ML HW6

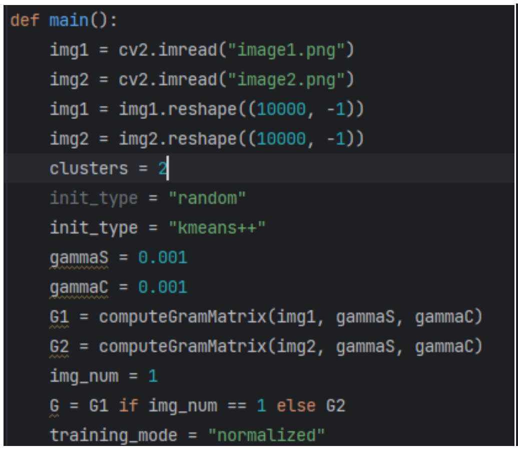
1. Code with detailed explanations (30%)

Part 1 & 2 & 3 & 4:

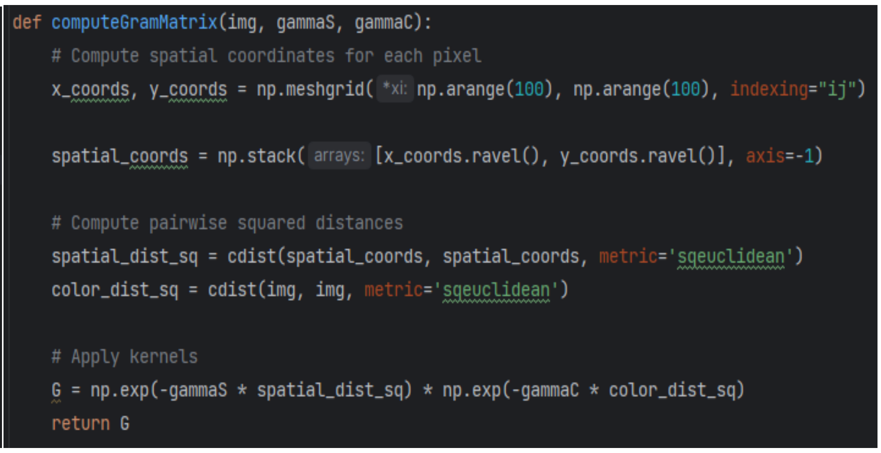
Initialization:

In main function, we choose the image that we want to do the clustering, clustering method and corresponding parameters of the kernel and number of clustering **(Part 2)** and then we compute the Gram matrix by using computeGramMatrix function. The kernel is defined as follows:

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



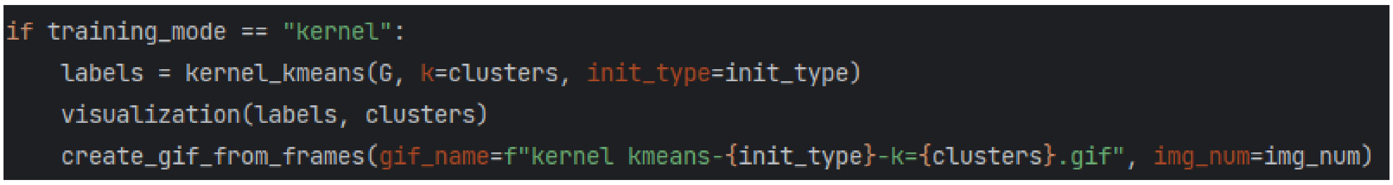
```python
def main():
    img1 = cv2.imread("image1.png")
    img2 = cv2.imread("image2.png")
    img1 = img1.reshape((10000, -1))
    img2 = img2.reshape((10000, -1))
    clusters = 2
    init_type = "random"
    init_type = "kmeans++"
    gammaS = 0.001
    gammaC = 0.001
    G1 = computeGramMatrix(img1, gammaS, gammaC)
    G2 = computeGramMatrix(img2, gammaS, gammaC)
    img_num = 1
    G = G1 if img_num == 1 else G2
    training_mode = "normalized"
```



```python
def computeGramMatrix(img, gammaS, gammaC):
    # Compute spatial coordinates for each pixel
    x_coords, y_coords = np.meshgrid(np.arange(100), np.arange(100), indexing="ij")

    spatial_coords = np.stack([x_coords.ravel(), y_coords.ravel()], axis=-1)

    # Compute pairwise squared distances
    spatial_dist_sq = cdist(spatial_coords, spatial_coords, metric='sqeuclidean')
    color_dist_sq = cdist(img, img, metric='sqeuclidean')

    # Apply kernels
    G = np.exp(-gammaS * spatial_dist_sq) * np.exp(-gammaC * color_dist_sq)
    return G
```

Kernel kmeans:

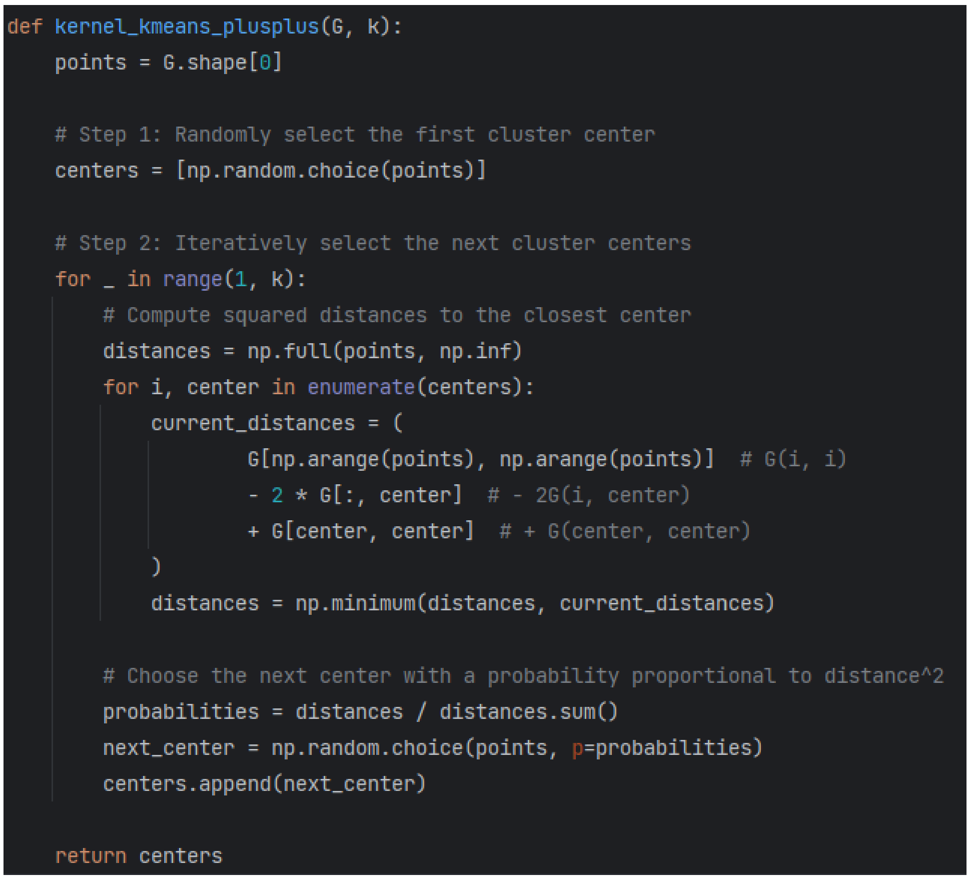
In main function:



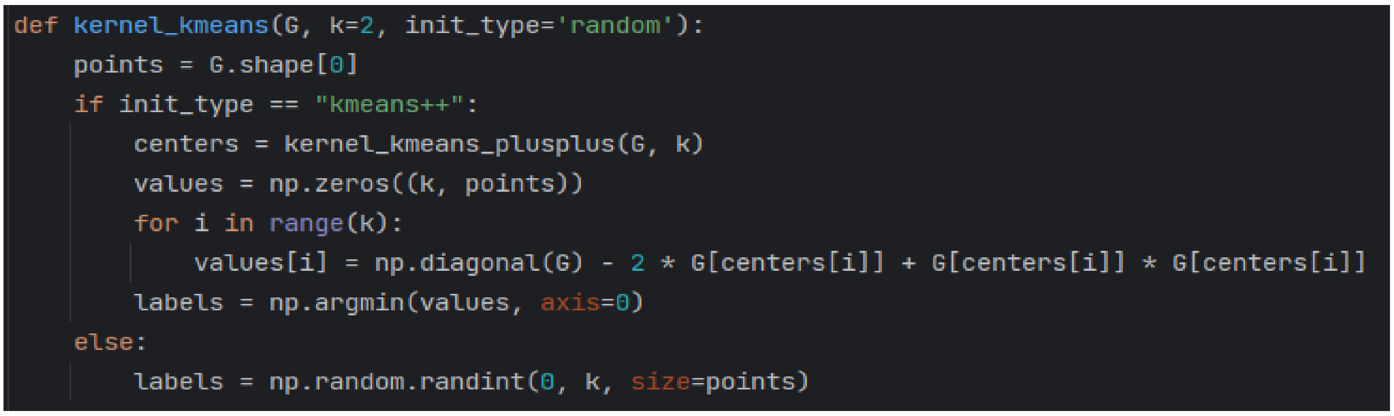
```python
if training_mode == "kernel":
    labels = kernel_kmeans(G, k=clusters, init_type=init_type)
    visualization(labels, clusters)
    create_gif_from_frames(gif_name=f"kernel kmeans-{init_type}-k={clusters}.gif", img_num=img_num)
```

a. Initialization method and give the corresponding label to the datapoint

(Part3) Based on the init_type, the initialization method is selected. If init_type is set to "kmeans++", the algorithm uses the k-means++ method to determine the k cluster centers and assigns labels to data points based on their closest center. Otherwise, data points are randomly assigned to k clusters.



```python
def kernel_kmeans_plusplus(G, k):
    points = G.shape[0]

    # Step 1: Randomly select the first cluster center
    centers = [np.random.choice(points)]

    # Step 2: Iteratively select the next cluster centers
    for _ in range(1, k):
        # Compute squared distances to the closest center
        distances = np.full(points, np.inf)
        for i, center in enumerate(centers):
            current_distances = (
                    G[np.arange(points), np.arange(points)]  # G(i, i)
                    - 2 * G[:, center]  # - 2G(i, center)
                    + G[center, center]  # + G(center, center)
            )
            distances = np.minimum(distances, current_distances)

        # Choose the next center with a probability proportional to distance^2
        probabilities = distances / distances.sum()
        next_center = np.random.choice(points, p=probabilities)
        centers.append(next_center)

    return centers
```



```python
def kernel_kmeans(G, k=2, init_type='random'):
    points = G.shape[0]
    if init_type == "kmeans++":
        centers = kernel_kmeans_plusplus(G, k)
        values = np.zeros((k, points))
        for i in range(k):
            values[i] = np.diagonal(G) - 2 * G[centers[i]] + G[centers[i]] * G[centers[i]]
        labels = np.argmin(values, axis=0)
    else:
        labels = np.random.randint(0, k, size=points)
```

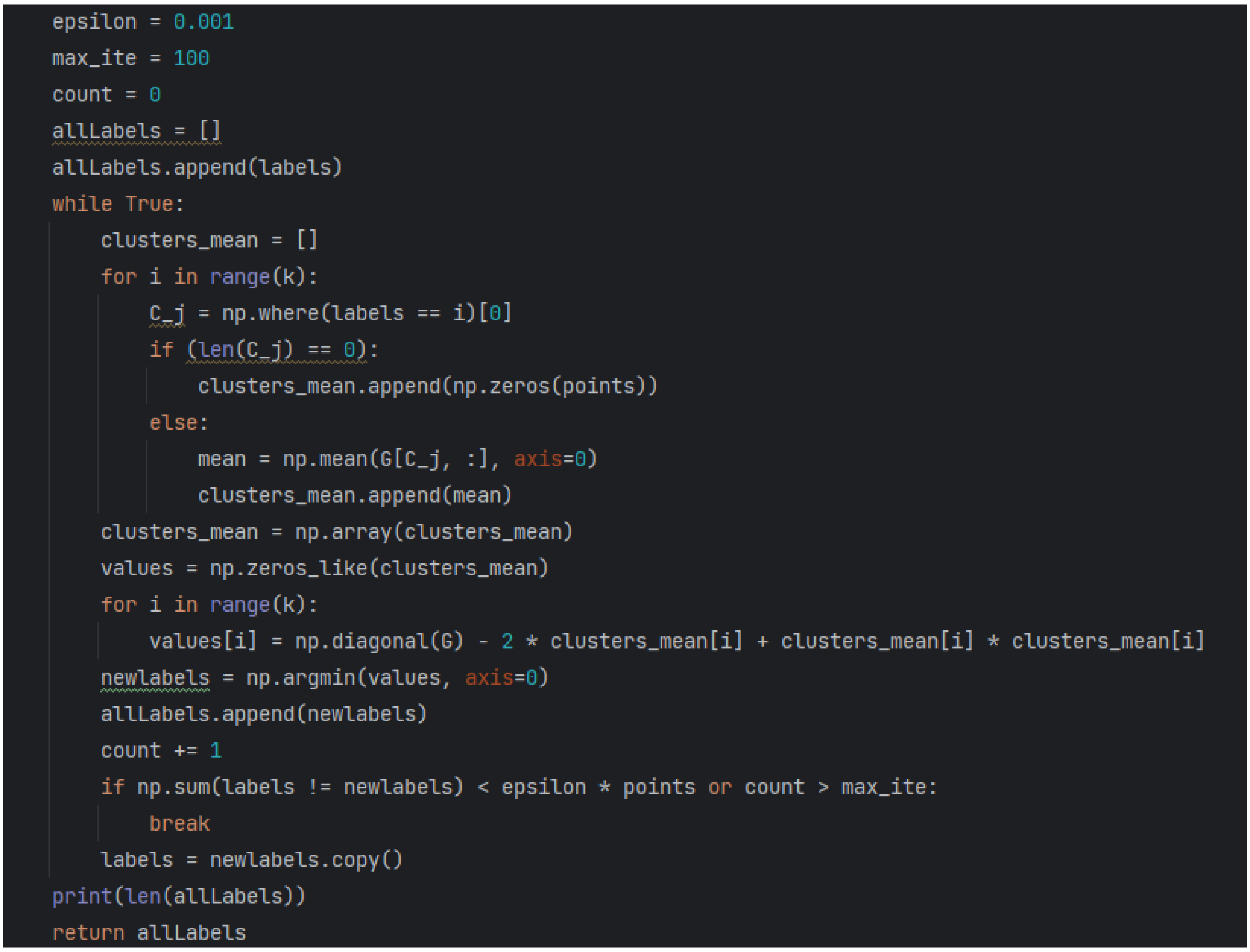
b. kmeans clustering

In each iteration, I calculate the mean of the data points within the same label group to serve as the center of that cluster. For each data point, I compute its distance to these centers and assign it the label of the closest center using the following formula:

$$\mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_p\sum_q \alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$

If the number of label changes between iterations is small enough, the algorithm is considered to have converged, and the loop is terminated.

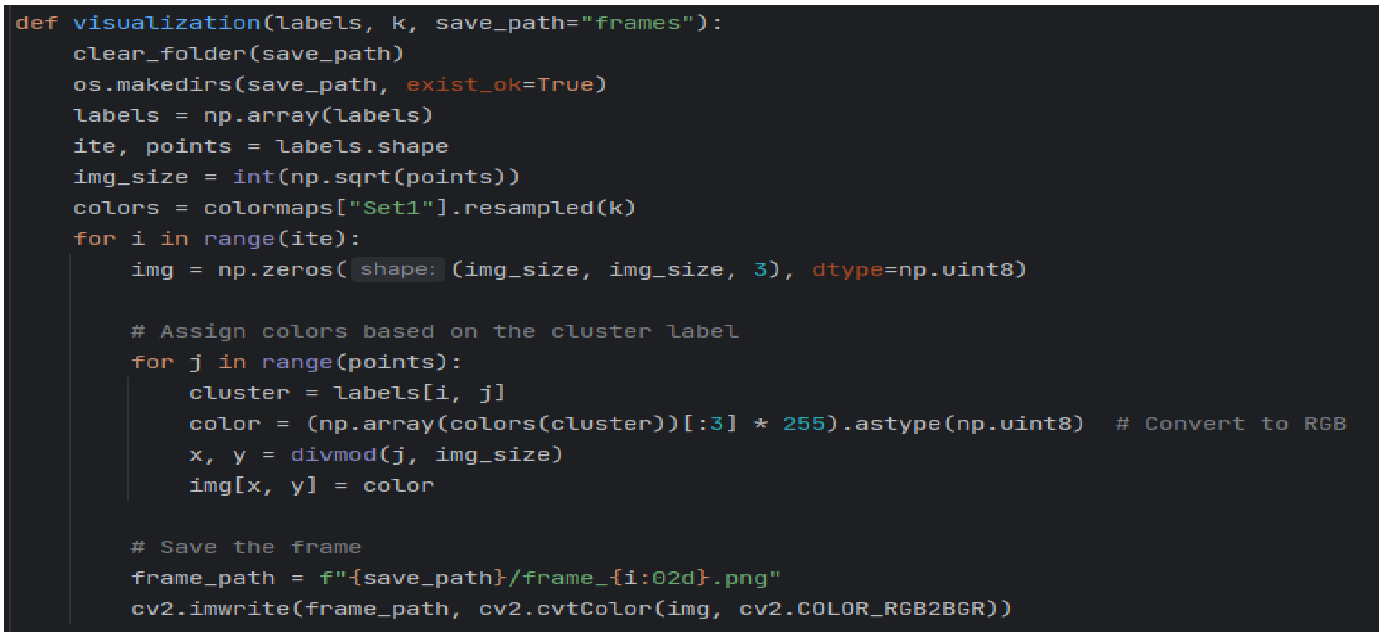
In kernel_kmeans:



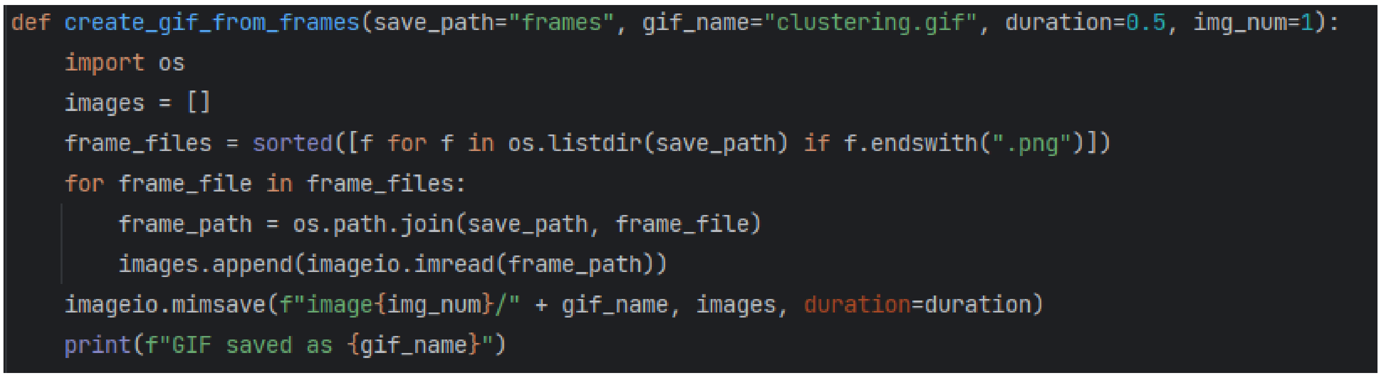
```
    epsilon = 0.001
    max_ite = 100
    count = 0
    allLabels = []
    allLabels.append(labels)
    while True:
        clusters_mean = []
        for i in range(k):
            C_j = np.where(labels == i)[0]
            if (len(C_j) == 0):
                clusters_mean.append(np.zeros(points))
            else:
                mean = np.mean(G[C_j, :], axis=0)
                clusters_mean.append(mean)
        clusters_mean = np.array(clusters_mean)
        values = np.zeros_like(clusters_mean)
        for i in range(k):
            values[i] = np.diagonal(G) - 2 * clusters_mean[i] + clusters_mean[i] * clusters_mean[i]
        newlabels = np.argmin(values, axis=0)
        allLabels.append(newlabels)
        count += 1
        if np.sum(labels != newlabels) < epsilon * points or count > max_ite:
            break
        labels = newlabels.copy()
    print(len(allLabels))
    return allLabels
```

c. visualization

After having each iteration labels, I use the labels to create the image of each iteration and save as gif



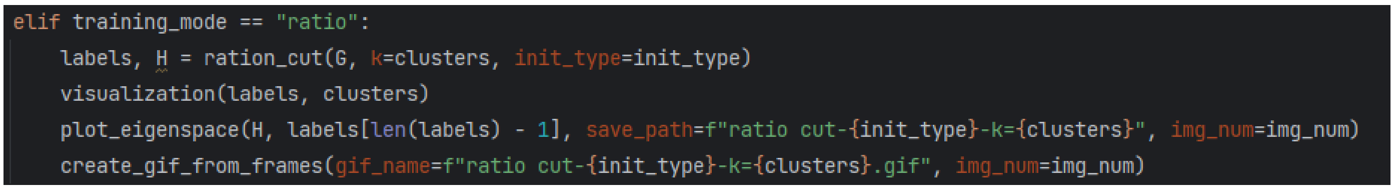
```
def visualization(labels, k, save_path="frames"):
    clear_folder(save_path)
    os.makedirs(save_path, exist_ok=True)
    labels = np.array(labels)
    ite, points = labels.shape
    img_size = int(np.sqrt(points))
    colors = colormaps["Set1"].resampled(k)
    for i in range(ite):
        img = np.zeros( shape: (img_size, img_size, 3), dtype=np.uint8)

        # Assign colors based on the cluster label
        for j in range(points):
            cluster = labels[i, j]
            color = (np.array(colors(cluster))[:3] * 255).astype(np.uint8)  # Convert to RGB
            x, y = divmod(j, img_size)
            img[x, y] = color

        # Save the frame
        frame_path = f"{save_path}/frame_{i:02d}.png"
        cv2.imwrite(frame_path, cv2.cvtColor(img, cv2.COLOR_RGB2BGR))
```



```python
def create_gif_from_frames(save_path="frames", gif_name="clustering.gif", duration=0.5, img_num=1):
    import os
    images = []
    frame_files = sorted([f for f in os.listdir(save_path) if f.endswith(".png")])
    for frame_file in frame_files:
        frame_path = os.path.join(save_path, frame_file)
        images.append(imageio.imread(frame_path))
    imageio.mimsave(f"image{img_num}/" + gif_name, images, duration=duration)
    print(f"GIF saved as {gif_name}")
```

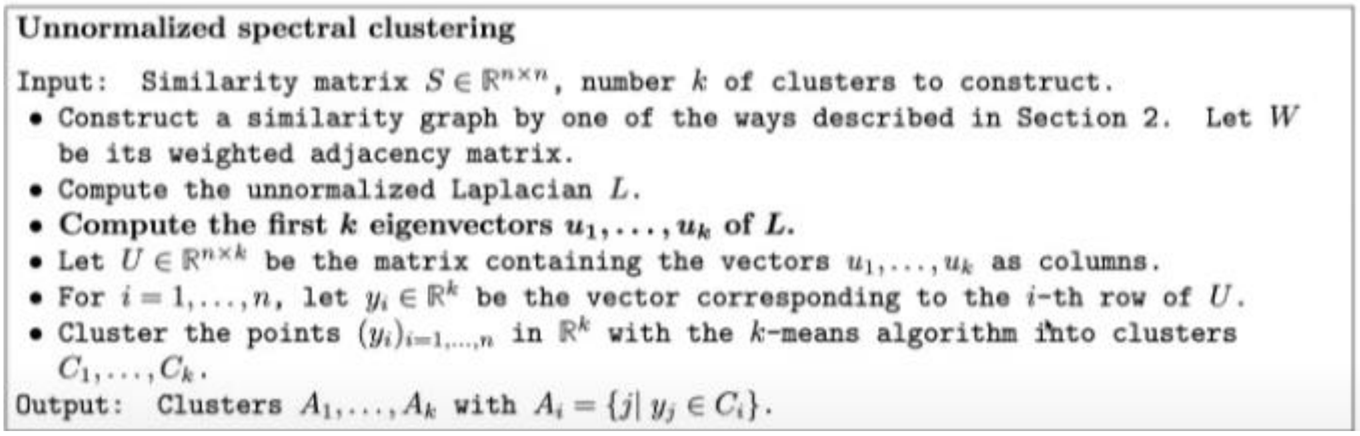
Ratio cut:

In main function:



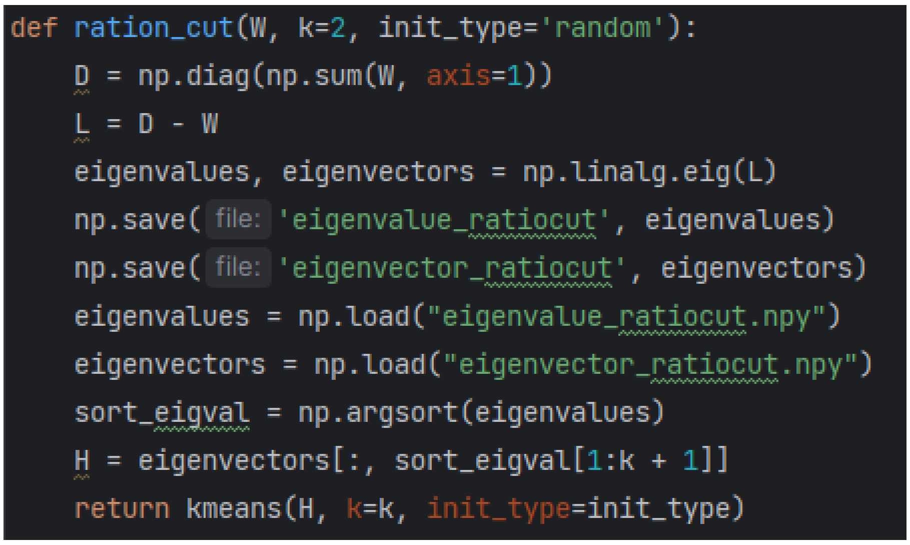
```python
elif training_mode == "ratio":
    labels, H = ration_cut(G, k=clusters, init_type=init_type)
    visualization(labels, clusters)
    plot_eigenspace(H, labels[len(labels) - 1], save_path=f"ratio cut-{init_type}-k={clusters}", img_num=img_num)
    create_gif_from_frames(gif_name=f"ratio cut-{init_type}-k={clusters}.gif", img_num=img_num)
```

The process is similar to unnormalized spectral clustering which is described below,



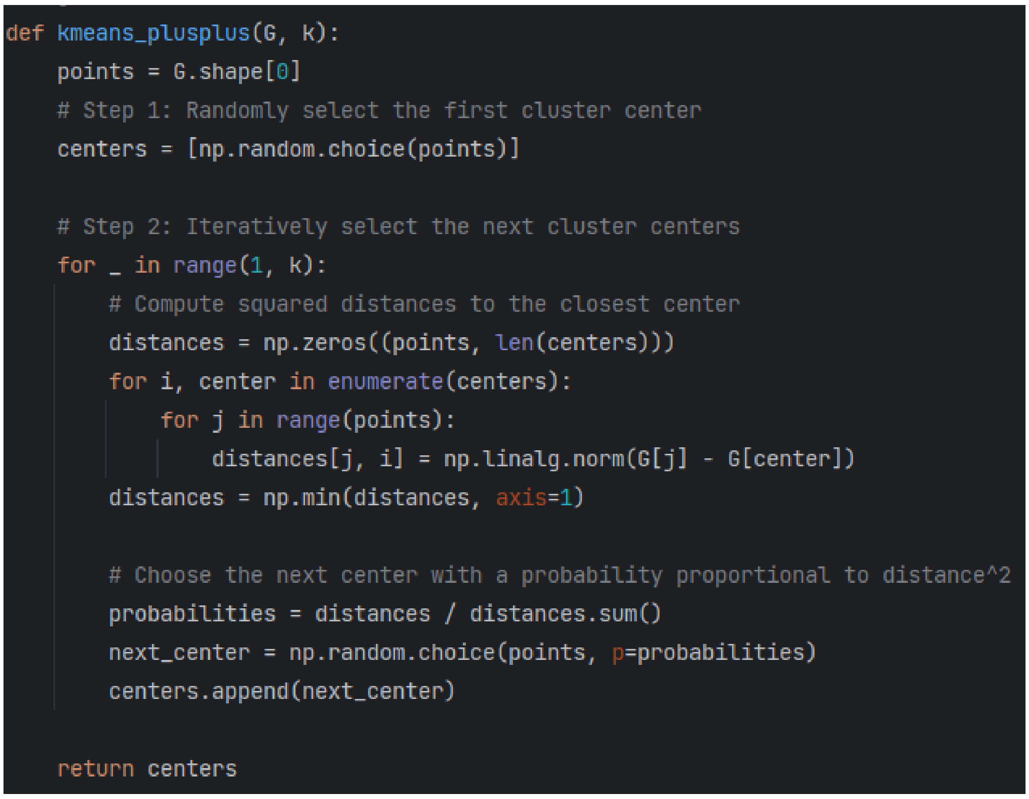
```
Unnormalized spectral clustering

Input: Similarity matrix $S \in \mathbb{R}^{n\times n}$, number $k$ of clusters to construct.
  • Construct a similarity graph by one of the ways described in Section 2. Let $W$
    be its weighted adjacency matrix.
  • Compute the unnormalized Laplacian $L$.
  • Compute the first $k$ eigenvectors $u_1, \dots, u_k$ of $L$.
  • Let $U \in \mathbb{R}^{n\times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ as columns.
  • For $i = 1, \dots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the $i$-th row of $U$.
  • Cluster the points $(y_i)_{i=1,\dots,n}$ in $\mathbb{R}^k$ with the $k$-means algorithm into clusters
    $C_1, \dots, C_k$.
Output: Clusters $A_1, \dots, A_k$ with $A_i = \{j \mid y_j \in C_i\}$.
```

Calculating L by using L=D-W, where D is diagonal matrix where d_{ii} = summation of ith row of W.

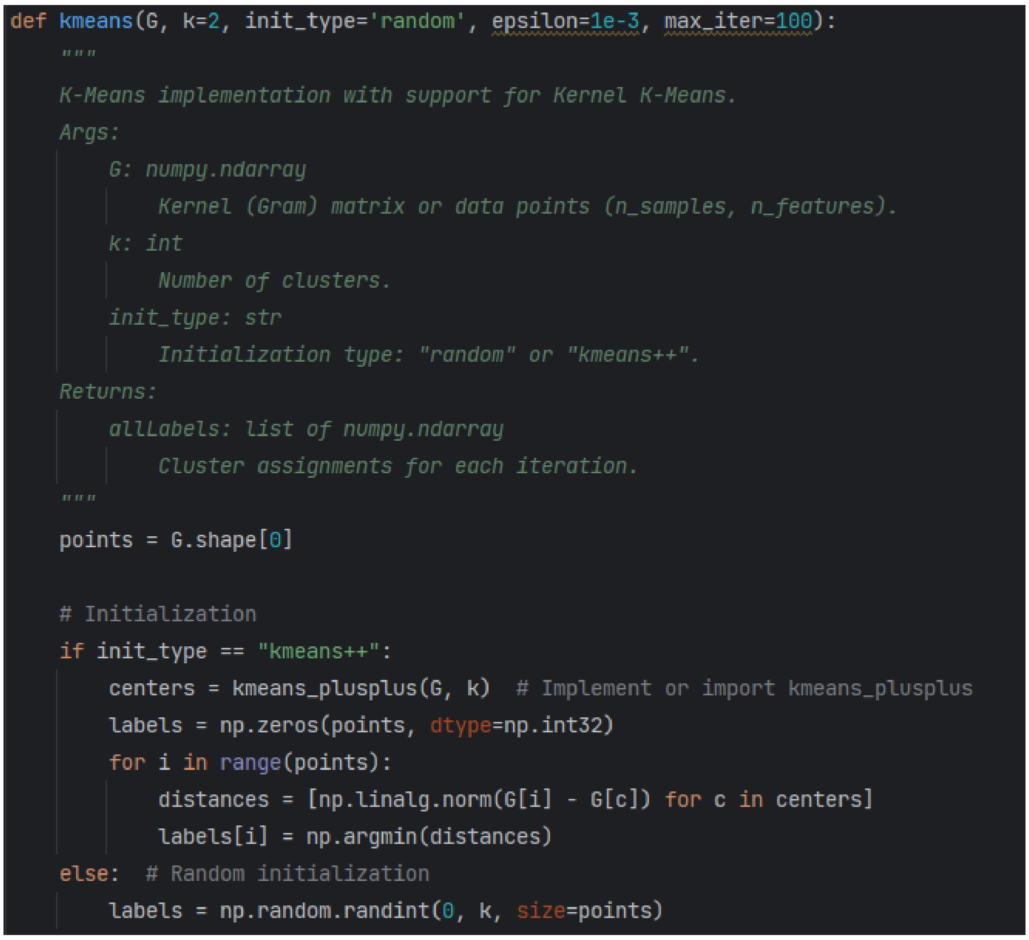


```python
def ration_cut(W, k=2, init_type='random'):
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    eigenvalues, eigenvectors = np.linalg.eig(L)
    np.save( file: 'eigenvalue_ratiocut', eigenvalues)
    np.save( file: 'eigenvector_ratiocut', eigenvectors)
    eigenvalues = np.load("eigenvalue_ratiocut.npy")
    eigenvectors = np.load("eigenvector_ratiocut.npy")
    sort_eigval = np.argsort(eigenvalues)
    H = eigenvectors[:, sort_eigval[1:k + 1]]
    return kmeans(H, k=k, init_type=init_type)
```

The code of kmeans algorithm in unnormalized spectral clustering is like kmeans clustering. Doing the initialization according init_type to assign the label to the datapoints. **(Part3)** If init_type is set to "kmeans++", the algorithm uses the k-means++ method to determine the k cluster centers and assigns labels to data points based on their closest center. Otherwise, data points are randomly assigned to k clusters.

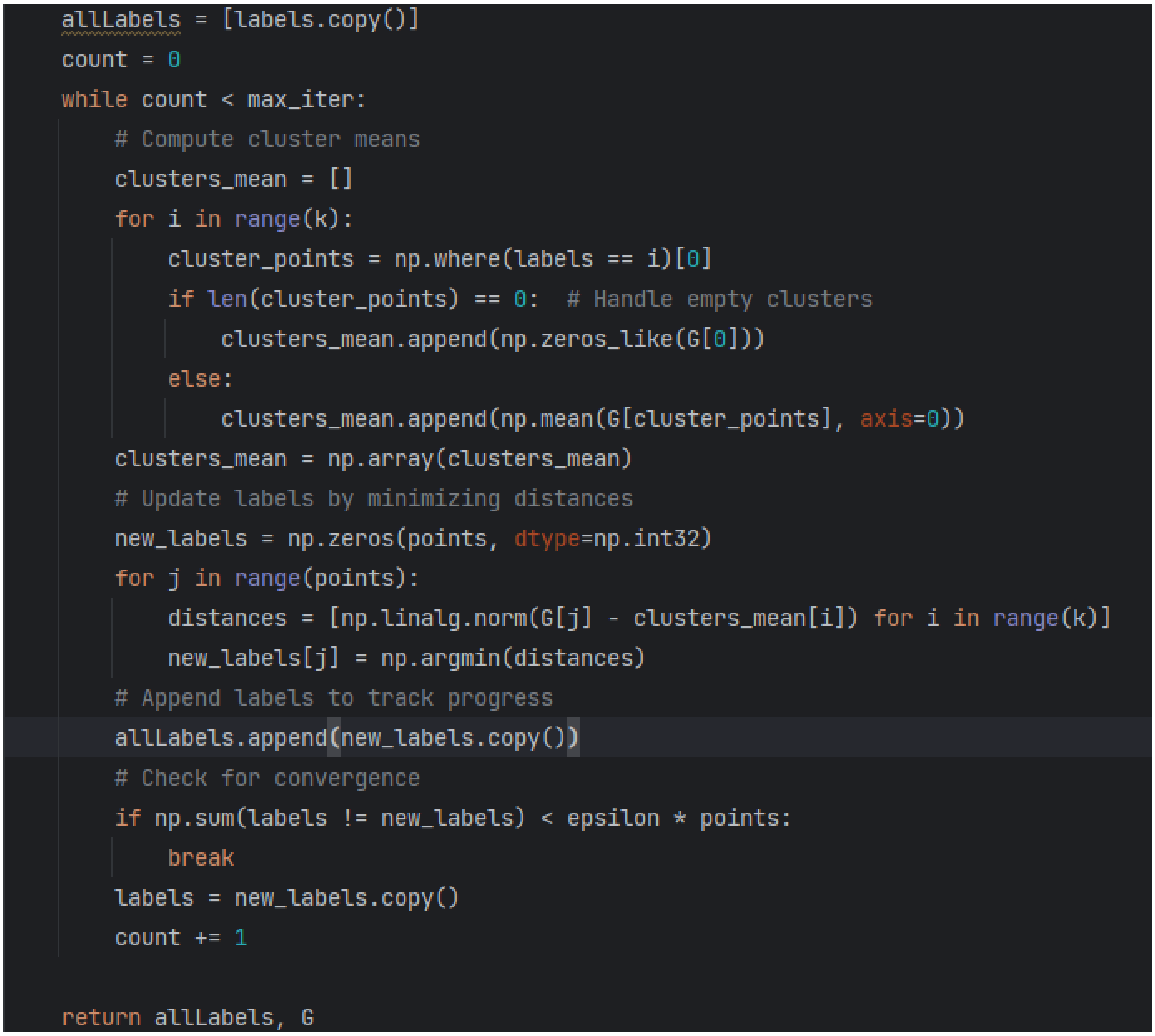


```python
def kmeans_plusplus(G, k):
    points = G.shape[0]
    # Step 1: Randomly select the first cluster center
    centers = [np.random.choice(points)]

    # Step 2: Iteratively select the next cluster centers
    for _ in range(1, k):
        # Compute squared distances to the closest center
        distances = np.zeros((points, len(centers)))
        for i, center in enumerate(centers):
            for j in range(points):
                distances[j, i] = np.linalg.norm(G[j] - G[center])
        distances = np.min(distances, axis=1)

        # Choose the next center with a probability proportional to distance^2
        probabilities = distances / distances.sum()
        next_center = np.random.choice(points, p=probabilities)
        centers.append(next_center)

    return centers
```



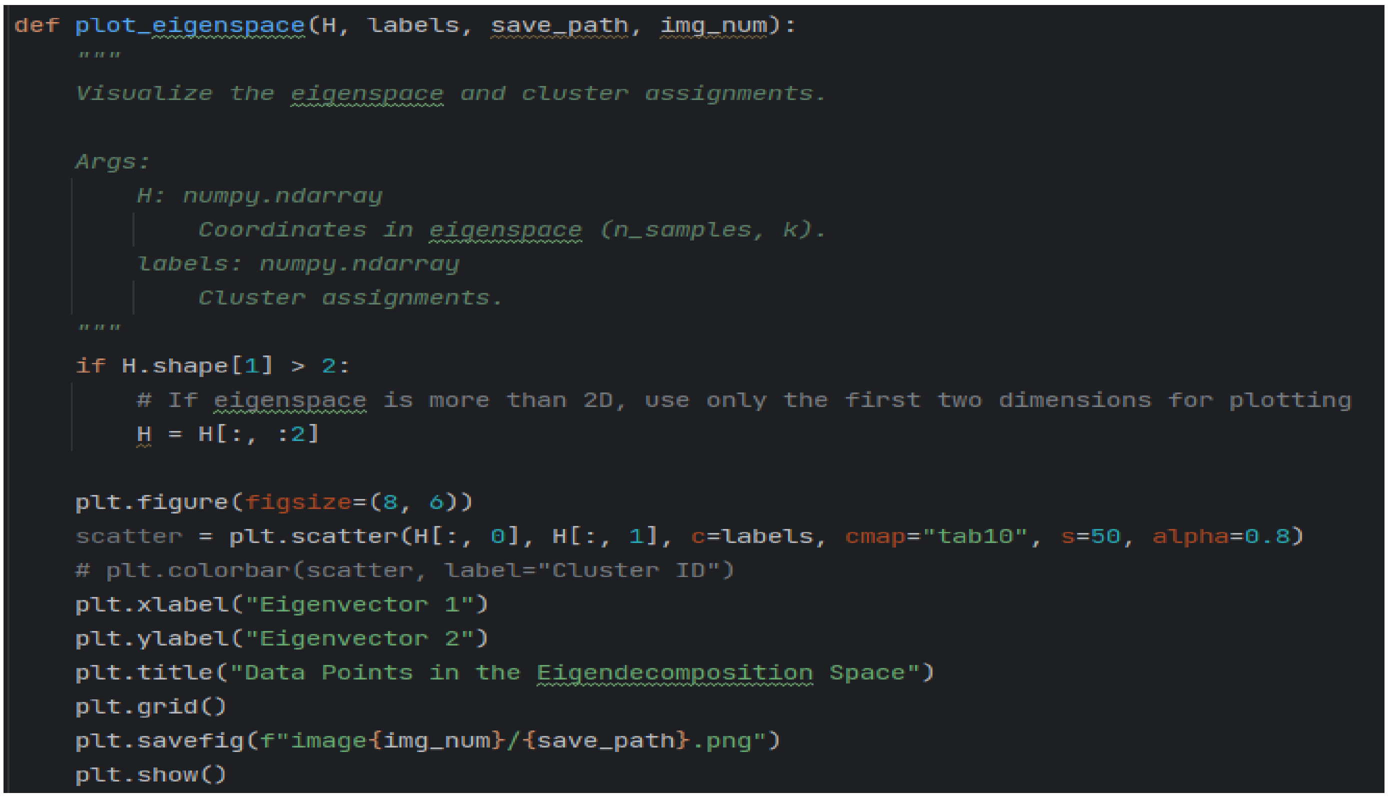
```python
def kmeans(G, k=2, init_type='random', epsilon=1e-3, max_iter=100):
    """
    K-Means implementation with support for Kernel K-Means.
    Args:
        G: numpy.ndarray
            Kernel (Gram) matrix or data points (n_samples, n_features).
        k: int
            Number of clusters.
        init_type: str
            Initialization type: "random" or "kmeans++".
    Returns:
        allLabels: list of numpy.ndarray
            Cluster assignments for each iteration.
    """
    points = G.shape[0]

    # Initialization
    if init_type == "kmeans++":
        centers = kmeans_plusplus(G, k)  # Implement or import kmeans_plusplus
        labels = np.zeros(points, dtype=np.int32)
        for i in range(points):
            distances = [np.linalg.norm(G[i] - G[c]) for c in centers]
            labels[i] = np.argmin(distances)
    else:  # Random initialization
        labels = np.random.randint(0, k, size=points)
```

After initialization, in each iteration, I calculate the mean of the data points within the same label group to serve as the center of that cluster. For each data point, I computed its distance to these centers and assigned it the label of the closest center.



```
    allLabels = [labels.copy()]
    count = 0
    while count < max_iter:
        # Compute cluster means
        clusters_mean = []
        for i in range(k):
            cluster_points = np.where(labels == i)[0]
            if len(cluster_points) == 0:  # Handle empty clusters
                clusters_mean.append(np.zeros_like(G[0]))
            else:
                clusters_mean.append(np.mean(G[cluster_points], axis=0))
        clusters_mean = np.array(clusters_mean)
        # Update labels by minimizing distances
        new_labels = np.zeros(points, dtype=np.int32)
        for j in range(points):
            distances = [np.linalg.norm(G[j] - clusters_mean[i]) for i in range(k)]
            new_labels[j] = np.argmin(distances)
        # Append labels to track progress
        allLabels.append(new_labels.copy())
        # Check for convergence
        if np.sum(labels != new_labels) < epsilon * points:
            break
        labels = new_labels.copy()
        count += 1

    return allLabels, G
```

After having each iteration label, I use the labels to create the image of each iteration and save as gif. The code of the visualization is the same as kernel kmeans except adding the plot of eigenvector.

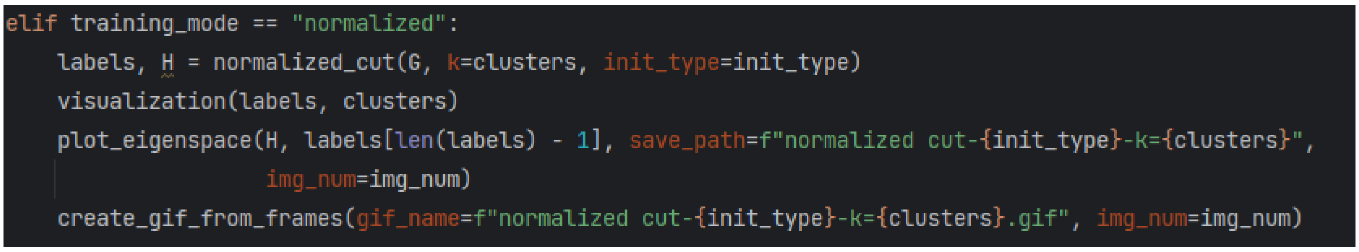


```python
def plot_eigenspace(H, labels, save_path, img_num):
    """
    Visualize the eigenspace and cluster assignments.

    Args:
        H: numpy.ndarray
            Coordinates in eigenspace (n_samples, k).
        labels: numpy.ndarray
            Cluster assignments.
    """
    if H.shape[1] > 2:
        # If eigenspace is more than 2D, use only the first two dimensions for plotting
        H = H[:, :2]

    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(H[:, 0], H[:, 1], c=labels, cmap="tab10", s=50, alpha=0.8)
    # plt.colorbar(scatter, label="Cluster ID")
    plt.xlabel("Eigenvector 1")
    plt.ylabel("Eigenvector 2")
    plt.title("Data Points in the Eigendecomposition Space")
    plt.grid()
    plt.savefig(f"image{img_num}/{save_path}.png")
    plt.show()
```

👆(Part4 function:plot_eigenspace)

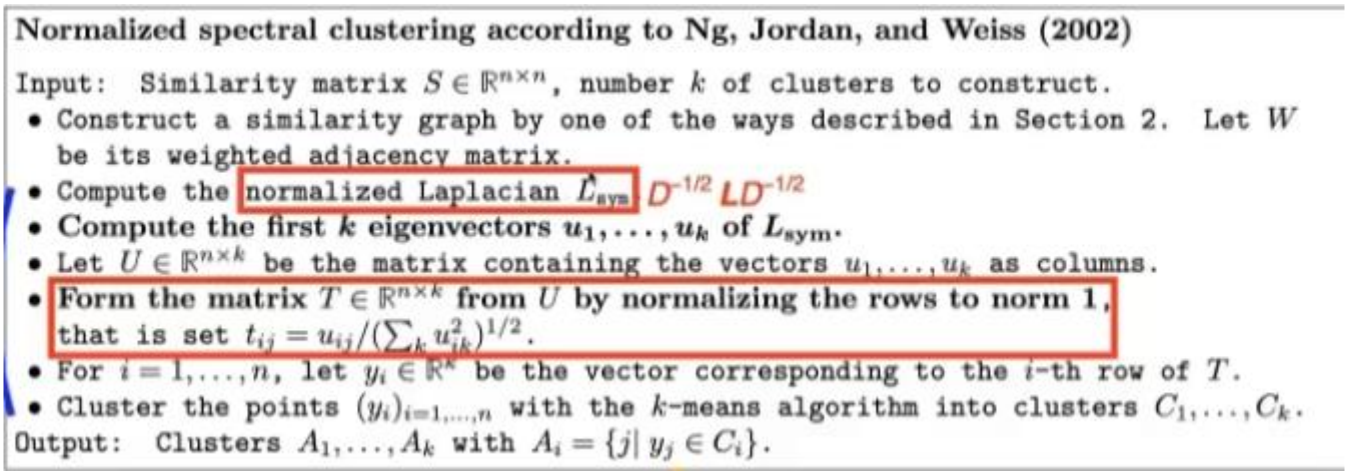
Normalized Cut:

In main function:



```python
elif training_mode == "normalized":
    labels, H = normalized_cut(G, k=clusters, init_type=init_type)
    visualization(labels, clusters)
    plot_eigenspace(H, labels[len(labels) - 1], save_path=f"normalized cut-{init_type}-k={clusters}",
                    img_num=img_num)
    create_gif_from_frames(gif_name=f"normalized cut-{init_type}-k={clusters}.gif", img_num=img_num)
```

The process is described as follows:



Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

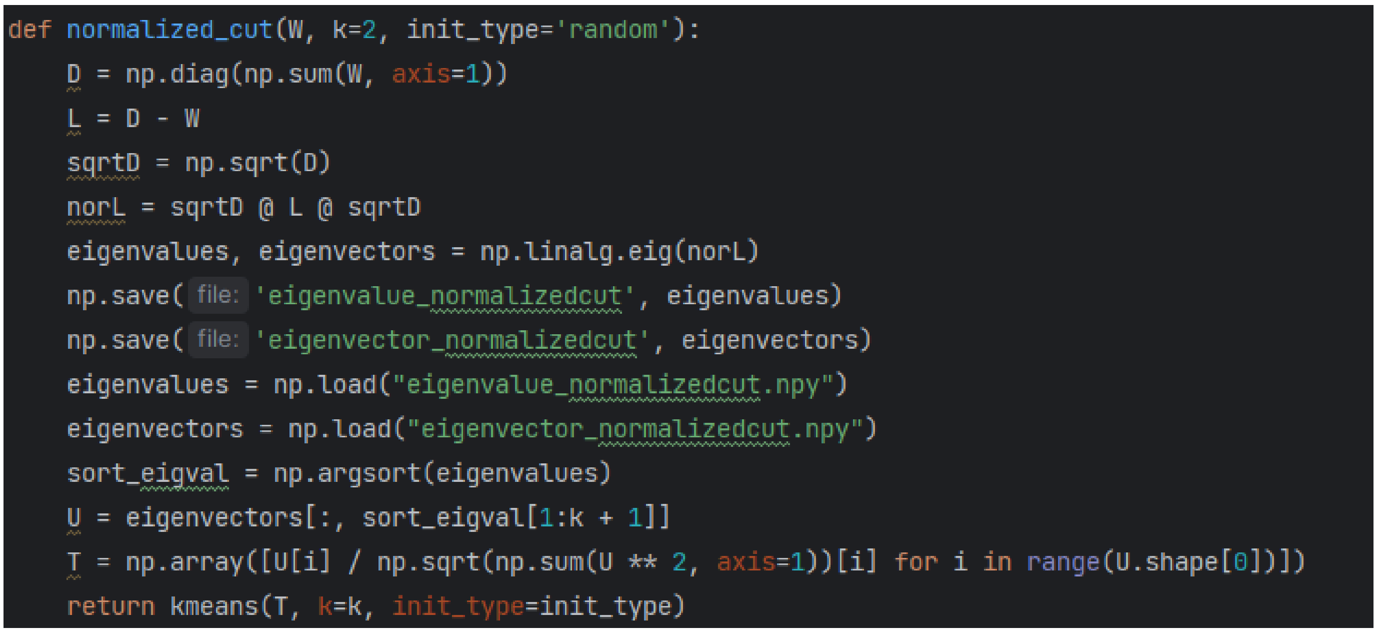
Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the $\boxed{\text{normalized Laplacian } L_{\text{sym}}}$ $D^{-1/2} LD^{-1/2}$
- Compute the first k eigenvectors $u_1, \dots, u_k$ of L_{sym}.
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ as columns.
- Form the matrix $T \in \mathbb{R}^{n \times k}$ from U by normalizing the rows to norm 1, that is set $t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}$.
- For $i = 1, \dots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of T.
- Cluster the points $(y_i)_{i=1,\dots,n}$ with the k-means algorithm into clusters $C_1, \dots, C_k$.

Output: Clusters $A_1, \dots, A_k$ with $A_i = \{j \mid y_j \in C_i\}$.

Calculating L by using L=D-W, where D is diagonal matrix where d_{ii} = summation of ith row of W.

Moreover, calculating the normalized L using D$^{-1/2}$L D$^{-1/2}$.



```
def normalized_cut(W, k=2, init_type='random'):
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    sqrtD = np.sqrt(D)
    norL = sqrtD @ L @ sqrtD
    eigenvalues, eigenvectors = np.linalg.eig(norL)
    np.save( file: 'eigenvalue_normalizedcut', eigenvalues)
    np.save( file: 'eigenvector_normalizedcut', eigenvectors)
    eigenvalues = np.load("eigenvalue_normalizedcut.npy")
    eigenvectors = np.load("eigenvector_normalizedcut.npy")
    sort_eigval = np.argsort(eigenvalues)
    U = eigenvectors[:, sort_eigval[1:k + 1]]
    T = np.array([U[i] / np.sqrt(np.sum(U ** 2, axis=1))[i] for i in range(U.shape[0])])
    return kmeans(T, k=k, init_type=init_type)
```

The kmeans function is the same as in the ratio cut, also the code of visualization is as same as in the ratio cut.

2. Experiments settings and results (30%) & discussion (20%)

hyperparameters : γ_s=0.001, γ_c=0.001

For all gif file, the naming rule is as follows:

f"{clustering method}-{initialization type}-{number of clustering}"

clustering method: kernel represent kernel kmeans, ratio represent ratio cut, normalized represent normalized cut

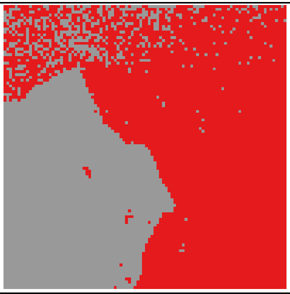
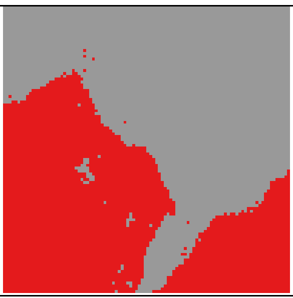
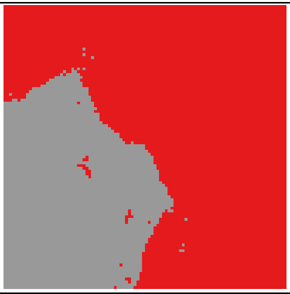
initialization type: kmean++ represent kmeans++ strategy, random represent random assign the label from 1 to number of clustering to each datapoints

number of clustering : 2 or 3

Part1

Init_type: random

i.image1

Final image	Kernel kmeans	Ratio cut	Normalized cut
2 clusters			

ii.image2

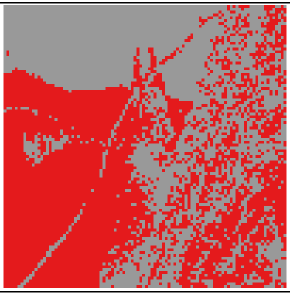
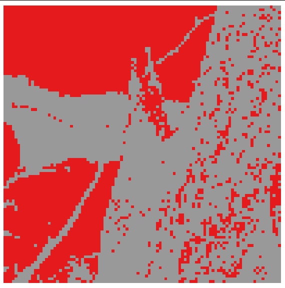
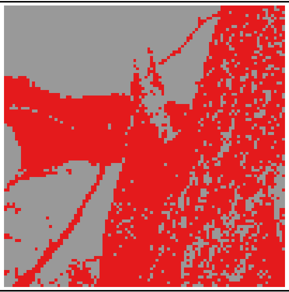
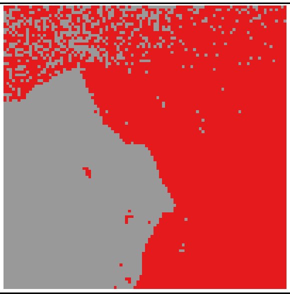
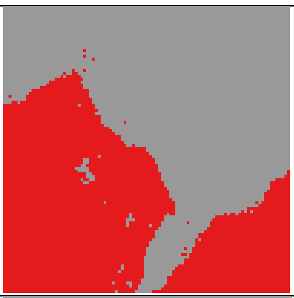
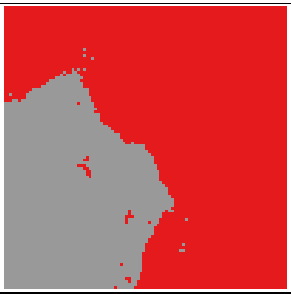
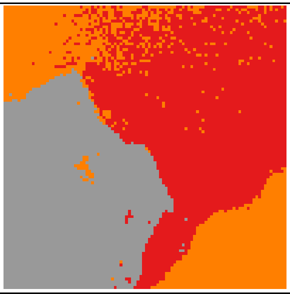
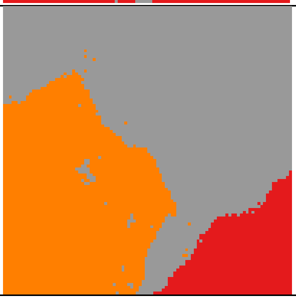
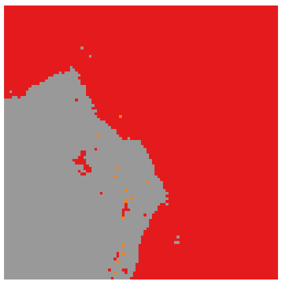
Final image	Kernel kmeans	Ratio cut	Normalized cut
2 clusters			

Image 1 can clearly distinguish between the island and the ocean, likely because of distinct color or texture differences. However, in Image 2, the presence of numerous white dots on the tree causes kernel kmeans difficulties in distinguishing the background from objects like the tree and the animal while ratio cut and normalized cut perform better.

Part2

Init_type: random

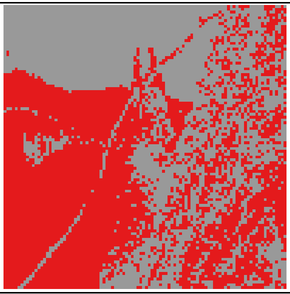
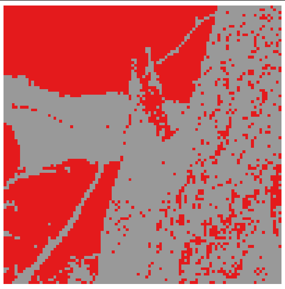
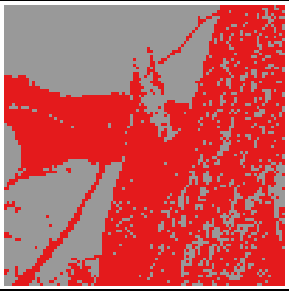
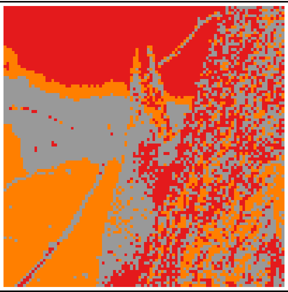
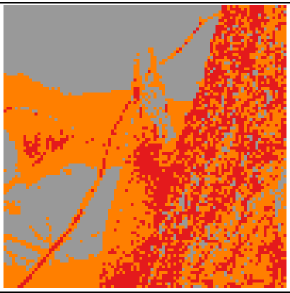
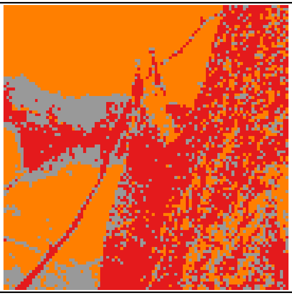
i.image1

Final image	Kernel kmeans	Ratio cut	Normalized cut
2 clusters			
3 clusters			

The reason that normalized cut in cluster 2 and 3 does not change too much is the initialization problem of using random initialization. Random initialization can lead to an imbalanced starting condition, where certain clusters are initially underpopulated or empty.

ii.image2

Final image	Kernel kmeans	Ratio cut	Normalized cut

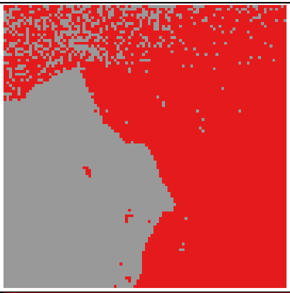
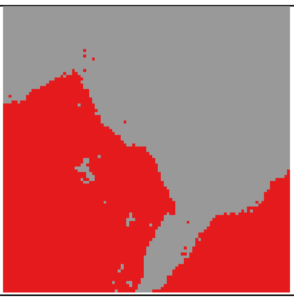
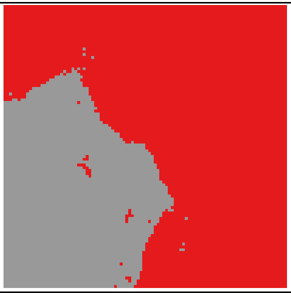
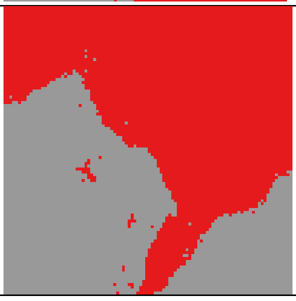
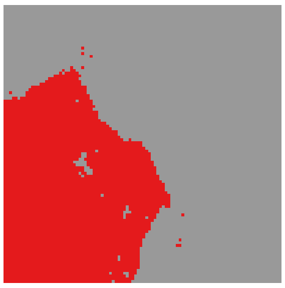
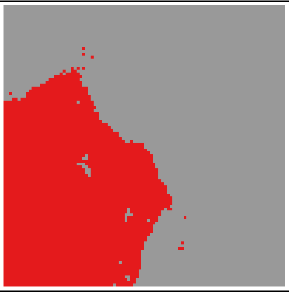
2 clusters			
3 clusters			

The performance of 2 clusters seems better than in 3 clusters. Especially the part of the tree, the white noise may blend the features, reducing the contrast needed for effective clustering.

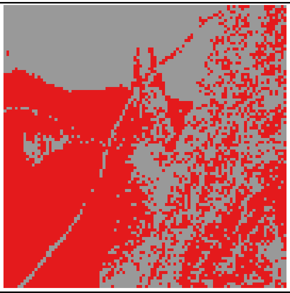
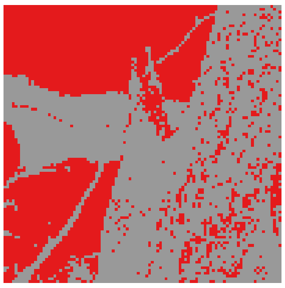
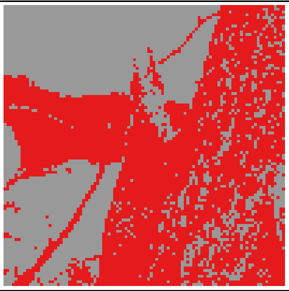
Part3

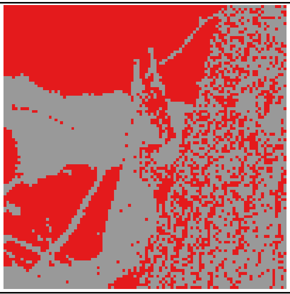
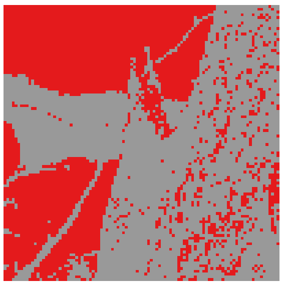
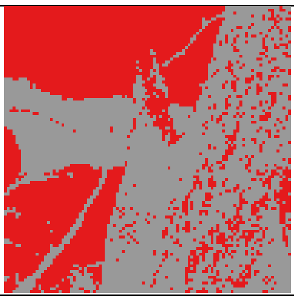
Number of clusters:2

i.image1

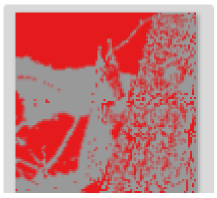
Final image	Kernel kmeans	Ratio cut	Normalized cut
Random			
Kmeans++			

ii.image2

Final image	Kernel kmeans	Ratio cut	Normalized cut
Random			

Kmeans++			

The outcome for kmeans and random does not have significant difference. However, their initial image is very different from what you can see in the below:



the left image is the start image of random initialization,while the right image is using kmeans++.

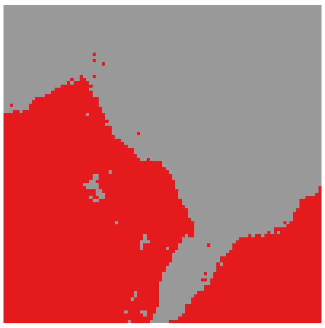
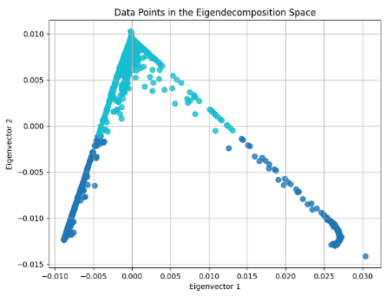
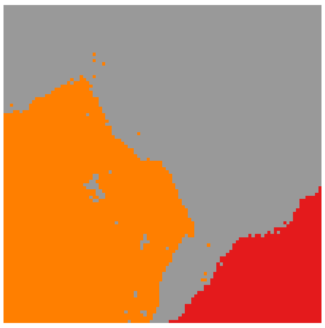
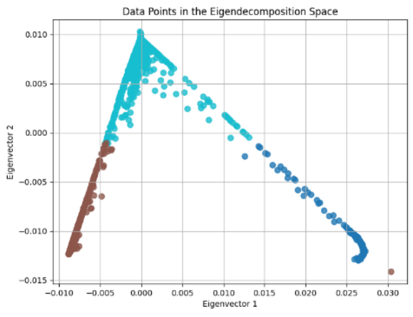
Most of the time the convergence speed of kmean++ is faster than random initialization.

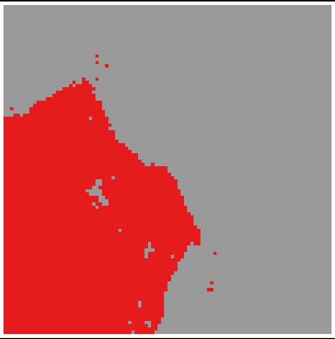
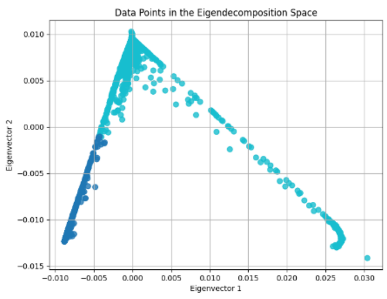
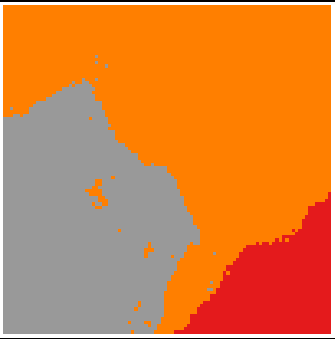
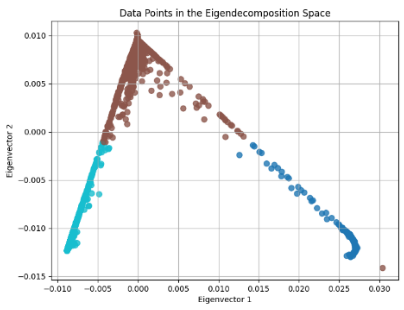
Part4

i.image1

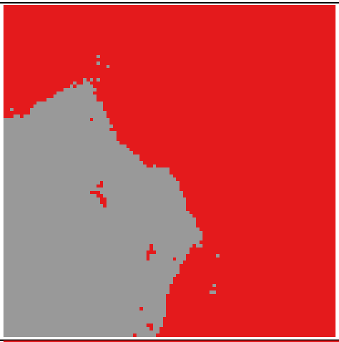
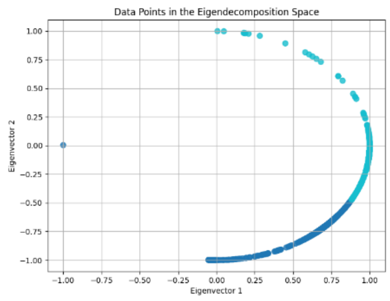
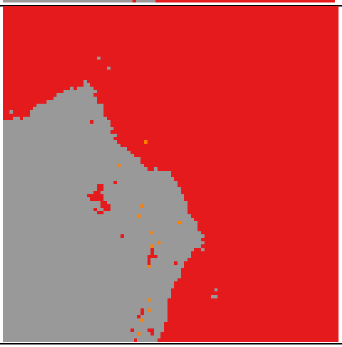
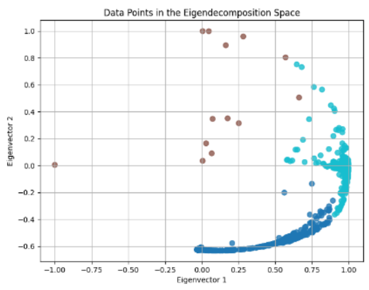
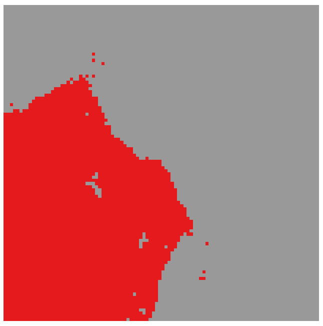
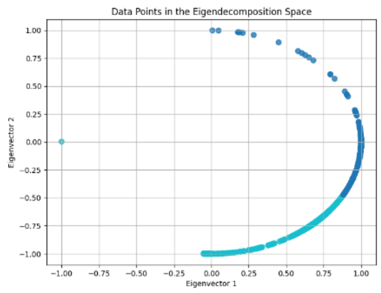
I only use two eigenvector to show the result.

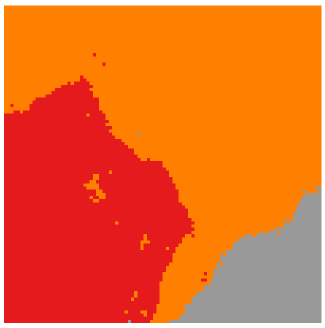
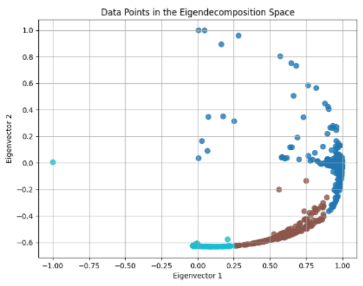
ratio cut:

	Final image	Plot of eigenvector
Random k=2		
Random k=3		

Kmeans++ k=2		
Kmeans++ k=3		

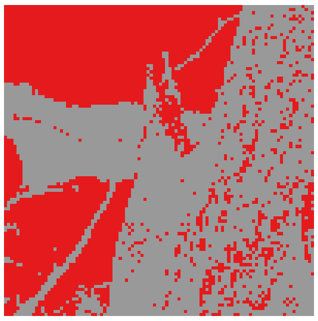
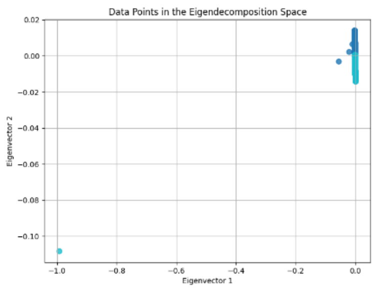
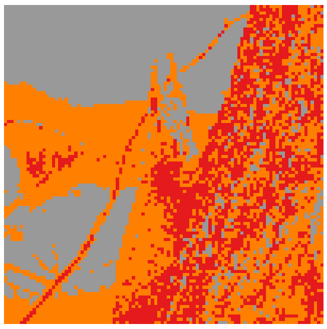
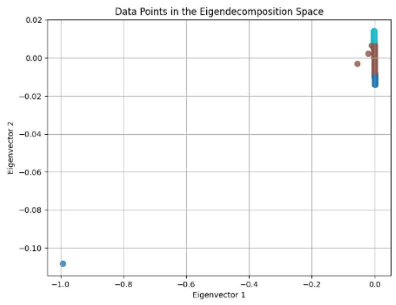
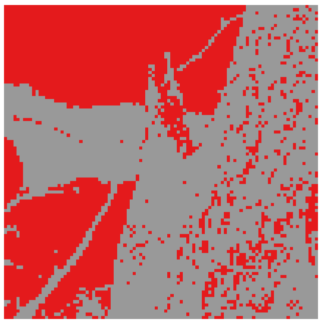
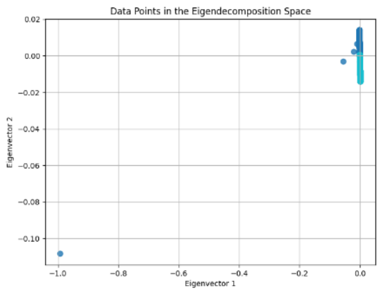
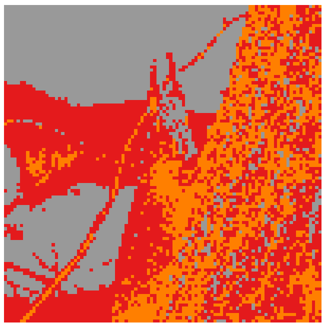
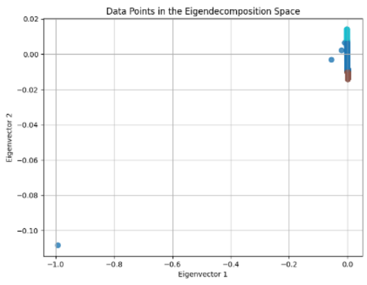
Normalized cut:

	Final image	Plot of eigenvector
Random k=2		
Random k=3		
Kmeans++ k=2		

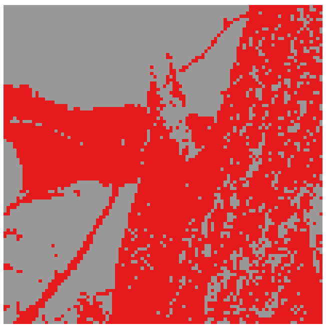
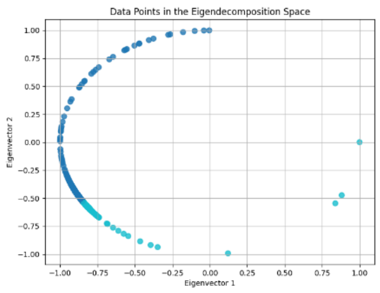
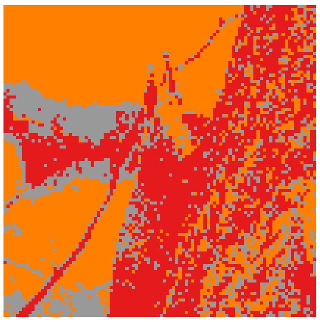
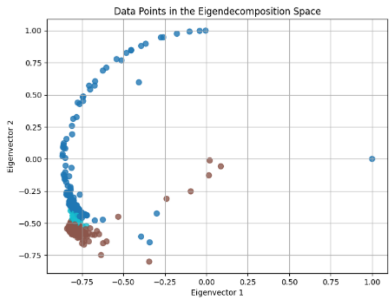
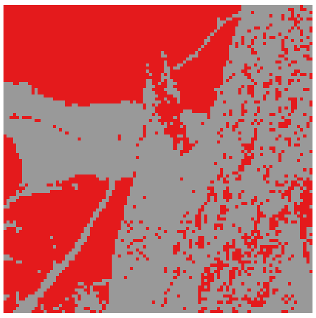
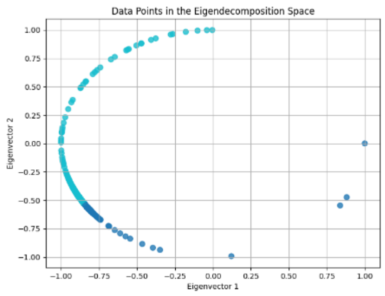
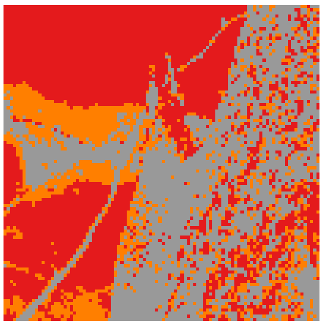
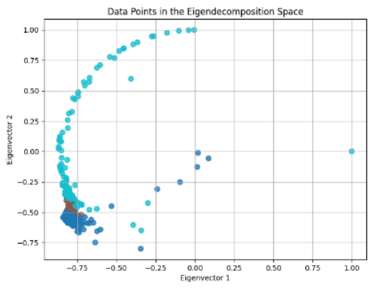
Kmeans++ k=3		

ii.image2

ratio cut:

	Final image	Plot of eigenvector
Random k=2		
Random k=3		
Kmeans++ k=2		
Kmeans++ k=3		

Normalized cut:

	Final image	Plot of eigenvector
Random k=2		
Random k=3		
Kmeans++ k=2		
Kmeans++ k=3		

Points sharing the same label tend to be close to each other in the eigenspace. However, the eigenspace representations for ratio cut and normalized cut differ significantly.

3. Observations and discussion (20%)

   i. Compare the performance between different clustering methods.

      In Image 1, the color distribution is relatively simple, resulting in similar performance across the models. However, in Image 2, the color distribution is more complex, with objects potentially having varying colors. As a result, kernel k-means struggles to perform well, while spectral clustering achieves better results.

ii. Compare the execution time of different settings.

Since the eigen decomposition in spectral clustering requires O(n^3), it generally takes longer than kernel k-means. The most time-consuming step in kernel k-means is computing the Gram matrix, which requires O(n^2). As a result, spectral clustering is computationally more expensive, especially for large datasets, due to its reliance on eigen decomposition.

iii. Anything you want to discuss.

If the difference between γ_s and γ_c is too large, the clustering performance will significantly degrade. Based on current testing, setting these parameters to be equal yields the best results.