I. Kernel K-means

compute_distance(pos, m, img_length): Converts the flat index pos into two-dimensional coordinates (x, y).Iterates over each cluster center to compute the distance between pos and the center, updating the minimum distance found.

initialize_centers(img, img_size, K, METHOD, img_length):

Random: Selects **K** random indices from the image to serve as initial cluster centers.

K-means++: Selects the first center randomly, then uses a probability distribution based on the distance to existing centers to select subsequent centers.

Naive Sharding: Splits the image into **K** segments based on a sorted attribute and calculates the mean index and color for each segment to define the centers.

init(img, K, METHOD='random'): Calculates image dimensions and initializes cluster centers using initialize_centers. Initializes an array alpha to track which cluster each pixel belongs to.

```python
def compute_distance(pos, m, img_length):
    x, y = pos % img_length, pos // img_length
    min_distance = float('inf')
    for center in m:
        center_x, center_y = center[0] % img_length, center[0] // img_length
        distance = (x - center_x) ** 2 + (y - center_y) ** 2
        if distance < min_distance:
            min_distance = distance
    return min_distance

def initialize_centers(img, img_size, K, METHOD, img_length):
    centers = []
    if METHOD == "random":
        indices = np.random.choice(img_size, K, replace=False)
        centers = [[index, img[index]] for index in indices]

    elif METHOD == "kmeans++":
        first_center = np.random.randint(img_size)
        centers = [[first_center, img[first_center]]]
        for _ in range(1, K):
            distances = np.array([compute_distance(n, centers, img_length) for n in range(img_size)])
            probabilities = distances / distances.sum()
            cumulative_probabilities = np.cumsum(probabilities)
            random_value = np.random.rand()
            next_center = np.searchsorted(cumulative_probabilities, random_value)
            centers.append([next_center, img[next_center]])

    elif METHOD == "naive_sharding":
        attributes = np.array([[n] + list(img[n]) for n in range(img_size)])
        attributes_sorted = attributes[np.argsort(attributes[:, 1])]
        slice_size = img_size // K
        for i in range(K):
            slice_indices = attributes_sorted[i * slice_size:(i + 1) * slice_size, 0]
            mean_index = int(np.mean(slice_indices))
            mean_color = np.mean(attributes_sorted[i * slice_size:(i + 1) * slice_size, 2:], axis=0).astype(np.uint8)
            centers.append([mean_index, mean_color])

    return centers
```

```python
def init(img, K, METHOD='random'):
    img_length = int(np.sqrt(len(img)))
    img_size = len(img)
    centers = initialize_centers(img, img_size, K, METHOD, img_length)
    alpha = np.zeros((img_size, K), dtype=np.uint8)
    for i, center in enumerate(centers):
        alpha[center[0], i] = 1
    c = updateCk(alpha)
    return centers, c, alpha
```

Compute the Kernel Matrix:

Use a double loop to traverse every pair of pixels in the image, calculating only the upper triangle (including the diagonal) because the kernel matrix is symmetric.

For each pair of pixels **(p, q)**, calculate the kernel value between **p** and **q**, and store this value in both **kernel[p][q]** and **kernel[q][p]**.

spatial_similarity(p1, p2): Computes the square of the Euclidean distance between two pixel positions **p1** and **p2**. It uses the image length **img_length** to convert one-dimensional indices into two-dimensional coordinates.

color_similarity(c1, c2): Computes the square of the Euclidean distance between the color values **c1** and **c2** of two pixels. The color values are represented in RGB format, and here the color values are converted to **uint32** type for computation.

kernel_function(x1, x2): Combines spatial and color similarities to calculate the kernel value. The kernel value is the product of two exponential functions, whose exponents are the negative spatial and color similarity measures, multiplied by the corresponding hyperparameters **gamma_s** = **1 / img_size** and **gamma_c = 1 / (256 * 256)**.

$$k(x, x') = e^{-\gamma_s \|S(x)-S(x')\|^2} \times e^{-\gamma_c \|C(x)-C(x')\|^2}$$

```python
def compute_kernel(img):
    kernel = np.zeros((img_size, img_size))

    def spatial_similarity(p1, p2):
        return ((p1 // img_length - p2 // img_length) ** 2 +
                (p1 % img_length - p2 % img_length) ** 2)

    def color_similarity(c1, c2):
        return np.sum((np.array(c1, dtype=np.uint32) - np.array(c2, dtype=np.uint32)) ** 2)

    def kernel_function(x1, x2):
        spatial_sim = np.exp(-gamma_s * spatial_similarity(x1[0], x2[0]))
        color_sim = np.exp(-gamma_c * color_similarity(x1[1], x2[1]))
        return spatial_sim * color_sim


    for p in trange(img_size):
        for q in range(p, img_size):
            kernel[p][q] = kernel[q][p] = kernel_function([p, img[p]], [q, img[q]])

    return kernel
```

The function **distance** computes the distance for each cluster and each point in the image.

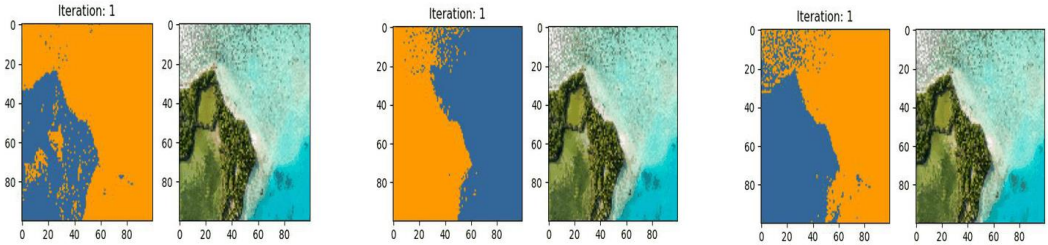
$$
\begin{aligned}
&||\phi(x_j) - \mu_k^{\phi}|| \\
&= ||\phi(x_j) - \frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)|| \\
&= k(x_j, x_j) - \frac{2}{|C_k|}\sum_{n}\alpha_{kn}K(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q}\alpha_{kp}\alpha_{kq}k(x_p, x_q)
\end{aligned}
$$

```python
def distance(c, alpha):
    dist = np.ones((K, img_size))

    for k in range(K):
        cluster_mask = alpha[:, k].reshape(-1, 1)

        second_term = (2 / c[k]) * (cluster_mask.T @ kernel).flatten()

        third_term = (1 / c[k] ** 2) * np.sum(cluster_mask.T @ kernel @ cluster_mask)

        dist[k] -= second_term
        dist[k] += third_term

    return dist
```

Result

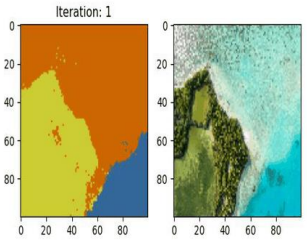
image1.png

2-cluster

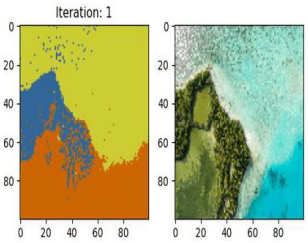


Random *Kmeans++* *naïve sharding*

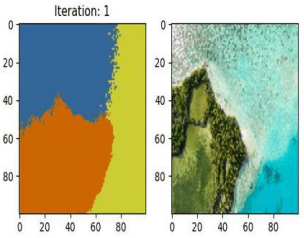
3-cluster



Random

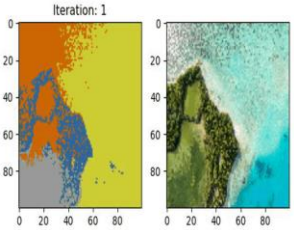


Kmeans++

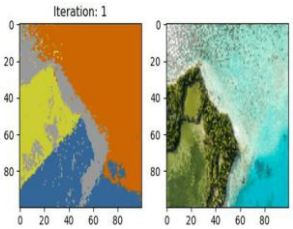


naïve sharding

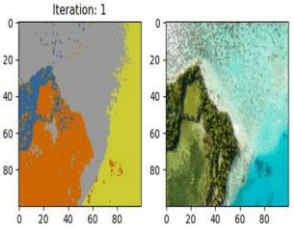
4-cluster



Random



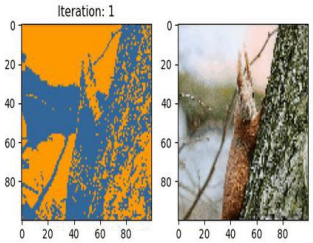
Kmeans++



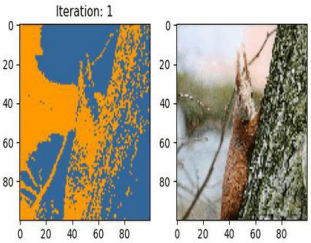
naïve sharding

image2.png

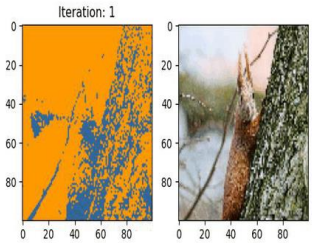
2-cluster



Random

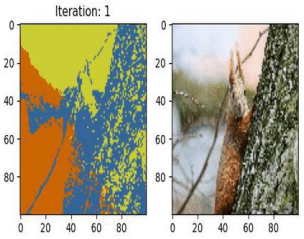


Kmeans++

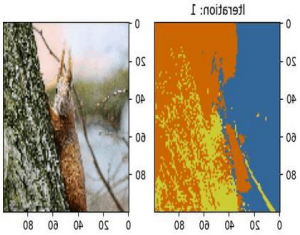


naïve sharding

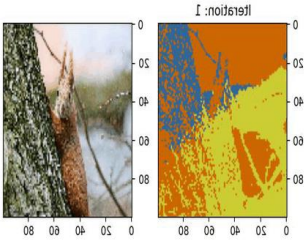
3-cluster



Random

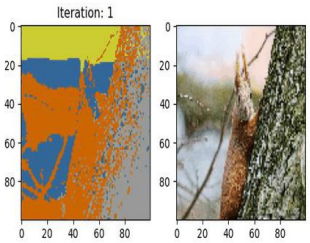


Kmeans++

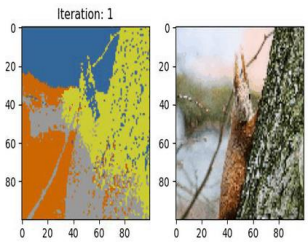


naïve sharding

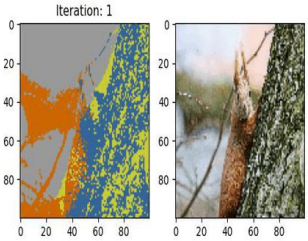
4-cluster



Random



Kmeans++



naïve sharding

II. Spectral Clustering

compute_laplacian_and_eigen(W, normalized=True): Computes the Laplacian matrix and performs eigen decomposition to facilitate clustering in the spectral clustering algorithm.

Normalized

$$L = I - D^{-1/2} W D^{-1/2}$$

Unnormalized(ratio cut)

$$L = D - W$$

then calculates eigenvalues and eigenvectors of the Laplacian matrix, returns the first K+1 smallest eigenvectors.

```python
if __name__ == "__main__":
    img = readImg(f'image{IMAGE}.png')

    W = compute_kernel(img)

    U = compute_laplacian_and_eigen(W, normalized=(CUT == "normalized"))

    kMeans(U, img)
```

```python
def compute_laplacian_and_eigen(W, normalized=True):
    D = np.diag(np.sum(W, axis=1))
    if normalized:
        with np.errstate(divide='ignore', invalid='ignore'):
            D_inv_sqrt = np.diag(1 / np.sqrt(np.diag(D)))
            D_inv_sqrt[np.isinf(D_inv_sqrt)] = 0
        L = np.identity(len(W)) - D_inv_sqrt @ W @ D_inv_sqrt
    else:  # "ratio"
        L = D - W
    eigenvalues, eigenvectors = np.linalg.eigh(L)
    return eigenvectors[:, 1:K+1]
```

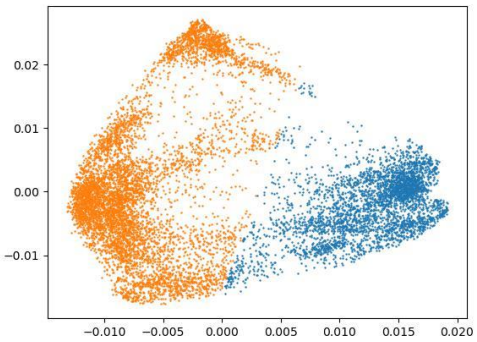
Initialization for k-means clustering is like above.

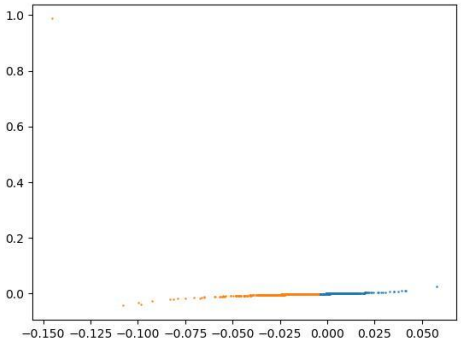
```python
def min_distance(means, point):
    distances = np.sum((means - point)**2, axis=1)
    return np.min(distances)

def initKMeans(U):
    img_size, features = U.shape
    m = []
    cluster = np.full(img_size, -1, dtype=np.uint32)

    if METHOD == "random":
        indices = np.random.choice(img_size, K, replace=False)
        m = U[indices].tolist()  # Store initial centers

    elif METHOD == "kmeans++":
        first_index = np.random.randint(img_size)
        m = [U[first_index]]
        for _ in range(1, K):
            distances = np.array([min_distance(m, U[n]) for n in range(img_size)])
            probabilities = distances / np.sum(distances)
            cumulative_probabilities = np.cumsum(probabilities)
            next_index = np.searchsorted(cumulative_probabilities, np.random.rand())
            m.append(U[next_index])

    elif METHOD == "naive_sharding":
        sorted_indices = np.argsort(U[:, 0])
        slice_size = img_size // K
        for i in range(K):
            slice_indices = sorted_indices[i * slice_size:(i + 1) * slice_size]
            m.append(np.mean(U[slice_indices], axis=0))  # Compute mean of each slice

    return m, cluster
```

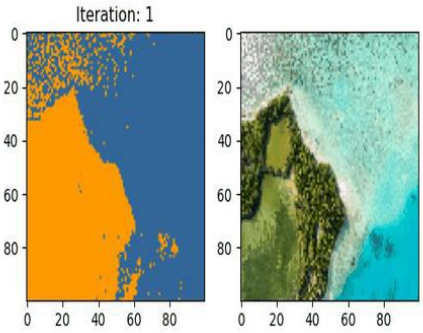
Result

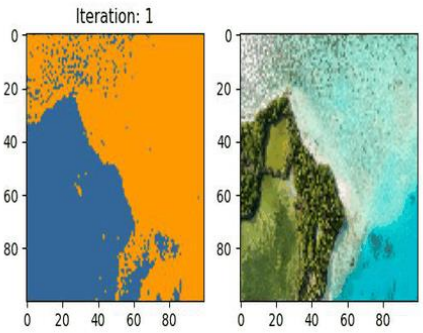
image1.png

2-cluster-random



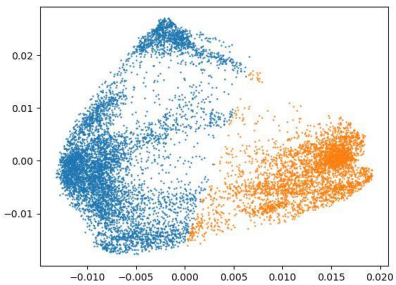


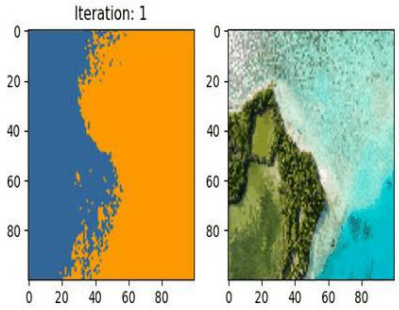




normalized *ratio*

2-cluster-kmeans++





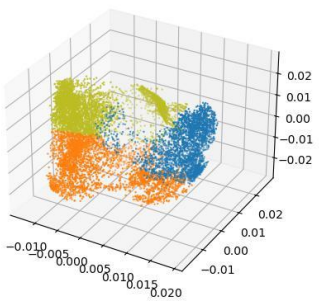
normalized

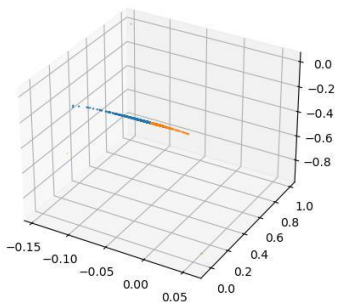
Spectral Clustering: As shown in the figure above, the spectral clustering results using "2-cluster-random" clearly display two distinct clusters. In the eigenspace, the data points of these two clusters show significant separation.

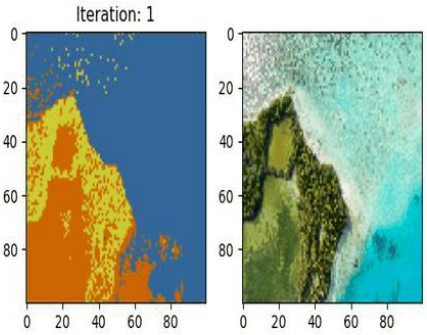
Normalized Method: The clusters in this method show a tighter and more uniform clustering effect, indicating that normalization helps enhance the internal consistency of the clusters.

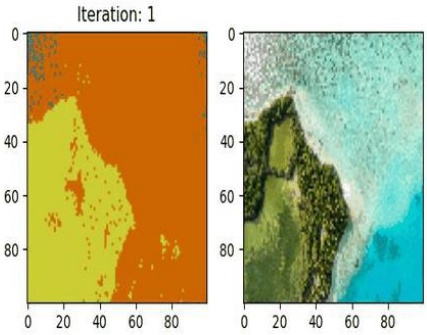
Ratio Cut Method: Although this method also effectively separates different clusters, the degree of separation between clusters is slightly lower than that of the normalized method.

3-cluster-random



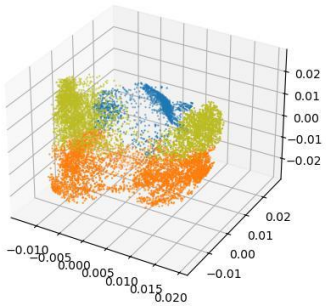


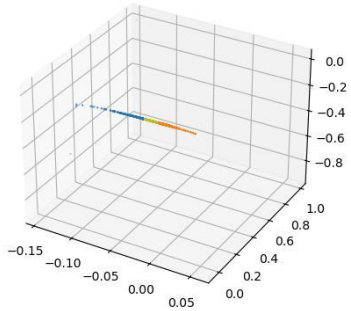


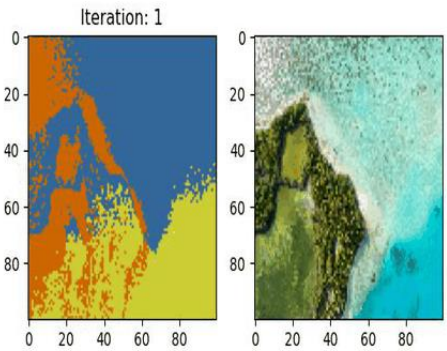


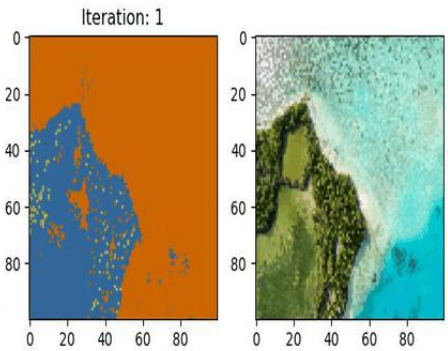
normalized *ratio*

3-cluster-kmeans++









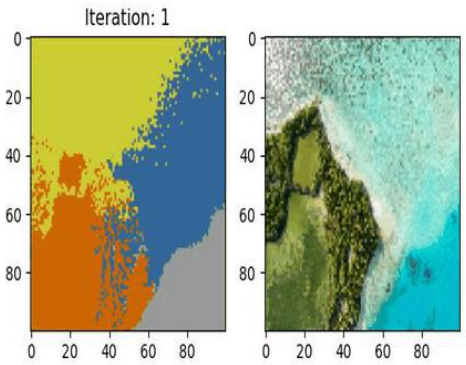
normalized *ratio*

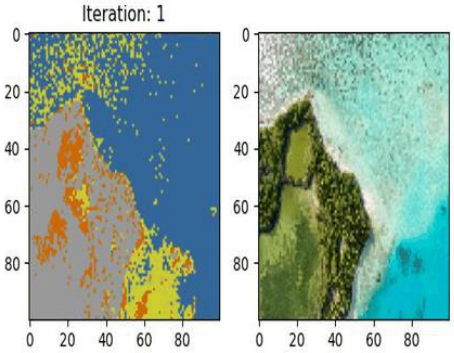
Spatial Distribution in Eigenspace: In both the normalized and ratio cut methods, clusters appear distinct in the eigenspace plots. Above plots demonstrate how data points are grouped together, implying that points within the same cluster tend to share similar coordinates in the eigenspace.

Normalized vs. Ratio Cut Method: The normalized method tends to produce tighter clusters with clearer boundaries, suggesting that this approach might be more effective at ensuring that data points within the same cluster share similar eigenspace coordinates.

The ratio cut method, while effective at clustering, shows a slightly less compact clustering, which might indicate some variations in the coordinates within the same cluster in the eigenspace.

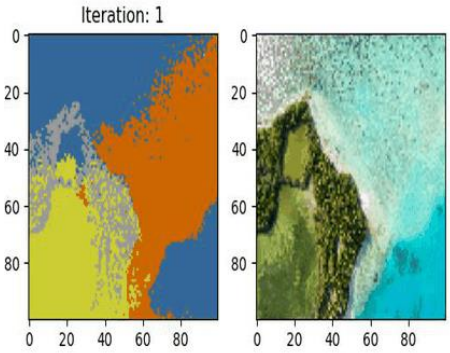
4-cluster-random

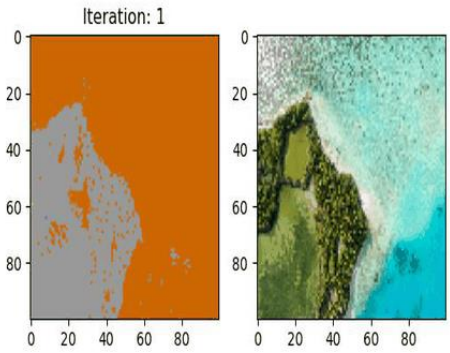




normalized *ratio*

4-cluster-kmeans++

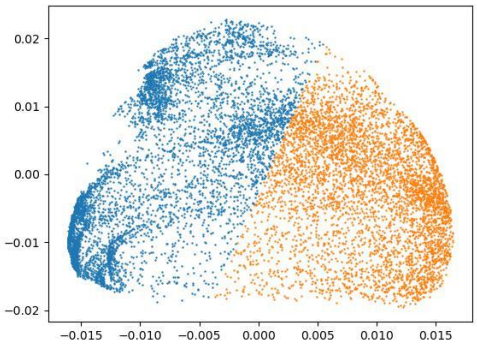


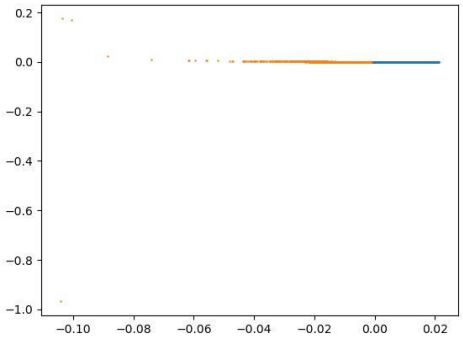


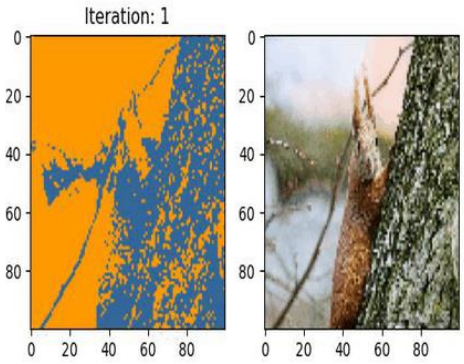
normalized *ratio*

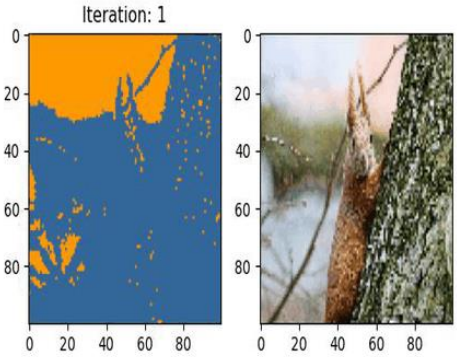
image2.png

2-cluster-random



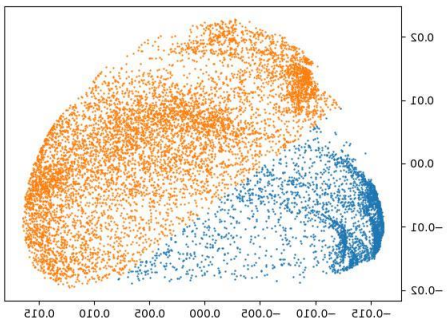


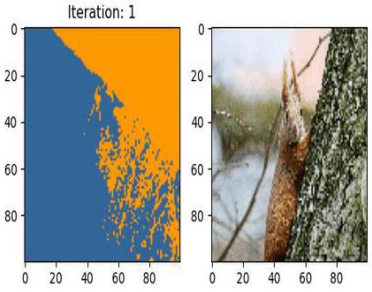




normalized *ratio*

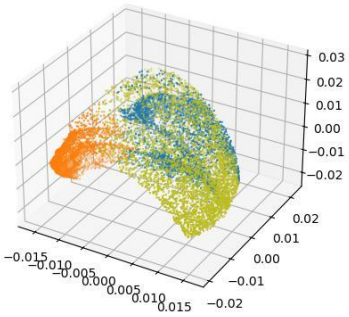
2-cluster-kmeans

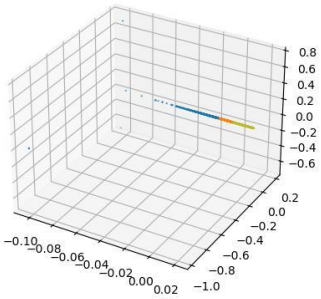


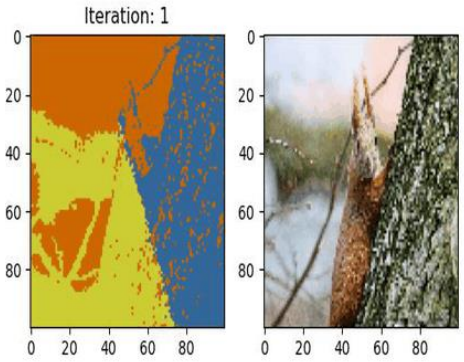


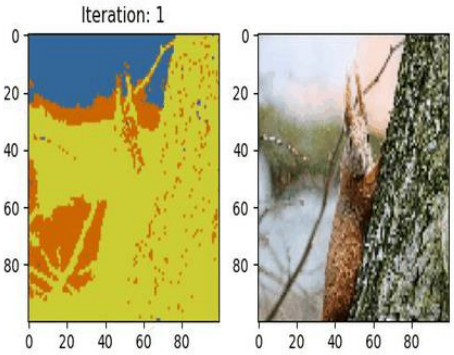
normalized

3-cluster-random



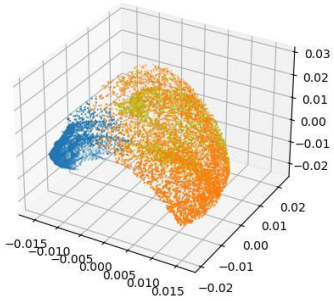


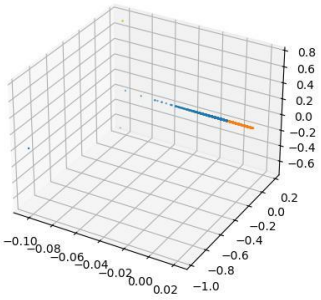


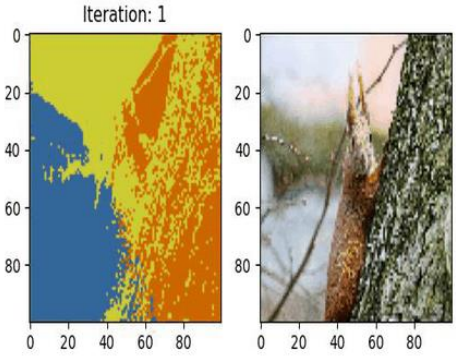


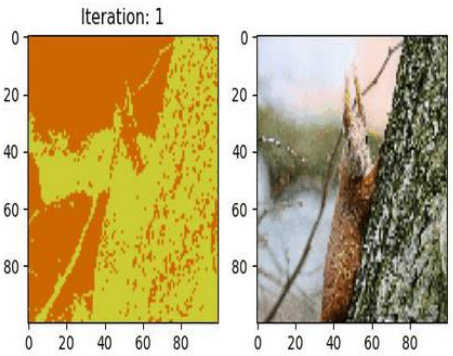
normalized *ratio*

3-cluster-kmeans



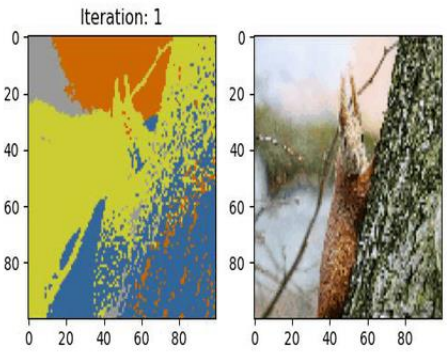


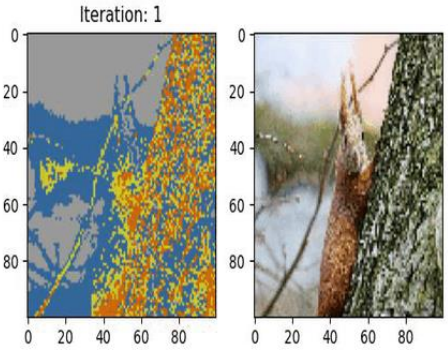




normalized *ratio*

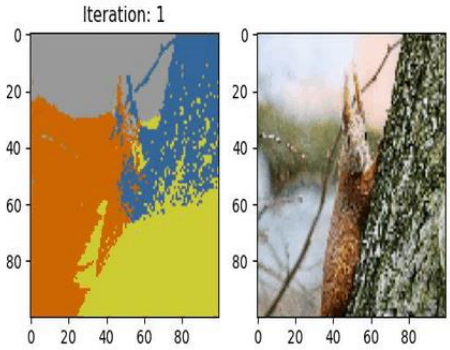
4-cluster-random

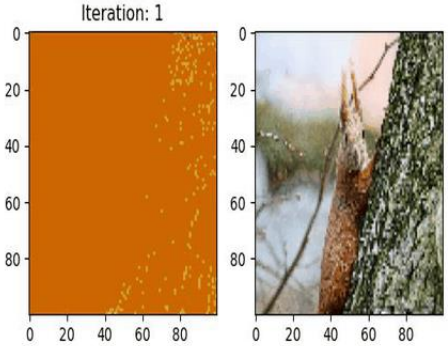




normalized *ratio*

4-cluster-kmeans





normalized *ratio*