m_1 and m_2 for the pixels in G_1 and G_2 respectively.
4. Compute a new threshold value midway between m_1 and m_2 :

$$T_{\text{new}} = \frac{1}{2} \times (m_1 + m_2) \quad \text{Equation 02}$$

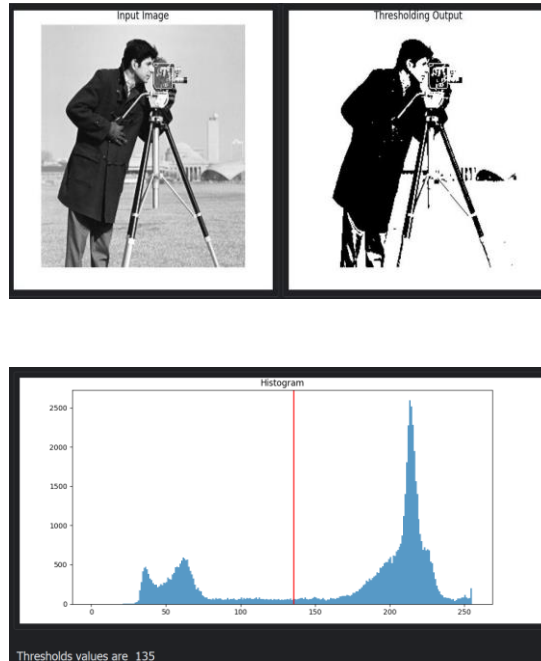
5. Repeat steps 2 through 4 until the difference between values of T in successive iterations is smaller than a predefined value, ΔT . In our application, $\Delta T = 0$.

Experiment

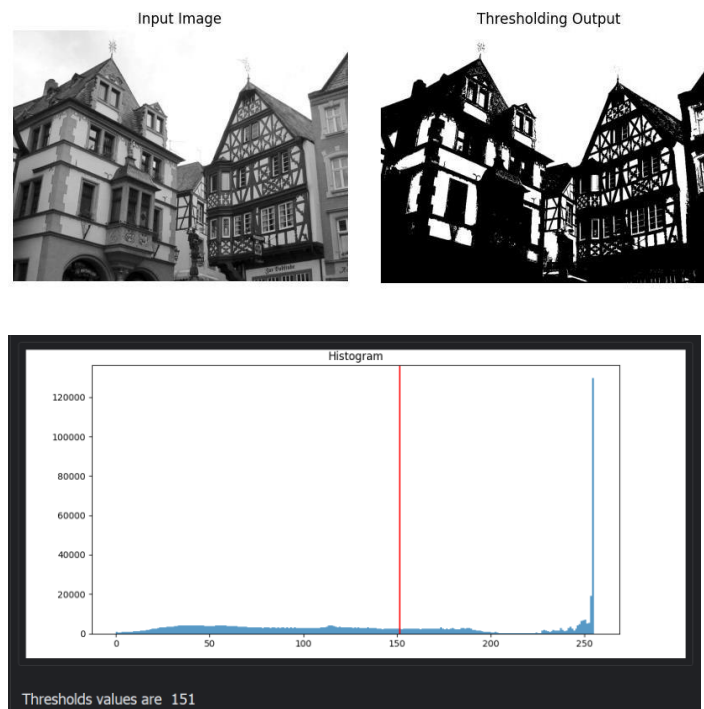
The only parameter the optimal thresholding takes is the image itself. We shall see different outputs and their results thresholding value.

Global Optimal Thresholding

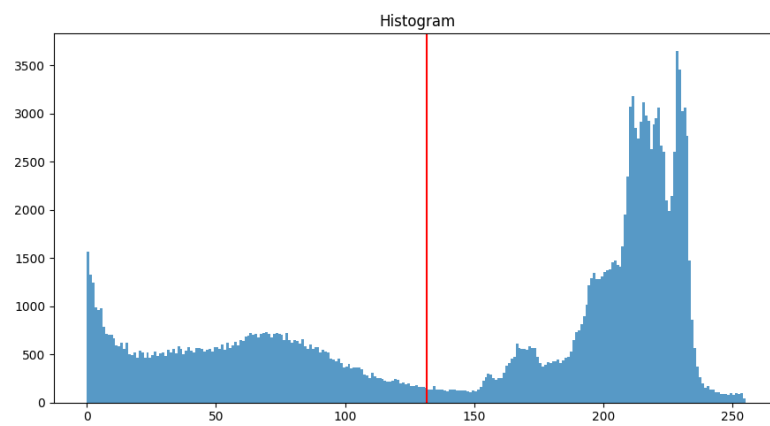
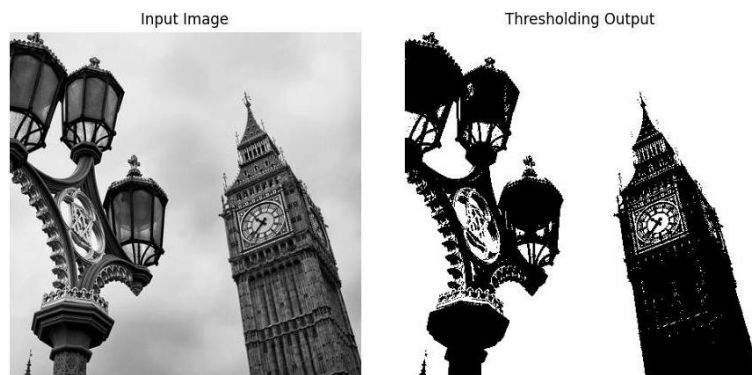
Camera Man image



Two Houses Image



Big Ben



Thresholds values at 131

Otsu Thresholding (Binary and Multi Modal)

Brief

Thresholding may be viewed as a **statistical-decision theory problem** whose objective is to minimize the average error incurred in assigning pixels to two or more groups (also called classes). This problem is known to have an elegant closed-form solution known as the Bayes decision function. The solution is based on only two parameters: the probability density function (PDF) of the intensity levels of each class, and the probability that each class occurs in a given application. This can be done through **Gaussian Mixture Models** using Estimation Maximization Algorithm. Unfortunately, estimating PDFs is not a trivial matter, so the problem usually is simplified by making workable assumptions about the form of the PDFs, such as assuming that they are Gaussian functions. Even with simplifications, the process of implementing solutions using these assumptions can be complex and not always well-suited for real-time applications.

The approach in the following discussion, called **Otsu's method** (Otsu [1979]), is an **attractive alternative**. The method is optimum in the sense that it maximizes the ***Between-Class Variance***, a well-known measure used in statistical discriminant analysis. The basic idea is that properly thresholded classes should be distinct with respect to the intensity values of their pixels and, conversely, that a threshold giving the best separation between classes in terms of their intensity values would be the best (optimum) threshold. In addition to its optimality, Otsu's method has the important property that it is based entirely on **computations performed on the histogram of an image**, an easily obtainable 1-D array.

Global Otsu Thresholding

Multi Modal Thresholding Algorithm

Between-Class Variance

- The Algorithm maximizes the **Between-Class Variance**, which is calculated as shown in Equation 03 for 2 classes/regions as follows:

$$\sigma_B^2 = P_1 (m_1 - m_G)^2 + P_2 (m_2 - m_G)^2 \quad \text{Equation 03}$$

Which can also be written as:

$$\sigma_B^2 = P_1 P_2 (m_1 - m_2)^2 \quad \text{Equation 04}$$

Where for the first candidate threshold k

- (The probability of class 1 occurring/having pixels assigned to class 1)

$$P_1 = \sum_{i=0}^k p_i \quad \text{Equation 05}$$

Where:

$$p_i = \frac{n_i}{M \times N} \quad \text{Equation 06: Normalized histogram}$$

Where:

- n_i = number of pixels having intensity value $i \in [0, L - 1]$ having $L = 255$ for an 8-bit image,
- M, N are the dimensions of the 2D greyscale image (width and height).

Following the same concept for any Class j between thresholds k_{j-1} or 0 and k_j .

- (Mean intensity value of the pixels in c_1)

$$m_1(k) = \sum_{i=0}^k i P(i|c_1) = \frac{1}{P_1(k)} \times \sum_{i=0}^k i p_i \quad \text{Equation 07}$$

Following the same concept for any $m_i(k)$.

Separability Measure

We can evaluate the effectiveness of the threshold at level k using the following normalized (ranges from 0 to 1) dimensionless measure:

$$\eta = \frac{\sigma_B^2}{\sigma_G^2} \quad \text{Equation 08}$$

Where

$$\sigma_G^2 = \sum_{i=0}^{L-1} (i - m_G^2)^2 p_i \quad \text{Equation 09}$$

$$m_G = \sum_{i=0}^{L-1} i p_i \quad \text{Equation 10}$$

The Optimum Threshold Value in Otsu Thresholding

For two classes only problem/one optimal candidate, that candidate is the one that has the highest ***Between-Class Variance***.

$$\sigma_B^2(k^*) = \max \sigma_B^2(k) \text{ for } k \in [0, L - 1] \quad \text{Equation 11}$$

Candidate Choice Optimization

For a single threshold k , k is tested over the range $[0, L - 1]$ with a step of 1 between each candidate and another one. If $k > 1$, candidates will increase nearly exponentially as a function of k . As in such a case, we would have to make all the candidates constant, moving one till the end of the range, then increasing the following one of the constants by 1, then increasing the last till the end of the range, and so on and so forth. To make it faster and more optimized, we give the user the chance to change the step of k , instead of 1 as a constant value.

Notes on sigma squared calculation and Multi Modal Thresholding.

As we can see in Equations 03 and 04, there are 2 formulas to calculate the ***Between-Class Variance*** metric. We'll refer to equation 03 as sigma squared, and equation 04 as between-class variance in this section. We found that it's advised to use sigma squared equation but in our trials, we found that between-class variance gives better results in terms of separability measure and overall performance, especially for Multi Modal Thresholding. Below is the comparison on 2 images.

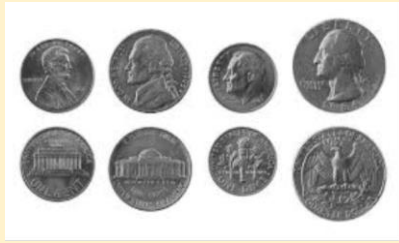


Figure 2, Input image: Coins Image

The minimum between classes variance is at index 178 with eta value 0.938
The minimum sigma squared eta is at index 146 with eta value 0.571

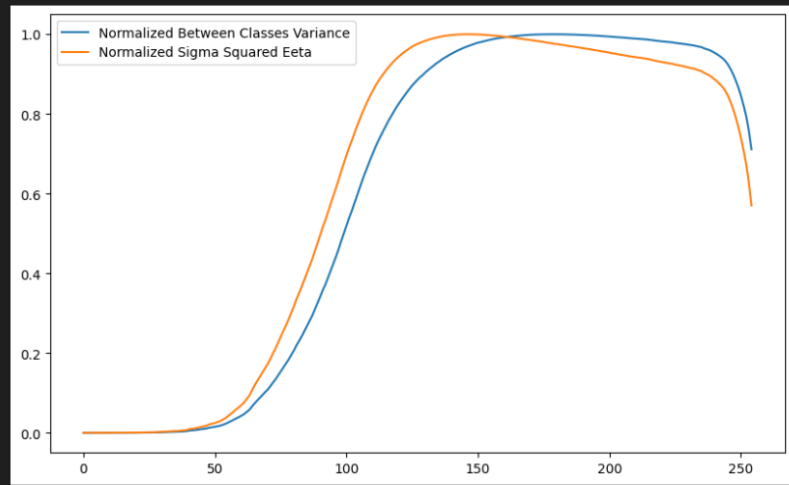


Figure 3, The results of Binary Thresholding (number_of_thresholds =1, step =1) on Coins Image.



Figure 4, Input Image: Brandenburg gate

The minimum between classes variance is at index 1606 with eta value 0.941
The minimum sigma squared eta is at index 2414 with eta value 0.895

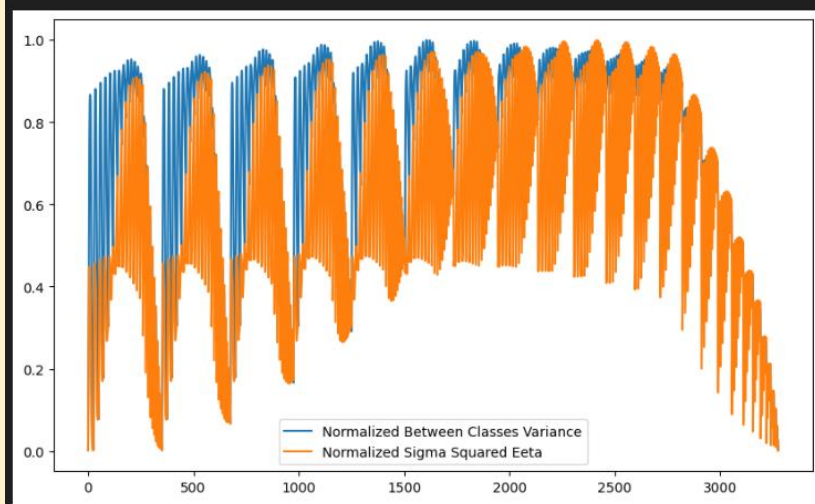


Figure 5, The results of Multi Modal Thresholding (number_of_thresholds =3, step =10) on Brandenburg gate

Visualizing the classes

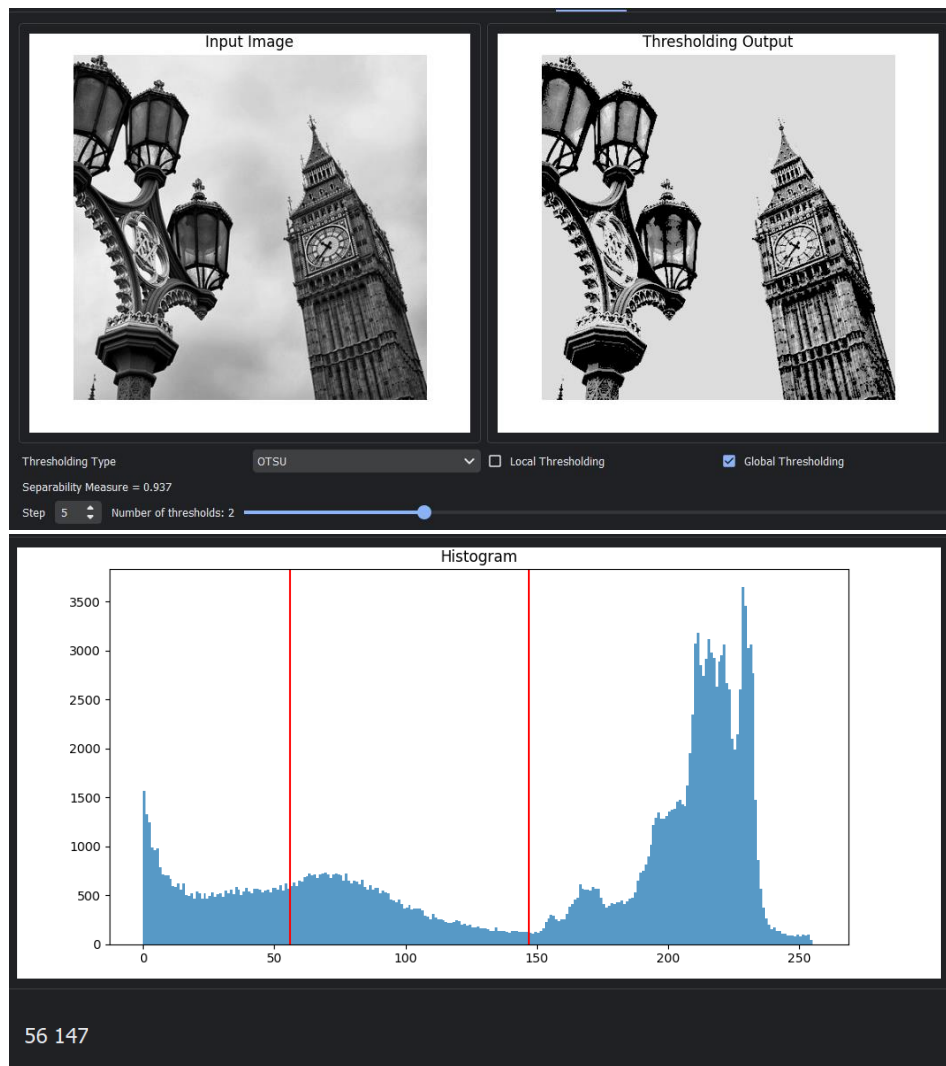
We currently visualize the resulting image as a 2D greyscale image which means that each region/class has a single value $\in [0, 255]$ assigned to it. The list of levels depends on the number of thresholds, following that:

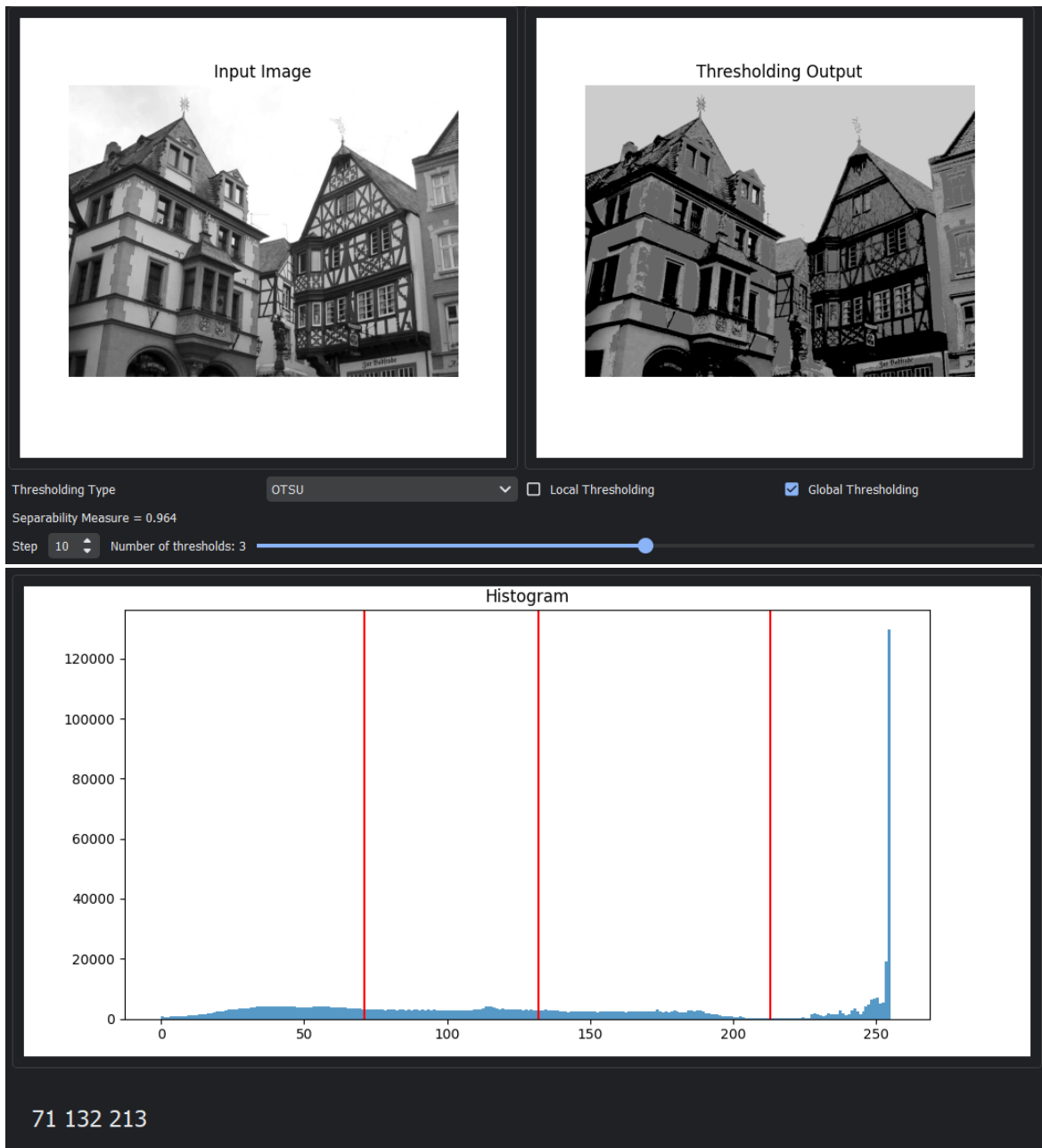
- Start = 0
- End = 255
- Step = $255/\text{number of thresholds}$

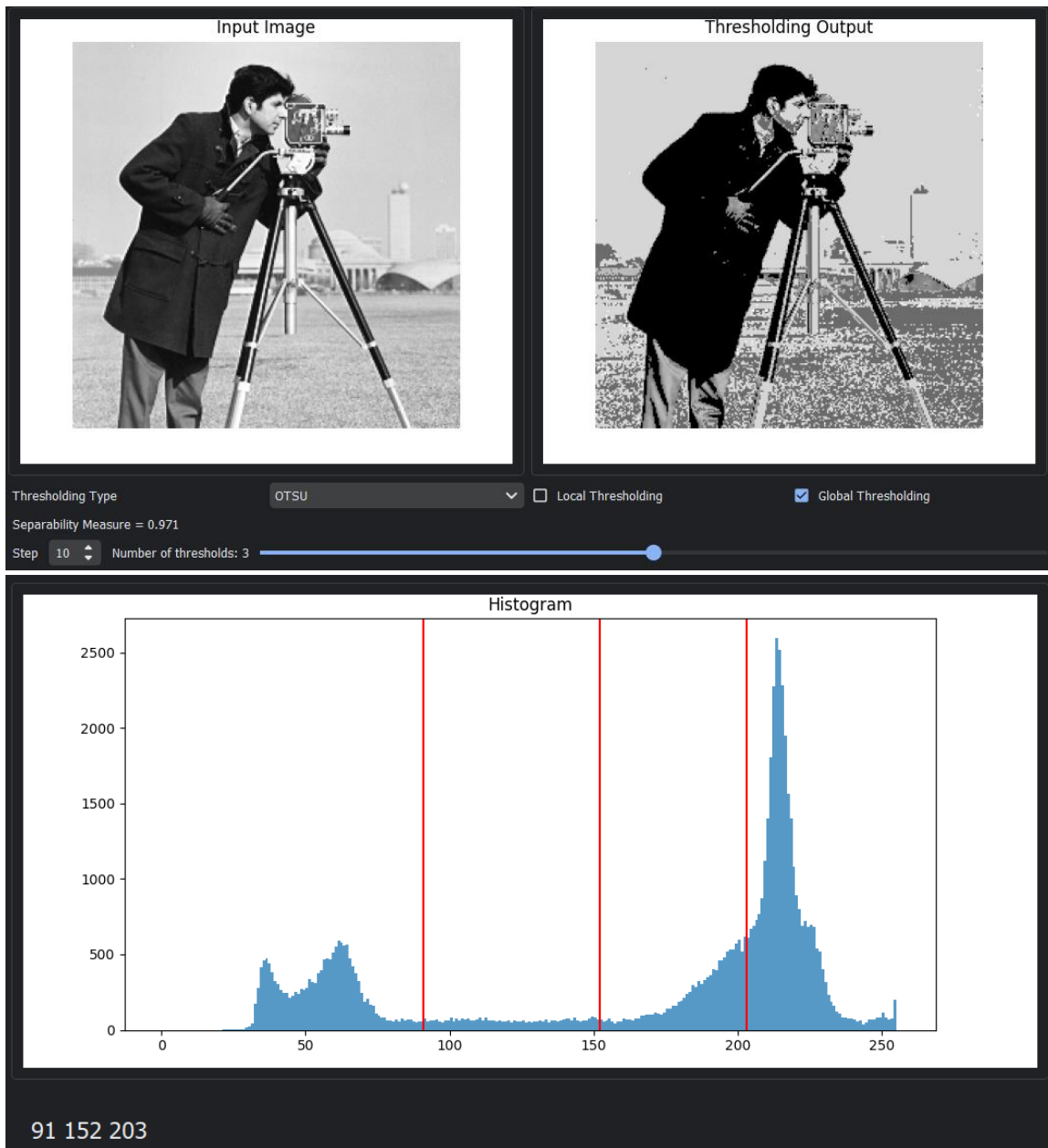
For example, for number of thresholds = 2, the levels are {0, 128, 255}.

Results

In our application, the user can manipulate the number of thresholds, 6 maximum for better value and the step taken in each combination of threshold values for optimization purposes.







Local Thresholding

Brief

A basic approach to variable thresholding is to compute a threshold at every point, $f(x, y)$ in the image based on one or more specified properties in a neighborhood of $f(x, y)$. Although this may seem like a laborious process, modern algorithms and hardware allow for fast neighborhood processing, especially for common functions such as logical and arithmetic operations.

The Algorithm

A window, not the full image.

It applies the same previous methods (Optimal Thresholding and Otsu Thresholding) on a window of the image. The dimension of the window is defined using a kernel. So the first step in the process is padding the image to be equally divided by the kernel.

Optimization Tricks

The traditional method is assigning one pixel value in the output image based on the kernel output, but we have applied an optimization trick to avoid that. In our application, we changed the step and the pixels to be assigned, instead of one each time, it moves and assigns **half of the kernel size** each time. We noticed that that **increased the speed by 4x** in one of our machines with less noisy/good output.

Also, we detected when all the window pixels have the same value, we assign the resulting image directly as there's no meaning of applying a thresholding algorithm.



Figure 6, Optimization Trick result comparison on Leena Image, the image resulting from optimization is on the right.

Results of Local Thresholding

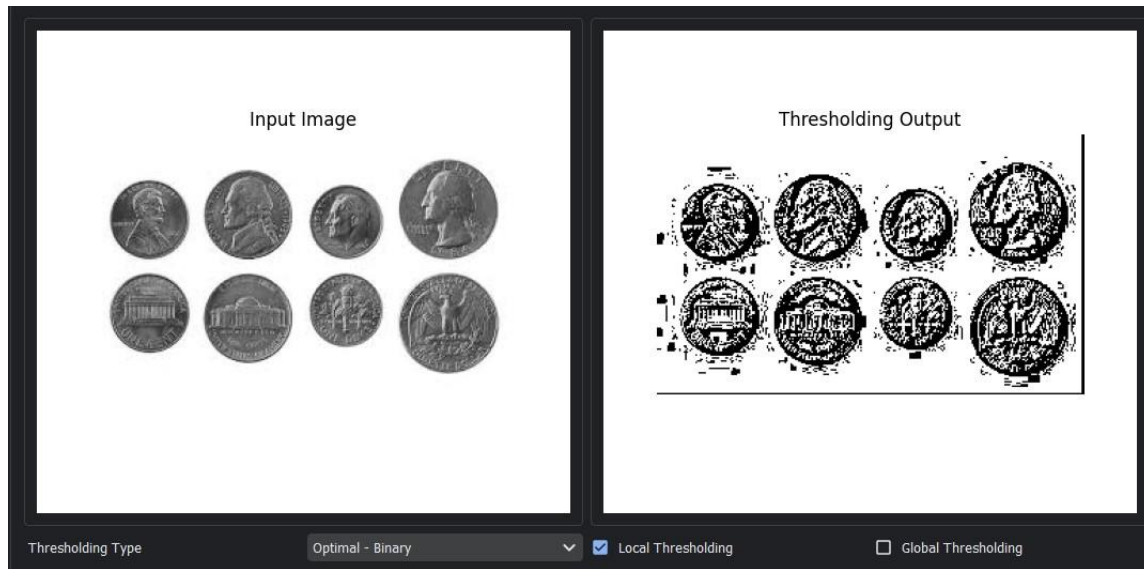


Figure 7, The result of local thresholding using Optimal Thresholding method on Coins Image.

Region Growing Segmentation

Overview

The region-growing algorithm is a segmentation method where users pick specific points in the image as seeds. These seeds then expand to form the desired region. Here's a summary of how it works:

First, a seed pixel is chosen, and the algorithm looks at a window of pixels around it. It compares the intensity of these pixels with the mean intensity of the region, which initially is just the intensity of the seed. If a pixel's intensity is similar to the region's mean within a certain threshold, it's marked as visited and becomes part of the region. Otherwise, it's only marked as visited.

Then, each visited pixel identified as part of the region becomes a new seed, and the process repeats, updating the mean intensity with each iteration. When the algorithm can't find any more neighbouring pixels to add to the region, it stops and outlines the region.

Region growing is useful for detecting irregular shapes, but it relies on accurate seed placement by the user, which can introduce errors.

Algorithm's Syntax Explanation

1. Initialization:

- Initialize a mask called "visited" to keep track of visited pixels. It has the same dimensions as the input grayscale image and is initialized with all false values.
- Initialize an empty segmented image, also with the same dimensions as the input image.

2. Seed Point Selection:

- For each seed point selected by the user:
 - Get the grayscale intensity of the seed point as the initial region mean.
 - Initialize a queue with the seed point.

3. Region Growing Loop:

- Start a loop that continues until the queue is empty.
- Within the loop:
 - Pop a pixel from the queue.
 - Check if the pixel is within the image bounds and has not been visited.
 - If conditions are met:

- Mark the pixel as visited.
- Check if the grayscale intensity of the pixel is similar to the region mean, based on the specified threshold.
- If the similarity condition is satisfied:
 - Add the pixel to the segmented region.
 - Update the region mean based on the pixels in the segmented region.
 - Enqueue the neighbouring pixels for further processing.

4. Segmented Image Visualization:

- Plot the segmented image by finding the contours of the segmented regions.
- Draw the contours on a copy of the input image.
- Display the output image with contours drawn.

Results

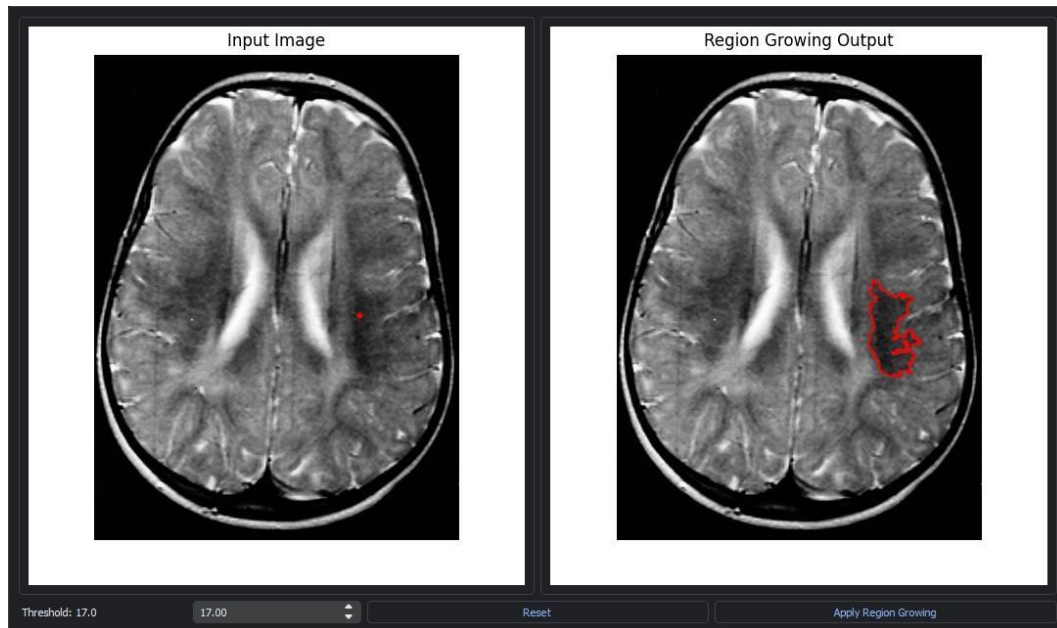


Figure 8, the region-growing result using one seed and an intensity threshold of 17

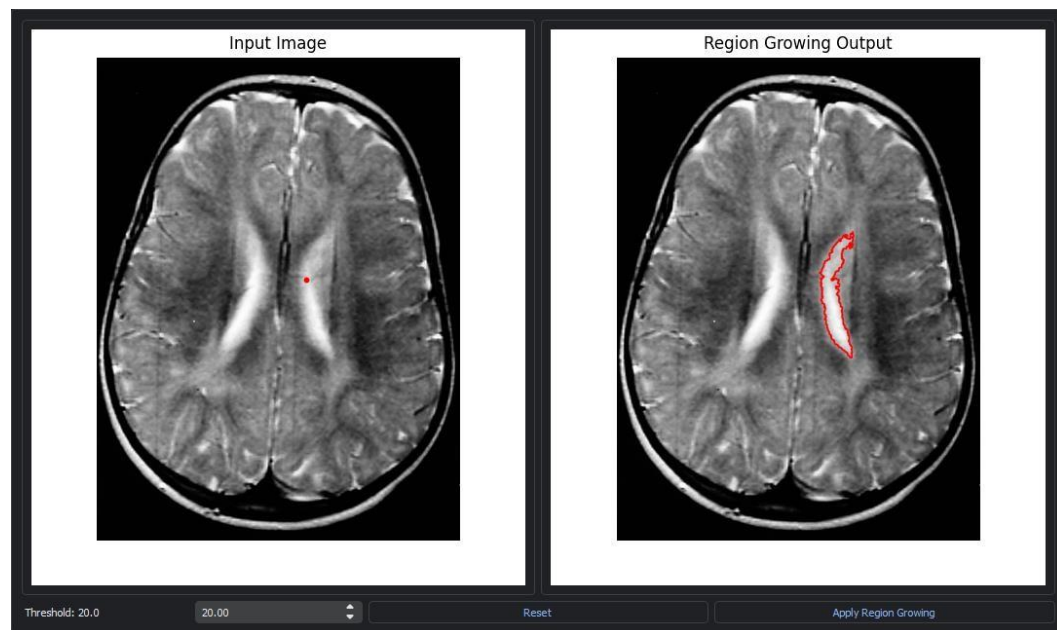


Figure 9, the region-growing result using one seed and a threshold of 20

Agglomerative Clustering

Overview

Agglomerative clustering is a hierarchical clustering method where each data point initially represents a single cluster, and at each iteration, the two closest clusters are merged based on a specified distance metric. This process continues until all data points belong to a single cluster or until a specified number of clusters is reached. Agglomerative clustering produces a dendrogram, allowing visualization of the clustering process and enabling the determination of the optimal number of clusters.

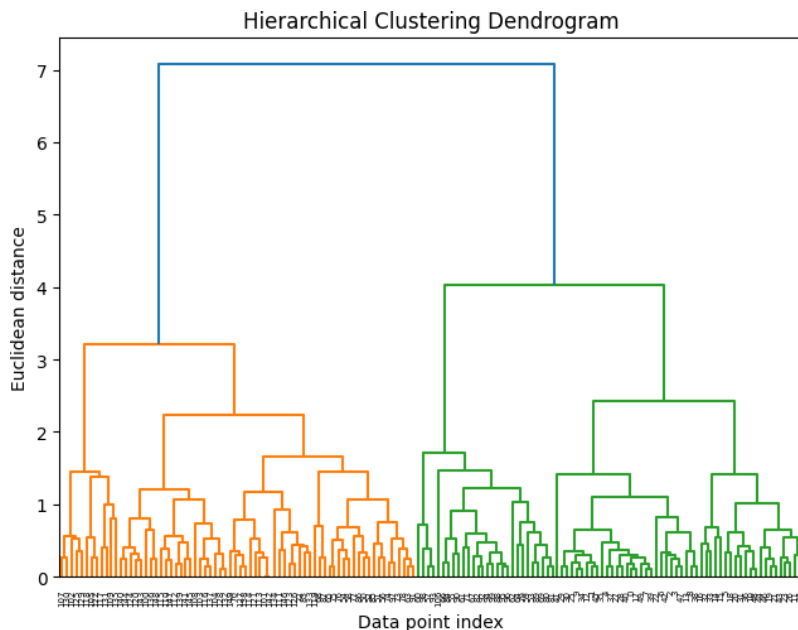


Figure 10, Example of a hierarchical clustering dendrogram

Algorithm's Steps Explanation

1. **Parameter Setup:** This step involves configuring parameters such as whether to downsample the image, the desired number of clusters, the initial number of clusters the pixels will be partitioned to, and the scale factor for downsampling.
2. **Downsampling [optional]:** If downsampling is enabled, the image is resized to a smaller version to reduce computational complexity. This is done by dividing the original image dimensions by the scale factor to obtain new dimensions.
3. **Reshape Image:** The image is reshaped into a format suitable for clustering. Typically, each pixel is represented as a point in a high-dimensional space, with each dimension corresponding to a colour channel (e.g., red, green, blue).

4. **Define Distance Metrics:** Functions are implemented to calculate distances between points and clusters. Euclidean distance is commonly used, measuring the straight-line distance between two points in the colour space.
5. **Initial Clustering:** Pixels are initially partitioned into clusters based on colour similarity. This is often done by iteratively assigning each pixel to the cluster with the closest centroid colour. By default, the pixels are partitioned into 25 clusters but the user can manipulate this property through the GUI of the app.
6. **Merge Clusters:** The agglomerative clustering process begins. At each iteration, the two closest clusters are merged. The distance between clusters can be computed based on the distance between individual points or the centroids of the clusters.
7. **Assign Cluster Numbers:** After merging clusters, each pixel is reassigned to its new corresponding cluster. This step ensures that all pixels are associated with the correct cluster after merging.
8. **Compute Cluster Centers:** The centroids of the newly formed clusters are recalculated. The centroid of a cluster is the mean of the points (pixels) belonging to that cluster.
9. **Apply Clustering:** The agglomerative clustering process is applied to the image pixels. Each pixel is reassigned to the cluster represented by its closest centroid.
10. **Display Result:** The segmented image, where each pixel is coloured according to its corresponding cluster centroid, is visualized. This provides a clear representation of the image segmentation achieved by the clustering algorithm.

Results

Based on the prior explanation and experimentation, it's evident that this process can be time-consuming. Thus, determining the initial number of clusters for pixel partitioning and the desired final number of clusters is crucial. Additionally, downsampling the image with an appropriate factor can expedite results. Analysis from Figures 4 and 5 indicates that maintaining a fixed initial cluster count while increasing the expected cluster count reduces processing time, aligning with expectations.

A general rule emerges: widening the gap between the initial and expected cluster counts prolongs processing time. Moreover, Figures 6 and 7 demonstrate how downsampling aids in time reduction.



Figure 11, Agglomerative Clustering with an initial number of clusters of 25 and 2 clusters as the termination criteria, the time elapsed is 7 minutes and 14 seconds



Figure 12, Agglomerative Clustering with an initial number of clusters of 25 and 4 clusters as the termination criteria, the time elapsed is 7 minutes and around 4 seconds

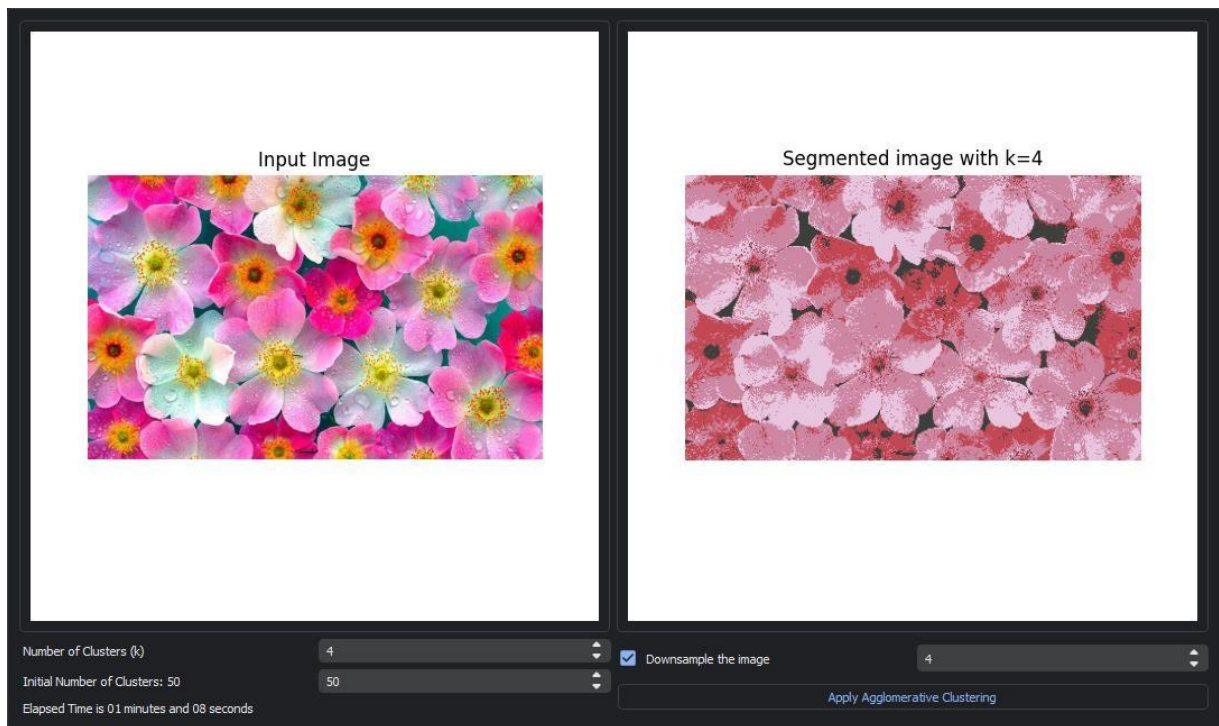


Figure 13, Agglomerative Clustering with an initial number of clusters of 50 and 4 clusters as the termination criteria with downsampling by the factor of 4, the time elapsed is highly reduced to 1 minute and 8 seconds

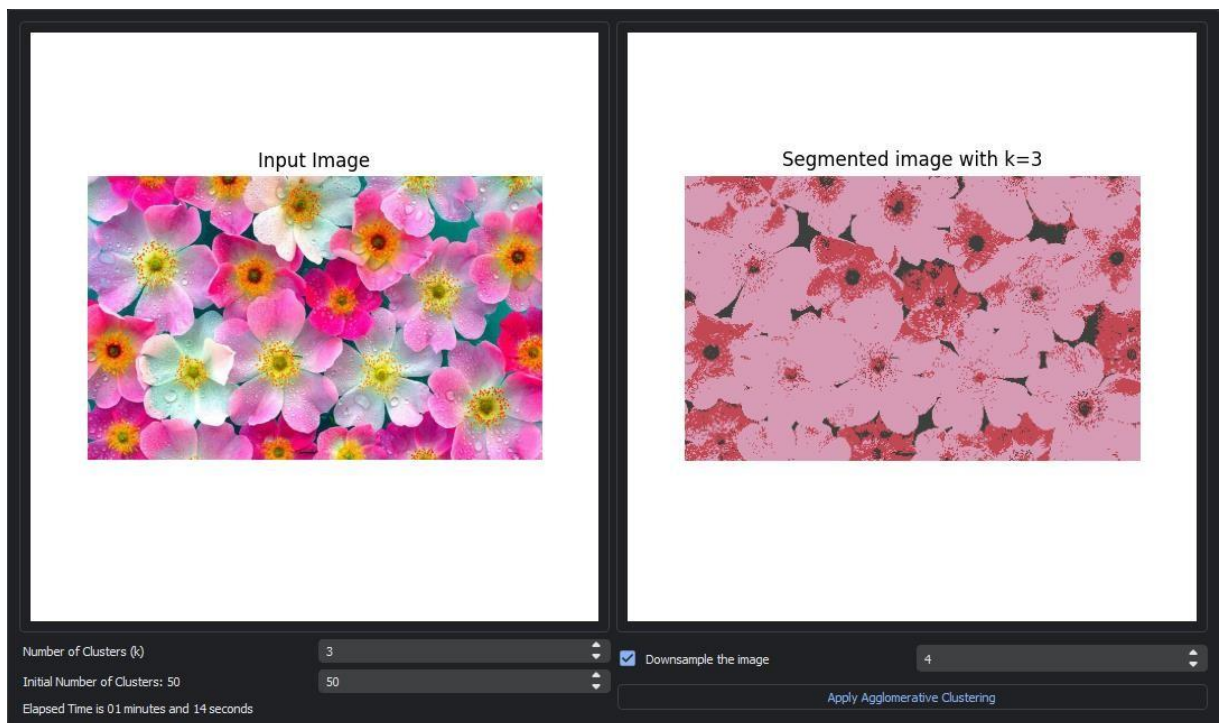


Figure 14, Agglomerative Clustering with an initial number of clusters of 50 and 3 clusters as the termination criteria with downsampling by the factor of 4, the time elapsed is highly reduced to 1 minute and 14 seconds

K-Means Clustering

The k-means clustering algorithm is fairly simple algorithm that enable us to segment a coloured image in the colour space. The algorithm works as the following:

1. Construct the feature space (colour space).
2. Randomly select (k) points in the colour space to be our initial centroids of the (k) clusters.
3. Start iterating over the entire points in the colour space and compute their distances from the centroids.
4. Form clusters by assigning each point in the space to a cluster based on its proximity from one of the centroids.
5. Recompute the centroids of each cluster by computing the average of each cluster.
6. Get back to step (3) and keep preceding until reaching the maximum number of iterations.

This is a pseudocode of the algorithm that is widely used. However, we have added some additional steps and modified others steps to optimize the algorithm as well as to enforce some constraints as desirable, namely:

1. Optimization of centroid initialization.
2. Inclusion of spatial segmentation

Optimization of Centroid Initialization

Instead of randomly initializing the centroids of the clusters, we will optimize this step by selecting centroids that are maximally separated from each other. This can be achieved by computing the distances from each obtained centroids and other points, then selecting from the most probabilistic point that might achieve this objective. Below is a code sample that illustrate this technique, the reason for such implementation is to optimize computational time as possible.

```

1 def initialize_centroids_indices(img_as_features, nclusters):
2     centroids_indices = np.zeros(nclusters, dtype=int)
3
4     centroids_indices[0] = np.random.choice(img_as_features.shape[0])
5
6     for i in range(1, nclusters):
7         distances = np.min(
8             np.linalg.norm(
9                 img_as_features[centroids_indices[:i]][:, None] - img_as_features,
10                axis=2,
11            )
12            ** 2,
13            axis=0,
14        )
15
16        probabilities = distances / np.sum(
17            distances
18        ) # represent distances as probability
19
20        next_centroid_index = np.random.choice(
21            np.arange(len(img_as_features)), p=probabilities
22        ) # select by probability
23        centroids_indices[i] = next_centroid_index
24
25    return centroids_indices

```

Figure 8, Code sample of initialization technique adopted in the algorithm

After initializing the centroids, we will apply the k-means algorithm on a sub-set of the feature space. By this way, we would ensure well-positioning of centroids, thus, well-formed clusters. This could be achieved by recursive call of the function with maximally separated centroid.



Figure 9, Input Sample Image



*Figure 10, Sample output with **no centroid optimization***



*Figure 11, Sample output with **centroid optimization***

There is no dramatic difference between them in case of segmenting in the colour space only. Note that we have used in the previous experiment ($k = 10$ & max no of iterations = 10). The running in case of no centroid optimization is (2 sec), while in the other case it is equal to (2.6 sec). As we can observe, there is no dramatic increase in computational time.

Inclusion of Spatial segmentation

Instead of segmenting in the colour space only, we will consider the spatial info of each pixel. This will help use form a more uniform segment in terms of space. Therefore, instead of forming clusters that might be scattered in the image, we will form clusters that are concentrated in specific region in the image. Definitely, we can control the amount of spatial information included in the segmentation process by tuning the spatial factor. Below are the experimental results with and without centroids optimization besides including spatial data in the segmentation.

As we can observe, the spatial segmentation inclusion has helped in creating more homogeneous segmented regions in the output image. Moreover, we can now get a sense of the effect of centroids

optimization, where the red cluster that is apparent in the output of segmentation with optimization turned on is completely vanished in the case of no optimization.

If we have increased the factor of spatial inclusion up to 1, we would see a more homogeneous segmented regions as shown below.



*Figure 12, Sample output with spatial segmentation included at a factor of 0.5 and **without centroid optimization***



*Figure 13, Sample output with spatial segmentation included at a factor of 0.5 and **with centroid optimization***

As we can observe, the spatial segmentation inclusion has helped in creating more homogeneous segmented regions in the output image. Moreover, we can now get a sense of the effect of centroids optimization, where the red cluster that is apparent in the output of segmentation with optimization turned on is completely vanished in the case of no optimization.

If we have increased the factor of spatial inclusion up to 1, we would see a more homogeneous segmented regions as shown below.



Figure 14, Sample output with spatial segmentation at a factor of 1 with centroids optimization.

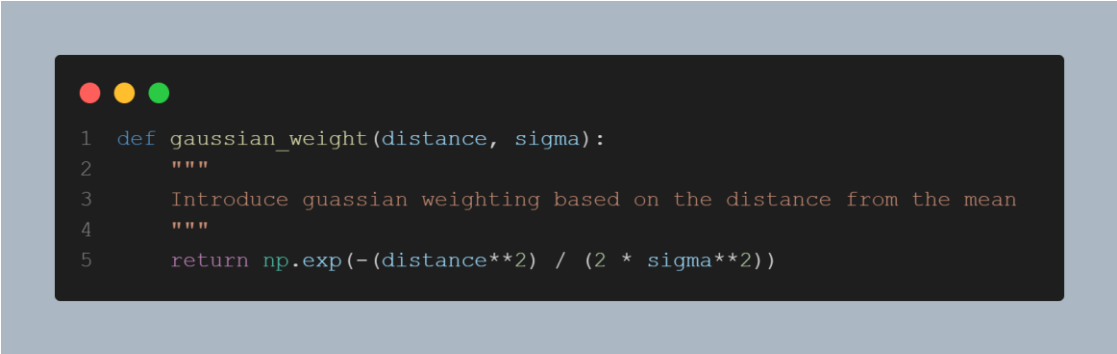
Mean-Shift Clustering

It's a clustering technique that makes use of points' density to determine the centroids of clusters, where the region of high density signifies the existence of candidate centroid. The algorithm goes like the following:

1. Constructing the feature space (colour space).
2. Iteration over all the points in the space that have not added to a cluster YET.
3. Randomly select point and consider it the initial mean.
4. Compute the distances between that point and the rest data points in the colour space.
5. Finding all the points that has a distance from that selected point that is less than half of window size of specific length.
6. Marking all the nearby points as visited. (Belonging to the same cluster)
7. Averaging the data points in that vicinity.
8. Assign the result as the new mean.
9. If the new mean is different from the initial mean with some threshold value, then assign the initial mean with the new mean and go to step 3.
10. Else we would check if the current new mean has been reached before from any other starting point, if no, then we would construct a new cluster and assign the centroid of it to the new mean and all the points that have been visited along the way to its cluster.
11. If yes, then we would search for the cluster that has the same centroid and merge their clusters.

All we need to pass for this algorithm is the window size and threshold of convergence. We have added another parameter, which is the sigma of the Gaussian distribution that is used in weighting the points which contributing to the new mean from the points that are within the window size around the initial mean. This would help in considering the density around the initial mean, if it is denser, this means that the new mean would be far away from the initial mean and vice versa.

Sample Code



```
1 def gaussian_weight(distance, sigma):
2     """
3     Introduce gaussian weighting based on the distance from the mean
4     """
5     return np.exp(-(distance**2) / (2 * sigma**2))
```

Figure 15, This function is used for density consideration

Results



Figure 16, Sample output with window size of 200, threshold of 10 and sigma of 20



Figure 17, Sample output with window size of 300, threshold of 10 and sigma of 20



Figure 18, Sample output with window size of 200, threshold of 50, and sigma of 20