1. Code with detailed explanations

- **Kernel k-means**

First, construct the new kernel given in the spec:

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

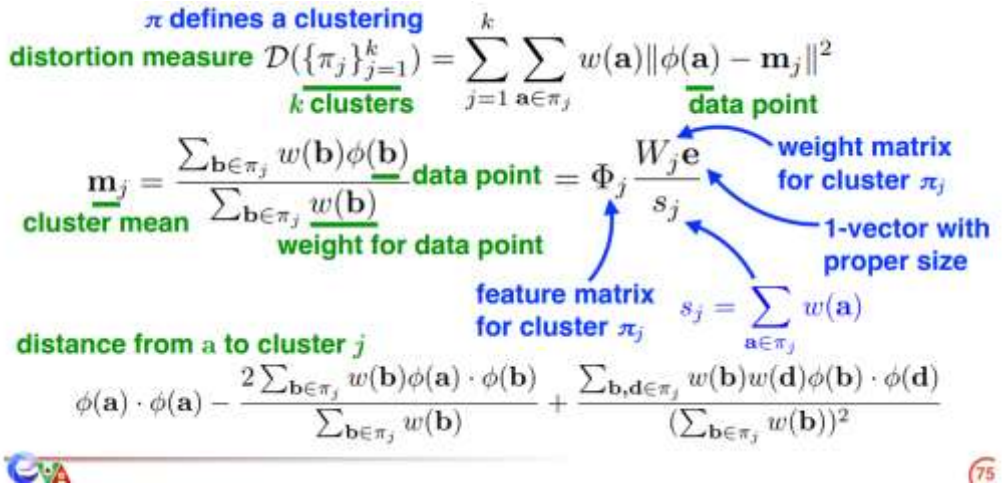
```python
def custom_kernel(image, gamma_s, gamma_c):
    h, w, c = image.shape
    n_points = h * w
    S = np.zeros((n_points, 2))  # (10000, 2)
    C = np.zeros((n_points, c))  # (10000, 3)

    for idx in range(n_points):
        S[idx] = [idx // w, idx % w]
        C[idx] = image[idx // w, idx % w]

    rbf_s = squareform(np.exp(-gamma_s * pdist(S, 'sqeuclidean')))
    rbf_c = squareform(np.exp(-gamma_c * pdist(C, 'sqeuclidean')))
    K = rbf_s * rbf_c  # (10000, 10000)

    return K
```

For the kernel k-means algorithm, usually, the coordinates of the centroids won't be explicitly written. However, the distance between a point and a cluster can be calculated using the formula on page 75 of the Unsupervised_Learning.pdf. ↓



↑ The formula can be broken down from left to right into three parts: the first, the

second, and the third term. The first term, the weight between the point and itself, is a constant, so it doesn't need to be computed. To calculate the second and third terms, it is necessary first to identify all the data points belonging to the specific cluster. It is worth noting that the third term is solely related to the members within a cluster. Therefore, these remaining two terms can be handled in separate loops to avoid redundant computations during computation.

In the kernel k-means algorithm, updating cluster centroids is unnecessary. The distance from a point to a cluster only requires considering the weights between the data point and the cluster members. When updating cluster assignments, the centroids are implicitly updated. The algorithm can be configured with maximum iterations (**max_iter**) and an early stopping criterion. The process terminates when the number of data points changing clusters falls below a threshold (**tolerance**). ↓

```
def kernel_kmeans(K, n_clusters, init_method, image_shape,
                  max_iter=100, tolerance=10):
    n_points = K.shape[0]
    clusters = initialization(K, n_clusters, init_method)
    gif_frames = []

    colormap = plt.get_cmap("tab10", n_clusters)
    colormap = (colormap(np.arange(n_clusters))[:, :3] *
                255).astype(np.uint8)  # convert to RGB format

    for iter in range(max_iter):
        print(f"Iteration: {iter + 1}")
        prev_clusters = clusters.copy()
        term_3 = np.zeros(n_clusters)

        for i in range(n_clusters):
            cluster_members = np.where(clusters == i)[0]

            if len(cluster_members) > 0:
                term_3[i] = np.sum(K[np.ix_(cluster_members,
```

```
                            cluster_members)]) / (len(cluster_members)**2)

        for i in range(n_points):
            distance_to_clusters = np.zeros(n_clusters)

            for j in range(n_clusters):
                cluster_members = np.where(clusters == j)[0]

                if len(cluster_members) > 0:
                    distance_to_clusters[j] += (-2 * np.sum(K[i,
                                    cluster_members]) / len(cluster_members))
                    distance_to_clusters[j] += term_3[j]

                else:
                    distance_to_clusters[j] = np.inf

            clusters[i] = np.argmin(distance_to_clusters)

        cluster_result = np.zeros((n_points, 3), dtype=np.uint8)
        for i in range(n_points):
            cluster_result[i] = colormap[clusters[i]]

        cluster_result = cluster_result.reshape(image_shape)
        gif_frames.append(cluster_result)

        clusters_diff = np.sum(clusters != prev_clusters)
        print(f"Cluster changes: {clusters_diff}")

        if clusters_diff < tolerance:
            print("Converged!")
            break

    return clusters, gif_frames
```

The approach involves assigning each data point a random initial cluster label for random initialization. For part 3, a different initialization method, k-means++, was used. The algorithm process was referenced from this link (link directs to cnblogs). ↓

K-means++算法
Step 1: 从数据集中随机选取一个样本作为初始聚类中心c_1；
Step 2: 首先计算每个样本与当前已有聚类中心之间的最短距离(即与最近的一个聚类中心的距离)，用$D(x)$表示；接着计算每个样本被选为下一个聚类中心的概率 $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$。最后，按照轮盘法选择出下一个聚类中心；
Step 3: 重复第 2 步直到选择出共K个聚类中心；
之后的过程与经典 K-means 算法中第 2 步至第 4 步相同

In summary, the larger the shortest distance between a sample and the currently selected cluster centers, the higher the probability of the sample being chosen as the next cluster center. There are two considerations for implementation:

1. Due to the summation term in the denominator of the probability values, it is necessary to define the candidate set of centroids beforehand. This means that we have to record the positions of the centroids explicitly.
2. During initialization, data points have not yet been labeled, preventing using the previously mentioned formula for the distance between points and cluster centroids. Since the kernel matrix represents similarity (higher similarity implies smaller distance) and the custom kernel ranges from 0 to 1, with overlapping points yielding 1, I use 1 - custom kernel as the distance measure to simplify calculations when implementing the k-means++ algorithm. ↓

```
def initialization(K, n_clusters, init_method):
    n_points = K.shape[0]

    if init_method == 'random':
        np.random.seed(42)
        clusters = np.random.randint(low=0, high=n_clusters, size=n_points)

    elif init_method == 'kmeans++':
        np.random.seed(42)
        centers = [np.random.randint(low=0, high=n_points)]

        for _ in range(n_clusters - 1):
```

```
            min_distances = 1 - np.max(K[:, centers], axis=1)
            min_distances = min_distances ** 2  # D(x)^2
            probabilities = min_distances / min_distances.sum()
            # np.random.seed(42)
            next_center = np.random.choice(n_points, p=probabilities)
            centers.append(next_center)

        clusters = np.full(n_points, -1)
        for i in range(n_points):
            clusters[i] = np.argmax([K[i, centers[j]] for j in
                                     range(n_clusters)])

    else:
        raise ValueError("Unsupported initialization method. Choose
                         'random' or 'kmeans++'.")

    return clusters
```

Other details unrelated to the kernel k-means algorithm, such as coloring the clustering results, saving the results, and creating .gif files, are omitted in this report.

- **Spectral clustering**

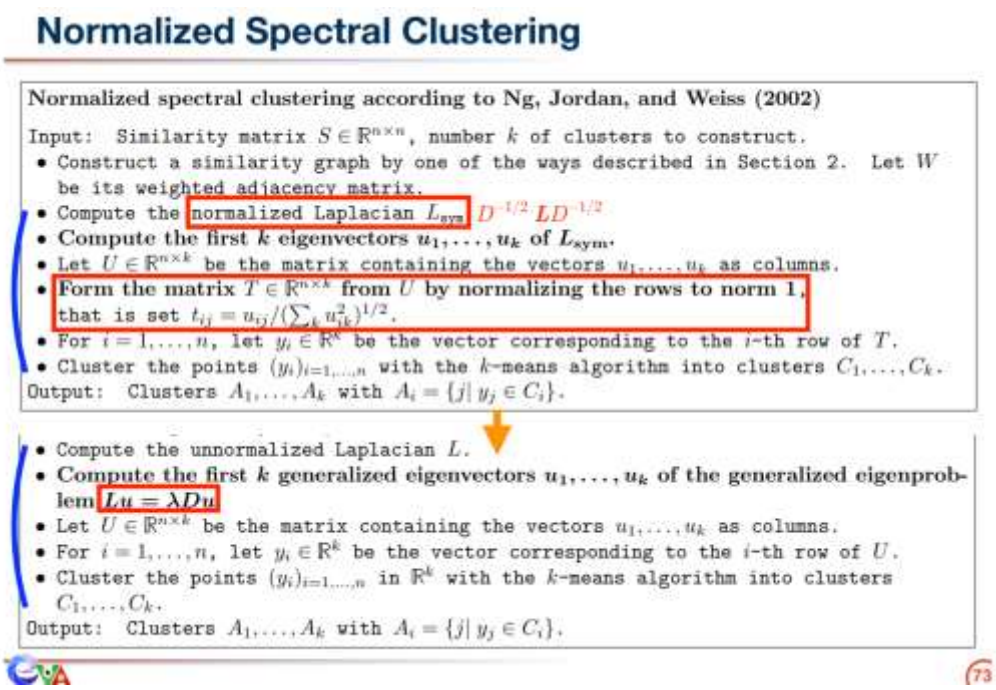
The process of ratio cut and normalized cut algorithm can be found on pages 56 and 73 of the Unsupervised_Learning.pdf, respectively. ↓

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 $u_1, \ldots, u_k$ of L.
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of U.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ in $\mathbb{R}^k$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

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



Some repetitive function definitions (such as **custom_kernel**) and tool functions unrelated to the spectral clustering algorithm (such as visualization components) will be omitted here. (However, the visualization function for Part 4 will still be included.)

The process of ratio cut begins by calculating the Laplacian matrix L and then obtaining its eigenvectors. After excluding the eigenvector corresponding to the eigenvalue 0, the eigenvectors corresponding to the most minor k non-zero eigenvalues are collected into a matrix U. Each row of U is then treated as a data point for k-means clustering. For normalized cut, the difference lies in calculating the normalized Laplacian matrix L_{sym}. Additionally, U undergoes a normalization step to obtain the matrix T. Other steps in the process remain largely the same for both algorithms. ↓

```
def main(n_clusters, init_method, spectral_method, gamma_s, gamma_c,
         max_iter, tolerance, image_path):
    file_name_prefix = image_path.split(".")[0]
    image = cv2.imread(image_path)
    start_time = time.time()

    print("Calculating custom kernel...")
    W = custom_kernel(image, gamma_s, gamma_c)
```

```
    D = np.diag(np.sum(W, axis=1))  # Degree matrix
    L = D - W  # Laplacian matrix

    if spectral_method == 'ratio':
        print("Performing spectral clustering using the ratio cut
               method...")
        U = eigen_decomposition(L, n_clusters)
        clusters, gif_frames = kmeans(U, init_method, image.shape,
                                      max_iter, tolerance)

        if n_clusters in [2, 3]:
            output_eigen_dir = "./spectral_clustering/ratio_cut/eigenspace"
            eigenspace_visualization(U, init_method, clusters,
                                     output_eigen_dir, file_name_prefix)

    elif spectral_method == 'normalized':
        print("Performing spectral clustering using the normalized cut
               method...")
        D_inv_sqrt = np.diag(1.0 / np.diag(np.sqrt(D)))
        L_sym = D_inv_sqrt @ L @ D_inv_sqrt
        U = eigen_decomposition(L_sym, n_clusters)
        T = U / np.linalg.norm(U, axis=1, keepdims=True)  # normalization
        clusters, gif_frames = kmeans(T, init_method, image.shape,
                                      max_iter, tolerance)

        if n_clusters in [2, 3]:
            output_eigen_dir =
                          "./spectral_clustering/normalized_cut/eigenspace"
            eigenspace_visualization(T, init_method, clusters,
                                     output_eigen_dir, file_name_prefix)

    else:
        raise ValueError("Unsupported spectral clustering method. Choose
                          'ratio' or 'normalized'.")

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Total execution time: {elapsed_time:.2f} seconds")
```

```
    output_gif_dir = f"./spectral_clustering/{spectral_method}_cut/gif"
    output_png_dir = f"./spectral_clustering/{spectral_method}_cut/png"
    os.makedirs(output_gif_dir, exist_ok=True)
    os.makedirs(output_png_dir, exist_ok=True)

    gif_output_path = os.path.join(output_gif_dir,
             f"{file_name_prefix}_{init_method}_{n_clusters}clusters.gif")
    save_gif(gif_output_path, gif_frames)

    png_output_path = os.path.join(output_png_dir,
             f"{file_name_prefix}_{init_method}_{n_clusters}clusters.png")
    save_last_frame_as_png(png_output_path, gif_frames[-1])
```

Eigendecomposition can be easily performed by calling **np.linalg.eigh()**. The k-means algorithm and its initialization process are similar to what was discussed in the previous section (kernel k-means). The main difference is that the centroids' coordinates can now be more intuitively and explicitly computed in the eigenspace. ↓

```
def eigen_decomposition(L, n_clusters):
    eigenvalue, eigenvector = np.linalg.eigh(L)
    return eigenvector[:, 1:1+n_clusters]  # exclude first eigenvector


def initialization(spectrum, init_method):
    n_points, n_clusters = spectrum.shape

    if init_method == 'random':
        np.random.seed(42)
        centroids = spectrum[np.random.choice(n_points, size=n_clusters,
                                              replace=False)]

    elif init_method == 'kmeans++':
        np.random.seed(42)
        centroids = [spectrum[np.random.choice(n_points)]]

        for _ in range(n_clusters - 1):
            min_distances = np.full(n_points, np.inf)
```

```
            for c in centroids:
                distances_to_c = np.linalg.norm(spectrum - c, axis=1)**2
                min_distances = np.minimum(min_distances, distances_to_c)

            probabilities = min_distances / np.sum(min_distances)
            next_centroid = np.random.choice(n_points, p=probabilities)
            centroids.append(spectrum[next_centroid])

        centroids = np.array(centroids)

    else:
        raise ValueError("Unsupported initialization method. Choose
                         'random' or 'kmeans++'.")

    return centroids


def kmeans(spectrum, init_method, image_shape, max_iter=100, tol=10):
    n_points, n_clusters = spectrum.shape
    centroids = initialization(spectrum, init_method)
    clusters = np.zeros(n_points)
    gif_frames = []

    colormap = plt.get_cmap("tab10", n_clusters)
    colormap = (colormap(np.arange(n_clusters))[:, :3] *
                255).astype(np.uint8)  # convert to RGB format

    for iter in range(max_iter):
        print(f"Iteration: {iter + 1}")
        prev_clusters = clusters.copy()
        distance_to_centroids = np.zeros((n_points, n_clusters))

        for i in range(n_points):
            for j in range(n_clusters):
                distance_to_centroids[i, j] = np.linalg.norm(spectrum[i]
                                                             - centroids[j])

        clusters = np.argmin(distance_to_centroids, axis=1)
        cluster_result = np.zeros((n_points, 3), dtype=np.uint8)
```

```
        for i in range(n_points):
            cluster_result[i] = colormap[clusters[i]]

        cluster_result = cluster_result.reshape(image_shape)
        gif_frames.append(cluster_result)

        clusters_diff = np.sum(clusters != prev_clusters)
        print(f"Cluster changes: {clusters_diff}")

        if clusters_diff < tol:
            print("Converged!")
            break
        else:
            centroids = np.array([spectrum[clusters == i].mean(axis=0)
                                  for i in range(n_clusters)])

    return clusters, gif_frames
```

Finally, for Part 4, the clustering results in the eigenspace will be visualized. ↓

```
def eigenspace_visualization(spectrum, init_method, clusters,
                             output_dir, file_name_prefix):
    n_clusters = spectrum.shape[1]
    os.makedirs(output_dir, exist_ok=True)

    if n_clusters == 2:
        plt.figure(figsize=(8, 6))

        for cluster in range(n_clusters):
            cluster_points = spectrum[clusters == cluster]
            plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
                        label=f"Cluster {cluster}")

        plt.title("Clustering in Eigenspace (2D)")
        plt.legend()
        plt.grid(True)

        output_path = os.path.join(output_dir,
                                   f"{file_name_prefix}_{init_method}_2D.png")
        plt.savefig(output_path)
```

```
        print(f"2D eigenspace clustering visualization saved to:
              {output_path}")
        plt.close()

    elif n_clusters == 3:
        fig = plt.figure(figsize=(10, 8))
        ax = fig.add_subplot(111, projection='3d')

        for cluster in range(n_clusters):
            cluster_points = spectrum[clusters == cluster]
            ax.scatter(cluster_points[:, 0], cluster_points[:, 1],
                       cluster_points[:, 2], label=f"Cluster {cluster}")

        ax.set_title("Clustering in Eigenspace (3D)")
        ax.legend()

        output_path = os.path.join(output_dir,
                                   f"{file_name_prefix}_{init_method}_3D.png")
        plt.savefig(output_path)
        print(f"3D eigenspace clustering visualization saved to:
              {output_path}")
        plt.close()

    else:
        print("Eigenspace visualization is only supported for k=2 or k=3
              dimensions.")
```

2. Experiments settings and results & discussion

- **Part1 – Part3**

The output .gif and .png files (the last frame of the .gif files) will be stored in the following directories:

- ./kernel_kmeans/gif
- ./kernel_kmeans/png
- ./spectral_clustering/normalized_cut/gif

- ./spectral_clustering/normalized_cut/png
- ./spectral_clustering/ratio_cut/gif
- ./spectral_clustering/ratio_cut/png

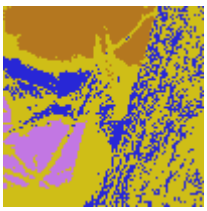
The naming convention for each file is:

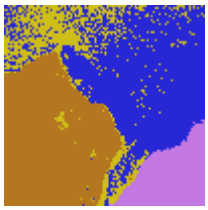
- f"{file_name_prefix}_{init_method}_{n_clusters}clusters.gif"
- f"{file_name_prefix}_{init_method}_{n_clusters}clusters.png"

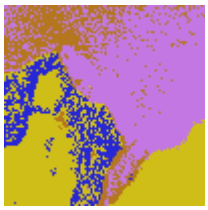
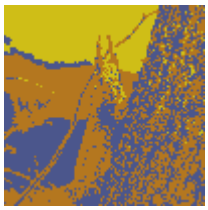
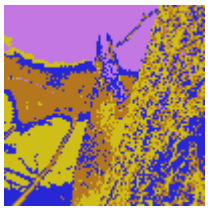
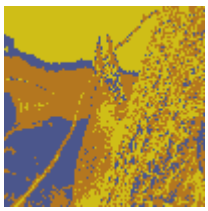
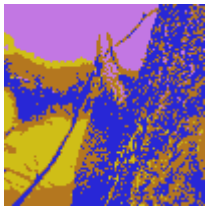
e.g. image1_kmeans++_2clusters.gif, image2_random_3clusters.png

The clustering results are presented in the table below. Due to space limitations, please refer to the above descriptions for the image paths, which are not listed individually in the table. ↓

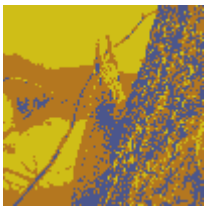
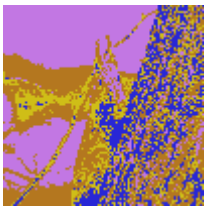
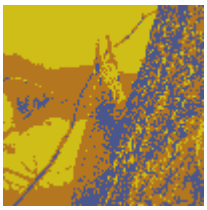
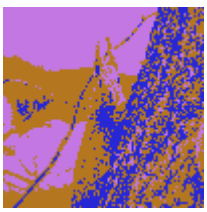
Algorithm	File Name	Initialization	# of Clusters	Result
Kernel K-means	image1	Random	2	
Kernel K-means	image1	Random	3	
Kernel K-means	image1	Random	4	

Kernel K-means	image1	K-means++	2	
Kernel K-means	image1	K-means++	3	
Kernel K-means	image1	K-means++	4	
Kernel K-means	image2	Random	2	
Kernel K-means	image2	Random	3	
Kernel K-means	image2	Random	4	
Kernel K-means	image2	K-means++	2	

Kernel K-means	image2	K-means++	3	
Kernel K-means	image2	K-means++	4	
Spectral Clustering (Normalized Cut)	image1	Random	2	
Spectral Clustering (Normalized Cut)	image1	Random	3	
Spectral Clustering (Normalized Cut)	image1	Random	4	
Spectral Clustering (Normalized Cut)	image1	K-means++	2	
Spectral Clustering (Normalized Cut)	image1	K-means++	3	

Spectral Clustering (Normalized Cut)	image1	K-means++	4	
Spectral Clustering (Normalized Cut)	image2	Random	2	
Spectral Clustering (Normalized Cut)	image2	Random	3	
Spectral Clustering (Normalized Cut)	image2	Random	4	
Spectral Clustering (Normalized Cut)	image2	K-means++	2	
Spectral Clustering (Normalized Cut)	image2	K-means++	3	
Spectral Clustering (Normalized Cut)	image2	K-means++	4	

Spectral Clustering (Ratio Cut)	image1	Random	2	
Spectral Clustering (Ratio Cut)	image1	Random	3	
Spectral Clustering (Ratio Cut)	image1	Random	4	
Spectral Clustering (Ratio Cut)	image1	K-means++	2	
Spectral Clustering (Ratio Cut)	image1	K-means++	3	
Spectral Clustering (Ratio Cut)	image1	K-means++	4	
Spectral Clustering (Ratio Cut)	image2	Random	2	

Spectral Clustering (Ratio Cut)	image2	Random	3	
Spectral Clustering (Ratio Cut)	image2	Random	4	
Spectral Clustering (Ratio Cut)	image2	K-means++	2	
Spectral Clustering (Ratio Cut)	image2	K-means++	3	
Spectral Clustering (Ratio Cut)	image2	K-means++	4	

- **Part4**

  The output .png files will be stored in the following directories:

  - ./spectral_clustering/normalized_cut/eigenspace
  - ./spectral_clustering/ratio_cut/eigenspace

  The naming convention for each file is:

  - f"{file_name_prefix}_{init_method}_2D.png"
  - f"{file_name_prefix}_{init_method}_3D.png"

e.g. image1_random_2D.png, image2_kmeans++_3D.png

Please note that visualization is performed only for cases where the number of clusters is 2 or 3. The clustering results on eigenspace are presented in the table below. Due to space limitations, please refer to the above descriptions for the image paths, which are not listed individually in the table. ↓

- Spectral Clustering (Normalized Cut)

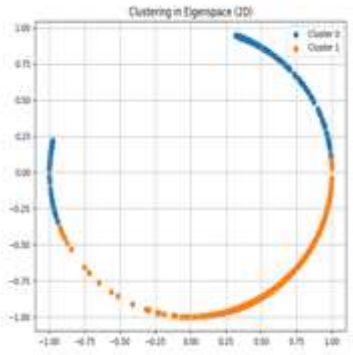
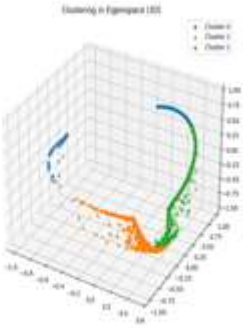
File Name	Initialization	# of Clusters	Result (Eigenspace)
image1	Random	2	
image1	Random	3	

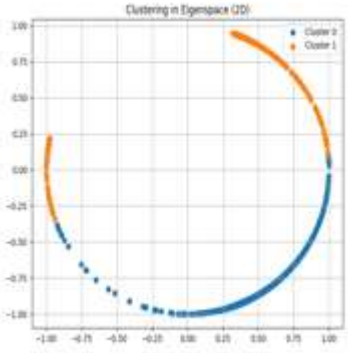
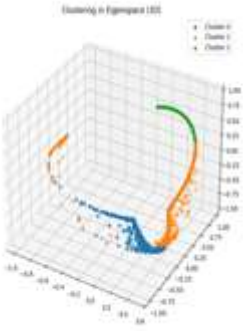
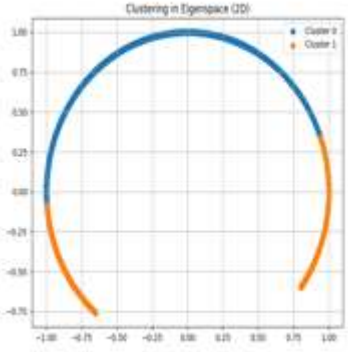
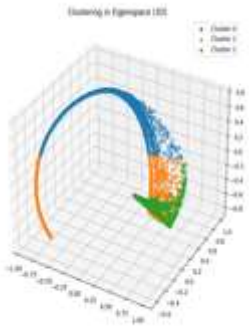
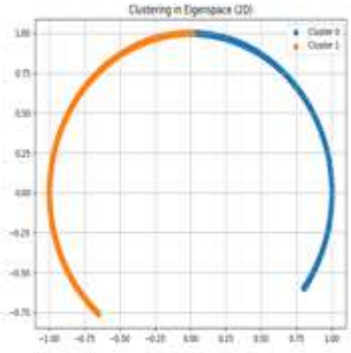
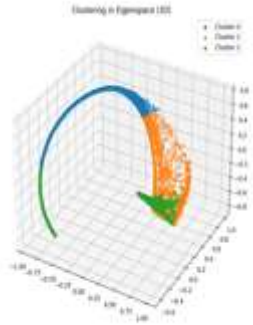
image1	K-means++	2	
image1	K-means++	3	
image2	Random	2	
image2	Random	3	

image2	K-means++	2	
image2	K-means++	3	

- Spectral Clustering (Ratio Cut)

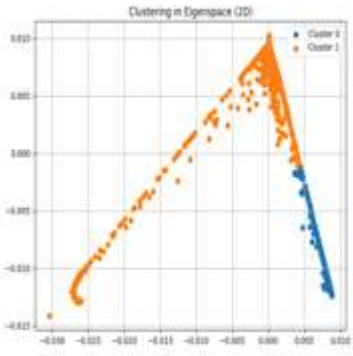
File Name	Initialization	# of Clusters	Result (Eigenspace)
image1	Random	2	

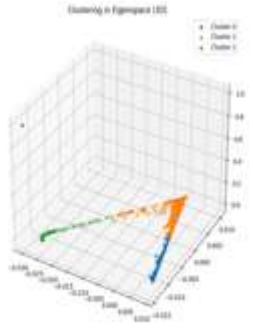
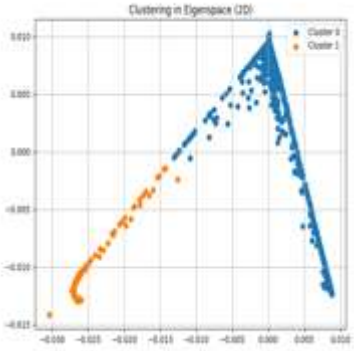
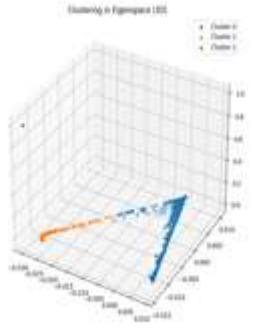
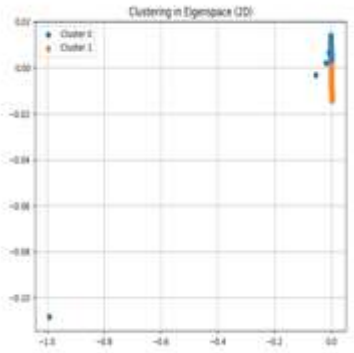
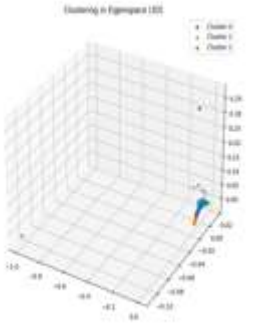
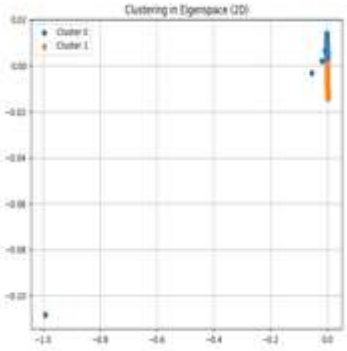
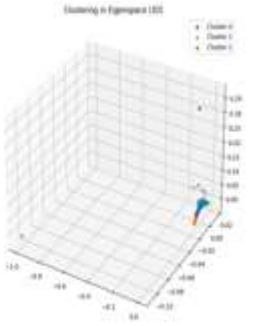
image1	Random	3	
image1	K-means++	2	
image1	K-means++	3	
image2	Random	2	

image2	Random	3	
image2	K-means++	2	
image2	K-means++	3	

↑ We can easily observe from the visualization that points close to each other in the eigenspace tend to have the same label, forming clusters.

3. Observations and discussion

- **Compare the performance between different clustering methods**

Due to the nature of the clustering methods required for this assignment, where the results vary depending on the initial values and lack ground truth, I will summarize

the performance of each method based on a more subjective evaluation:

- On average, the kernel k-means method requires more iterations to converge under the same termination conditions than spectral clustering approaches.
- The methods of spectral clustering tend to produce clustering results where the sizes of clusters are more balanced. However, strange cases may still occur (e.g., the clustering result of image1 under ratio cut, which will be discussed in the last part of this report). These anomalies seem to be caused more by the initial value settings than the algorithm itself. On the other hand, in the kernel k-means method, even after using different sets of random seeds, the clustering results remain less effective in distinguishing between the background and the object than spectral clustering, such as in the case of image2.
- Based on my experimental results, the clustering outcomes of normalized cut exhibit more balanced cluster sizes than those of ratio cut. This can be more clearly explained through the visualization of the eigenspace, which will be elaborated on in the last part of this report.

• Compare the execution time of different settings

- Kernel K-means

Settings	Convergence iterations	Execution time (s)
image1, random, k = 2	13	19.59
image1, random, k = 3	12	21.87
image1, random, k = 4	54	100.04
image1, k-means++, k = 2	18	26.11
image1, k-means++, k = 3	12	20.99

image1, k-means++, k = 4	31	58.04
image2, random, k = 2	15	23.10
image2, random, k = 3	10	19.85
image2, random, k = 4	27	54.00
image2, k-means++, k = 2	6	11.37
image2, k-means++, k = 3	26	46.27
image2, k-means++, k = 4	54	102.80

- Spectral Clustering (Normalized Cut)

Settings	Convergence iterations	Execution time (s)
image1, random, k = 2	3	77.07
image1, random, k = 3	7	79.50
image1, random, k = 4	9	77.69
image1, k-means++, k = 2	3	70.13
image1, k-means++, k = 3	11	83.08
image1, k-means++, k = 4	12	73.50
image2, random, k = 2	5	73.63
image2, random, k = 3	9	81.74
image2, random, k = 4	16	84.38
image2, k-means++, k = 2	19	81.63
image2, k-means++, k = 3	9	79.52
image2, k-means++, k = 4	5	83.19

- Spectral Clustering (Ratio Cut)

Settings	Convergence iterations	Execution time (s)

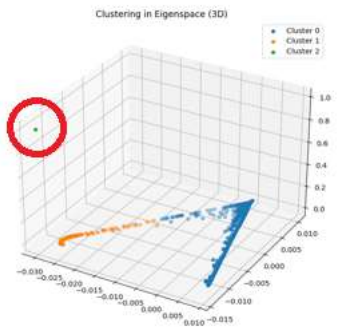
image1, random, k = 2	2	65.20
image1, random, k = 3	4	74.69
image1, random, k = 4	9	76.31
image1, k-means++, k = 2	2	74.14
image1, k-means++, k = 3	2	76.89
image1, k-means++, k = 4	2	71.87
image2, random, k = 2	7	75.61
image2, random, k = 3	6	75.86
image2, random, k = 4	11	64.66
image2, k-means++, k = 2	3	73.70
image2, k-means++, k = 3	6	60.28
image2, k-means++, k = 4	5	64.74

The total execution time may vary depending on the differences in CPU performance. However, it is still possible to observe the relative speed of each method under different settings. The following two points summarize the findings:

- Using k-means++ initialization does not necessarily guarantee faster convergence of the whole process. However, we can observe that during the initial few iterations, the number of data points changing cluster assignments is significantly less than random initialization, indicating that k-means++ indeed has better stability.
- It can be observed that, among the two spectral clustering methods, although the number of convergence iterations may differ under various settings, the overall execution times are similar. In contrast, there is a significant difference in execution times across different settings for the kernel k-means

method. This is because, in spectral clustering, most of the total execution time is spent on the eigendecomposition calculation, while the subsequent k-means algorithm executes quickly. When excluding the eigendecomposition step, if we purely compare the clustering execution speeds of kernel k-means and spectral clustering, spectral clustering is faster.

- **Anything you want to discuss**
  - By adjusting np.random.seed(), it was observed that clustering results can still change under the same parameter settings, confirming that different initial values can indeed lead to different convergence outcomes. Conversely, different clustering methods can yield similar results, as seen with the normalized cut and ratio cut methods in the "image2, k-means++, k=2" case.
  - As mentioned earlier, the experimental results revealed an imbalance in cluster sizes during ratio cuts. In the "image1, k-means++, k=3" setting, visual analysis suggests the presence of only two distinct and prominent categories despite the algorithmic requirement to partition the data into three clusters. The reason for this can be traced back to its eigenspace:



We can observe outliers marked by red circles in the eigenspace. If, during the initialization process, the centroid of one cluster is placed near these outliers, there may be no other data points assigned to this cluster during

the iterative process. As a result, the centroid will not be updated, and only a few points will ultimately form this cluster.

In the corresponding original image, the area marked by the black circle contains a small group of data points whose color is slightly different, which is difficult for the naked eye to detect. These points likely correspond to the ones marked by the red circles in the eigenspace. 

- Finally, there are some minor details. I initially thought the collection of all eigenvectors was called the spectrum, so I named the variable passed in the spectral clustering as "spectrum." However, I later realized that this term should refer to the collection of eigenvalues (but I was too lazy to change the code). 😂