Faculty of Engineering
Cairo University

Third Year – 2nd Term

Systems & Biomedical Eng. Department

Cairo University

COMPUTER VISION TASK 4: SEGMENTATION WITH THRESHOLDING AND CLUSTERING REPORT

Team 5

Salma Ashraf

Sarah Mohamed

Habiba Salama

Hager Samir

Aya Eyad

May 1, 2024

Setup & Installation

To set up the Filtering and Edge Detection Studio environment, follow these steps:

1. **Install the Requirements**

   Open your terminal or command prompt and enter the following command to install the required packages: `pip install -r requirements.txt`
   This command will install the necessary libraries and dependencies for the studio environment.

2. **Run the Main Script**

   After installing the required packages, run the `main.py` file to start the Filtering and Edge Detection Studio. Use the following command: `python main.py`
   This command will execute the main script and launch the studio application.

The following libraries are included in the `requirements.txt` file and will be installed during the setup process:

- PyQt5
- pyqtgraph
- scipy
- opencv-python
- matplotlib
- numpy

Note: these libraries are essential for the functionalities and features of the Filtering and Edge Detection Studio.

K-Mean Cluster Technique

The “KMeans” class provides a complete set of functions for KMeans clustering, including initialization, training, prediction, and visualization. This comprehensive functionality makes it a versatile tool for clustering tasks, enabling users to efficiently analyze data, identify patterns, and visualize clustering results.

❖ **Class Functions:**

1. getCentroids:

Parameters:

None

Purpose:

Returns the positions of the centroids.

Observation:

This function is useful for obtaining the centroid positions after fitting the KMeans model. It enables users to inspect the final centroids for interpretation or further analysis.

2. getClusters:

Parameters:

None

Purpose:

Returns clusters with shape (1, Npts), where each data point belongs to a specific cluster.

Observation:

This function provides access to the cluster assignments of each data point after fitting the KMeans model. Users can use these cluster labels for various downstream tasks such as classification or anomaly detection.

3. fit:

Parameters:

- max_iter (int): maximum number of iterations for the KMeans algorithm.
- init_state (np.ndarray):optional parameter for specifying initial centroid positions. If None, random positions are generated.

Purpose:

Trains the KMeans model, finds the ultimate positions for centroids, and creates clusters.

Observation:

This function implements the KMeans algorithm by iteratively updating centroid positions to minimize the within-cluster sum of squares. Users can control the maximum number of iterations and optionally provide initial centroid positions for customization.

4. predict:

Parameters:

data (np.ndarray): data points with shape (Npts, Ndim).

Purpose:

Predicts clusters for data points based on the current centroid positions.

Observation: This function assigns each data point to the nearest centroid's cluster, facilitating the clustering of new data points after the model has been trained.

5. visualize:

Parameters:

None

Purpose:

Plots data points and the final position of centroids. This function needs to be called after training.

Observation:

This method offers a visual representation of the clustering results, aiding in the interpretation and assessment of the model's performance.

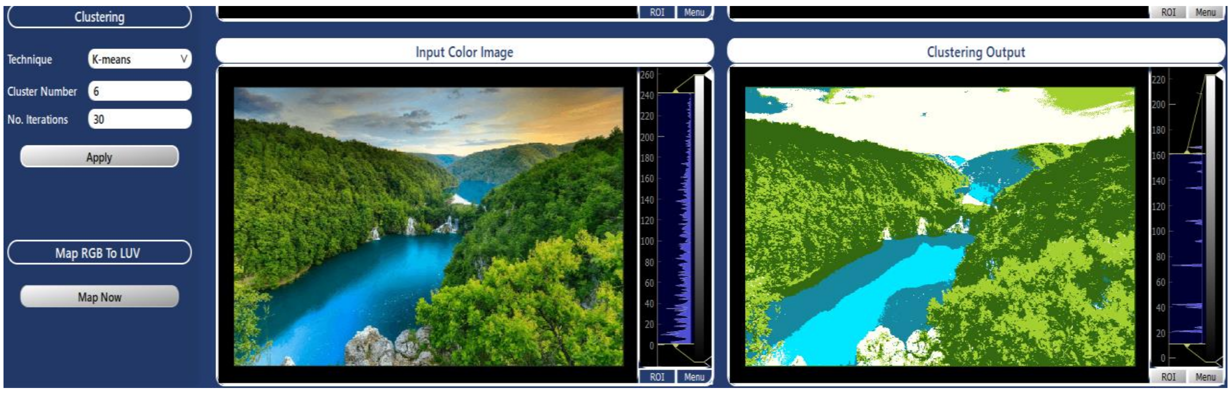
It generates a scatter plot with data points in black and centroids in blue, allowing users to inspect how well the data points are clustered around the centroids.

Experiment:

➢ **Apply K-Mean Clustering with**

cluster_no = 6

max_iteration = 30



Technique: K-means

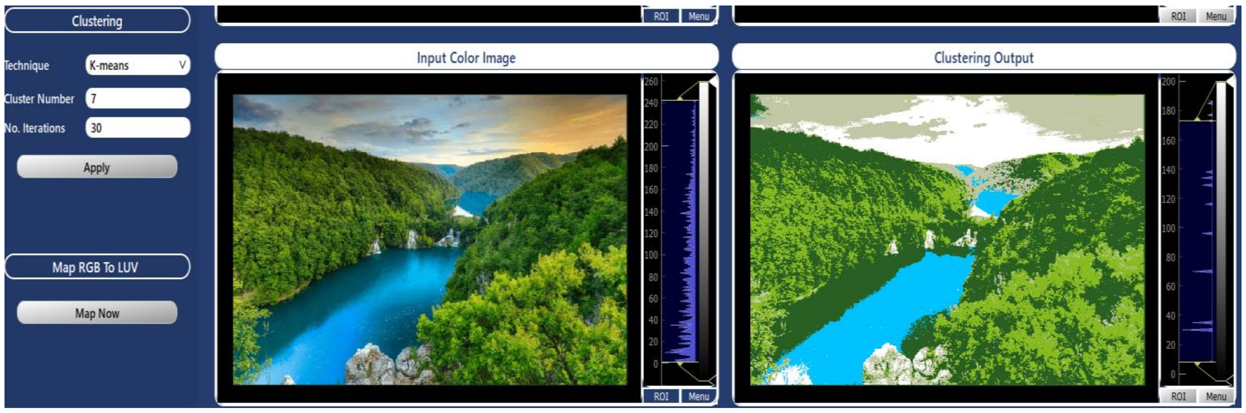
Cluster Number: 6

No. Iterations: 30

➢ **Apply K-Mean Clustering with**

cluster_no = 7 (more no of K more efficient cluster output)

max_iteration = 30



Technique: K-means

Cluster Number: 7

No. Iterations: 30

RGB TO LUV

1. linearize:

Parameters:

x (numpy.ndarray): Input array of RGB values.

Description:

This function transforms the input RGB values from a nonlinear to a linear color space. It operates on each element of the input array x. First, it identifies elements of x that are less than or equal to 0.04045 and divides them by 12.92. For elements greater than 0.04045, it applies a nonlinear transformation defined by the formula :

This transformation enhances the accuracy of color representation by ensuring a linear relationship between the input RGB values and the resulting color intensity. Finally, the function returns the linearized array x.

Observation:

The linearize function plays a crucial role in the color space conversion process, contributing to the accurate representation of colors by linearizing the input RGB values before further processing.

2. rgb_to_luv:

Parameters:

image (numpy.ndarray): Input RGB image.

Description:

This function takes an RGB image as input and performs the conversion to the Luv color space. It first normalizes the RGB channels using the linearize function. The linearize function transforms the RGB values to linear space to ensure accurate color representation. Then, the function converts the RGB values to XYZ space using predetermined conversion factors.
From the XYZ values, it calculates the lightness component (L)according to CIE:
Constants for the u and v components are defined(u′n = 0.2009, v′n = 0.4610).
It computes the u and v components according to (L) value :

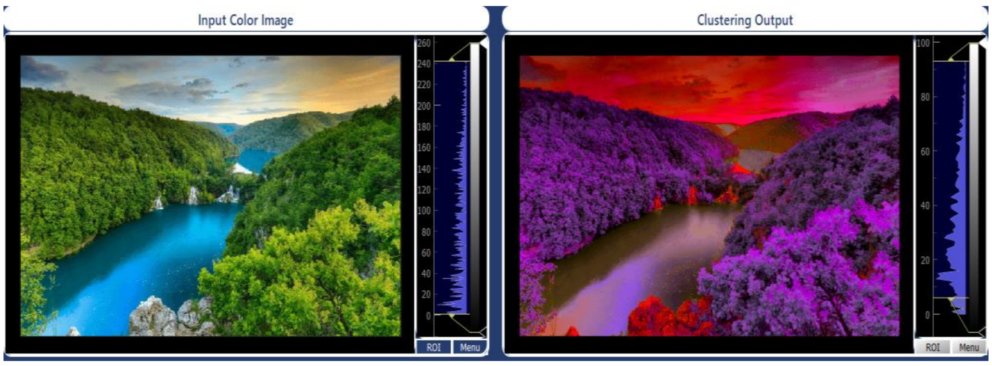
$$u^* = 13L^*(u' - u'_n)$$
$$v^* = 13L^*(v' - v'_n)$$

Finally,it constructs the Luv image by combining the L, u, and v channels.

Observation:

The function handles color space conversion from RGB to Luv, allowing for better representation and analysis of color information in the image data.

Experiment :



Mean shift Cluster technique

The “MeanShift” class represents an implementation of the Mean Shift clustering algorithm, a non-parametric clustering technique used for data clustering and image segmentation. This class provides functionality to cluster input data, such as images, into groups based on their underlying density distributions.

1. apply_mean_shift

Parameters:

- bandwidth (float): Specifies the bandwidth parameter for the Mean Shift algorithm, determining the size of the region to search for candidate points during clustering.
- tolerance (float): Sets the convergence tolerance for the Mean Shift algorithm, determining when the algorithm considers the clustering process to have converged.
- sigma (float): Specifies the standard deviation parameter for the Gaussian kernel used in Mean Shift clustering, influencing the influence of neighboring points on the mean shift computation.

Description:

Initializes a MeanShift object with the provided bandwidth, tolerance, and sigma values.

2. fit:

Parameters:

- image (np.ndarray): The input image to be clustered**.**

Returns:

- list: A list of dictionaries representing the clusters found by Mean Shift. Each dictionary contains keys 'points' and 'center', where:
  - 'points' is a boolean array indicating points belonging to the cluster.
  - 'center' is the centroid of the cluster.

Description: Applies the Mean Shift clustering algorithm to the input image.

3. gaussian_kernel:

Parameters:

- distances (numpy.ndarray): An array of distances.

Returns:

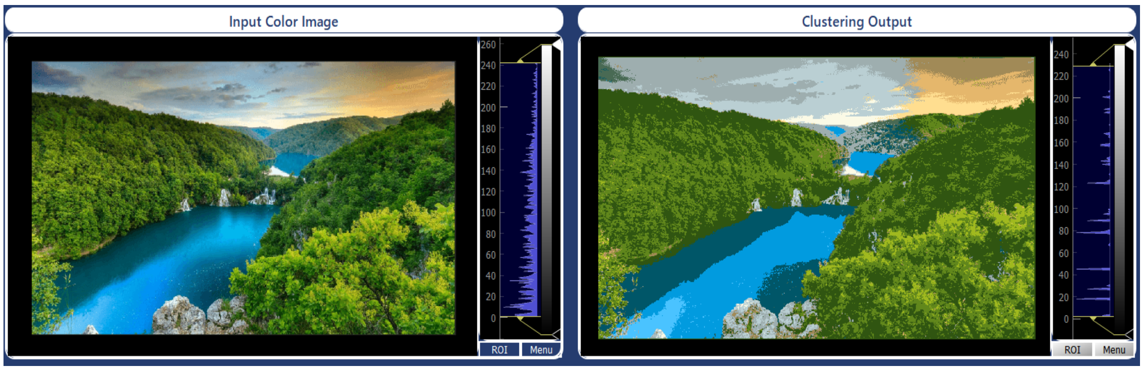
- numpy.ndarray: An array of Gaussian kernel weights.

Observations:

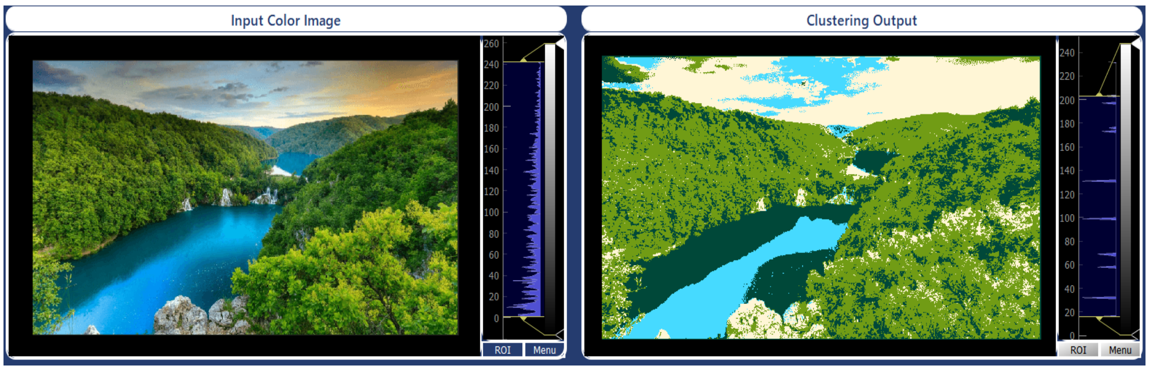
- ➢ The MeanShift class provides an encapsulated implementation of the Mean Shift clustering algorithm, allowing users to perform clustering tasks on input data.
- ➢ By adjusting the parameters such as bandwidth, tolerance, and sigma, users can fine-tune the clustering process to suit different datasets and applications.
- ➢ The fit method is the core functionality of the class, performing the actual clustering operation on input images.
- ➢ The class provides flexibility and control over the clustering process, making it suitable for various clustering tasks in image processing and data analysis.

Experiments:

➢ **Apply Mean Shift Clustering with bandwidth = 100 and threshold= 10 :**



➢ **Apply Mean Shift Clustering with bandwidth = 200 and threshold= 10 :**



Agglomerative Hierarchical Clustering

The "Agglomerative" class provides a group of functions for carrying out the full process f agglomerative clustering as combining clusters and calculating the distance between 2 clusters and a principal function that carries executes the whole algorithm.

❖ **Main Function:**

1. apply_agglomerative:

Parameters:

- image (np.ndarray): a numpy array of 3 channels (BGR) input by the user.
- no_clusters (int): The number of clusters the algorithm will stop at. Must be input by the user. The algorithm starts with a number of clusters equal to the number of points and combines them till it reaches this number.
- image_size (int): The image input by the user will be resized to (image_size x image_size) to allow computational time to decrease. The resized image will replace the original input image in the UI. This value is hardcoded and chosen empirically using trial and error.

Purpose:

This function resizes the input image as explained, creates an instance of the Agglomerative class, and calls the apply_agg function which executes the algorithm and displays the output.

❖ **Class Functions:**

2. calculate_distance_between_clusters:

Parameters:

- c1, c2 (lists carrying the 3 color values for each cluster)

Purpose:

Calculates the Euclidean distance between the centers of 2 clusters.

Observation:

This function is useful for obtaining the similarity between 2 points to judge whether or not they belong to the same cluster.

3. get_new_center_for_clusters:

Parameters:

- c1, c2 (lists carrying the 3 color values for each cluster)

Purpose:

Used to combine 2 clusters by averaging their values to produce the central value for the new cluster.

4. apply_agg:

Parameters:

- input_image (int): maximum number of iterations for the KMeans algorithm.
- number_clusters: The number of clusters the algorithm will stop at. Must be input by the user. The algorithm starts with a number of clusters equal to the number of points and combines them till it reaches this number.
- output_image_gs (np.ndarray): a 2D numpy array. We chose to display the output segmented classes using greyscale values which will be put in this array. It has the same shape as the input image but with one grayscale channel instead of RGB.

- clusters (list): A list which initially carries all pixel values and then their combinations as clusters are formed.
- labels (np.ndarray): a 2D array the same size of the input image but with on channel in the third dimention, carries the index of the cluster corresponding to each pixel.

Purpose:

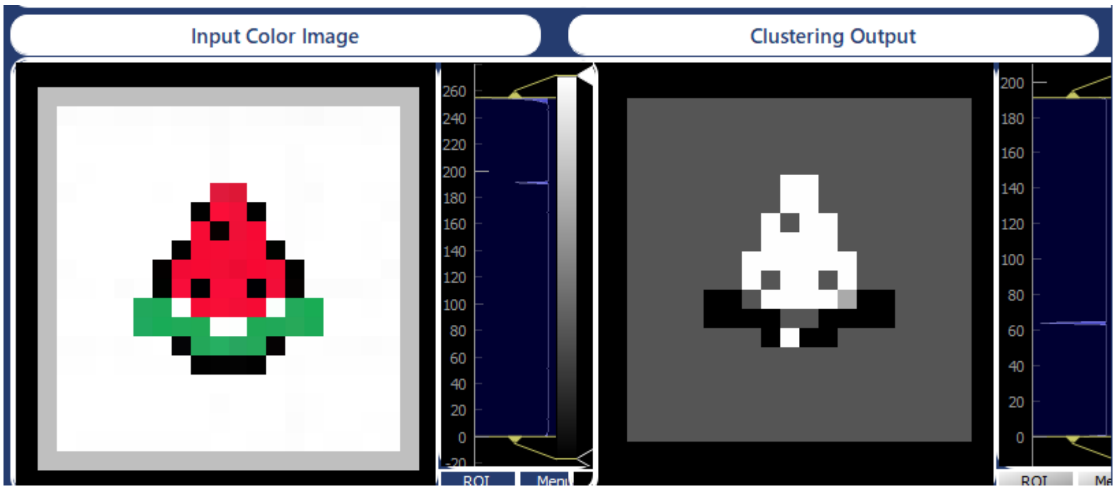
This function executes the agglomerative clustering algorithm.

Steps:

1. Conversion of the input image from BGR to CIELUV space.
2. Initialize the clusters with all pixels as single clusters.
3. Combine clusters till number_clusters is reached inside the clusters list:
    a. Loop on all clusters calculating the distance between each 2 and keeping track of the minimum distance found and the indices of the clusters comprising the minimum distance.
    b. After all combinations have been checked, combine these 2 clusters (using the get_new_center_for_clusters fucntion) and repeat.
4. Forming the labels list which contain the cluster index for each pixel.
5. Generating the colors_gs list according to number_clusters from 0 to 255 (greyscale).
6. Assigning each cluster index in labels to a greyscale value from colors_gs to be put in the output_image.

Experiment:

➢ **Apply Agglomerative Clustering with cluster_no = 4 on a 20x20 image.**



Observation:

This algorithm produced less precise results also due to the fact of it being very computationally exhaustive due to the need for distance calculation between all clusters before any update, making it very hard to run on an image of considerable size. The biggest we could run was on a 25x25 pixel image.

Region Growing Method

❖ **Main Function:**

1. grow_region:

Parameters:

- image (numpy.ndarray): Input RGB image.
- threshold: Similarity threshold for region growing.

Description: Performs region growing on the input image based on a given similarity threshold.

Steps:

1. The function converts the input RGB image to the Luv color space and uses the L channel for region growing.
2. It iteratively grows the region from seed points until no more seed points are available.
3. The region growing process considers 8-neighbours and checks their similarity with the seed point.
4. Once the region growing process is complete, it constructs an output image where the grown region is marked.
5. The function utilizes a stack (seed_points) to keep track of seed points, allowing for iterative processing of the image.

❖ **Helper functions:**

2. get_8neighbours:

Parameters:

- x: x-coordinate of the point.
- y: y-coordinate of the point.
- max_x: Maximum value of x-coordinate (image width).
- max_y: Maximum value of y-coordinate (image height).

Description: This function calculates the 8-neighbour coordinates of a given point within an image.

Purpose:

➢ It provides a straightforward approach to obtain the coordinates of the 8-neighbours of a given pixel.

➢ The function ensures that the calculated neighbours are within the boundaries of the image.

3. get_similarity

Parameters:

- image: Input image (numpy.ndarray).
- x_seed: x-coordinate of the seed point.
- y_seed: y-coordinate of the seed point.
- x_neighbour: x-coordinate of the neighbour.
- y_neighbour: y-coordinate of the neighbour.

Description: Calculates the similarity between a seed point and its neighbour using intensity difference.

Purpose:

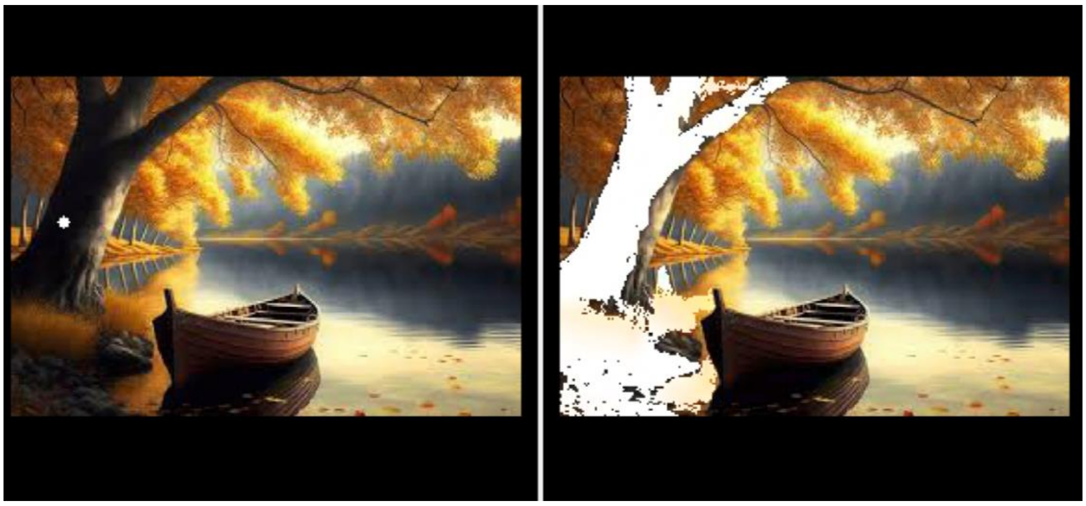
This function computes the absolute intensity difference between the seed point and its neighbor.

Experiments:

- **Region growing applied with**
  similarity threshold = 5 at (a)
  similarity threshold = 3 at (b)
  one initial seed point

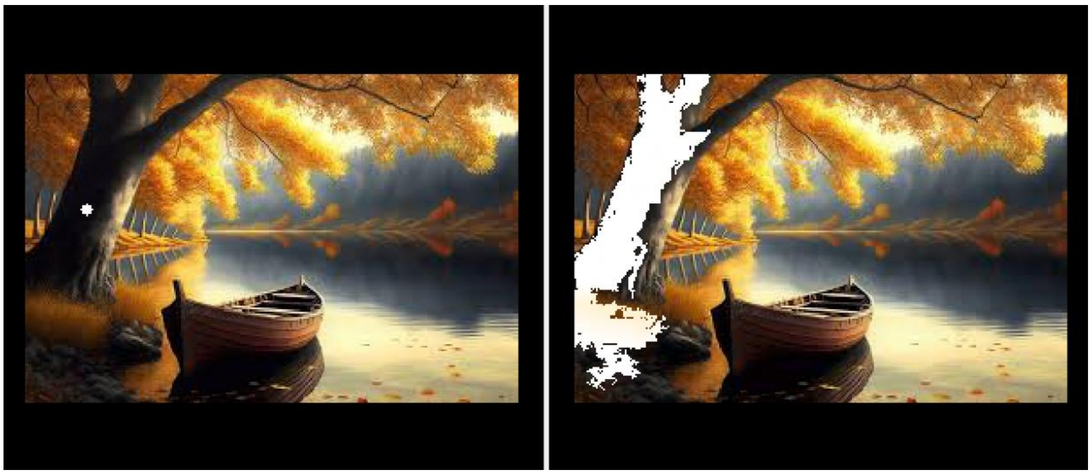
Input Image *Output Image*

(a)



Input Image *Output Image*

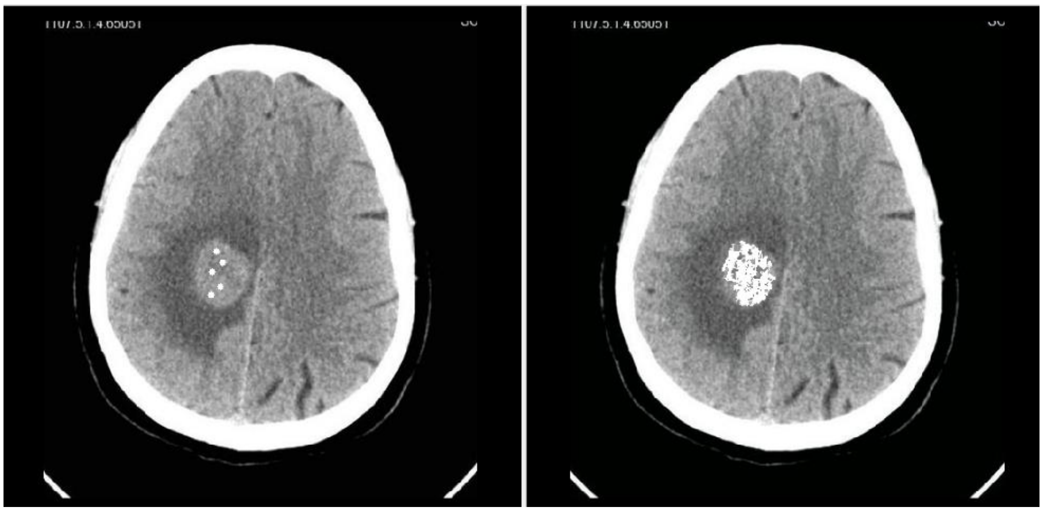
(b)



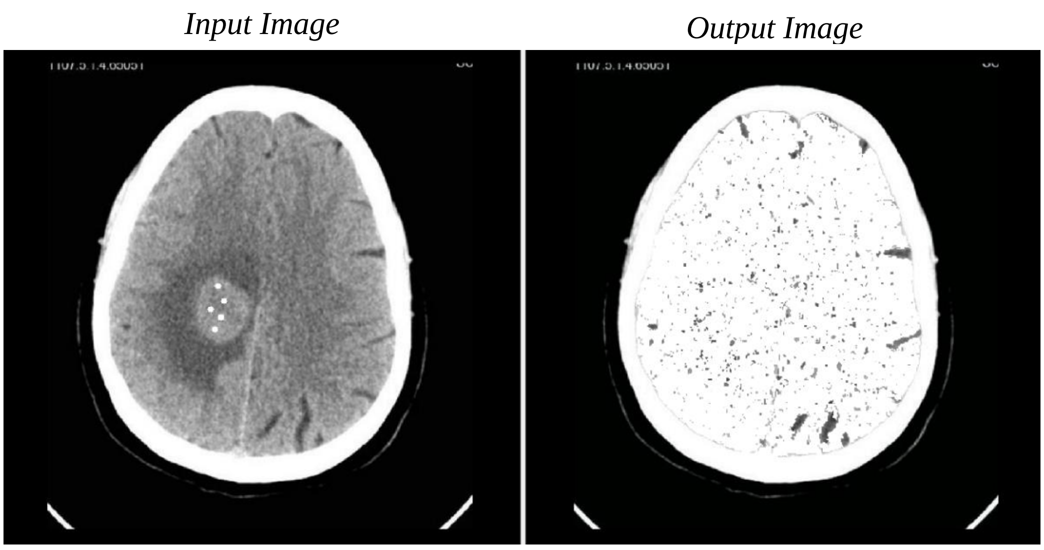
- **Brain tumor segmentation using region growing method with**
  similarity threshold = 4 at (a)
  similarity threshold = 6 at (b)
  5 initial seed points

Input Image *Output Image*

(a)



(b)



Observations:

- The similarity calculation relies solely on intensity, which might be limited for certain types of images or applications.
- A higher threshold allows more neighboring pixels to be included, resulting in a larger region, the output region includes a broader range of intensities, leading to a larger segmented region with less distinct boundaries leading to over-segmentation.
- Conversely, too low threshold restricts the inclusion of neighboring pixels, leading to a smaller region, slower expansion of the region and result in fragmented or incomplete segmentation.
- Portions of the image that are relatively similar but slightly different in intensity may not be included in the segmented region (under-segmentation).

Optimal Thresholding

Optimal thresholding is a technique in image processing that separates an image into foreground and background using a calculated threshold value. It's effective for images with bi-modal histograms but may struggle with non-uniform illumination or noise. It's widely used in applications like medical imaging and machine vision.

Global

optimal_threshold:

Parameters:

- img (np.ndarray): input image as a numpy array.

Description:

This function applies an iterative thresholding method to the image to separate the background and objects. It uses the mean pixel values of the background and objects to calculate an optimal threshold value.

Steps:

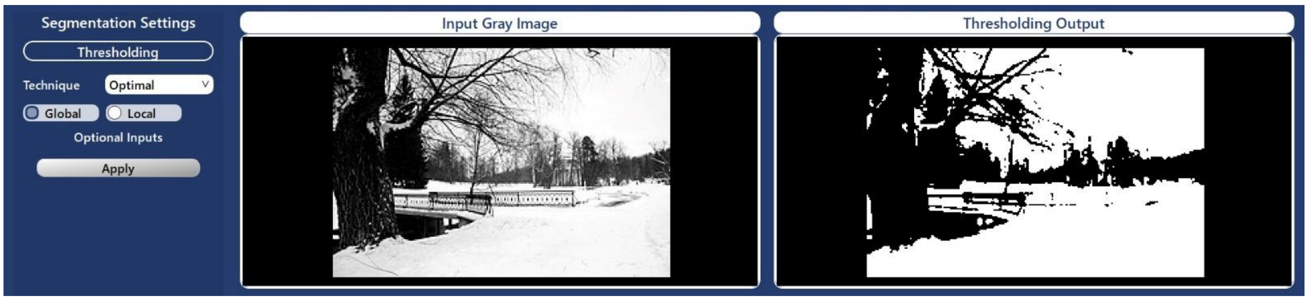
1. Enhances the image by adjusting the contrast and brightness.
2. Applies a Gaussian blur to the image to reduce noise.
3. Initializes the background as the pixels at the four corners of the image and the objects as the rest of the pixels.
4. Calculates the initial threshold as the average of the mean pixel values of the background and objects.
5. Enters a loop where it updates the pixels of background and objects based on the current threshold and recalculates the threshold as the average of the new mean pixel values. This loop continues until the threshold value stabilizes.

$$\mu_{b_{current}} = \frac{\sum_{(i,j)\in background} img(i,j)}{\#background\_pixles} \ , \ \mu_{o_{current}} = \frac{\sum_{(i,j)\in objects} img(i,j)}{\#objects\_pixles}$$

$$T_{new} = \frac{\mu_{b_{current}} + \mu_{o_{current}}}{2}$$

6. Finally, it applies the threshold to the image, setting pixels below the threshold to 0 (background) and pixels above the threshold to 255 (objects). Assuming that the background is darker than the objects.

Experiment:



Technique: Optimal

[✓ Global] Local

Optional Inputs

Local

local_optimal_threshold:

Parameters:

- image (np.ndarray): The source grayscale image as a NumPy array.
- num_x_sections, num_y_sections (int): The number of regions to divide the image into along the x and y axes.

Description:

This function applies a local optimal thresholding method to the image. It divides the image into several sections and applies the optimal_threshold function to each section independently. This allows it to handle variations in lighting conditions across different parts of the image.

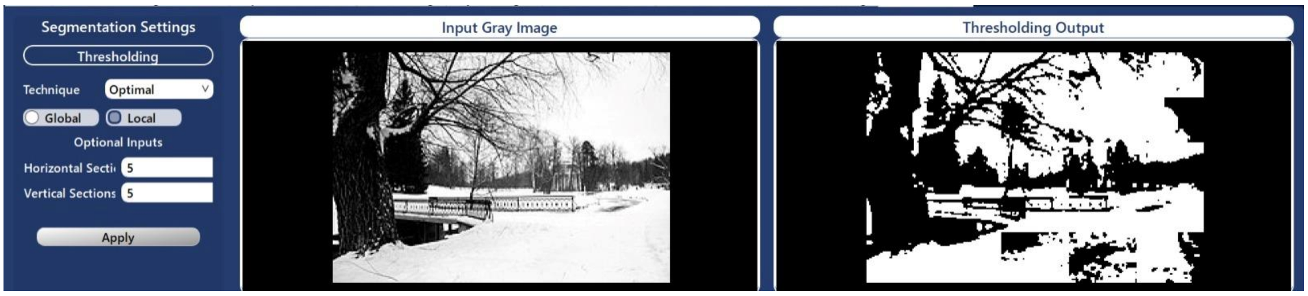
Steps:

1. Convert the image to grayscale if it is not already.
2. Initialize an output image of the same size as the input image.
3. Divide the image into sections based on the specified number of x and y sections.
4. Calls the optimal_threshold function to each section of the image.
5. Return the thresholded image. Each pixel in the image is classified as either background or object based on the local optimal threshold for its section. This allows the function to handle variations in lighting conditions across different parts of the image.

Observations:

- ➢ Useful for images with non-uniform illumination, where a global threshold might not be effective. By applying the threshold locally, it can adapt to different lighting conditions in different parts of the image.
- ➢ The choice of the number of sections to divide the image into is a critical parameter that can significantly affect the results. Too few sections might not capture the variations in lighting conditions, while too many sections might over-segment the image.

Experiment:


Segmentation Settings
Thresholding
Technique: Optimal
Global [✓ Local]
Optional Inputs
Horizontal Secti: 5
Vertical Sections: 5
Apply

Otsu Thresholding

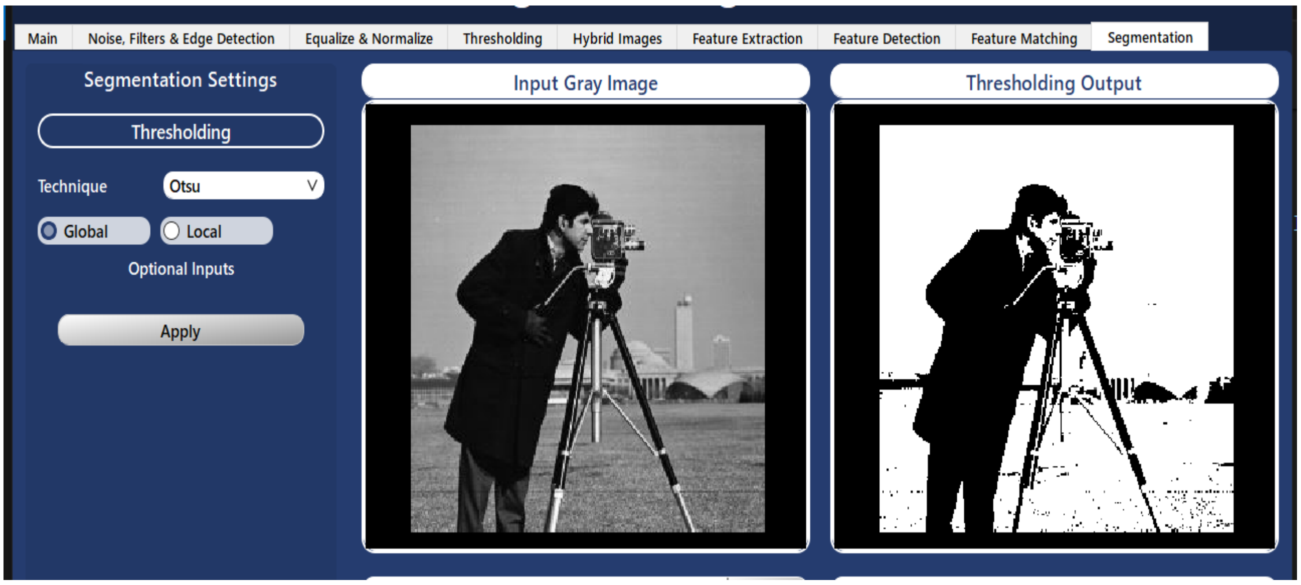
Otsu's thresholding is an image processing method used for automatic binarization or segmentation of an image. It determines an optimal threshold by minimizing intra-class intensity variance, or equivalently, maximizing inter-class variance. This method works well for bimodal histograms but may struggle with images having overlapping intensities or non-uniform illumination.

Global

Steps:

1. **Compute Histogram**: Calculate the histogram of the grayscale image. This histogram represents the distribution of pixel intensities in the image.
2. **Compute Cumulative Distribution Function (CDF):** Compute the cumulative distribution function (CDF) of the histogram.
3. **Initialize Variables**: Initialize variables to store the maximum between-class variance and the optimal threshold value.
4. **Iterate Over Thresholds**: Iterate over all possible threshold values (intensity levels) from 0 to 255.
5. **Compute Class Probabilities and Means**: For each threshold value, calculate the probabilities and means of the two classes: pixels with intensities below the threshold and pixels with intensities above the threshold.
6. **Calculate Between-Class Variance:** Using the probabilities and means calculated in the previous step, compute the between-class variance. This variance represents the separability between the two classes.
7. **Update Maximum Variance and Threshold**: Compare the computed between-class variance with the maximum variance obtained so far. If the current variance is greater, update the maximum variance and store the corresponding threshold value.
8. **Apply Thresholding:** Once the optimal threshold value is determined, apply binary thresholding to the grayscale image using this threshold. Pixels with intensities below the threshold are assigned one value (0 or 255), while pixels with intensities above the threshold are assigned another value.
9. **Output Result**: Return the thresholded image along with the optimal threshold value.

Experiment:



Technique: Otsu

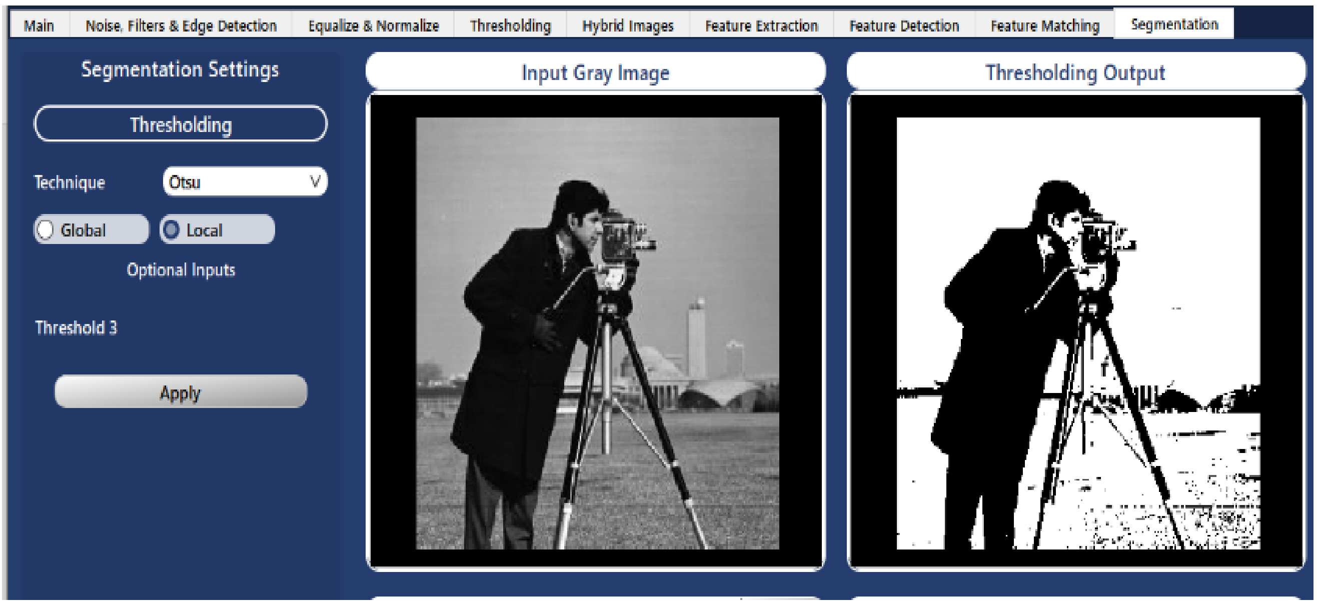
[✓ Global] Local

Local

Steps:

1. **Divide the Image into Sections**: Split the input image into smaller sections.
   In our current implementation, we have divided the image into 4 sections. However, this can be adjusted based on the specific requirements of your image processing task. Increasing the number of sections can capture more local variations in the image, while decreasing it can give a more global view. The optimal number of sections may vary depending on the image and the application.
2. **Iterate Over Sections**: For each section in the image:
   - Calculate the histogram of the current section.
   - Compute the cumulative distribution function (CDF) of the histogram.
   - Initialize variables to store the maximum between-class variance and the optimal threshold value for the current section.
3. **Iterate Over Thresholds**: Iterate over all possible threshold values (intensity levels) from 0 to 255 for each section.
4. **Compute Class Probabilities and Means**: For each threshold value, calculate the probabilities and means of the two classes: pixels with intensities below the threshold and pixels with intensities above the threshold, within the current section.
5. **Calculate Between-Class Variance**: Using the probabilities and means calculated in the previous step, compute the between-class variance for the current section.
6. **Update Maximum Variance and Threshold**: Compare the computed between-class variance with the maximum variance obtained so far for the current section. If the current variance is greater, update the maximum variance and store the corresponding threshold value.
7. **Apply Thresholding:** Once the optimal threshold value is determined for each section, apply binary thresholding to the corresponding section of the grayscale image using this threshold. Pixels with intensities below the threshold are assigned one value (0 or 255), while pixels with intensities above the threshold are assigned another value.
8. **Reconstruct the Image:** After thresholding each section individually, reconstruct the final thresholded image by combining the thresholded sections.
9. **Output Result:** Return the final thresholded image along with the optimal threshold values for each section.

Experiment:



Technique: Otsu

Global [✓ Local]

Optional Inputs

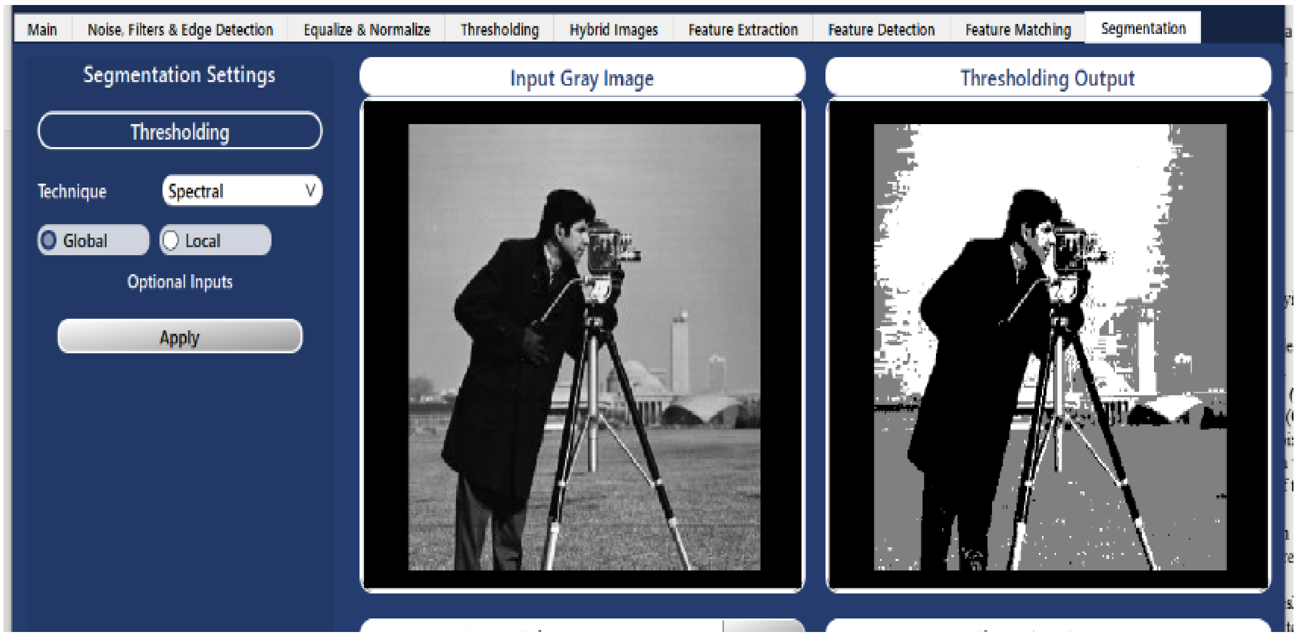
Threshold 3

Spectral Thresholding

Global

Steps:

1. **Image Copy:** Make a copy of the input image to avoid modifying the original image.
2. **Compute Histogram and Cumulative Distribution Function (CDF):** Calculate the histogram and cumulative distribution function (CDF) of the grayscale image. The histogram represents the frequency of pixel intensity values, while the CDF gives the cumulative sum of histogram values.
3. **Compute Image Mean:** Calculate the mean intensity value of the image, which is used in subsequent calculations.
4. **Initialize Variables**: Initialize variables to store the maximum between-class variance (max_value), as well as the optimal low and high thresholds (optimal_low and optimal_high, respectively).
5. **Loop Over Possible Thresholds:** Iterate over all possible threshold values within a certain range (from 1 to 254 for k1, and from k1 + 1 to 255 for k2).
6. **Compute Weights and Means of Classes**: For each pair of thresholds (k1 and k2), calculate the weights and means of the three classes: pixels with intensity values below k1, between k1 and k2, and above k2.
7. **Calculate Variance**: Using the weights and means calculated in the previous step, compute the variance using the formula for between-class variance.
8. **Update Optimal Thresholds:** If the calculated variance (sigma2) is greater than the current maximum variance (max_value), update the maximum variance and store the corresponding optimal low and high thresholds.
9. **Threshold the Image: U**sing the optimal low and high thresholds, threshold the input image. Pixels with intensity values below optimal_low are set to 0, those between optimal_low and optimal_high are set to 128, and those above optimal_high are set to 255.
10. **Return Results**: Return the optimal low and high thresholds along with the thresholded image.



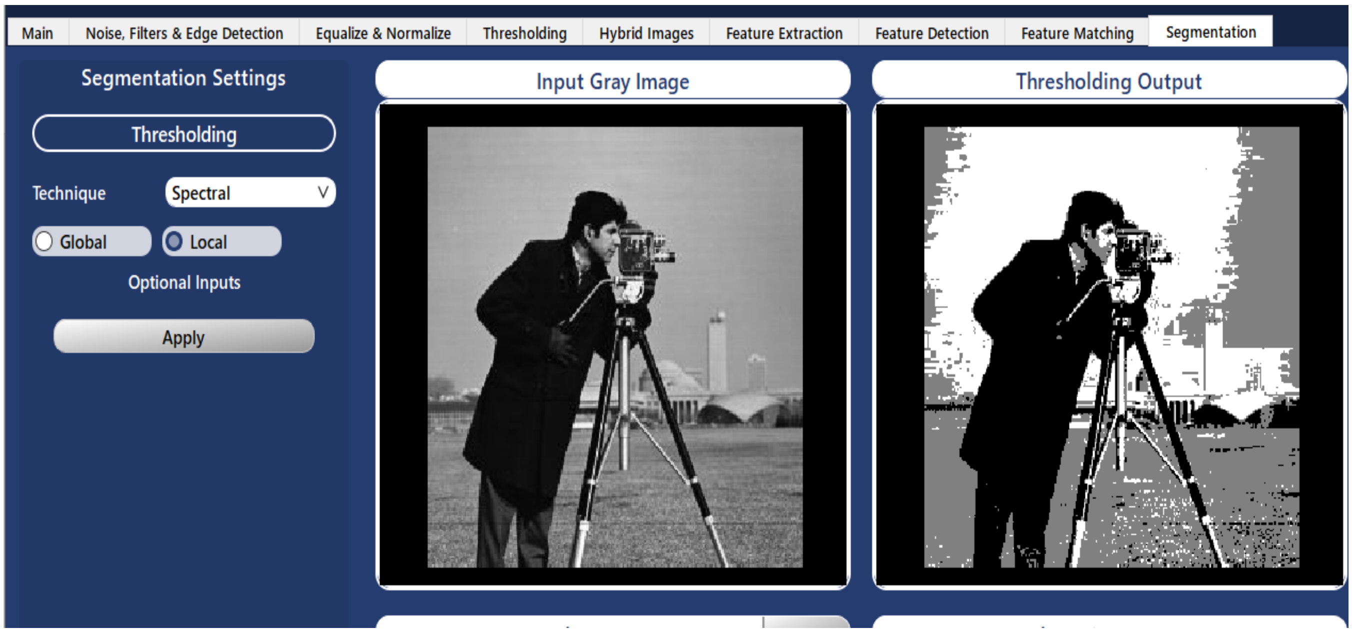
Spectral Thresholding

Local

The same algorithm local thresholding splits the image into sections and calls the global spectral thresholding method for each section then concatenates them.

In our current implementation, we have divided the image into 4 sections. However, this can be adjusted based on the specific requirements of your image processing task. Increasing the number of sections can capture more local variations in the image, while decreasing it can give a more global view. The optimal number of sections may vary depending on the image and the application.

Experiments:



Technique: Spectral

Global [✓ Local]