Computer Vision
Task 4
Team 18

Ahmed Kamal Mohamed	Sec.1
Amgad Atef Abd Al-Hakeem	Sec.1
Mahmoud Mohamed Ali	Sec.2
Mahmoud Magdy Mohamed	Sec.2
Mohanad Emad El-Sayed	Sec.2

Supervised By

Dr. Ahmed Badawi

Eng. Lila Abbas

Eng. Omar Hesham

TABLE OF CONTENTS

1 Mean Shift Algorithm:

1.1 Initialization:

1. **Bandwidth Initialization:** Choose an appropriate bandwidth h for the kernel function.
2. **Mean Shift Vector Initialization:** Initialize the mean shift vector for each data point to zero.

1.2 Mean Shift Iteration:

1. **Kernel Computation:** Compute the kernel function for each data point.

2. **Weighted Average Calculation:** Compute the weighted average of nearby data points using the kernel values as weights.

3. **Mean Shift Vector Update:** Update the mean shift vector for each data point based on the computed weighted average.

1.3 Convergence Check:

Threshold Check:

1. Calculate the magnitude of the mean shift vector.
2. Compare it with a predefined convergence threshold.
3. If the magnitude is below the threshold for all data points, consider convergence.

1.4 Cluster Assignment:

1. **Mode Identification**:
   - Identify the modes (peaks) in the data space.
   - Each mode represents a cluster center.

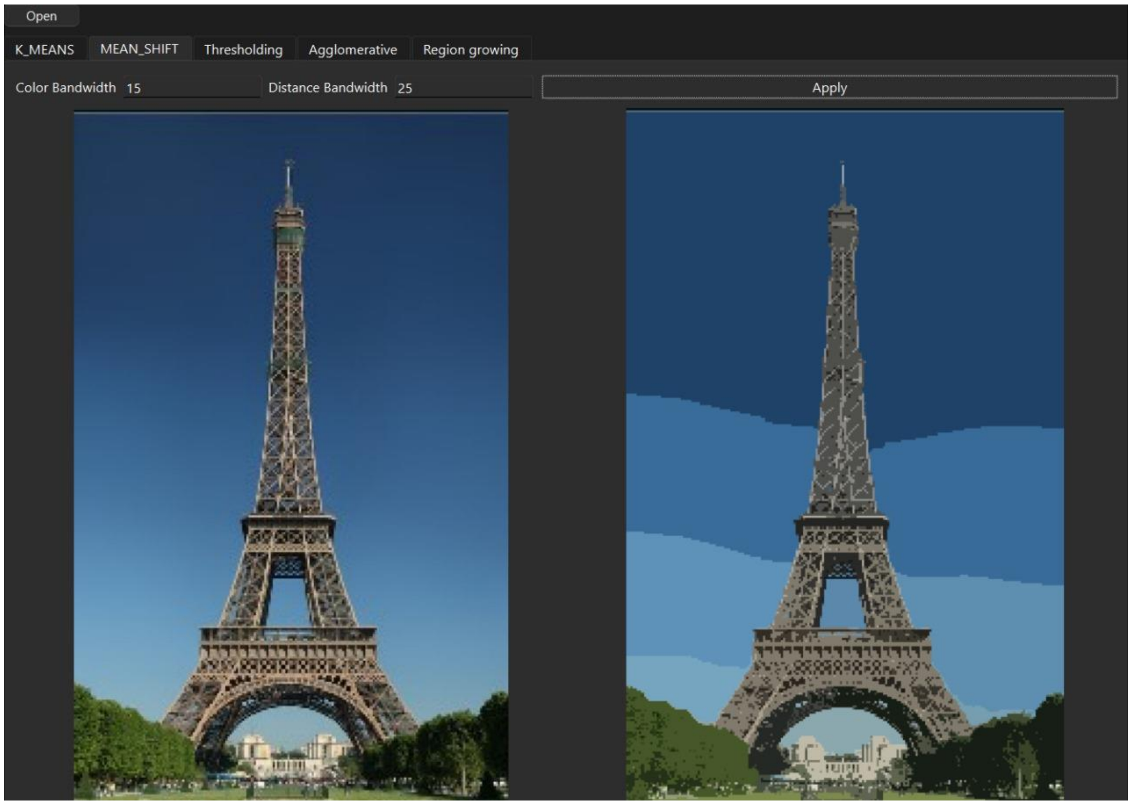
1. **Nearest Mode Assignment**:

   - Assign each data point to the nearest mode based on distance in the data space.

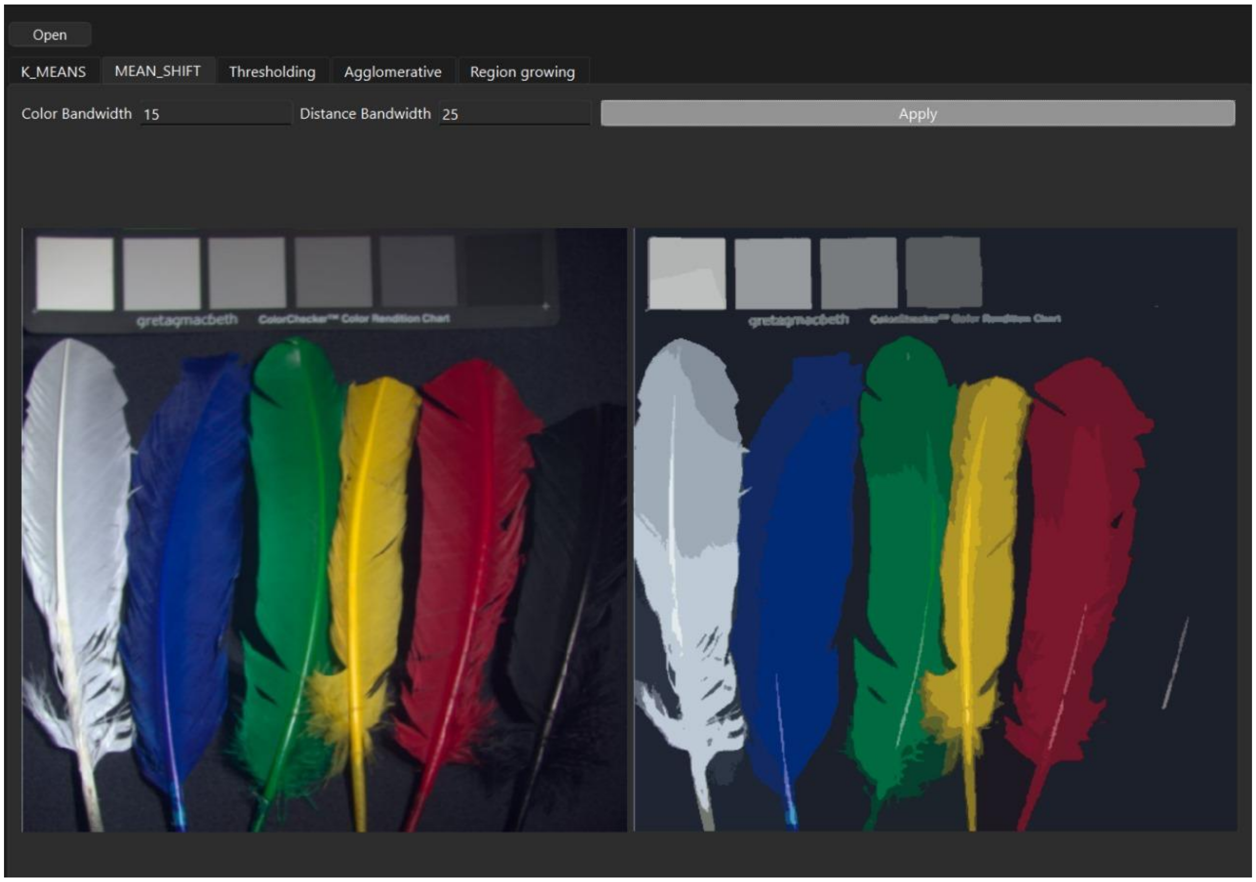
2. **Cluster Labeling**:

   - Label each data point with the cluster it belongs to based on the assigned mode.

1.5 output:



Color Bandwidth 15 Distance Bandwidth 25



Color Bandwidth 15 Distance Bandwidth 25

2 Agglomerative:

2.1 Read and Prepare Data:

1. Read the input image data.
2. Convert the image data to a suitable format (e.g., grayscale).
3. Reshape the image data into a one-dimensional array.

2.2 Initialization:

1. Choose the number of clusters, k.
2. Initialize each data point as a separate cluster.

2.3 Compute Pairwise Distances:

Compute the Euclidean distance between each pair of clusters.

2.4 Merge Closest Clusters:

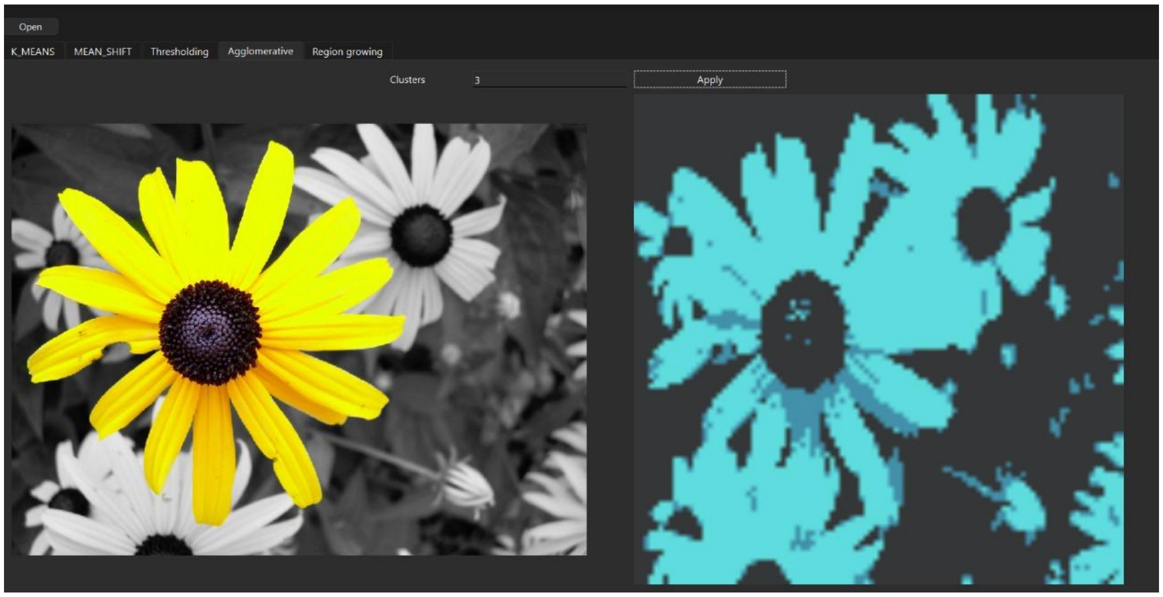
1. Find the closest pair of clusters based on distance.
2. Merge the closest pair of clusters into a single cluster.

2.5 Update Cluster Membership:

1. Update the cluster membership for each data point.
2. Assign each data point to the nearest cluster centroid.

2.6 Repeat Until Convergence:

1. Repeat steps 4-6 until the desired number of clusters (k) is reached.
2. Terminate when the number of clusters reaches k.



Clusters 3

3 K-Means:

3.1 Initialization:

Random Initialization: Generate k random centroids within the data range.

3.2 Iterations:

1. **Iteration Counter:** Initialize an iteration counter to track the number of iterations.

2. **Convergence Criteria:** Define a convergence criterion (e.g., maximum number of iterations or a small change in centroids).

3.3 Assignment Step:

1. **Data Point Iteration**: Iterate over each data point in the dataset.
2. **Distance Calculation**: Calculate the distance between each data point and all centroids.
3. **Nearest Centroid**: Assign each data point to the centroid with the minimum distance.

3.4 Update Step:

Centroid Update:

For each cluster:

- Calculate the mean of all data points assigned to the cluster.
- Update the centroid to the mean value.

3.5 Convergence Check:

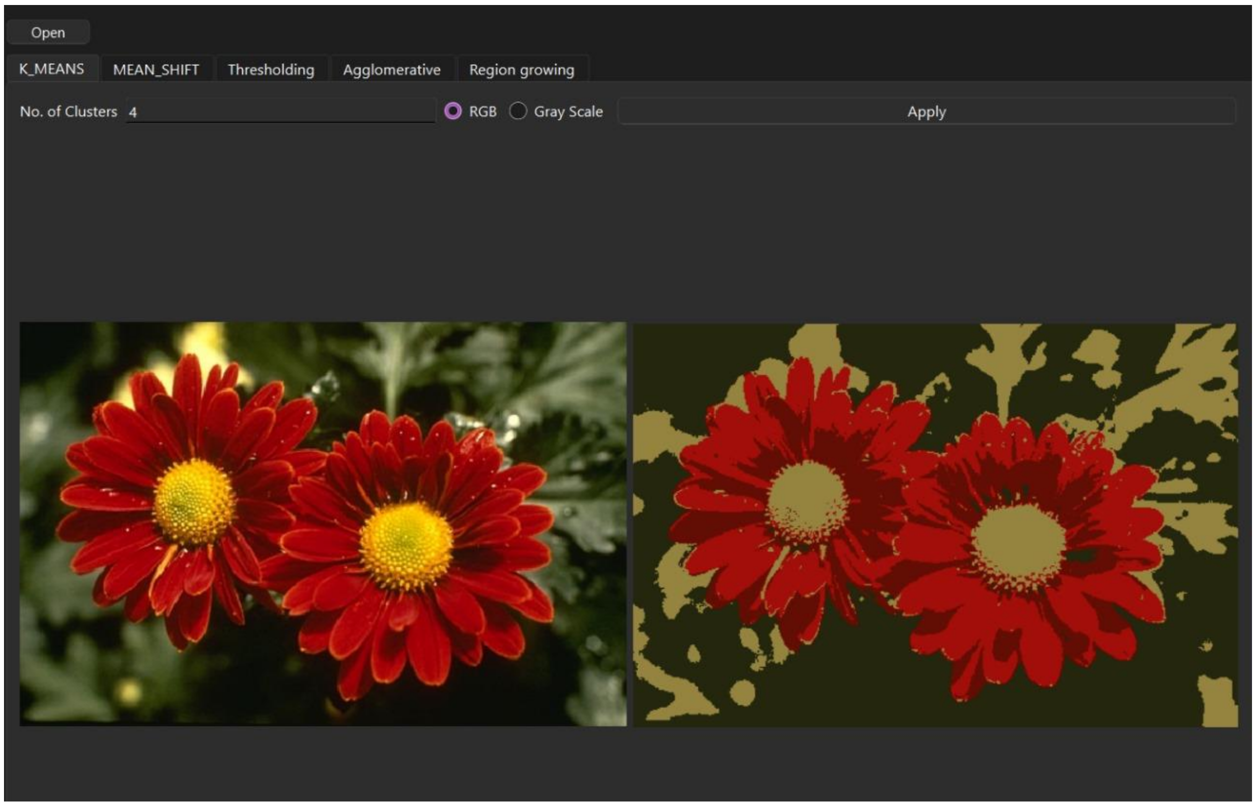
1. **Centroid Change Calculation:** Calculate the change in centroids' positions between consecutive iterations.

2. **Convergence Criterion Check:** Check if the change in centroids is below a predefined threshold or if the maximum number of iterations is reached.

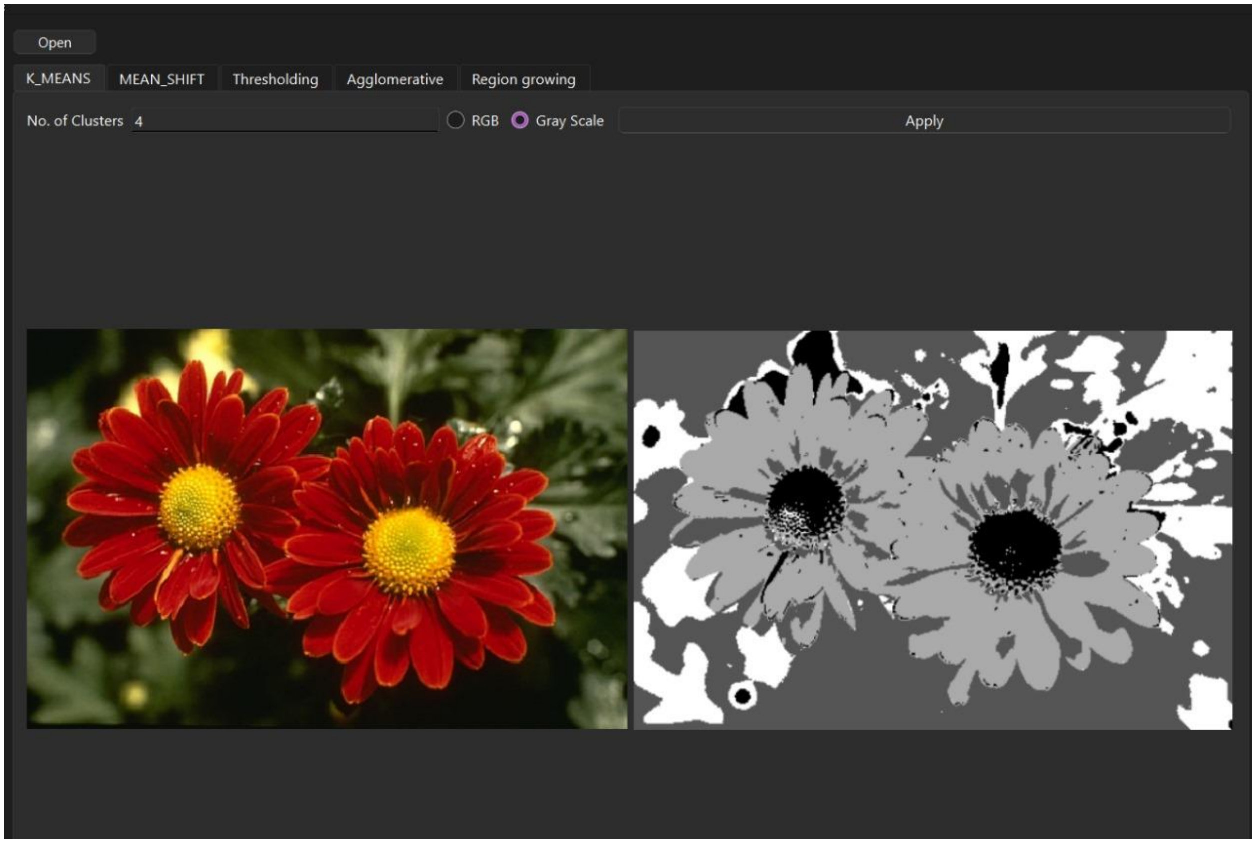
3.6 Output:

1. **Final Centroids:** Return the final centroid positions after convergence.

2. **Cluster Assignments:** Return the cluster assignments for each data point based on the final centroids.



No. of Clusters 4 [✓ RGB] Gray Scale



No. of Clusters 4 RGB [✓ Gray Scale]

4 Region Growing:

4.1 color_region Function:

1. **Iterate Over Pixels:** Iterate over each pixel in the image.
2. **Colorization Based on Labels:**
    - Retrieve the label of the current pixel from the label matrix.
    - **If `colorflag` is true:** Copy the pixel value from the input image to the output image based on the label.
    - **If `colorflag` is false**:
        - Assign a unique color to each region based on its label.
        - Colorize the pixel with the assigned color in the output image.

4.2 region_detection Function:

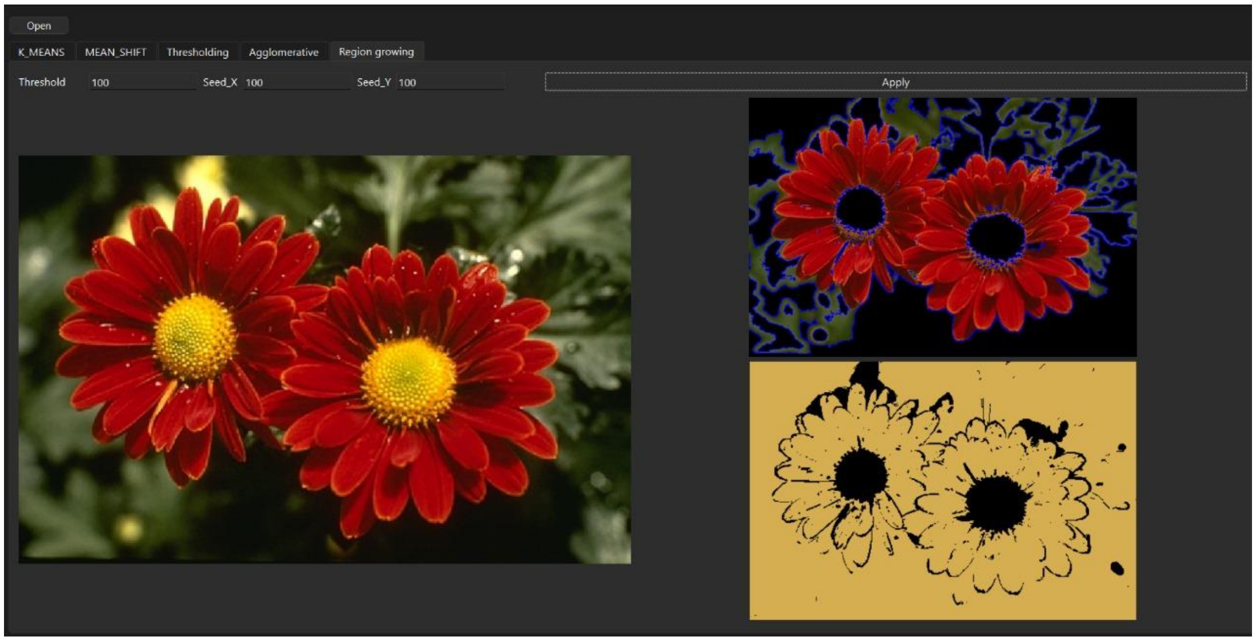
1. **Region Detection Loop:** Iterate while the queue Q is not empty.
2. **Dequeue Point:** Dequeue the next point from Q.
3. **Neighbor Pixel Check:** Check the 8-connected neighborhood of the current point.
4. **Region Growing:**
    - If `colorflag` is true:
        - Calculate the absolute difference between the current pixel value and its neighbors.
        - If the difference is within the threshold, assign the current label to the neighboring point and add it to the queue.
    - If `colorflag` is false:
        - Calculate the difference between the current pixel value and its neighbors.
        - If the difference is within the threshold, assign a new label to the neighbor and add it to the queue.

4.3 regionGrowingHelper Function:

1. **Label Matrix Initialization:** Create an empty label matrix.
2. **Queue Initialization:** Create a queue and initialize it with the seed point.
3. **Region Growing:** Call `region_detection` to perform region growing based on the seed point.
4. **Colorization:**
    - If `colorflag` is true, convert the image to grayscale and call `region_detection.`
    - Create an output image and colorize regions based on the label matrix.

4.4 regionGrowingMultiSeed Function:

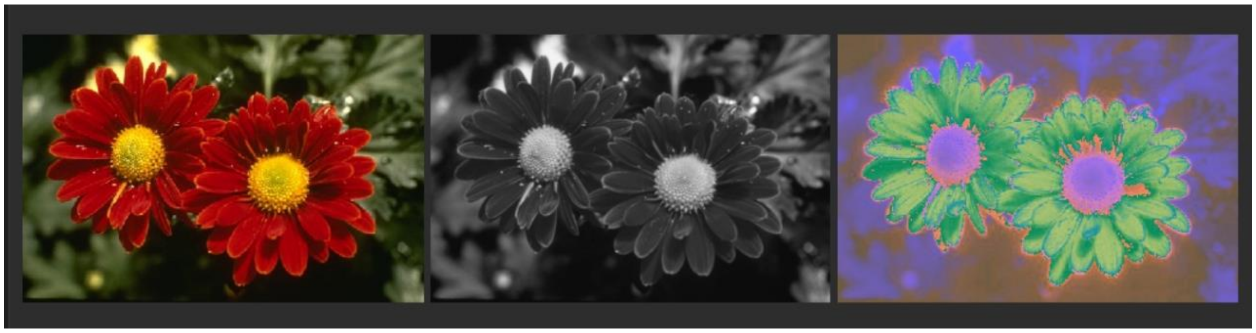
1. **Label Number Initialization:** Initialize the label number to 1.
2. **Seed Point Iteration:** Iterate over each seed point in `seedPoints`.
3. **Region Growing for Each Seed:** Call `regionGrowingHelper` for each seed point to grow the region.
4. **Label Number Increment:** Increment the label number for the next seed point.



Threshold 100 Seed_X 100 Seed_Y 100

5 RGB to LUV:

```
x = 0.412453 * pixel[2] / 255.0 + 0.357580 * pixel[1] / 255.0 + 0.180423 * pixel[0] / 255.0;
y = 0.212671 * pixel[2] / 255.0 + 0.715160 * pixel[1] / 255.0 + 0.072169 * pixel[0] / 255.0;
z = 0.019334 * pixel[2] / 255.0 + 0.119193 * pixel[1] / 255.0 + 0.950227 * pixel[0] / 255.0;
```



6 Global Thresholding:

6.1 Otsu's Thresholding:

Explanation:

Otsu's method computes an optimal threshold value to separate an image into foreground and background regions by maximizing the inter-class variance of pixel intensities.

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$$

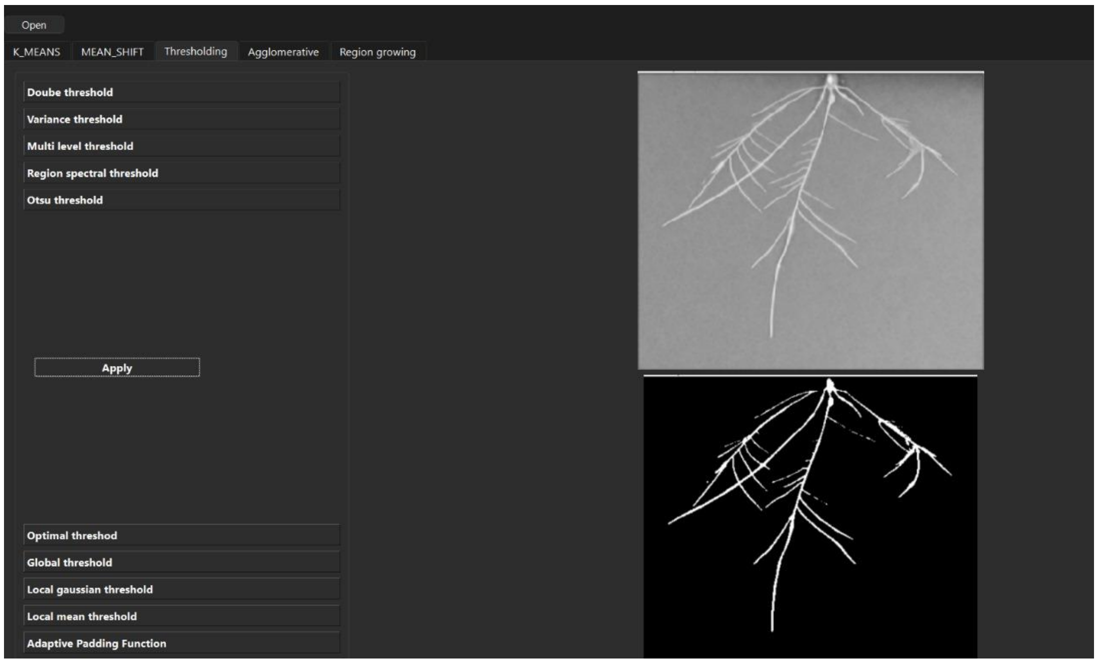
Code Explanation:

computeOtsuThreshold:

- Computes Otsu's threshold using the histogram of pixel intensities.
- Iterates through the histogram and calculates the threshold that maximizes the between-class variance.
- Returns the optimal threshold value.

otsuThreshold:

- Computes the histogram of pixel intensities in the input grayscale image.
- Calls `computeOtsuThreshold` to find the optimal threshold.
- Applies thresholding to the image based on the computed threshold.
- Returns the thresholded image.



6.2 Entropy Thresholding:

Explanation:

Entropy thresholding calculates the threshold that maximizes the information gain, measured by the entropy, between the background and foreground regions.

$$t_{opt} = argmax\left[s_q^A(t) + s_q^B(t) + (1-q) \, . \, s_q^A(t) \, . \, s_q^B(t)\right]$$

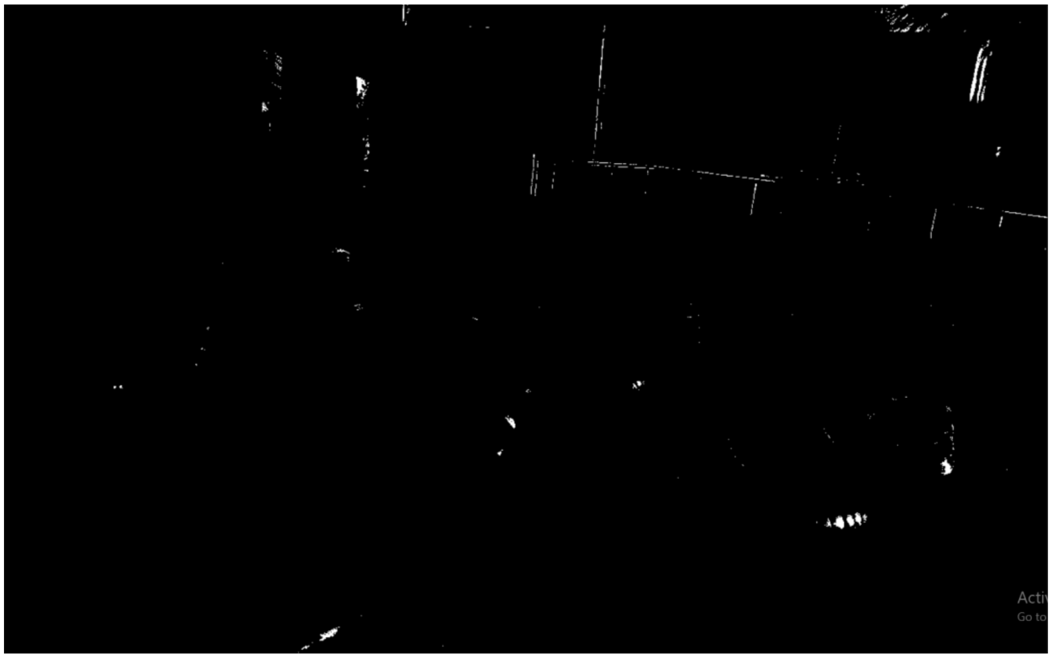
Code Explanation:

computeEntropy:

- Computes the entropy of the image using its histogram.
- Calculates the entropy based on the probability distribution of pixel intensities.
- Returns the entropy value.

entropyThreshold:

- Computes the histogram of pixel intensities in the input grayscale image.
- Calls computeEntropy to calculate the entropy.
- Finds the threshold that maximizes the information gain.
- Applies thresholding to the image based on the computed threshold.
- Returns the thresholded image.



6.3 Global threshold:

Global thresholding is a technique used in image processing to segment an image into binary regions based on a single threshold value. It's a simple yet effective method to separate objects from the background in an image. Global thresholding assumes that the intensity histogram of the image has distinct peaks corresponding to the foreground and background. The threshold value is chosen to separate these peaks, effectively dividing the image into two classes: foreground (object) and background.

Input Parameters:

Threshold value: The threshold value used for segmentation.

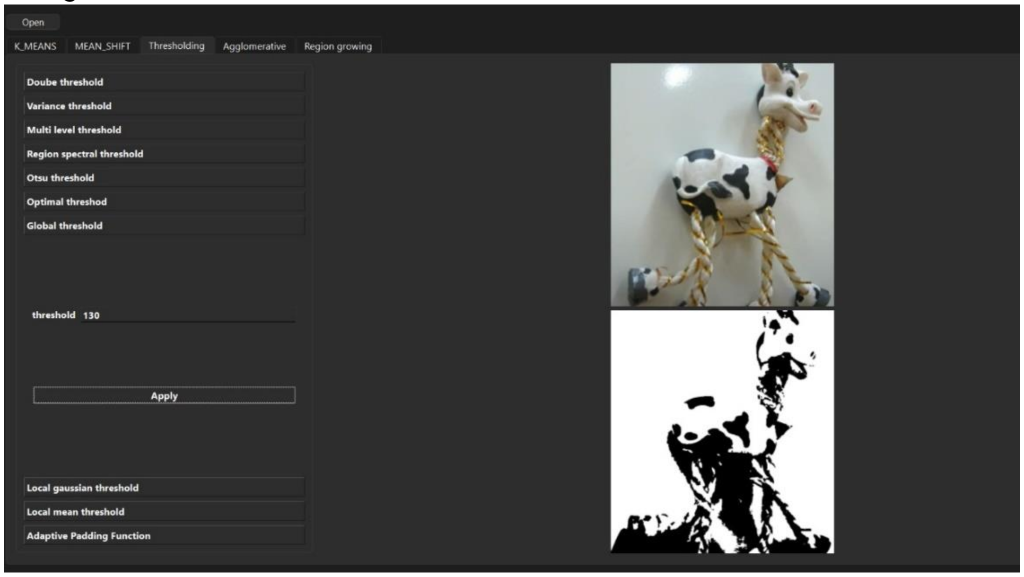
Maximum value: The maximum pixel value for the binary image (usually 255 for 8-bit images).

Minimum value: The minimum pixel value for the binary image (usually 0 for binary images).

$$\text{Binary Image}(x,y) = \begin{cases} 1 & \text{if Image}(x,y) \geq \text{Threshold} \\ 0 & \text{otherwise} \end{cases}$$

Pseudocode:

- The function clones the input image to ensure the original image is not modified.
- It iterates over each pixel of the input image.
- For each pixel, if its intensity value is greater than the threshold value, the corresponding pixel value in the result image is set to maximum value (typically 255), indicating foreground (object). Otherwise, it is set to minimum value (typically 0), indicating background



threshold 130

7 Local Thresholding:

7.1 Local Spectral Thresholding:

Explanation:

Local spectral thresholding calculates Otsu's threshold for small regions (neighborhoods) within the image instead of the entire image.

Code Explanation:

computeRegionHistogram:

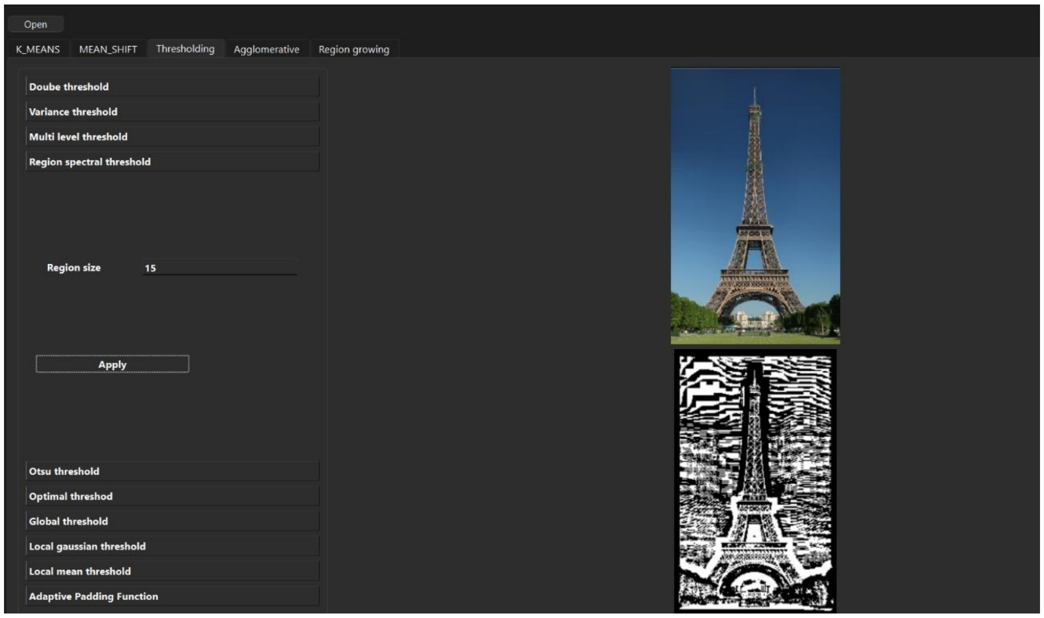
- Computes the histogram of a local region around a specified pixel.
- Considers a square region centered at the specified pixel with a given size.
- Returns the histogram of pixel intensities in the region.

computeRegionOtsuThreshold:

- Computes Otsu's threshold for a local region using its histogram.
- Finds the threshold that maximizes the between-class variance within the region.
- Returns the optimal threshold value.

localSpectralThreshold:

- Iterates over each pixel in the image.
- Computes the histogram and Otsu's threshold for the local region around each pixel.
- Applies thresholding to the pixel based on the computed threshold.
- Returns the thresholded image.

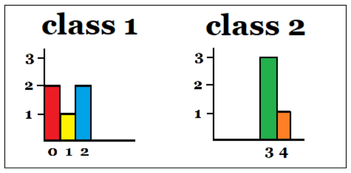


7.2 Variance Thresholding:

Explanation:

Variance thresholding applies a threshold based on the variance of pixel intensities within a local region.

# if pixels are classified into 2 classes,
then the between class variance $(V_b) = W_1W_2(\mu_1-\mu_2)^2$



$W_1 = 5/9 \quad W_2 = 4/9$

$\mu_1 = 1 \quad \mu_2 = 13/4$

$$V_b = \frac{5}{9} * \frac{4}{9} * \left(1-\frac{13}{4}\right)^2 = 1.25$$

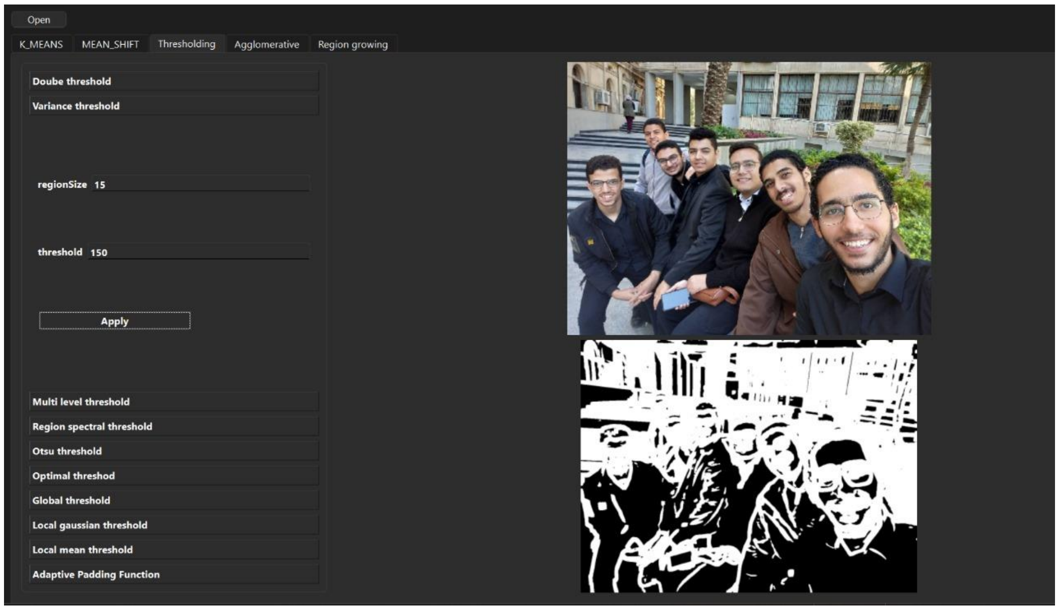
Code Explanation:

computeRegionVariance:

- Computes the variance of pixel intensities within a local region centered at a specified pixel.
- Considers a square region with a given size centered at the specified pixel.
- Returns the variance of pixel intensities in the region.

varianceThreshold:

- Iterates over each pixel in the image.
- Computes the variance for the local region around each pixel.
- Applies thresholding based on the computed variance.
- Returns the thresholded image.



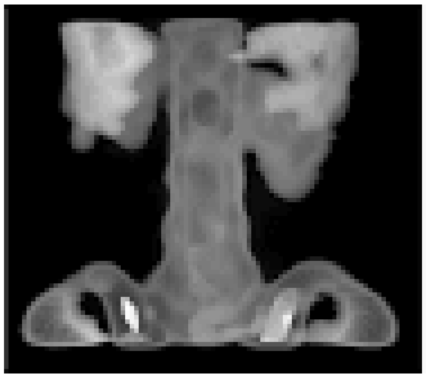
regionSize 15

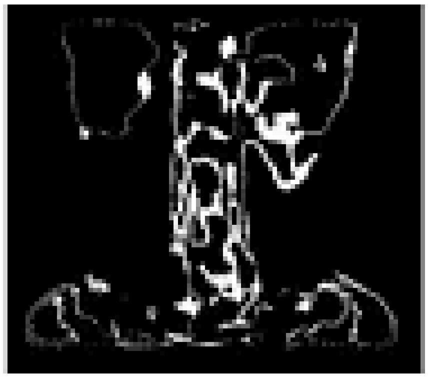
threshold 150

7.3 Double Thresholding:

Explanation:

Double thresholding segments the image into three regions: **low-intensity, medium-intensity**, and **high-intensity regions**.

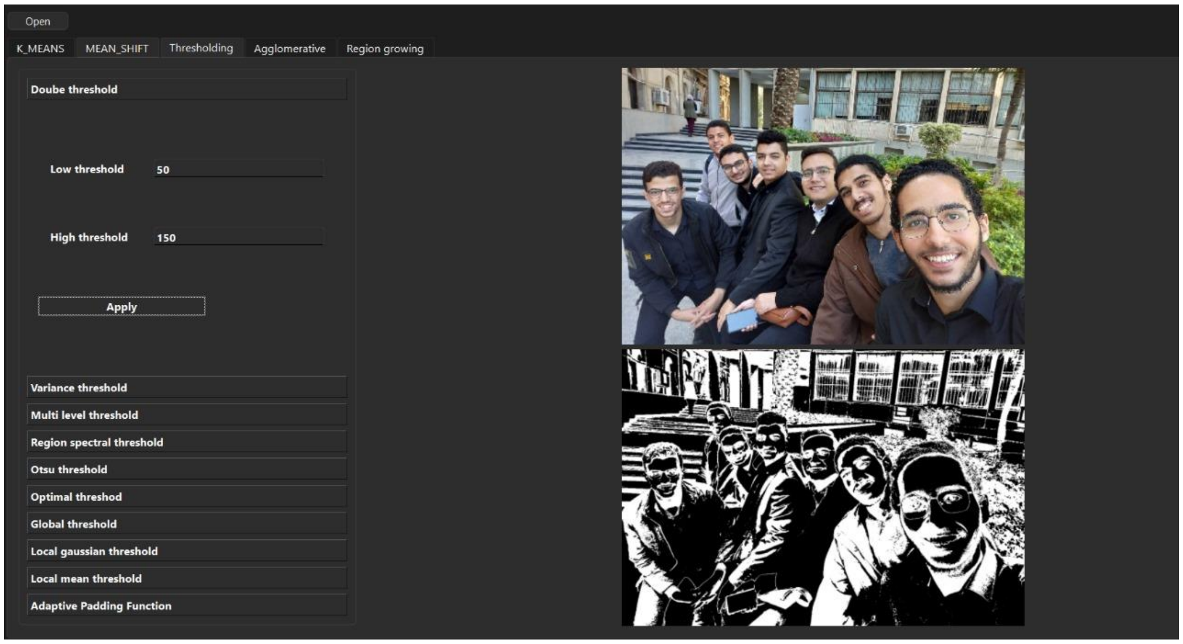




Code Explanation:

doubleThresholding:

- Iterates over each pixel in the image.
- Classifies pixels based on their intensity relative to two specified thresholds.
- Pixels with intensities between the two thresholds are considered medium intensity.
- Returns the thresholded image.



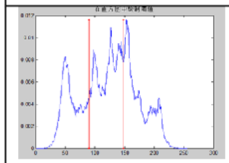
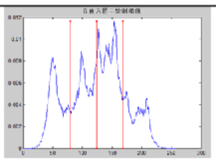
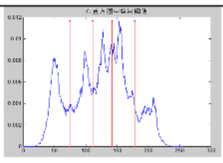
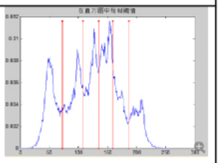
Low threshold 50

High threshold 150

7.4 Multi-level Thresholding:

Explanation:

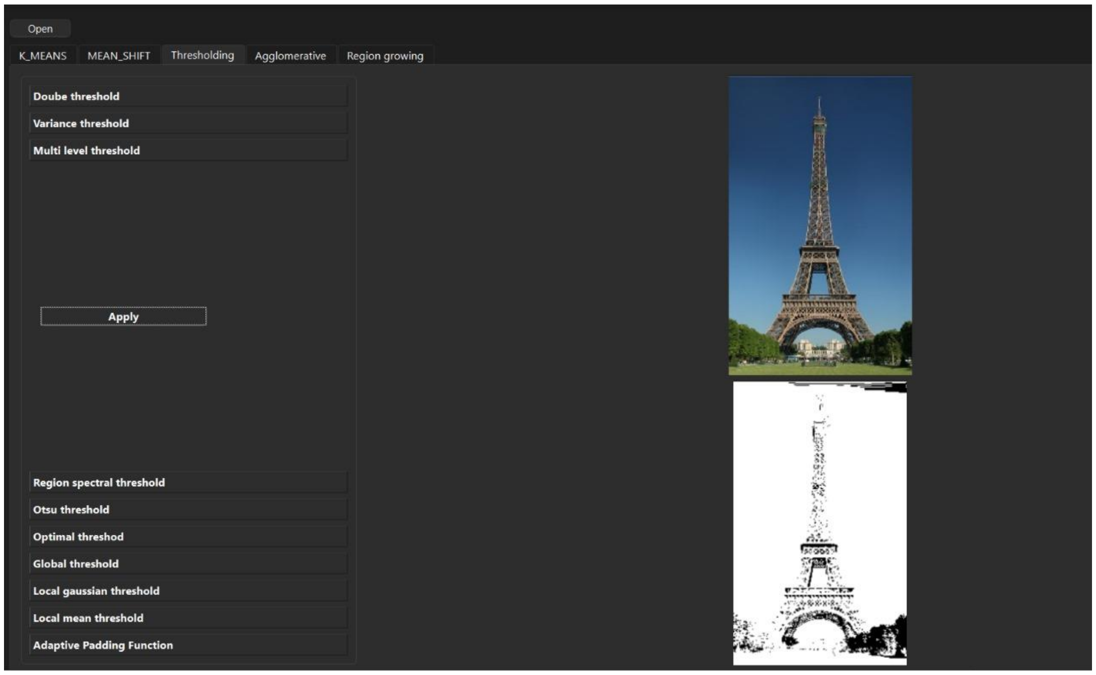
Multi-level thresholding divides the image into multiple segments using predefined threshold values.

th = 2	th = 3	th = 4	th = 5
			

Code Explanation:

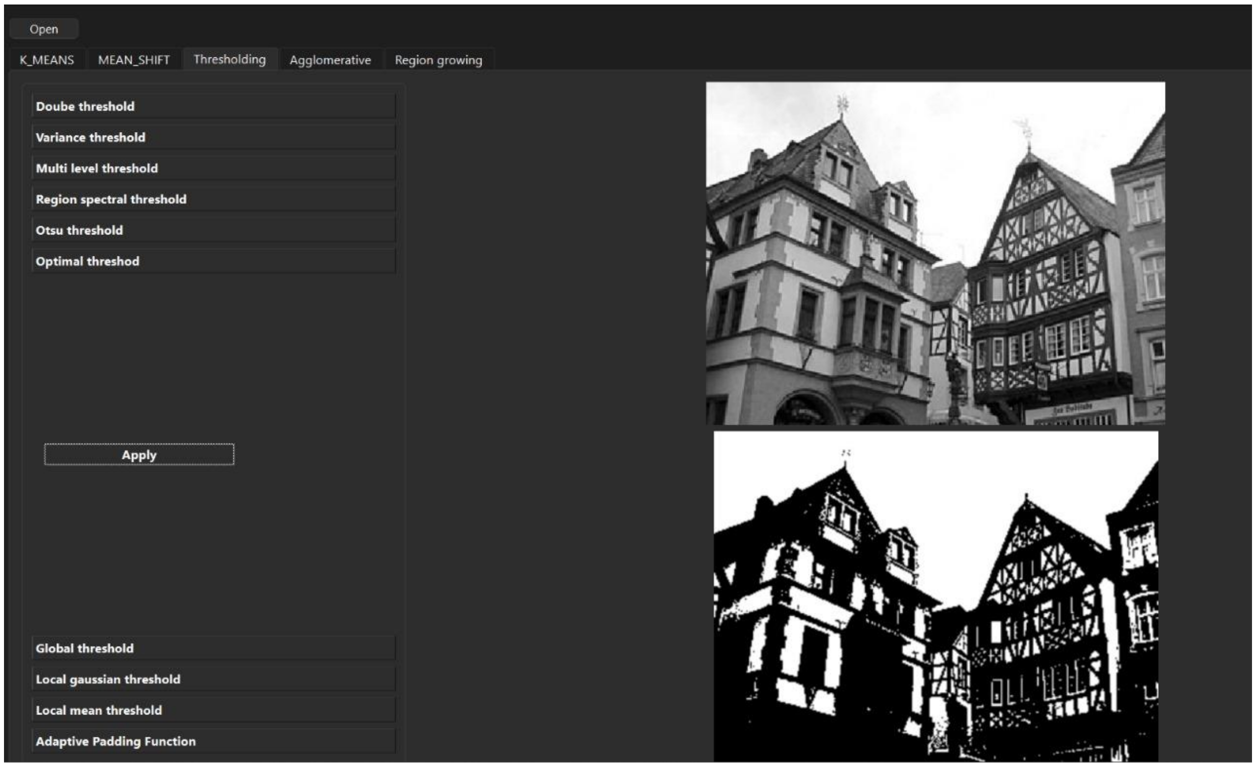
multiLevelThresholding:

- Iterates over each pixel in the image.
- Compares the intensity of each pixel against a set of predefined threshold values.
- Classifies pixels into multiple intensity levels or regions based on the thresholds.
- Returns the thresholded image.



7.5 Optimal Thresholding:

- Initialization: Convert the input image to grayscale and create a matrix (background_vs_object) to represent the classification of pixels as background or object. Initialize the corner pixels of this matrix as background.
- Initial Calculation: Calculate the mean intensity values for background and object regions in the image.
- Threshold Iteration Loop: Iterate through a loop to refine the threshold value. Within each iteration:
- Classify pixels based on whether their intensity is above or below the current threshold.
- Recalculate the mean intensity values for background and object regions.
- Update the threshold based on the new mean intensity values.
- Check for convergence: If the new threshold is close enough to the previous threshold, exit the loop.
- Conversion and Return: Convert the background_vs_object matrix to the appropriate data type (CV_8U) and return it.



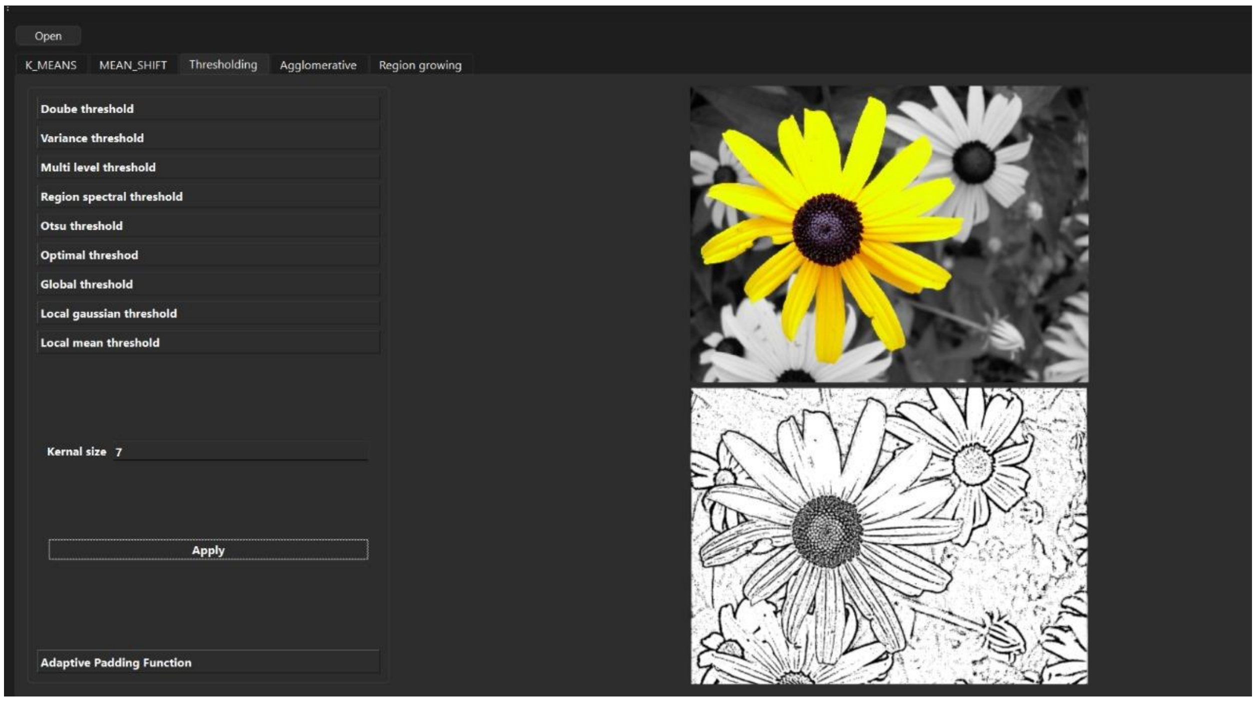
7.6 Local adaptive mean Thresholding:

Adaptive thresholds use a threshold value that varies across the image. This local threshold depends on the local average, computed in a squared portion of the image of kernel size by kernel size pixels, and on the offset relative to that local average.

Pseudocode:

```
1. Calculate the half size of the kernel for edge handling.
2. Iterate over each pixel in the input image, excluding boundary pixels:
   a. Compute the mean intensity value of the neighboring pixels within the kernel.
   b. Compare the intensity value of the current pixel with the mean minus a constant value.
   c. If the intensity value of the current pixel is greater than the mean minus the constant:
      - Set the corresponding pixel value in the result image to the maximum value.
   d. Otherwise:
      - Set the corresponding pixel value in the result image to the minimum value.
```

$$\mu = \frac{\sum_{i,j} I(x+i, y+j)}{(\text{kernelSize} \times \text{kernelSize})}$$



Kernal size 7

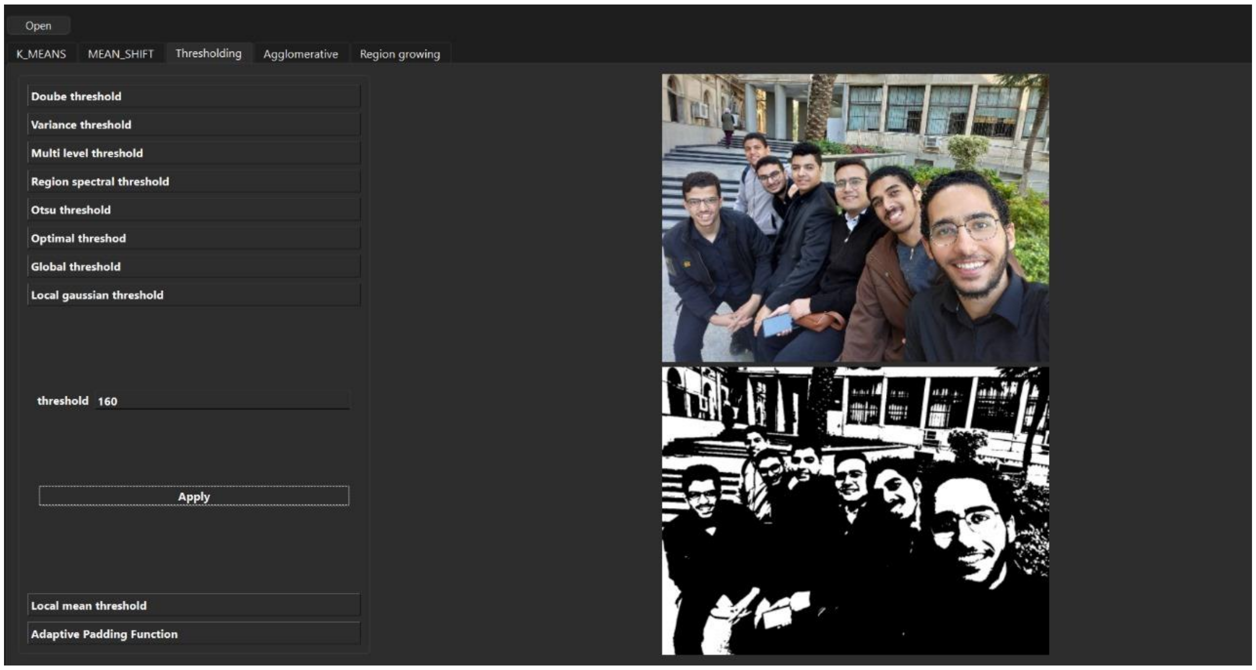
7.7 Local adaptive gaussian Thresholding:

Adaptive thresholds use a threshold value that varies across the image. This local threshold depends on the local average, computed in a squared portion of the image of kernel size by kernel size pixels, and on the offset relative to that local average.

Pseudocode:

```
1. Calculate the half size of the kernel for edge handling.
2. Generate a Gaussian kernel with the specified kernel size and sigma.
3. Iterate over each pixel in the input image, excluding boundary pixels:
    a. Compute the local weighted sum of pixel intensities using the Gaussian kernel.
    b. Compute the local Gaussian weighted mean intensity value.
    c. Determine the threshold using the local mean and the constant.
    d. Apply thresholding to the pixel:
        - If the intensity value is greater than the threshold, set it to 255 (foreground).
        - Otherwise, set it to 0 (background).
```

$$\mu = \frac{\sum_{i,j} I(x+i, y+j) \cdot K(i,j)}{\sum_{i,j} K(i,j)}$$

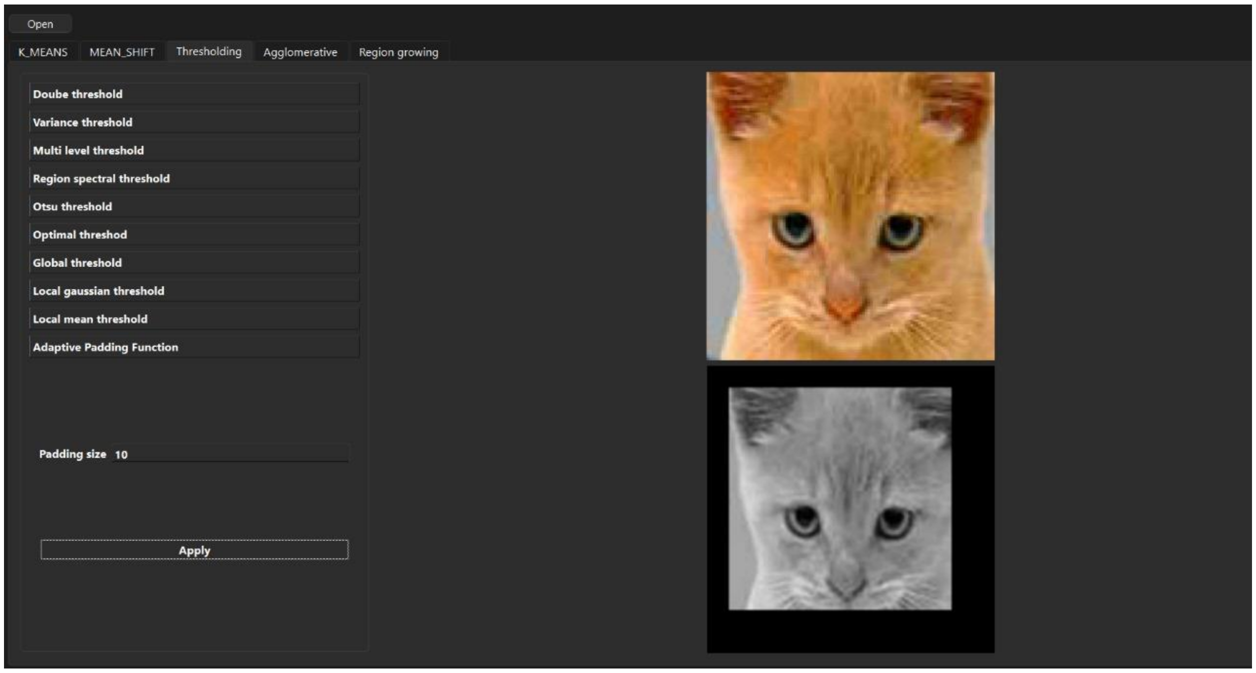


threshold 160

7.8 Adaptive Padding:

Pseudocode:

```
1. Create a new matrix (result) with dimensions enlarged by 2 * padding Size in both width and height, initialized with zeros.
2. Copy the input image onto the central region of the result matrix, starting at coordinates (padding Size, padding Size).
3. Copy the top padding Size rows of the result matrix and paste them below the original image.
4. Copy the leftmost padding Size columns of the result matrix and paste them to the right of the original image.
```



Padding size 10