THRESHOLDING AND SEGMENTATION

TEAM 14

- AHMED MOHAMED ALI
- ALI SHERIF BADRAN
- MUHANNAD ABDALLAH
- OSAMA MOHAMED ALI

SUBMITTED TO

- DR. AHMED M. BADAWI
- ENG. LAILA ABBAS
- ENG. OMAR DAWAH

TABLE OF CONTENTS

OPTIMAL THRESHOLDING

Introduction

Optimal thresholding is a method used in image processing to segment images by finding the best threshold that separates foreground from background. The optimal threshold is computed by iteratively refining an initial guess until it converges to a stable value. This technique is particularly useful for images with varied intensity distributions, where a single fixed threshold might not be effective. It aims to balance the mean intensity values of the background and foreground, making it adaptable to different types of images.

Algorithm Overview

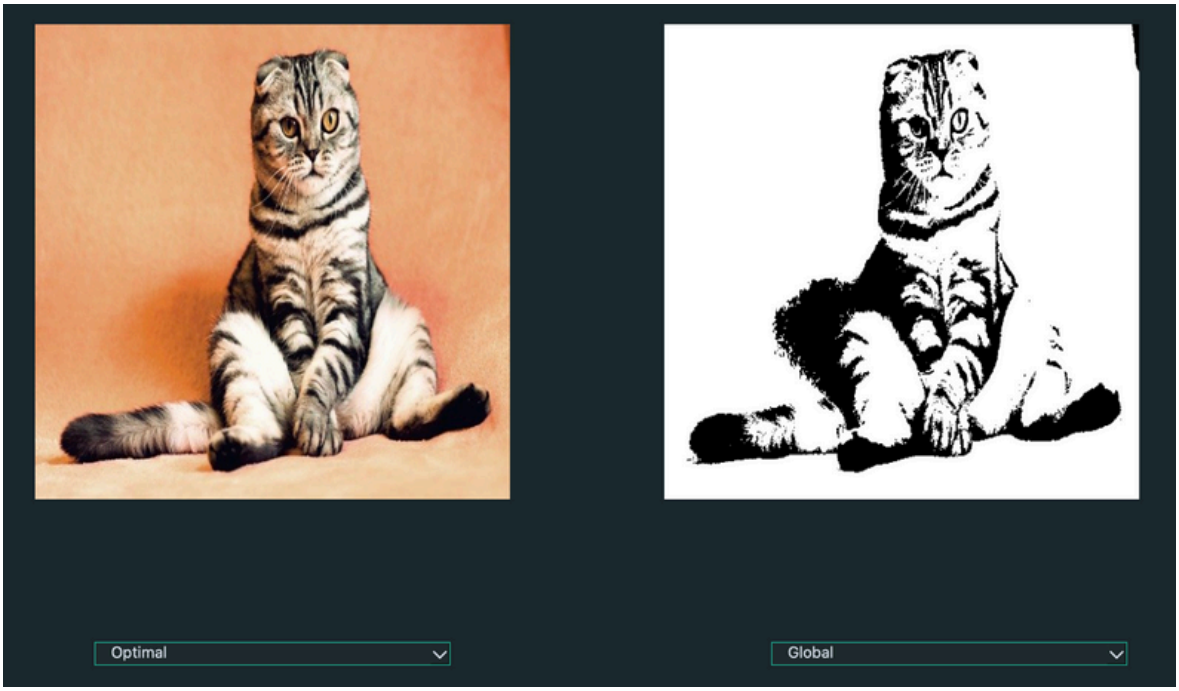
The implementation of optimal thresholding starts with an initial estimate of the background and object means. The approach then calculates a threshold that is the average of these two means. This process involves the following steps:

1. Input Image: A grayscale image is taken as input. If the image is in color, it is converted to grayscale.
2. Background Mean: The initial background mean is computed using the pixel intensities at the four corners of the image. This step assumes that these corners represent the background.
3. Object Mean: The object mean is calculated by summing all pixel intensities and subtracting the contribution from the background corners, then dividing by the total number of pixels excluding the corners.
4. Initial Threshold: An initial threshold is computed as the average of the background and object means.
5. Iterative Refinement: The algorithm refines the threshold by applying the compute_optimal_threshold function to the image, which calculates the new threshold based on the current estimate. This process continues until the old and new thresholds converge to the same value.

```python
def optimal_thresholding(image: np.ndarray):
img = np.copy(image)
if len(img.shape) > 2:
    img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

# Compute the initial background mean
x = img.shape[1] - 1
y = img.shape[0] - 1
background_mean = np.mean([img[0, 0], img[0, x], img[y, 0], img[y, x]])

total = img.sum() - background_mean * 4
count = img.size - 4
object_mean = total / count

# Initial threshold and iterative refinement
threshold = (background_mean + object_mean) / 2
new_threshold = compute_optimal_threshold(img, threshold)
old_threshold = threshold

# Iteratively refine the threshold until it converges
while old_threshold != new_threshold:
    old_threshold = new_threshold
    new_threshold = compute_optimal_threshold(img, old_threshold)

# Apply the final threshold to create the binary image
binary_image = np.zeros_like(img)
binary_image[img > new_threshold] = 255

return binary_image
```

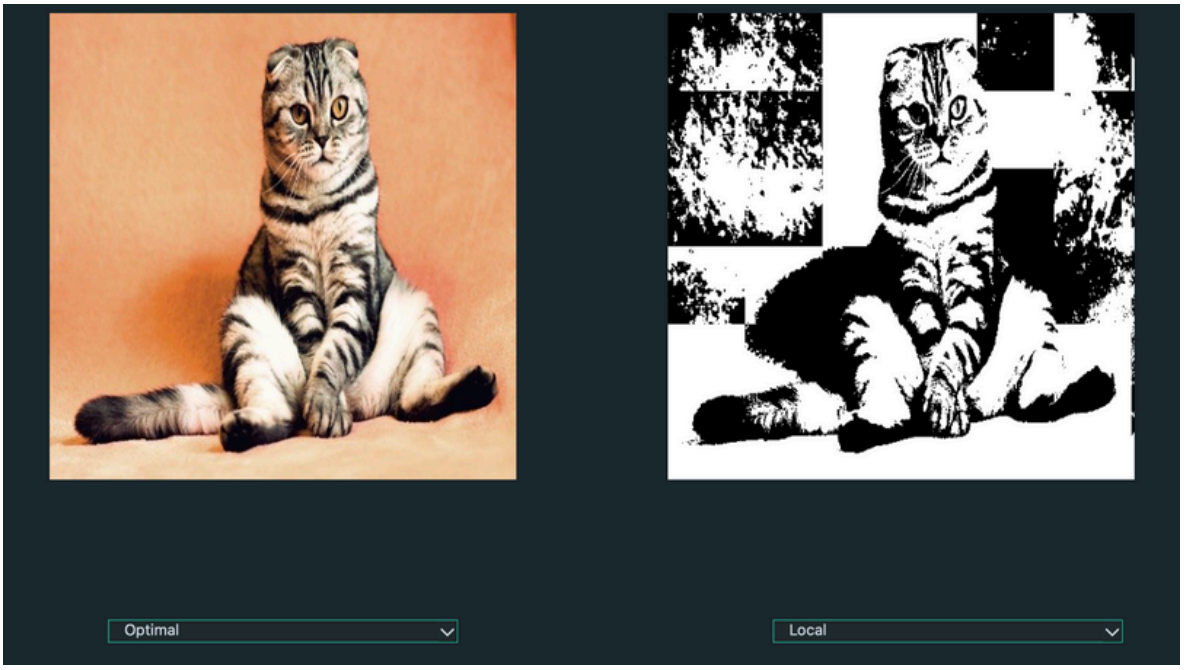
Results

Results and Applications Optimal thresholding provides a robust and adaptable way to segment images, accommodating variations in intensity distribution. And here you can see the result of global optimal thresholding :



Local thresholding

Local thresholding divides an image into smaller regions and applies a separate thresholding operation to each section. This method helps segment images with uneven lighting or varying backgrounds. By calculating step sizes, the image is divided into a grid, and a specific thresholding function is applied to each region. The results are then combined to create the final thresholded image. The approach provides improved accuracy over global thresholding, allowing for more precise segmentation in complex images. And here is the result of dividing the image into 6 regions in both x and y axes.



OTSU THRESHOLDING

Introduction

Otsu's thresholding is a widely used method for automatically finding an optimal threshold to convert a grayscale image into a binary image. The approach aims to minimize the within-class variance while maximizing the between-class variance, thereby achieving effective segmentation into foreground and background. Unlike traditional thresholding methods that require manual tuning, Otsu's thresholding determines the best threshold based on the image's intensity distribution.

Algorithm Overview

The key to Otsu's thresholding is the ability to identify an optimal threshold by evaluating the between-class variance for each possible threshold value. The method calculates a histogram of the image's pixel intensities and uses this information to compute various statistical metrics, such as the cumulative sum and cumulative mean, which are then used to determine the between-class variance.
Here's a high-level breakdown of the algorithm:

1. Calculate Histogram:
    - Compute the 256-bin histogram for the grayscale image, representing the frequency of each intensity level.
2. Normalize Histogram:
    - Normalize the histogram to create a probability distribution. This step is essential for computing cumulative sums and means.
3. Calculate Cumulative Sums and Means:
    - Compute the cumulative sum of the normalized histogram, representing the cumulative probability of pixel intensities.
    - Compute the cumulative mean, indicating the weighted sum of pixel intensities.
4. Calculate Global Mean:
    - Determine the global mean of the entire image, which is the final value of the cumulative mean.
5. Compute Between-Class Variance:
    - Calculate the between-class variance for each possible threshold. This step involves the global mean, cumulative sum, and cumulative mean to compute the variance numerators and denominators.
    - Handle division by zero by adding a small constant (1e-10) to any zero or very small denominator.
6. Find Optimal Threshold:
    - The optimal threshold is the value that maximizes the between-class variance, determined by np.argmax.
7. Generate Binary Image:
    - Apply the optimal threshold to create a binary image, setting pixels above or equal to the threshold to 255 (white) and those below to 0 (black).

```python
def otsu_threshold(image):

    # Ensure the image is grayscale
    if len(image.shape) > 2:
        raise ValueError("Otsu's thresholding requires a grayscale image.")

    # Compute the histogram of the grayscale image
    hist, _ = np.histogram(image, bins=256, range=(0, 256))

    # Normalize the histogram to create a probability distribution
    hist_norm = hist / float(np.sum(hist))

    # Compute the cumulative sum and cumulative mean of the normalized histogram
    cum_sum = np.cumsum(hist_norm)  # Cumulative sum
    cum_mean = np.cumsum(hist_norm * np.arange(256))  # Cumulative mean

    # Compute the global mean of the image
    global_mean = cum_mean[-1]  # Last value of cum_mean

    # Compute the between-class variance for each threshold
    variance_numerators = global_mean * cum_sum - cum_mean
    variance_denominators = cum_sum * (1 - cum_sum)
    # Avoid division by zero or very small denominators
    variance_denominators[variance_denominators == 0] = 1e-10
    between_class_variance = (variance_numerators ** 2) / variance_denominators

    # Find the threshold that maximizes the between-class variance
    optimal_threshold = np.argmax(between_class_variance)

    # Apply the calculated threshold to create a binary image
    binary_image = (image >= optimal_threshold).astype(np.uint8) * 255

    return binary_image, optimal_threshold
```

Results

Otsu's thresholding provides an efficient and automated way to segment images into foreground and background without manual intervention. It works well for images with distinct intensity distributions but might not perform optimally for those with complex backgrounds or uneven lighting. And here are the results of global and local thresholding respectively.

SPECTRAL THRESHOLDING

Introduction

Spectral thresholding is a technique used in image processing to determine optimal thresholds for dividing an image into multiple classes. Unlike other thresholding methods, spectral thresholding calculates two optimal threshold values, creating a double-threshold system. This approach can effectively separate an image into background, weak signal, and strong signal regions.

Algorithm Overview

Spectral thresholding computes optimal low and high thresholds based on the image's histogram and the between-class variance. The method involves the following steps:

1. Grayscale Conversion:
   - Convert the input image to grayscale if it is in RGB format. This ensures that spectral thresholding operates on a single-channel image.
2. Histogram Calculation:
   - Compute the histogram of the grayscale image, which provides the distribution of pixel intensities across 256 levels.
3. Global Mean Calculation:
   - Calculate the global mean of the image, representing the average pixel intensity.
4. Iterative Optimal Thresholding:
   - Iterate through all possible pairs of low and high thresholds to find the optimal combination. For each pair, compute the class weights (proportion of pixels in each segment) and the class means (average intensity of each segment).
   - Calculate the between-class variance for each pair of thresholds.
   - Keep track of the thresholds that produce the highest between-class variance.
5. Double Thresholding:
   - Apply double thresholding to the image using the optimal low and high thresholds.
   - Classify pixels into three categories: background, weak signal, and strong signal.
6. Return Binary Image:
   - Create a binary image with pixel values set according to the calculated thresholds:
     - Pixels below the low threshold are set to 0 (background).
     - Pixels between the low and high thresholds are set to 128 (weak signal).
     - Pixels above the high threshold are set to 255 (strong signal).

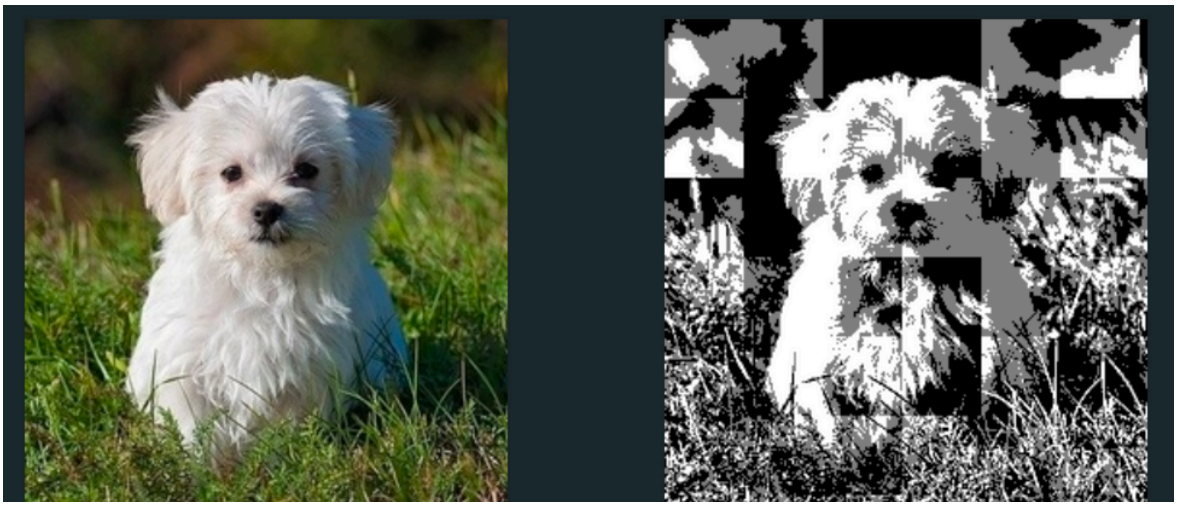
```python
def spectral_thresholding(img):

    if len(img.shape) > 2 and img.shape[2] == 3:
        img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Compute the histogram and the global mean
    hist, _ = np.histogram(img.flatten(), 256, [0, 256])
    global_mean = np.sum(hist * np.arange(256)) / img.size

    # Initialize variables for optimal thresholds and max variance
    optimal_high_threshold = 0
    optimal_low_threshold = 0
    max_variance = 0

    # Iterate through potential high and low thresholds to find the optimal ones
    for high in range(1, 256):
        for low in range(1, high):
            # Calculate class weights and means
            w0 = np.sum(hist[0:low])
            if w0 == 0:
                continue
            mean0 = np.sum(np.arange(low) * hist[0:low]) / w0

            w1 = np.sum(hist[low:high])
            if w1 == 0:
                continue
            mean1 = np.sum(np.arange(low, high) * hist[low:high]) / w1

            w2 = np.sum(hist[high:])
            if w2 == 0:
                continue
            mean2 = np.sum(np.arange(high, 256) * hist[high:]) / w2

            # Compute the between-class variance
            variance = (w0 * (mean0 - global_mean) ** 2 +
                        w1 * (mean1 - global_mean) ** 2 +
                        w2 * (mean2 - global_mean) ** 2)

            # Update optimal thresholds if variance is the highest so far
            if variance > max_variance:
                max_variance = variance
                optimal_low_threshold = low
                optimal_high_threshold = high

    # Apply double thresholding with the optimal thresholds
    binary = np.zeros(img.shape, dtype=np.uint8)
    binary[img < optimal_low_threshold] = 0  # Background
    binary[(img >= optimal_low_threshold) & (img < optimal_high_threshold)] = 128  # Weak
signbinary[img >= optimal_high_threshold] = 255  # Strong signal

    return binary
```

Results

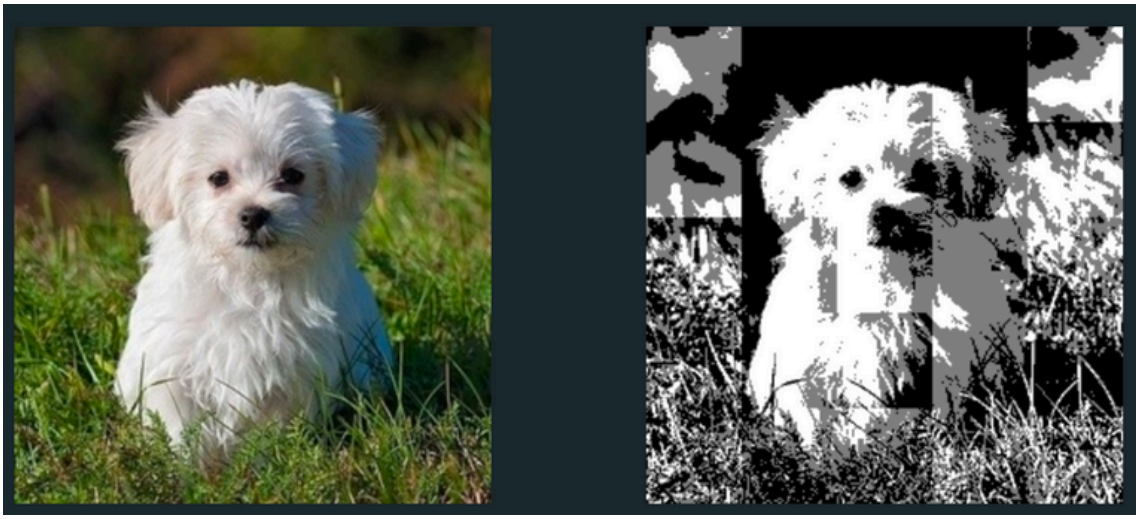
Spectral thresholding provides a flexible approach to image segmentation, allowing for fine-tuned control over thresholds. It is particularly useful when dealing with images that contain multiple distinct regions. The double thresholding approach helps to maintain a balance between signal and background, reducing noise and enhancing feature extraction.

Local thresholding

When applying local spectral thresholding, the final binary image should exhibit more adaptability to varying lighting conditions compared to global spectral thresholding. This method can capture nuanced transitions within different parts of the image, ensuring that regions with lower or higher contrast are appropriately thresholded. And here is the result of applying local spectral thresholding.



And we can experiment different results with different regions values, as changing the number of regions in local thresholding directly impacts the sensitivity and adaptability of the thresholding process. When dividing an image into smaller regions, the algorithm's ability to capture local variations in contrast or illumination improves. And here is the result of reducing the number of regions from 6 to 5.



REGION GROWING

Introduction

Region Growing Segmentation is a technique employed in image processing to partition an image into regions based on predefined criteria. It operates on the principle of grouping pixels with similar properties or characteristics, typically in terms of intensity or color, to form coherent regions.

Algorithm Overview

The Region Growing Segmentation algorithm proceeds through the following steps:

1. Initialization: Initialize a visited matrix and an empty region matrix to keep track of visited pixels and the segmented region, respectively.
2. Neighbor Definition: Define neighbor offsets for 8-connectivity to access adjacent pixels during region growing.
3. Queue Initialization: Create a queue and enqueue the seed pixel to initiate the region growing process.
4. Region Growing: While the queue is not empty, dequeue pixels and perform the following steps:
5. Mark the pixel as visited.
6. Check if the pixel meets the similarity criteria based on the intensity threshold compared to the seed pixel.
7. If the criteria are met, add the pixel to the region and enqueue its neighbors.
8. Termination Condition: Check if the size of the segmented region exceeds a predefined limit.

```
    def region_growing_segmentation(self, image, seed, threshold):
        visited = np.zeros_like(image, dtype=np.uint8)
        region = np.zeros_like(image, dtype=np.uint8)
        height, width = image.shape[:2]
        # Define neighbors offsets (8-connectivity)
        offsets = [(-1, -1), (-1, 0), (-1, 1),
                   (0, -1),           (0, 1),
                   (1, -1), (1, 0), (1, 1)]
        queue = [seed]
        image = np.float32(image)
        while queue:
            current_pixel = queue.pop(0)
            if current_pixel is not None:
                x, y = current_pixel
                not_visited_condition = visited[y, x] == 0
                # Check if pixel is within image bounds and not visited
                if 0 <= x < width and 0 <= y < height and not_visited_condition:
                    visited[y, x] = 1
                    # Check if pixel intensity meets similarity criteria
                    if abs(image[y, x] - image[seed[1], seed[0]]) <= threshold:
                        # Add pixel to region
                        region[y, x] = image[y, x]
                        for dx, dy in offsets:
                            new_x, new_y = x + dx, y + dy
                            if 0 <= new_x < width and 0 <= new_y < height:  # Ensure neighbor is within
image bounds
                                queue.append((new_x, new_y))
            # Check if region size exceeds limit
            if np.count_nonzero(region) > 350 * 350:
                break
        return region
```

REGION GROWING

Function Procedure

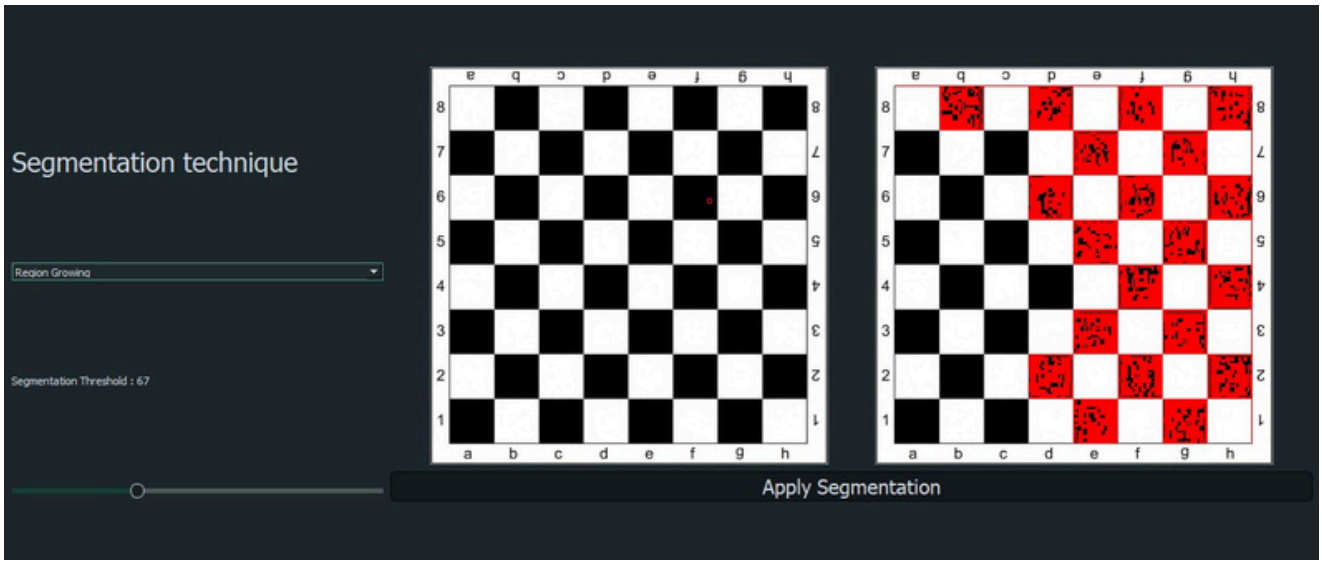
1. Convert Image to np.float32: The input image is converted to a numpy array of data type np.float32 for numerical operations.
2. Region Growing: The main procedure of the region growing algorithm is implemented using a while loop. It iteratively explores neighboring pixels and adds them to the region if they satisfy the similarity condition.
3. Termination: The region growing process terminates either when all eligible pixels have been visited or when the size of the region exceeds a specified limit.

Parameters and Thresholding

The Region Growing Segmentation algorithm accepts three parameters:

- image: Input image for segmentation.
- seed: Seed pixel coordinates to initiate the region growing process.
- threshold: Intensity threshold used to determine pixel similarity.

Results



Results demonstrate the effectiveness of the Region Growing Segmentation algorithm in segmenting images based on similarity criteria. In this image we selected a pixel in F6 Black tile after apply the segmentations with these [parameters we can now observe the segmented black tiles

AGGLOMERATIVE CLUSTERING

Introduction

Agglomerative Clustering is a hierarchical clustering technique used to group similar data points into clusters. In the context of image processing, agglomerative clustering can be applied to segment images by grouping pixels with similar characteristics into clusters.

Algorithm Overview

The Agglomerative Clustering algorithm proceeds through the following steps:

1. Initialization: Initialize clusters with individual pixels or small groups of pixels.
2. Cluster Pair Selection: Select pairs of clusters to merge based on a specified criterion, often the mean distance between cluster centroids.
3. Cluster Merge: Merge the selected clusters into a single cluster.
4. Iteration: Repeat steps 2 and 3 until the desired number of clusters is achieved.

Function Procedure

1. Euclidean Distance Calculation: The function calculate_distance computes the Euclidean distance between two points in multidimensional space.
2. Cluster Distance Calculation: The function clusters_distance calculates the distance between two clusters as the maximum distance between any two points in the clusters.
3. Mean Distance Calculation: The function clusters_mean_distance computes the mean distance between two clusters as the Euclidean distance between their centroids.
4. Initial Clusters Formation: The function initial_clusters initializes clusters by grouping image pixels based on their color.
5. Cluster Center Retrieval: The function get_cluster_center retrieves the center of the cluster to which a given point belongs.
6. Agglomerative Clustering: The function get_clusters applies the agglomerative clustering algorithm to group image pixels into a specified number of clusters.
7. Segmentation Application: The function apply_agglomerative_clustering applies agglomerative clustering to the input image and returns the segmented image.

AGGLOMERATIVE CLUSTERING

```python
def apply_agglomerative_clustering(image, clusters_number, initial_clusters_number):
    """
    Applies agglomerative clustering to the image and returns the segmented image.

    Parameters:
    image : array_like
        The input image.
    clusters_number : int
        The number of clusters to form.
    initial_clusters_number : int
        The number of initial clusters.

    Returns:
    array_like
        The segmented image.
    """
    flattened_image = np.copy(image.reshape((-1, 3)))
    cluster, centers = get_clusters(flattened_image, clusters_number, initial_clusters_number)
    output_image = []
    for row in image:
        rows = []
        for col in row:
            rows.append(get_cluster_center(list(col), cluster, centers))
        output_image.append(rows)
    output_image = np.array(output_image, np.uint8)
    return output_image
```

```python
def get_clusters(image_clusters, clusters_number, initial_clusters_number):
    clusters_list = initial_clusters(image_clusters, initial_clusters_number)
    cluster = {}
    centers = {}

    while len(clusters_list) > clusters_number:
        min_distance = float('inf')  # Initialize minimum distance to positive infinity
        for i, c1 in enumerate(clusters_list):
            for c2 in clusters_list[:i]:  # Avoid comparing a cluster with itself and avoid duplicate
comparisons
                distance = clusters_mean_distance(c1, c2)
                if distance < min_distance:
                    min_distance = distance
                    cluster1, cluster2 = c1, c2

        # Remove the two clusters from the list and merge them
        clusters_list = [cluster_itr for cluster_itr in clusters_list if not
np.array_equal(cluster_itr, cluster1) and not np.array_equal(cluster_itr, cluster2)]
        merged_cluster = cluster1 + cluster2
        clusters_list.append(merged_cluster)
    # Replace color key with cluster numbers for each cluster group (each pixel in the group as well)
    for cl_num, cl in enumerate(clusters_list):
        for point in cl:
            cluster[tuple(point)] = cl_num

    # Calculate the center of each cluster to determine the representative color of the cluster
    for cl_num, cl in enumerate(clusters_list):
        centers[cl_num] = np.average(cl, axis=0)

    return cluster, centers
```

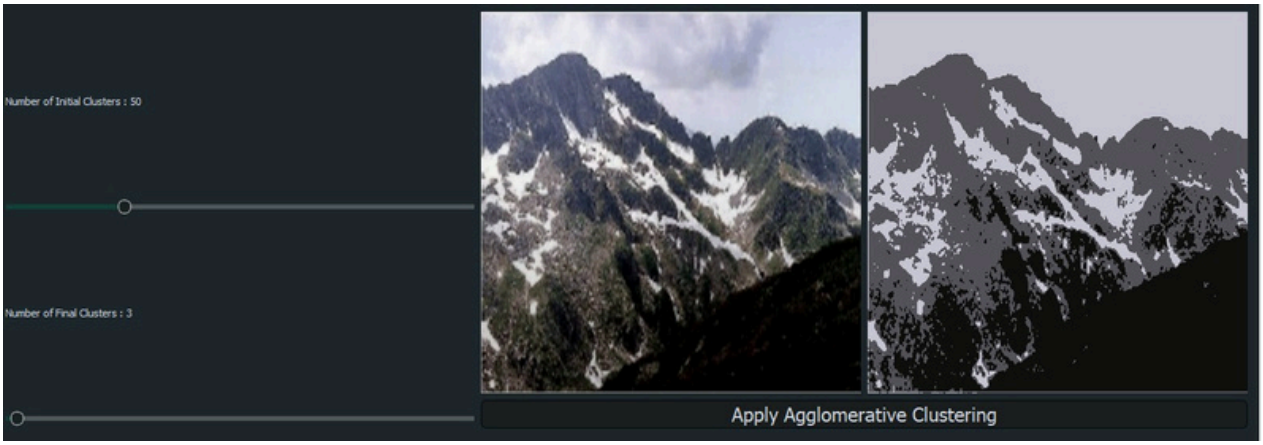
Parameters and Thresholding

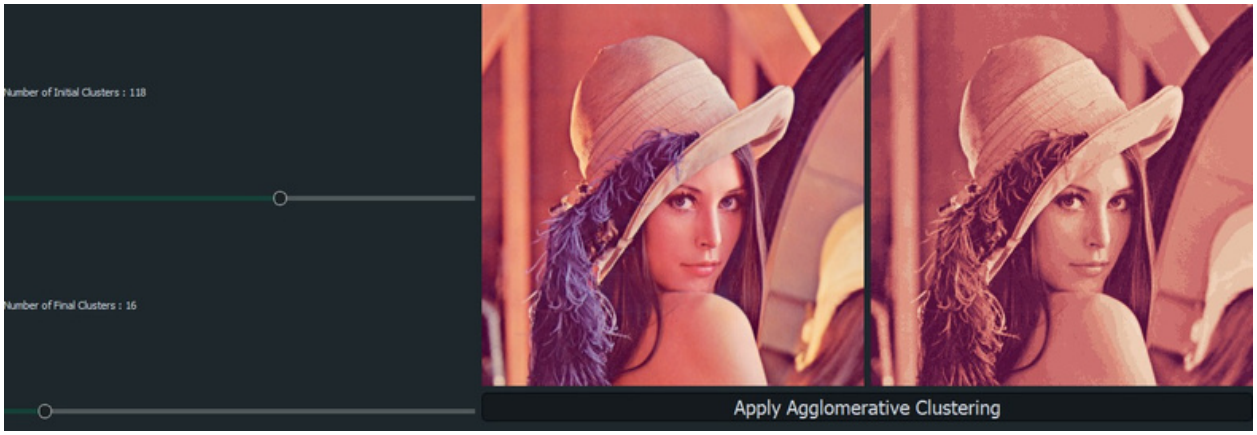
The Agglomerative Clustering algorithm accepts the following parameters:

- image: Input image for segmentation.
- clusters_number: The desired number of clusters to form.
- initial_clusters_number: The number of initial clusters used for clustering.

AGGLOMERATIVE CLUSTERING

Results





Results demonstrate the effectiveness of Agglomerative Clustering in segmenting images based on color similarity. The algorithm's performance may vary depending on the choice of parameters and the complexity of the input image.

RGB TO LUV

Introduction

The LUV color space, derived from CIELUV, offers a perceptually uniform color representation, making it particularly valuable in computer vision tasks such as segmentation. Unlike RGB, which lacks uniformity in terms of human perception, LUV separates luminance (brightness) from chrominance (color information), allowing for more accurate analysis of image features. In segmentation, where distinguishing objects based on color and brightness is crucial, LUV's uniformity facilitates better discrimination between foreground and background elements, leading to more precise segmentation results.

Algorithm Overview

The process of converting RGB to LUV involves several steps. Initially, the RGB image is rescaled to have pixel values within the range [0, 1]. Next, the RGB values are transformed into the XYZ color space using predefined coefficients in the matrix below. From XYZ, the luminance (L) and chrominance (U and V) components in the LUV color space are calculated. These components are determined based on formulas involving the XYZ values along with reference values for white point adaptation. Finally, the LUV components are normalized and adjusted to fit within the range of 0 to 255 to obtain the final LUV image.

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

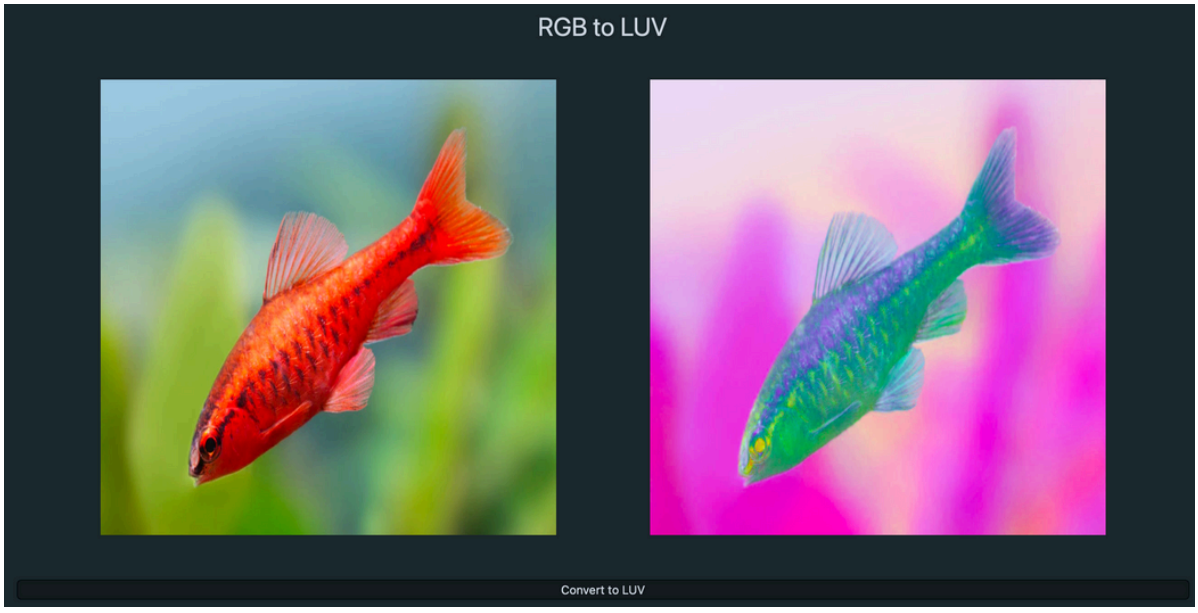
Function Procedure

1. Rescaling Pixel Values: Initially, the RGB image is rescaled so that pixel values fall within the range [0, 1]. This normalization step ensures consistency in the subsequent calculations, as many color spaces, including LUV, expect input values in this range.
2. Conversion to XYZ: The RGB values are then transformed into the XYZ color space using specific conversion coefficients. This conversion separates the color information from luminance, facilitating further processing.
3. Calculation of LUV Components: Once in the XYZ space, the algorithm computes the luminance (L) and chrominance (U and V) components. These calculations involve complex formulas derived from the CIELUV color space definition. Key to these computations are the reference white point values, which are constants representing the standard illuminant used for color calibration.
4. Normalization and Adjustment: After computing LUV components, they are normalized and adjusted to fit within the range of 0 to 255. This step ensures that the resulting LUV image can be properly displayed and processed within typical image processing pipelines, as most image formats and libraries expect pixel values to fall within this range.
5. Conversion to uint8: Finally, the LUV image is cast to the uint8 data type before being returned. This conversion ensures that the image is represented as 8-bit unsigned integers, which is a standard format for storing and processing images. By converting to uint8, the resulting LUV image can be easily visualized and manipulated using common image processing tools and libraries in Python.

RGB TO LUV

```python
def RGB_to_LUV(img):
    # Rescale pixel values to range [0, 1]
    img = img / 255.0

    # Convert RGB to XYZ
    r, g, b = img[:, :, 0], img[:, :, 1], img[:, :, 2] # split channels
    x = r * 0.4124564 + g * 0.3575761 + b * 0.1804375
    y = r * 0.2126729 + g * 0.7151522 + b * 0.0721750
    z = r * 0.0193339 + g * 0.1191920 + b * 0.9503041

    # Convert XYZ to LUV
    x_ref, y_ref, z_ref = 0.95047, 1.0, 1.08883

    u_prime = (4 * x) / (x + 15 * y + 3 * z)
    v_prime = (9 * y) / (x + 15 * y + 3 * z)

    u_ref_prime = (4 * x_ref) / (x_ref + 15 * y_ref + 3 * z_ref) #0.19793943
    v_ref_prime = (9 * y_ref) / (x_ref + 15 * y_ref + 3 * z_ref) #0.46831096

    L = np.where(y > 0.008856, 116 * np.power(y / y_ref, 1.0 / 3.0) - 16.0, 903.3 * y)
    u = 13 * L * (u_prime - u_ref_prime)
    v = 13 * L * (v_prime - v_ref_prime)

    L = (255.0 / (np.max(L) - np.min(L))) * (L - np.min(L))
    u = (255.0 / (np.max(u) - np.min(u))) * (u - np.min(u))
    v = (255.0 / (np.max(v) - np.min(v))) * (v - np.min(v))

    img_LUV = np.stack((L, u, v), axis=2)

    # Convert LUV to uint8 data type
    img_LUV = img_LUV.astype(np.uint8)

    return img_LUV
```

Results



K MEANS SEGMENTATION

Introduction

K-means segmentation is a widely-used method in image processing for automatically grouping pixels into clusters based on color similarity. By iteratively refining cluster centroids to minimize intra-cluster variance, the algorithm effectively partitions the image into visually coherent segments. Its simplicity and computational efficiency make it suitable for various applications, although its performance depends on factors like the number of clusters chosen and centroid initialization. Despite these considerations, k-means segmentation remains a fundamental technique in image analysis, serving as a basis for more advanced segmentation approaches.

Algorithm Overview

The process commences with the conversion of the input image into a manageable numpy array, reshaped into a 2D array where each row corresponds to a pixel and its color channels. This preprocessing step enables efficient handling and manipulation of pixel data.

Following initialization, k centroids are randomly seeded by selecting k unique indices from the flattened array of pixel values. These centroids serve as the initial cluster centers, around which pixels will be grouped during segmentation. Careful selection of initial centroids significantly impacts segmentation outcomes, influencing convergence and final cluster assignments.

Subsequently, pixels are assigned to the nearest centroid based on the Euclidean distance between their color values and the centroids. This assignment step yields a 1D array of labels, with each element denoting the index of the closest centroid for the corresponding pixel. Pixels with akin color characteristics coalesce, forming preliminary clusters.

Refinement of centroids ensues through iterative optimization aimed at minimizing the distance between pixels and their assigned centroids. Within each iteration, centroids are updated by computing the mean color values of the pixels assigned to each centroid. This iterative process continues until convergence or until a predefined maximum number of iterations is attained.

Convergence, pivotal for algorithm termination, is ascertained by evaluating changes in centroid positions against a specified threshold. If centroids exhibit minimal variation, indicative of stable cluster assignments, the algorithm halts prematurely; otherwise, it continues iterating until convergence or the maximum iteration limit is reached.

Upon convergence, each pixel in the image inherits the color of its corresponding centroid, effectively segmenting the image into visually cohesive clusters. This segmentation facilitates subsequent analysis and interpretation, enabling tasks such as object detection, image retrieval, and image compression.

K MEANS SEGMENTATION

```python
def kmeans_segmentation(image, k, max_iterations=100, threshold=1e-4):
    # Convert the image into a numpy array
    img = np.array(image)

    # Reshape the numpy array into a 2D array
    img_2d = img.reshape(-1, img.shape[2])

    # Initialize k centroids randomly
    centroids_idx = np.random.choice(img_2d.shape[0], k, replace=False)
    centroids = img_2d[centroids_idx]

    # Assign each pixel to the closest centroid
    labels = np.argmin(np.linalg.norm(img_2d[:, None] - centroids, axis=2), axis=1)
    # Repeat the following steps until convergence
    for _ in range(max_iterations):
        # print("iteration")

        # Update centroids
        for i in range(k):
            centroids[i] = np.mean(img_2d[labels == i], axis=0) # update the centroid with the mean of
the pixels in the cluster

        new_labels = np.argmin(np.linalg.norm(img_2d[:, None] - centroids, axis=2), axis=1) # assign
each pixel to the closest new centroid

        # Check convergence
        if np.sum(np.abs(centroids - np.array([np.mean(img_2d[new_labels == i], axis=0) for i in
range(k)]))) < threshold: #check new centroids against old centroids
            break
        labels = new_labels

        segmented_img = np.zeros(img.shape)
    new_labels = new_labels.reshape(img.shape[0], img.shape[1])
    for i in range(k):
        segmented_img[new_labels == i] = centroids[i]
    segmented_img = segmented_img.astype(np.uint8)

    return segmented_img
```

Function Procedure

1. Initialization of Centroids:

- The algorithm starts by converting the input image into a numpy array and reshaping it into a 2D array where each row represents a pixel and its color channels.
- K centroids are initialized randomly by selecting k unique indices from the flattened array of pixel values. These centroids serve as the initial cluster centers.

2. Assigning Pixels to Centroids:

- Each pixel in the image is assigned to the nearest centroid based on the Euclidean distance between the pixel's color values and the centroids.
- This assignment step creates a 1D array of labels, where each element represents the index of the closest centroid for the corresponding pixel.

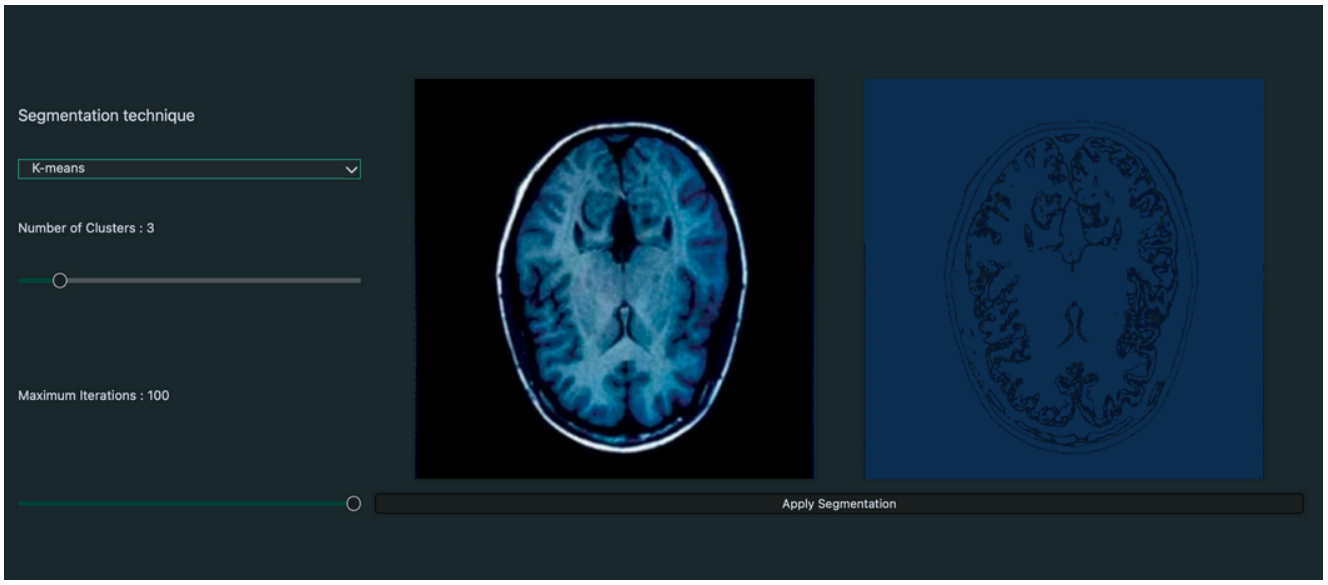
3. Iterative Refinement of Centroids:

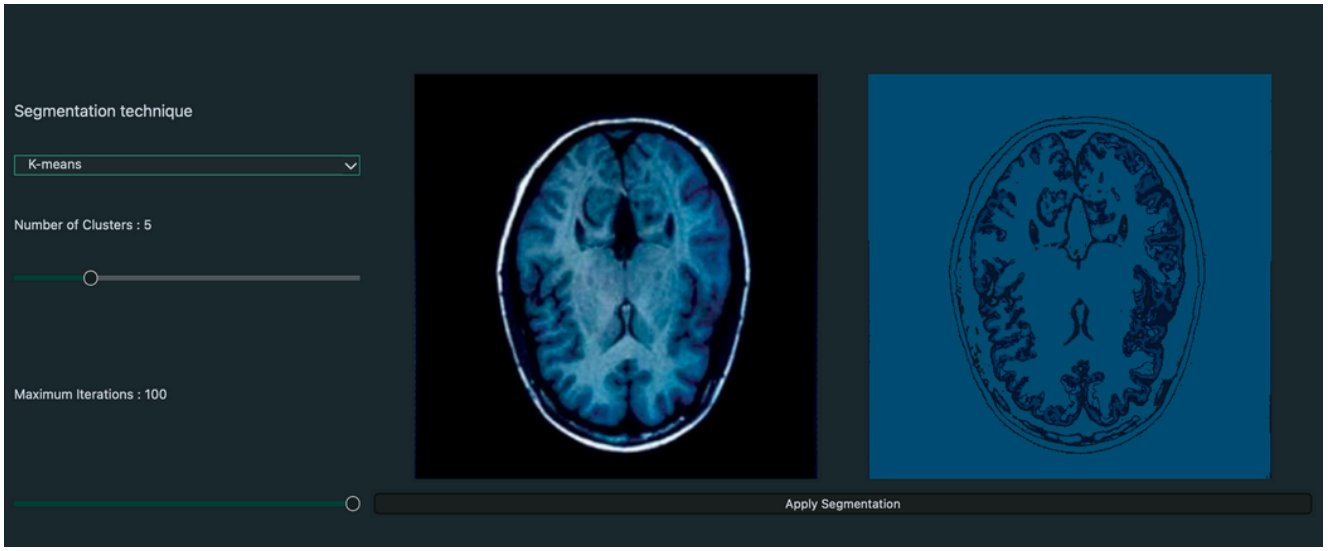
- The algorithm iteratively refines the centroids to minimize the distance between pixels and their assigned centroids.
- Within each iteration, it updates each centroid by computing the mean of the color values of the pixels assigned to that centroid.
- After updating the centroids, pixels are reassigned to the nearest centroids based on the updated centroid positions.

K MEANS SEGMENTATION

4. Convergence Check:
   - The iterative process continues until convergence or until the maximum number of iterations is reached.
   - Convergence is determined by checking if the change in centroid positions falls below a specified threshold. If the centroids stabilize, the algorithm terminates early.
5. Segmentation and Reconstruction:
   - Once convergence is achieved, each pixel in the image is assigned the color of its corresponding centroid, effectively segmenting the image into k clusters.
   - The segmented image is reconstructed by assigning each pixel the color of its centroid.
   - Finally, the segmented image's data type is converted to uint8 for proper display.

Results





MEAN SHIFT SEGMENTATION

Introduction

Mean shift clustering is a non-parametric technique employed in image processing for segmenting data points into distinct clusters based on their spatial density. This approach is particularly useful for image segmentation tasks where pixels with similar characteristics, such as color and texture, are grouped together. Unlike traditional clustering methods, mean shift clustering does not require a predefined number of clusters, making it adaptable to diverse datasets and scenarios.

Algorithm Overview

Mean shift clustering is a powerful technique for image segmentation that iteratively shifts data points towards local density peaks, effectively clustering similar points together. Here's a detailed breakdown of the algorithm's key steps:

1. Reshaping and Initialization:
    - Initially, the input image is reshaped into a 2D array, where each row represents a pixel and its color channels. This step simplifies subsequent computations.
    - Pointers are initialized to track visited points and to assign labels to clusters.
2. Window-based Search:
    - Mean shift operates within a local region defined by a window centered on each data point. The size of the window, typically determined empirically, influences the algorithm's performance.
    - Using a KD-tree data structure, the algorithm efficiently identifies points within the window for each data point. This step reduces computational complexity, particularly for large datasets.
3. Mean Calculation and Convergence:
    - For each data point, the mean of points within the window is computed, representing the new center. This mean shift operation effectively shifts the data point towards regions of higher density.
    - Convergence is determined by comparing the difference between the current and new centers. If the shift is below a predefined threshold, indicating convergence, the process halts for that data point.
4. Label Assignment and Reconstruction:
    - Once convergence is achieved for all data points, labels are assigned to points within each converged window, indicating their cluster membership.
    - The algorithm then reconstructs a new image where each pixel inherits the color of its corresponding cluster centroid. This step effectively segments the image into visually coherent regions, facilitating subsequent analysis and interpretation.

MEAN SHIFT SEGMENTATION

```python
def mean_shift(img, window_size=30, criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10,
1)):
    # Reshape the image into a 2D array of pixels
    img_to_2dArray = img.reshape(-1, 3)

    num_points, _ = img_to_2dArray.shape # Number of points in the image and the number of
dimensions(features)
    point_considered = np.zeros(num_points, dtype=bool)
    labels = -1 * np.ones(num_points, dtype=int) # Initialize the labels array
    label_count = 0
    tree = KDTree(img_to_2dArray)
    for i in range(num_points):
        if point_considered[i]: # Skip if the point has already been considered(visited)
            continue
        Center_point = img_to_2dArray[i] # Initialize the center point as the current point
        while True:
            # Find all points within the window centered on the current point
            # Use query_ball_tree for faster search
            in_window = tree.query_ball_point(Center_point, r=window_size) # Return an array of indices
of points within the window
            point_considered[in_window] = True #mark the points in the window as visited for faster
search
            # Calculate the mean of the points within the window
            new_center = np.mean(img_to_2dArray[in_window], axis=0) #assign the mean of the points in
the window as the new center
            # If the center has converged, assign labels to all points in the window
            if np.linalg.norm(new_center - Center_point) < criteria[1]:
                labels[in_window] = label_count
                label_count += 1 #update the label count(class label)
                break # Break out of the loop only if the center has converged
            Center_point = new_center # Update the center point to the new center if the center has not
converged
    # Reshape the labels array back to the original image shape
    labels = labels.reshape(img.shape[:2])
    new_img = np.zeros_like(img)
    for i in range(np.max(labels)+1):
        new_img[labels == i] = np.mean(img[labels == i], axis=0) # Calculate the mean of the points in
the cluster
    output_image = np.array(new_img, np.uint8)
    return output_image
```

Results