Report of Task 4

Introduction

In this report, we explore various image processing techniques applied to both grayscale and color images. We aim to understand and implement different thresholding and segmentation methods and observe their effects on image data.

We apply several thresholding techniques for grayscale images, including optimal thresholding, Otsu's method, spectral thresholding (for more than two modes), and local thresholding. Each of these methods has its strengths and weaknesses, and their effectiveness can vary depending on the specific characteristics of the image.

For color images, we delve into unsupervised segmentation techniques. We experiment with k-means clustering, region growing, agglomerative methods, and the mean shift algorithm. These methods allow us to partition the image into meaningful segments without prior knowledge about the number or characteristics of these segments.

Finally, we also explore the conversion of RGB images to the LUV color space. This conversion can be beneficial for certain image processing tasks, as the LUV color space more closely aligns with human color perception compared to the RGB color space.

I. Optimal Thresholding

Global

Optimal global thresholding is a vital technique in image processing used to automatically find the ideal threshold value for segmenting objects from the background in grayscale images. By iteratively analyzing pixel intensity distributions, this method seeks to identify a threshold that maximizes the distinction between foreground and background regions. Through this iterative process, optimal global thresholding efficiently produces binary images where objects of interest are accurately isolated.

Steps for Global Optimal Thresholding:

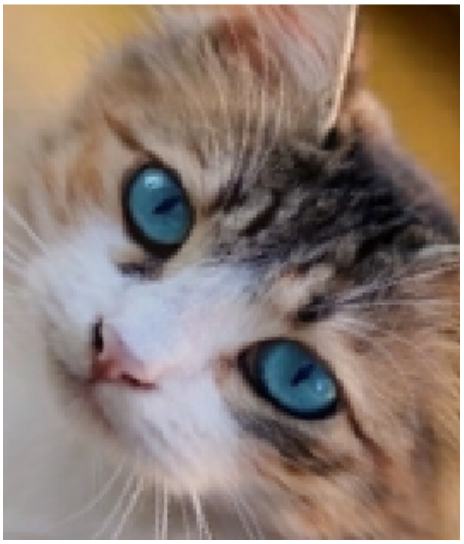
1) Convert the image to grayscale: Transform the input image to grayscale using OpenCV's cv2.cvtColor() function.

2) Compute initial background and foreground information: Calculate the initial sums of pixel values for the background and foreground regions. This involves considering pixel values at the corners of the image.

3) Calculate initial background and foreground means: Compute the mean intensity values for the background and foreground regions using the initial sums obtained in the previous step.

4) Iteratively update the threshold value: Begin an iterative process to update the threshold value. Inside the loop, calculate the mean intensity values for the background and foreground regions using the current threshold value. Update the threshold value as the average of the updated background and foreground means. Repeat this process until convergence, i.e., until the threshold value remains constant between iterations.

5) Generate the binary image: Apply the final threshold value to the grayscale image to create a binary image. Pixels with intensity values less than the threshold become 0 (background), while pixels with intensity values greater than or equal to the threshold become 255 (foreground).



<u>Local</u>

Local optimal thresholding is a technique used in image processing to segment images into foreground and background regions by adaptively determining threshold values for smaller blocks of the image. Unlike global thresholding methods, local optimal

thresholding considers the local characteristics of image regions, enabling finer segmentation of objects with varying intensity levels. This approach is particularly useful for images with non-uniform illumination or varying object sizes.

Steps for Local Optimal Thresholding:

1) Input Preparation: Convert the input image to grayscale, ensuring uniformity in processing.

2) Block Division: Divide the grayscale image into smaller blocks of a specified size (block_size), allowing for localized thresholding analysis.

3) Iterative Thresholding per Block:
For each block:

a) Calculate initial background and foreground means based on pixel intensities within the block.

b) Compute the initial threshold value as the average of background and foreground means.

c) Iterate to refine the threshold value:
d) Update the threshold iteratively based on the mean intensities of foreground and background pixels until convergence.

4) Threshold Application and Output Generation:
Apply the final threshold value to the respective block, creating a binary representation where pixel values above the threshold are set to 255 (foreground) and below to 0 (background).
Combine the binary representations of all blocks to generate the final segmented image.

Local Optimal Thresholding with Different Block Sizes:

To analyze the performance of local optimal thresholding with varying block sizes (20, 50, and 80), three images were processed using the algorithm. Each image underwent segmentation using block sizes corresponding to the aforementioned values. Below are the output images along with logical justifications for their segmentation results:

1) Block Size of 20:

The use of smaller block sizes allows for finer local analysis of pixel intensity variations within the image.

In regions with intricate details or sharp-intensity transitions, smaller blocks can capture nuances more effectively.

Consequently, objects with subtle intensity differences from the background are accurately segmented, resulting in sharper boundaries and enhanced object delineation.

2) Block Size of 50:

With a moderate block size, a balance between local analysis and computational efficiency is achieved.

The segmentation results tend to be smoother compared to smaller block sizes due to averaging effects over larger regions.

Objects with relatively uniform intensity distributions are well-segmented, while finer details may exhibit slight blurring or smoothing.

3) Block Size of 80:

Larger block sizes lead to more global analysis of pixel intensities, encompassing broader regions within the image.

While computational efficiency improves, finer details may be overlooked or merged with neighboring regions.

Segmentation results tend to exhibit smoother boundaries and may oversimplify object shapes, especially for objects with intricate textures or sharp-intensity transitions.

II. Otsu Thresholding

Global:

Global Otsu thresholding is a widely used method for automatically determining an optimal threshold value for image segmentation. This technique leverages the histogram of pixel intensities in a grayscale image to find a threshold that maximizes the separation between foreground and background classes. By minimizing the

intra-class variance, Otsu's method efficiently identifies the threshold that best discriminates between object and background regions.

Steps for Global Otsu Thresholding:

1) Conversion to Grayscale: Convert the input image to grayscale, ensuring uniform processing and simplifying intensity-based analysis.

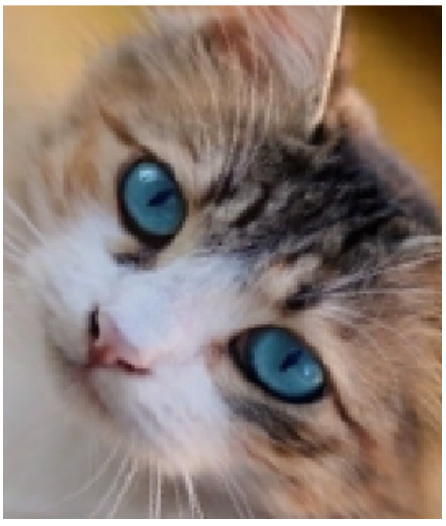
2) Histogram Calculation: Compute the histogram of pixel intensities in the grayscale image. The histogram represents the frequency distribution of pixel values across the entire image range.

3) Threshold Determination:

a) Iterate over all possible intensity levels (0 to 255).
b) For each intensity level, calculate the between-class variance, which quantifies the separation between foreground and background intensities if this intensity is chosen as the threshold.
c) Identify the intensity level that maximizes the between-class variance, representing the optimal threshold value.

4) Binarization:

a) Apply the optimal threshold to the grayscale image to create a binary representation.
b) Pixels with intensities above the threshold are classified as foreground (set to 256), while those below are considered background (set to 0).



Local:
Local Otsu

thresholding extends the concept of Otsu's
method to a localized context, enabling adaptive thresholding for images with spatially varying characteristics. Unlike global thresholding techniques, which determine a single threshold for the entire image, local Otsu thresholding divides the image into smaller blocks and computes optimal thresholds for each block independently. This approach allows for finer

adjustment of thresholds, resulting in more accurate segmentation of objects with varying intensity distributions and backgrounds.

Steps for Local Otsu Thresholding:

1) Image Preprocessing:
Convert the input image to grayscale to simplify intensity-based analysis.

2) Block Division:
Divide the grayscale image into square blocks of a specified size (block_size), ensuring that each block captures local image characteristics.

3) Local Otsu Threshold Computation:
For each block:

a) Compute the histogram of pixel intensities within the block.
b) Iterate over all possible intensity levels to calculate the between-class variance for each level.
c) Identify the intensity level that maximizes the between-class variance, representing the optimal threshold for the block.

4) Threshold Application:
Apply the optimal threshold obtained for each block to binarize the corresponding block, creating a binary representation.

5) Output Generation:
Combine the binary representations of all blocks to generate the final segmented image.

Local Otsu Thresholding with Different Block Sizes:

To assess the performance of local Otsu thresholding with varying block sizes (50 and 80), two images were subjected to segmentation using the algorithm. Each image underwent adaptive thresholding using block sizes corresponding to the aforementioned values. Below are the output images along with logical justifications for their segmentation results:

Block Size of 50:

a) The use of a moderately sized block facilitates localized analysis of pixel intensity variations within the image.
b) With smaller block sizes, finer details are captured, allowing for precise threshold adjustments in regions with subtle intensity changes.

c) Objects with varying textures and backgrounds are accurately segmented, preserving intricate details and boundaries.

Block Size of 80:

a) Larger block sizes offer a broader perspective of local image characteristics, encompassing larger regions for threshold computation.
b) While computational efficiency improves with larger blocks, finer details may be overlooked or smoothed out due to averaging effects.
c) Segmentation results may exhibit smoother boundaries and simplified object shapes, especially in regions with uniform textures or gradual intensity transitions.

III. Spectral Thresholding

In spectral thresholding (multilevel Otsu thresholding), instead of just separating the image into foreground and background using a single threshold value, the algorithm seeks to segment the image into multiple levels or regions based on the intensity distribution of the pixels. This allows for more nuanced segmentation, especially in images with varying lighting conditions.

Algorithm steps:

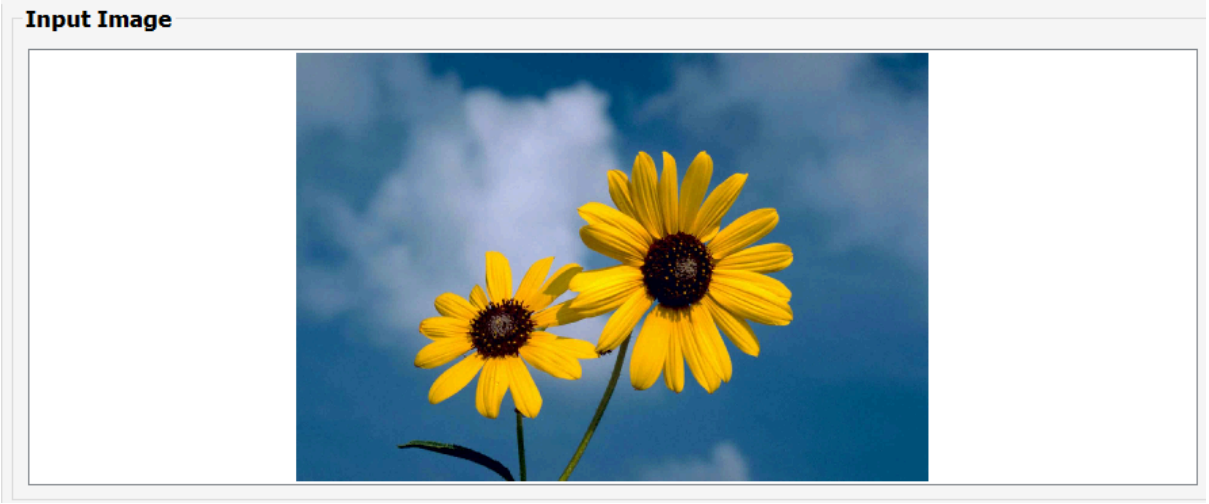
1. **Histogram Computation**: It computes the histogram of the input grayscale image. The histogram represents the distribution of **pixel intensities** in the image.
2. **Histogram Normalization**: It normalizes the histogram by dividing each bin count by the total number of pixels in the image. This step ensures that the sum of all bin counts **equals 1**, making it a **probability distribution.**

3. **Otsu's Thresholding**: It calculates the **optimal threshold** to separate the image into foreground and background classes. Otsu's method iterates through all possible threshold values and selects the one that maximizes the between-class variance.

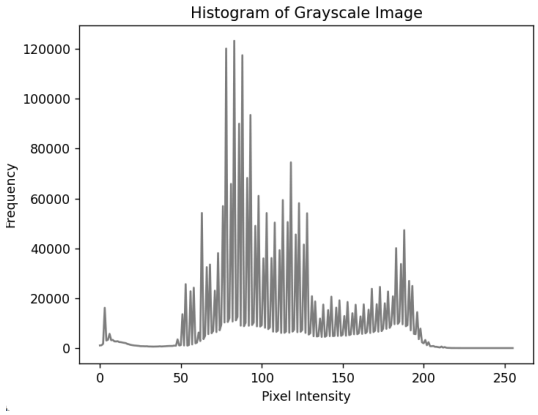
4. **Multiple Otsu Thresholds**: It extends Otsu's method to find multiple thresholds for segmenting the image into more than two classes. It repeatedly applies Otsu's method, each time considering the histogram of the remaining part of the image after the previous threshold.

5. Apply Thresholding: It applies the computed thresholds to the input image, segmenting it into different classes based on pixel intensity levels.

Trial:



For an image that has a histogram like this



We can see that there are three peaks between (50,100), (100,150), and (150, 200), so what will happen in the following cases?

1. Apply Spectral Thresholding with 2 classes (Otsu thresholding):

2. Apply Spectral Thresholding with 3 classes:

3. Increase the number of classes more and more:

Conclusion:
With each increment in the number of classes, the algorithm will compute an additional threshold value. These thresholds aim to segment the image into more distinct intensity levels or regions.

And because the image has almost three peaks, if the chosen number of classes equals 3 the algorithm will allow for finer segmentation.

And if we increase it more and more may lead to wasted computational resources and potentially noisy segmentation results, It's generally advisable to choose the number of classes based on the information present in the histogram of the image.

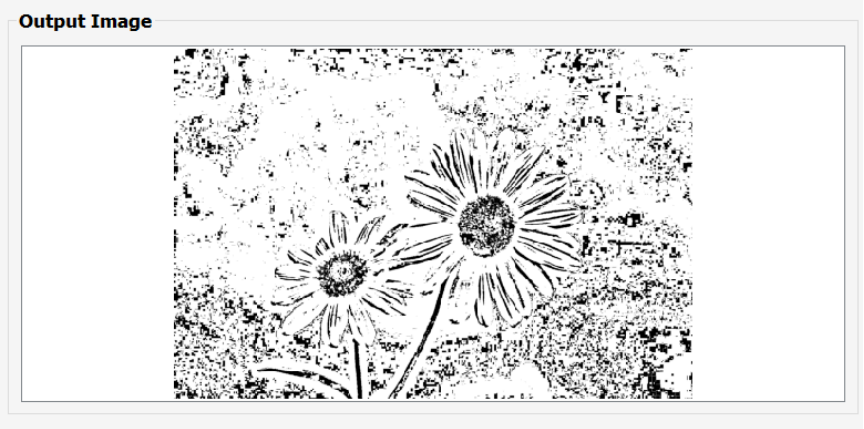
Spectral Thresholding with local thresholding:

Applying the multi-Otsu method with local thresholding allows for segmenting an image into multiple regions, each with its optimal threshold. This can lead to more precise segmentation, especially in images with varying lighting conditions or complex backgrounds. However, it may also increase computational complexity compared to global thresholding methods.

Algorithm steps:
The same algorithm of global spectral thresholding but we will divide the input image into overlapping local windows of a specified size. For each local window, the multi-Otsu thresholding algorithm is applied independently to obtain multiple threshold values. These thresholds are used to segment the local window into multiple intensity levels. The resulting thresholded image is obtained by combining the thresholded local windows.

Trial:

1. Smaller Window Size:
    - Smaller window sizes result in more localized thresholding.
    - Each local window covers a smaller region of the image, leading to finer details in the thresholded result.
    - However, smaller window sizes may lead to increased computational overhead due to a larger number of windows.
    - Smaller window sizes may result in slower performance due to the increased number of local thresholding operations.



2. Larger Window Size:
    - Larger window sizes result in more global thresholding.
    - Each local window covers a larger region of the image, smoothing out local variations in intensity.
    - This can lead to a more global overview of the image, but finer details may be lost.
    - Larger window sizes may lead to faster processing but may sacrifice detail in the thresholded result.

IV. K-means Clustering

K-means clustering is a popular unsupervised machine-learning technique that sorts similar data into groups, or clusters. The algorithm divides a set of n observations into k clusters, where each observation belongs to the cluster with the nearest mean. The ‘K’ in K-means represents the user-defined number of clusters. The goal of K-means clustering is to minimize the variances between the data points and the cluster’s centroid, thereby forming groups in a manner that maximizes the similarity amongst the data within each unique cluster.

The term “K-means” was first used by James MacQueen in 1967, though the idea goes back to Hugo Steinhaus in 1956. The standard algorithm was first proposed by Stuart Lloyd of Bell Labs in 1957 as a technique for pulse-code modulation.

The K-means clustering algorithm works as follows:

- **Initialization**: The process starts by randomly assigning each data point to an initial group. This is done by selecting random points from the data as the initial centroids.
- **Assignment**: Each observation is evaluated and assigned to the closest cluster. The definition of “closest” is that the Euclidean distance between a data point and a group’s centroid is shorter than the distances to the other centroids. The Euclidean distance between two points p and q in n-dimensional space is calculated as:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

- **Update**: When a cluster gains or loses a data point, the K-means clustering algorithm recalculates its centroid. If a cluster has no points, it reinitializes its centroid. The centroid of a set of n points in d-dimensional space is calculated as:

$$c = \frac{1}{n}\sum_{i=1}^{n} p_i$$

- **Iteration**: The algorithm repeats the assignment and update steps until it can no longer assign data points to a closer set, or until it reaches a maximum number of iterations.
- **Result**: When the K-means clustering algorithm finishes, all groups have the minimum within-cluster variance, which keeps them as small as possible. Sets with minimum variance and size have data points that are as similar as possible. The within-cluster variance is calculated as:

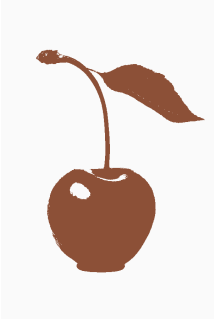
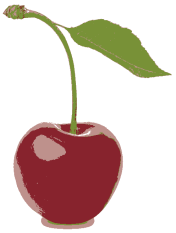
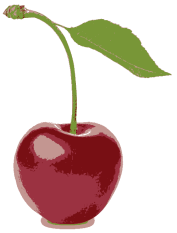
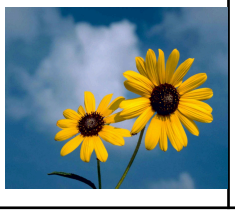
$$V(C) = \sum_{p \in C} (p - c)^2$$

In this formula, C is a cluster, p is a point in C, and c is the center of C. The goal of K-means is to minimize this variance.

For the kmeans_segmentation function:

- image = self.current_img: This line gets the current image from the class instance.
- pixels = image.reshape((-1, 3)): This line reshapes the image into a 2D array of pixels.
- pixels = np.float32(pixels): This line converts the pixel values to floating point.
- k = self.horizontalSlider.value(): This line gets the number of clusters from the user interface.
- iterations = self.horizontalSlider_2.value(): This line gets the maximum number of iterations from the user interface.
- centers = pixels[np.random.choice(pixels.shape[0], size=k, replace=False)]: This line initializes random centers.

- The for loop computes distances from pixels to centers, assigns each pixel to the closest center, and computes new centers as the mean of the assigned pixels. If a cluster has no points, it reinitializes its center.
- centers = np.uint8(centers): This line converts the cluster centers back to 8-bit values.
- segmented_image = centers[labels.flatten()]: This line maps the labels to the cluster centers to create the segmented image.
- segmented_image = segmented_image.reshape(image.shape): This line reshapes the segmented image back to the original image shape.
- self.display_filtered_img(segmented_image, self.output_graphicsView_2, self.output_scene_2): This line displays the segmented image in the user interface.
- colors = np.unique(segmented_image.reshape(-1, segmented_image.shape[2]), axis=0): This line finds the unique colors in the segmented image.
- print(f"# colors in the segmented image is {len(colors)}"): This line prints the number of unique colors in the segmented image.

Original Image	Clusters = 2	Clusters = 3	Clusters = 4	Clusters = 5
				
				

V. Region Growing Segmentation

Region growing is a technique used in image segmentation to partition an image into regions or segments based on similarity criteria, such as pixel intensity, color similarity, or texture. The basic idea is to start with a seed point (or seed points) and iteratively grow a region around these points by adding neighboring pixels that satisfy a predefined similarity criterion.

Steps:

1. Initialize seed point

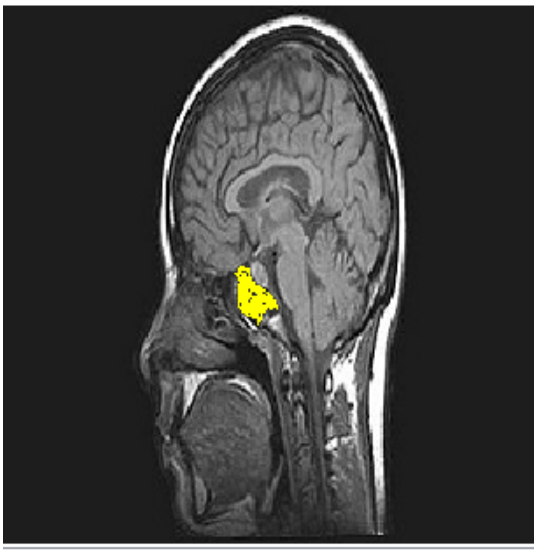
Choose an initial seed point from which the region growing process will start. This initial seed point is chosen manually by the user using the cursor. This allows the user to select a specific point within the image from which the segmentation process will begin.

2. Define Similarity Criterion: Determine the criterion for similarity between neighboring pixels. This criterion can be based on various attributes such as intensity, color, texture, or a combination of these.
   We chose a similarity criterion to be based on the color difference between the seed pixel and its neighbors.
3. Threshold
   The threshold determines the allowable difference in color between a seed pixel and its neighbors for them to be considered part of the same region during the region growing process.

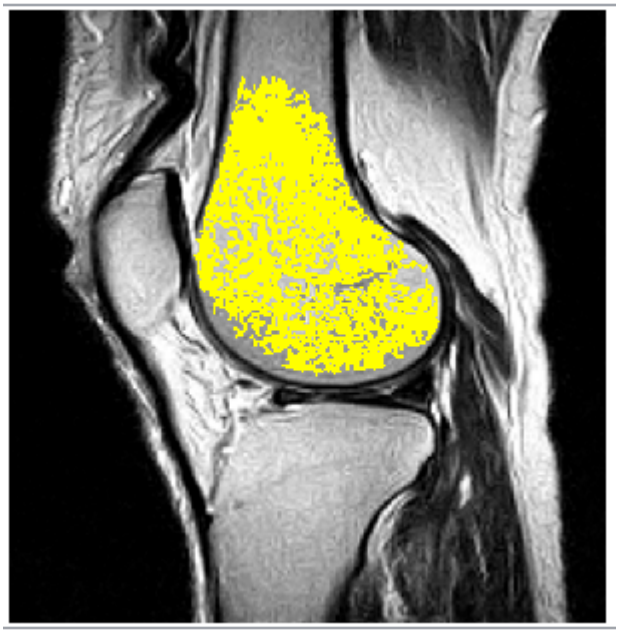
Results:

1. At threshold = 4:

2. At threshold = 25:



3. At threshold = 50:



For coloured images:

As the threshold increases the segmented region is expected to grow progressively larger, incorporating more neighbouring pixels that meet the heightened similarity criterion.

VI. Agglomerative Clustering

1. Introduction:

Agglomerative clustering is a hierarchical clustering technique widely used in data analysis and image processing. It aims to group similar data points or pixels together based on a defined distance metric.

2. Algorithm Overview:

The agglomerative clustering algorithm implemented here follows a bottom-up approach:

- It starts by initializing each data point or pixel as a separate cluster.
- Then, it iteratively merges the two most similar clusters until the desired number of clusters is reached.

3. Algorithm Components:

a. Euclidean Distance Calculation:

- The euclidean_distance function computes the Euclidean distance between two points in a multidimensional space.
- Parameters: x1, x2 (numpy arrays) representing the coordinates of two points.

b. Cluster Distance Calculation:

- The clusters_distance function calculates the Euclidean distance between the centroids of two clusters.
- Parameters: cluster1, cluster2 (lists of points) representing clusters of points.

c. Initial Clusters Generation:

- The initial_clusters function partitions the points into initial clusters based on color similarity.
- Parameters: points (list of points), initial_k (integer) representing the initial number of clusters.
- It uses a specified number of clusters (initial_k) and assigns each point to the closest cluster based on Euclidean distance.

d. Agglomerative Clustering:

- The fit_agglomerative function performs the main agglomerative clustering process.
- Parameters: points (list of points), clusters_num (integer) representing the desired number of final clusters.

- It iteratively merges the most similar clusters until the desired number of clusters is reached.

e. Prediction Functions:

- predict_cluster: Predicts the cluster number that a given point belongs to.
- predict_center: Predicts the center of the cluster that a given point belongs to.

4. Applying Agglomerative Clustering to Images:

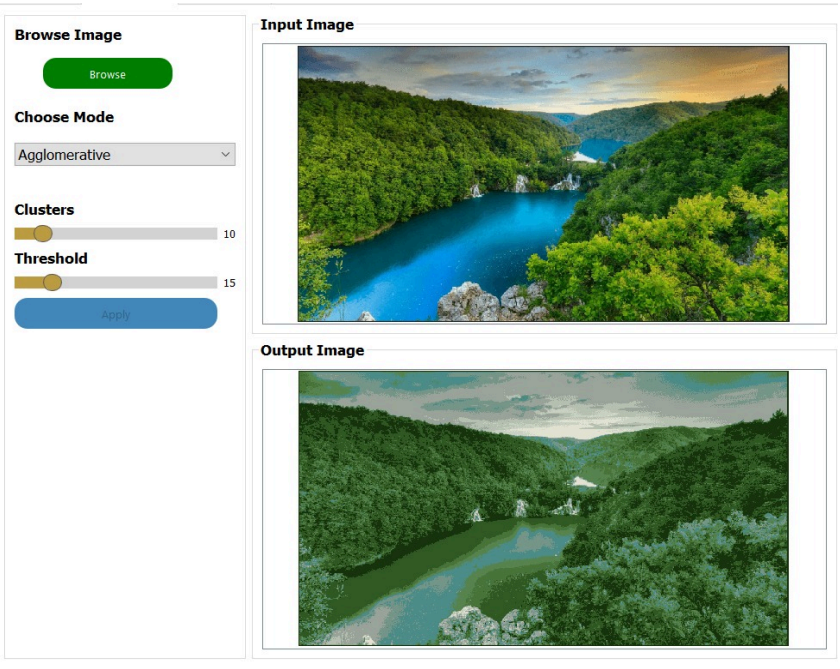
- The apply_agglomerative function applies the clustering algorithm to input images.
- Parameters:
    - source (numpy array representing the input image): The image to be clustered.
    - clusters_numbers (integer, default=2): The number of clusters to form.
    - initial_clusters (integer, default=25): The initial number of clusters.
- It reshapes the image, applies the clustering algorithm, and generates a clustered image where pixels are assigned to different clusters based on color similarity.

5. Output:

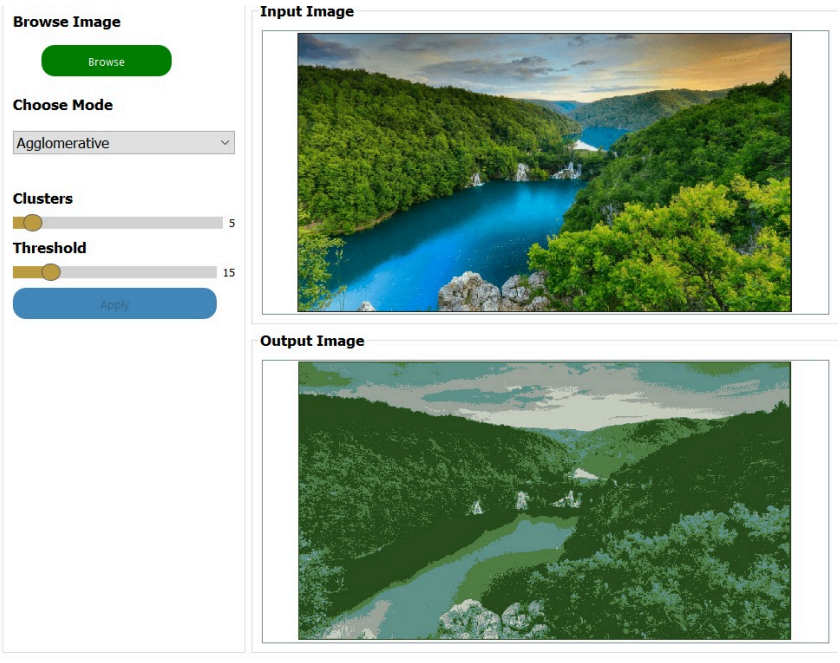
- The output of the algorithm is a clustered image, where pixels are assigned to different clusters based on their color similarity.
- Each pixel in the output image is represented by the center of the cluster it belongs to.

6. Impact of Parameters on Output Image:

- Number of Clusters (clusters_numbers):



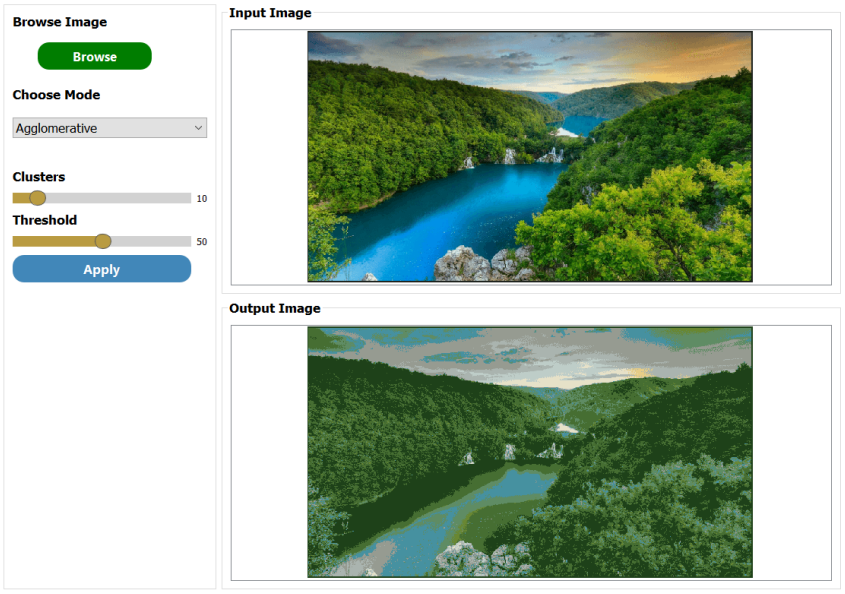
Num of Clusters = 10



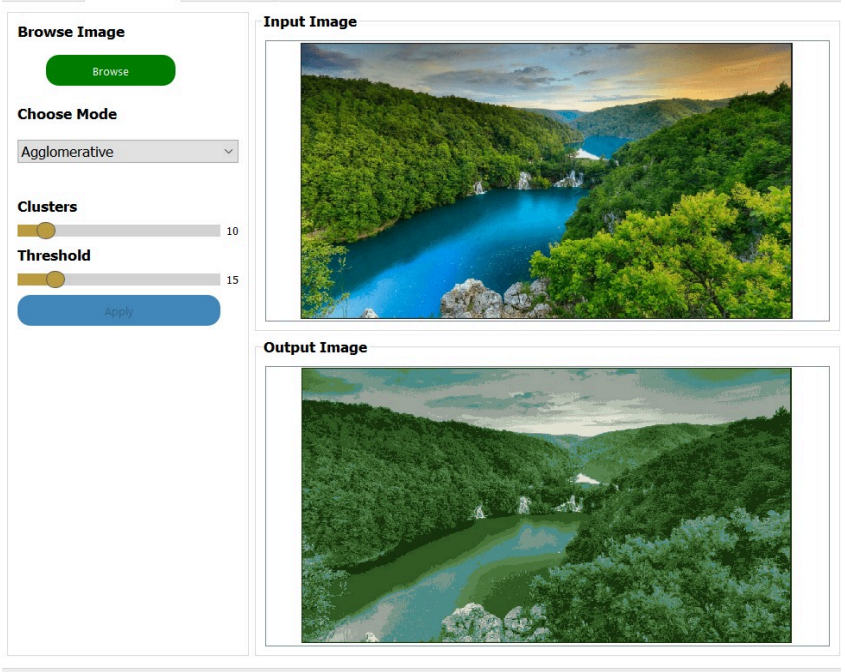
Num of Clusters = 5

Observation: Increasing the number of clusters will result in a **greater number of clusters** in the output image. Each cluster will represent a distinct color group, potentially leading to finer details but also **increasing complexity**.

- Initial Number of Clusters (initial_clusters):



Initial clusters = 50



Initial clusters = 15

Observation: A **higher** initial number of clusters leads to **finer initial clustering**, potentially capturing **more subtle color variations** in the image. However, it may also **increase** the computational **complexity** of the algorithm.

NOTE: Initializing each point as a separate cluster leads to a large number of initial clusters and The high number of initial clusters significantly increases computational complexity and memory usage, potentially leading to program

crashes so, to solve this problem partitioning the points into an initial set of clusters based on color similarity, using a predefined number of initial clusters (initial_k).

VII. Mean shift Clustering

Mean Shift Clustering is a non-parametric, density-based clustering algorithm that is used to identify clusters within a dataset. It works by iteratively shifting each data point towards the mode (i.e., the highest density) of the distribution of points within a certain radius. The algorithm continues these shifts until the points converge to a local maximum of the density function. These local maxima represent the clusters in the data. One of the main advantages of Mean Shift Clustering is that it does not require the number of clusters to be specified beforehand. It can handle arbitrary shapes and sizes of clusters. However, it can be sensitive to the choice of kernel and the radius of the kernel.

The Mean Shift Clustering algorithm can be understood in three main steps:

- Kernel Density Estimation: The first step involves estimating the underlying probability density function (PDF) of the data points. This is typically done using kernel density estimation, where each data point is represented by a kernel function centered at that point. The kernel function specifies the weight assigned to each data point in the density estimation process. The kernel function K(x) is typically a Gaussian function:

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

- Shifting Data Points: In the second step, the algorithm iteratively shifts the data points towards regions of higher density. The shift is determined by calculating the mean shift vector for each data point, which represents the direction and magnitude of the shift. The mean shift vector is calculated as the weighted average of the

$$M(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)} - x$$

differences between the data point and its neighboring points, where the weights are determined by the kernel function. The mean shift vector M(x) is given by:
where N(x) is the neighborhood of data points within the specified bandwidth of the data point x.
Each data point x is updated by adding the mean shift vector to it:

$$x_{\text{new}} = x + M(x)$$

- Convergence and Cluster Identification: The algorithm continues shifting the data points until convergence is reached. Convergence occurs when the mean shift vectors become very small or negligible. Once convergence is achieved, the final position of each data point represents a cluster center. The algorithm assigns each data point to the closest cluster center, thereby identifying the clusters within the data. This is done by calculating the Euclidean distance between x_new and x and checking if it's less than a threshold:

$$\|x_{\text{new}} - x\| < \text{threshold}$$

If a new mean is close to an existing cluster center c, the cluster center is updated to be the midpoint of c, and the new mean:

$$c_{\text{new}} = 0.5 * (c + x_{\text{new}})$$

The code implemented for the algorithm is as follows:

- def fit(self, data): This line defines the fit method that takes self and data as parameters.
- ball_tree = BallTree(data) This line creates a BallTree from the data for efficient nearest neighbor queries.
- while len(data) > 0: This line starts a loop that continues until all data points have been visited.
- current_mean_index = np.random.randint(0, len(data)) This line selects a random index from the data.
- current_mean = data[current_mean_index] This line assigns the data point at the selected index as the current mean.
- while True: This line starts an inner loop that continues until convergence is achieved.
- tracked_points_indices = ball_tree.query_radius(current_mean.reshape(1, -1), r=self.bandwidth)[0] This line finds all data points within the specified bandwidth of the current mean.

- tracked_points = data[tracked_points_indices] This line gets the tracked points from the data.
- new_mean = np.mean(tracked_points, axis=0) This line calculates the new mean of the tracked points.
- if np.linalg.norm(new_mean - current_mean) < self.threshold: This line checks if the new mean is close enough to the current mean.
- for i, c in enumerate(self.clusters): This line starts a loop over the existing clusters.
- if np.linalg.norm(c - new_mean) < 0.5 * self.bandwidth: This line checks if the new mean is close to the current cluster.
- self.clusters[i] = 0.5 * (c + new_mean) This line updates the current cluster to be the midpoint of the current cluster and the new mean.
- self.clusters.append(new_mean) This line adds the new mean as a new cluster.
- data = np.delete(data, tracked_points_indices, axis=0) This line removes the tracked points from the data.
- if len(data) == 0: This line checks if all data points have been visited.
- ball_tree = BallTree(data) This line updates the BallTree with the remaining data points.
- else: This line starts the else block for the if statement checking if the new mean is close enough to the current mean.
- current_mean = new_mean This line updates the current mean to the new mean.
- def predict(self, data): This line defines the predict method that takes self and data as parameters.
- labels = [] This line initializes an empty list to store the labels.
- for point in data: This line starts a loop over each data point.
- distances = cdist([point], self.clusters) This line calculates the distance from the point to all cluster centers.
- labels.append(np.argmin(distances)) This line assigns the point to the cluster whose center is closest.
- return labels This line returns the labels.
- def mean_shift_segmentation(self, image): This line defines the mean_shift_segmentation method that takes self and image as parameters.
- pixels = image.reshape(-1, 3) This line reshapes the image to a 2D array of pixels.
- self.fit(pixels) This line applies the Mean Shift algorithm to the pixels.
- labels = self.predict(pixels) This line predicts the cluster label for each pixel.
- labels = np.array(labels).reshape(image.shape[0], image.shape[1]) This line reshapes the labels to have the same dimensions as the original image.
- segmented_image = np.zeros_like(image) This line initializes an empty image with the same shape as the original image.
- for i in range(len(self.clusters)): This line starts a loop over each cluster.
- segmented_image[labels == i] = self.clusters[i] This line assigns each pixel in the cluster the value of its cluster center.
- self.display_filtered_img(segmented_image, self.output_graphicsView_2, self.output_scene_2) This line displays the segmented image.

A few optimization techniques were used to improve the efficiency of the Mean Shift Clustering algorithm:

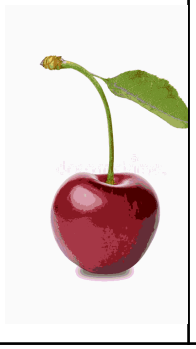
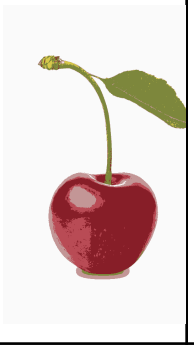
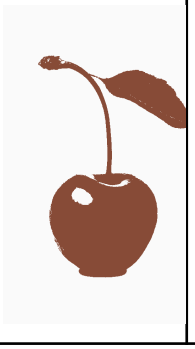
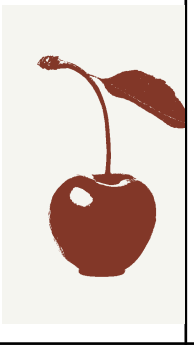
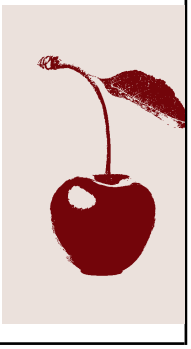
1. **BallTree for Nearest Neighbor Queries**: The code uses a BallTree data structure to efficiently perform nearest neighbor queries. A BallTree is a binary tree in which every node defines a D-dimensional hypersphere, or "ball", containing a subset of the points to be searched. Unlike a brute-force search for nearest neighbors (which would involve calculating the distance from a query point to every other point in the dataset), a BallTree allows for more efficient queries by pruning large portions of the tree when searching for nearest neighbors.
2. **Batch Update**: The code updates the data points in batches. In each iteration of the Mean Shift algorithm, it selects all data points within the specified bandwidth of the current mean and updates their positions. This batch update approach is more efficient than updating one point at a time.
3. **Early Stopping**: The code uses an early stopping criterion to terminate the algorithm before it has visited all data points. If the new mean is close enough to the current mean (as determined by a threshold), the algorithm stops updating the current mean and moves on to the next unvisited data point. This early stopping technique can significantly reduce the number of iterations needed for the algorithm to converge, especially in cases where data points are densely packed.
4. **Cluster Merging**: The code merges clusters that are close to each other. If the new mean is close to an existing cluster center, the algorithm merges the new mean with the closest cluster center by updating the cluster center to be the midpoint of the current cluster center and the new mean. This cluster merging technique helps to prevent over-segmentation of the data.
5. **Data Pruning**: After each iteration, the code removes all data points within the bandwidth of the new mean from the data. This data pruning technique reduces the size of the data to be processed in subsequent iterations, thereby speeding up the algorithm.
6. **Dynamic BallTree Update**: The code updates the BallTree with the remaining data points after each iteration. This dynamic update of the BallTree ensures that the tree always accurately represents the current state of the data, thereby improving the accuracy of the nearest neighbor queries.

The threshold value in the Mean Shift Clustering algorithm is a hyperparameter that determines when the algorithm should stop iterating. Specifically, it's used to decide when the shift of the mean is small enough to consider the algorithm as converged.

The choice of the threshold value can greatly affect the performance and results of the algorithm:

- If you **increase the threshold**, the algorithm will converge faster because it will stop iterating sooner. However, this might lead to underfitting, where the clusters are too broad and may not accurately represent the data distribution.

- If you **decrease the threshold**, the algorithm will take longer to converge because it will continue iterating until the mean shift is very small. This might lead to overfitting, where the clusters are too specific and may not generalize well to new data.

Original Image	Bandwidth = 60	Bandwidth = 100	Bandwidth = 180	Bandwidth = 260	Bandwidth = 350
					

VIII. RGB to LUV Conversion

The LUV color space, also known as CIE 1976 L*, u*, v* color space or simply CIELUV, is a color space that was adopted by the International Commission on Illumination (CIE) in 19761. It was designed as a simple-to-compute transformation of the 1931 CIE XYZ color space, attempting to achieve perceptual uniformity. This means the perceived difference between any two colors in this space should be proportional to their Euclidean distance.

The LUV color space is extensively used in applications such as computer graphics which deal with colored lights. It is beneficial in these applications because additive mixtures of different colored lights will fall on a line in CIELUV's uniform chromaticity diagram.

The algorithm for converting from RGB to LUV color space is based on the transformation matrix from RGB to XYZ, and then further transformations to compute L, u, and v values. This algorithm is derived from the definitions and formulas provided by the CIE when they adopted the LUV color space.

The algorithm is as follows:

1. **Normalize the RGB values**: The RGB values of the image are normalized to the range [0, 1] by dividing each value by 255.
2. **Convert RGB to XYZ**: The normalized RGB values are converted to XYZ color space. This is done using a transformation matrix, which is multiplied by the RGB values.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

where M is the transformation matrix:

$$M = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix}$$

3. **Compute u' and v'**: The u' and v' values are computed from the XYZ values using the formulas:

$$u' = \frac{4X}{X + 15Y + 3Z}$$
$$v' = \frac{9Y}{X + 15Y + 3Z}$$

4. **Compute L**: The L value is computed from the Y value of the XYZ color space using the piecewise function:
5. **Compute u and v**: The u and v values are computed from L, u', and v' using the

$$L = \begin{cases} 116 \cdot Y^{1/3} - 16 & \text{if } Y > 0.008856 \\ 903.3 \cdot Y & \text{if } Y \leq 0.008856 \end{cases}$$

formulas:

$$u = 13L \cdot (u' - u_n)$$
$$v = 13L \cdot (v' - v_n)$$

6. **Scale L, u, v to the 8-bit range**: The L, u, and v values are scaled to the 8-bit range [0, 255] using the formulas:

$$L = \frac{255}{100} \cdot L$$

$$u = \frac{255}{354} \cdot (u + 134)$$

$$v = \frac{255}{262} \cdot (v + 140)$$

7. **Reshape and display the image**: The LUV values are reshaped back to the original image shape and displayed.

For the rgb_to_luv function:

- image_rgb = self.image_rgb / 255.0: This line normalizes the RGB values to the range [0, 1].
- reshaped_rgb = image_rgb.reshape(-1, 3): This line reshapes the image to a 2D array so we can apply the transformation matrix.
- xyz = np.dot(reshaped_rgb, M.T): This line converts the RGB values to XYZ color space by multiplying with the transformation matrix.
- u_prime = 4 * xyz[:, 0] / (xyz[:, 0] + 15 * xyz[:, 1] + 3 * xyz[:, 2]) and v_prime = 9 * xyz[:, 1] / (xyz[:, 0] + 15 * xyz[:, 1] + 3 * xyz[:, 2]): These lines compute the u' and v' values from the XYZ values.
- L = np.where(xyz[:, 1] > 0.008856, 116 * np.cbrt(xyz[:, 1]) - 16, 903.3 * xyz[:, 1]): This line computes the L value from the Y value of the XYZ color space.
- u = 13 * L * (u_prime - 0.19793943) and v = 13 * L * (v_prime - 0.46831096): These lines compute the u and v values from L, u', and v'.
- L = 255 / 100 * L, u = 255 / 354 * (u + 134), and v = 255 / 262 * (v + 140): These lines scale the L, u, and v values to the 8-bit range [0, 255].
- self.display_filtered_img(np.dstack((L, u, v)).reshape(image_rgb.shape).astype(np.uint8), self.output_graphicsView_3, self.output_scene_3): This line reshapes the LUV values back to the original image shape and displays the image.

Result using rgb_to_luv	Result using cv2.COLOR_RGB2Luv