# Homework 6 report

## Machine Learning

Institute of Computer Science and Engineering

05/24/2024

***Produced by :***
PAULY Alexandre
alexandre.pauly@cy-tech.fr

# Contents

# 1 Introduction

**Context:**

As part of the Machine Learning module, an assignment focusing on Spectral clustering and K-means clustering has been initiated. This assignment aims to deepen understanding and practical implementation skills in these specific areas of unsupervised machine learning. Spectral clustering is a technique that leverages the eigenvalues of a similarity matrix to perform dimensionality reduction before clustering in fewer dimensions. On the other hand, K-means clustering is a partitioning method that divides a dataset into K distinct, non-overlapping clusters based on the distances between data points and cluster centroids.

**Objective:**

In this report, we focus on understanding and implementing Spectral clustering from scratch and K-means clustering using both custom implementation and available libraries. The main objective is to explain in detail each step of the implementation of these two techniques, including the underlying mathematical principles, computational procedures, and visualization techniques. Furthermore, we will illustrate the application of Spectral clustering and K-means clustering on the same dataset, demonstrating their effectiveness and the nuances in their practical use.

# 2 Dataset

The data used in this study contains only two images as shown by Figure 2:

**Figure 2:** Dataset

# 3 Implementation

## 3.1 Spectral clustering

Spectral clustering is a clustering technique that leverages the spectral information of data to group similar points in a low-dimensional space. Unlike traditional clustering methods like K-means, spectral clustering can capture complex and non-linear structures in data by relying on similarity relationships between points.

**1st step : Similarity matrix**

To implement this method, we start by loading the data in PNG format. Once the loading is done, we compute the similarity matrix using the kernel. This affinity matrix is constructed by using both color distance and spatial distances between pixels in the image. This allows us to capture both color and spatial location similarities between pixels.

```python
def kernel(img, img_size, gamma_s, gamma_c):
    """! Function to compute the affinity matrix using a combined spatial and color Gaussian kernel. """

    # Initialization of variables
    color_dist = cdist(img, img, 'sqeuclidean')
    coordinates = np.zeros((img_size, 2))

    # Creating a coordinate matrix for spatial distances
    for i in range(100):
        index = i * 100
        for j in range(100):
            coordinates[index + j][0] = i
            coordinates[index + j][1] = j

    # Calculate spatial distances between pixels
    spatial_distance = cdist(coordinates, coordinates, 'sqeuclidean')

    # Combine color and spatial distances with Gaussian kernels
    img_kernel = np.multiply(np.exp(-gamma_c * color_dist), np.exp(-gamma_s * spatial_distance))

    return img_kernel
```

***Code 1 :*** *Affinity matrix*

**2nd step : Laplacian matrix**

To apply spectral clustering, the Laplacian matrix is calculated with *compute_ laplacian* and is used to represent the connectivity between pixels in the image. It is calculated as the difference between the degree matrix and the affinity matrix.

```
def compute_laplacian(W):
    """! Function to compute the Laplacian matrix. """

    # Compute the degree matrix
    D = np.diag(np.sum(W, axis=1))

    # Compute the unnormalized Laplacian matrix
    L = D - W

    return L, D
```

***Code 2 :*** *Laplacian matrix*

**3rd step : Laplacian normalisation (if needed)**

In the case of normalized spectral clustering, the Laplacian matrix is normalized. This is done in the *normalize_ laplacian* function. Normalization makes the Laplacian matrix independent of the size and density of the graph.

```
def normalize_laplacian(L, D):
    """! Function to normalize the Laplacian matrix. """

    # Compute the inverse square root of the degree matrix
    sqrt_D_inv = np.diag(1.0 / np.sqrt(np.diag(D)))

    # Compute the normalized Laplacian matrix
    L_normalized = sqrt_D_inv @ L @ sqrt_D_inv

    return L_normalized, sqrt_D_inv
```

***Code 3 :*** *Laplacian normalisation*

**4th step : Eigen decomposition**

The eigen decomposition of the Laplacian matrix is performed in the *eigen_ decomposition* function. This allows us to obtain the eigenvectors associated with the smallest eigenvalues.

```
def eigen_decomposition(L, num_cluster):
    """! Function to perform eigen decomposition and select
       the first k eigenvectors. """

    eigenvalues, eigenvectors = np.linalg.eig(L)
    index = np.argsort(eigenvalues)
    eigenvectors = eigenvectors[:, index]
```

```
    return eigenvectors[:, 1:1 + num_cluster].real
```

***Code 4 :*** *Eigen decomposition*

**5th step : Spectral custering**

Spectral clustering is performed in the *kmeans* function. It uses the eigenvectors calculated earlier as features to perform clustering. The pixels are projected into a new space defined by the eigenvectors, and K-means clustering is applied in this space.

```
def kmeans(U, num_cluster, output_dir, img_size, method):
    """ This function performs K-means clustering on the data
        . """

    # Initialization of variables
    converge = 0   # Convergence flag
    iteration = 1  # Iteration counter
    means, clusters = init_kmeans(U, num_cluster, method,
       img_size)   # Initialize clusters

    while not converge:
        print(f'iteration: {iteration}')
        pre_clusters = clusters

        # E-step
        clusters = E_step(U, means)

        # M-step
        means = M_step(U, clusters, num_cluster)

        # Save the current clustering result
        save_picture(clusters, iteration, num_cluster,
           img_size, output_dir)

        # Check for convergence
        converge = check_converge(clusters, pre_clusters,
           img_size)

        iteration += 1

    return clusters
```

***Code 5 :*** *Special clustering*

In this step, each data point is assigned to the nearest cluster in terms of distance. In this context, it means calculating the distance between each data point and the current cluster centers, and then assigning each point to the cluster with the nearest center.

```python
def E_step(U, means):
    """! Function to perform the E-step of the K-means
       algorithm. """

    # Compute the squared distances between each point and
       each mean
    distances = np.linalg.norm(U[:, np.newaxis] - means, axis
       =2)**2

    # Assign each point to the cluster with the minimum
       distance
    new_clusters = np.argmin(distances, axis=1)

    return new_clusters
```

***Code 6 :*** *E-step*

In this step, the cluster centers are updated to better represent the data. This is done by recalculating the cluster centers as the average of the data points assigned to each cluster in step E. In other words, the cluster centers are moved towards the center of gravity of the points assigned to them.

```python
def M_step(U, clusters, num_cluster):
    """ Function to perform the M-step of the K-means
       algorithm. """

    # Initialize an array to store the new means of the
       clusters
    new_means = np.zeros((num_cluster, U.shape[1]))

    # For each cluster, compute the mean of the points
       assigned to that cluster
    for cluster in range(num_cluster):
        cluster_points = U[clusters == cluster]
        if len(cluster_points) > 0:
            new_means[cluster] = cluster_points.mean(axis=0)

    return new_means
```

***Code 7 :*** *M-step*

**$6^{th}$ step : Visualisation**

In order to properly analyze the clustering, the *draw_ eigenspace* function is used to visualize the results in the space of the eigenvectors. This helps understand the separation of clusters in this space.

```python
def draw_eigenspace(U, clusters, num_cluster, img_size,
   output_dir):
```

```
    """ This function visualizes the clustering result in
       eigenspace. """

    # Initialization of variables
    points_x, points_y, points_z = [], [], []  # Store
       coordinates
    color = ['c', 'm', 'grey']                 # Colors

    if num_cluster == 2:
        # Prepare lists for 2D coordinates
        for _ in range(num_cluster):
            points_x.append([])
            points_y.append([])

        # Assign points to clusters
        for pixel in range(img_size):
            points_x[clusters[pixel]].append(U[pixel][0])
            points_y[clusters[pixel]].append(U[pixel][1])

        # Plot the clusters
        for cluster in range(num_cluster):
            plt.scatter(points_x[cluster], points_y[cluster],
                c=color[cluster])

        # Save the plot
        plt.savefig(f'{output_dir}/eigenspace_{num_cluster}.
           png')
    elif num_cluster == 3:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')

        # Prepare lists for 3D coordinates
        for _ in range(num_cluster):
            points_x.append([])
            points_y.append([])
            points_z.append([])

        # Assign points to clusters
        for pixel in range(img_size):
            points_x[clusters[pixel]].append(U[pixel][0])
            points_y[clusters[pixel]].append(U[pixel][1])
            points_z[clusters[pixel]].append(U[pixel][2])

        # Plot the clusters in 3D
```

```
42          for cluster in range(num_cluster):
43              ax.scatter(points_x[cluster], points_y[cluster],
                    points_z[cluster], c=color[cluster])
44
45          # Save the plot
46          fig.savefig(f'{output_dir}/eigenspace_{num_cluster}.
                png')
47      plt.show()
```

***Code 8 :*** *Visualisation*

Also, images are saved as png file after each iterations of the algorithm. And a GIF is created to show the evolution.

## 3.2 K-means clustering

K-means clustering is a popular and unsupervised machine learning algorithm for partitioning a dataset into K distinct, non-overlapping clusters. The goal is to minimize the within-cluster variance by iteratively refining the cluster centers and reassigning data points to the nearest centers. The algorithm follows these steps :

**1st step : Initialisation**

After initializing the variables to manage the state of the clustering algorithm, including the convergence indicator and iteration counter, as well as the clusters and their counts, the algorithm can begin.

```
# Initialization of variables
converge = 0    # Boolean (0: False and 1: True)
iteration = 1   # Iterations counter

# Initialize clusters and cluster counts
clusters, C = init(W, mode, img_size, num_cluster, output_dir
    )
```

***Code 9 :*** *Initialization*

The function *init_cluster_c* initializes the clusters using an image kernel and given cluster centers. For each pixel, it finds the nearest cluster center and assigns the pixel to that cluster. Then, it constructs the cluster count array $C$ and saves the initial image of the clusters.

```
def init_cluster_c(img_kernel, centers, img_size, num_cluster
    , output_dir):
    """! Function to initialize clusters based on initial
        centers. """

    # Initialize cluster assignments
```

```
    clusters = np.zeros(img_size, dtype=int)

    # For each pixel
    for pixel in range(img_size):

        # Skip if the pixel is an initial center
        if pixel in centers:
            continue

        # Initialize minimum distance to infinity
        min_dist = np.Inf

        # For each cluster
        for cluster in range(num_cluster):

            # Get the current cluster center
            center = centers[cluster]
            temp_dist = img_kernel[pixel][center]

            # Update minimum distance and assign pixel to the
                nearest cluster
            if temp_dist < min_dist:
                min_dist = temp_dist
                clusters[pixel] = cluster

    # Construct cluster counts
    C = np.bincount(clusters, minlength=num_cluster)

    # Save initial clustering result
    save_picture(clusters, 0, num_cluster, img_size,
       output_dir)

    return clusters, C
```

***Code 10 :*** *Clusters initialization*

To accomplish this, it uses the *bincount* function from numpy to construct an array that counts the number of pixels in each cluster. It takes as input the cluster assignments for each pixel, the size of the image, and the number of clusters. It initializes a zero array CC and increments it based on the cluster assignments.

To encompass these steps, we use *init* which initializes the clusters according to the specified mode (random or K-means). For the random mode, it selects cluster centers randomly. For K-means, it chooses the centers to maximize the minimum distance between them. The clusters and the cluster count array are then initialized by calling *init_cluster_c*.

```python
def init(img_kernel, mode, img_size, num_cluster, output_dir):
    """! Function to initialize clusters based on initial centers. """

    if mode == 'random':
        centers = np.random.randint(img_size, size=num_cluster)
        clusters, C = init_cluster_c(img_kernel, centers, img_size, num_cluster, output_dir)
    elif mode == 'k-means':
        centers = np.zeros(num_cluster, dtype=int)
        centers[0] = np.random.randint(img_size, size=1)

        # For each cluster
        for i in range(1, num_cluster):
            distances = np.zeros(img_size)

            # For each pixel
            for pixel in range(img_size):
                min_dist = np.Inf

                for k in range(i):
                    temp_dist = img_kernel[centers[k]][pixel]

                    # Update minimum distance
                    if temp_dist < min_dist:
                        min_dist = temp_dist

                distances[pixel] = min_dist

            # Normalize distances
            distances = distances / np.sum(distances)

            # Select next center based on distances
            centers[i] = np.random.choice(10000, size=1, p=distances)
        clusters, C = init_cluster_c(img_kernel, centers)

    return clusters, C
```

***Code 11 :*** *Clusters initialization main function*

### 2nd step : Clustering k-means

The function *kernel_kmeans* performs Kernel K-means clustering. For each pixel, it calculates the distance to the cluster centers using the kernel values, *sigma_n*, and *sigma_pq* sums.

```
def kernel_kmeans(img_kernel, clusters, C, img_size,
    num_cluster):
    """! Perform Kernel K-means clustering. """

    # Initialization of variables
    new_clusters = np.zeros(img_size, dtype=int)          #
        New cluster
    pq = sigma_pq(img_kernel, clusters, C, num_cluster)  #

    # For each pixel
    for pixel in trange(img_size):
        distances = np.zeros(num_cluster)

        # For each cluster
        for cluster in range(num_cluster):
            distances[cluster] = img_kernel[pixel][pixel]
            distances[cluster] -= sigma_n(img_kernel[pixel,
                :], clusters, cluster, C)
            distances[cluster] += pq[cluster]

        new_clusters[pixel] = np.argmin(distances)

    # New cluster creation
    new_C = np.bincount(new_clusters, minlength=num_cluster)

    return new_clusters, new_C
```

***Code 12 :*** *Clustering k-means*

Using this two followings functions to compute sigma :

```
def sigma_n(pixel_kernel, clusters, k, C):
    """! Compute the sum of pixel_kernel values for pixels in
        cluster k. """

    # Use a boolean mask to select elements of pixel_kernel
        belonging to cluster k
    mask = clusters == k
    sum = np.sum(pixel_kernel[mask])

    return 2 / C[k] * sum
```

```python

def sigma_pq(img_kernel, clusters, C, num_cluster):
    """! Compute the sum of img_kernel values for each
       cluster, excluding inter-cluster values. """

    sum = np.zeros(num_cluster)

    # For each cluster
    for k in range(num_cluster):
        # Create a mask for pixels belonging to cluster k
        mask = clusters == k

        # Extract the submatrix of img_kernel corresponding
           to cluster k
        submatrix = img_kernel[mask][:, mask]

        # Sum the submatrix and scale it by the cluster size
           squared
        sum[k] = np.sum(submatrix) / (C[k] ** 2)

    return sum
```

***Code 13 :*** *Sigma*

**3rd step : Visualization**

Using this method, the classification was saved as a png for direct analysis in the following section. Each of the images generated highlights the different clusters detected in different colours. Also, a GIF is created to show the evolution of each iteration.

# 4 Results

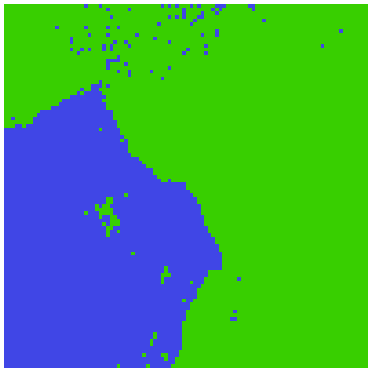
Here, all the results were obtained with a $\gamma_s$ of 0.0001 and a $\gamma_c$ of 0.0001.

## 4.1 Spectral clustering : Ratio cut

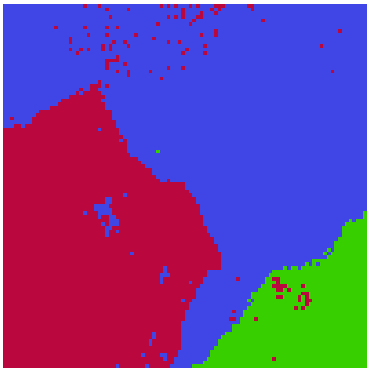
**Random :**

With this method, the results are very good and highlight the distinguished classes, delimiting the boundaries between each one. Although for image 1, having 3 or 4 clusters is fairly equivalent, there is no doubt that 3 clusters are needed to represent this image. Conversely, for image 2, 4 clusters are essential due to its complexity. This implies more information.

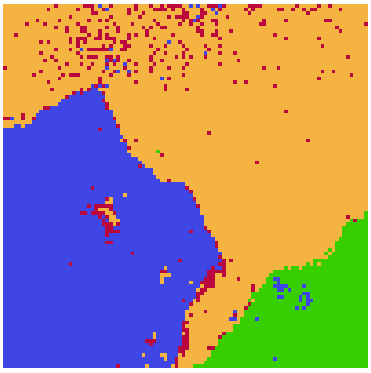
Moreover, the number of iterations for each of the images is completely different, which clearly demonstrates the difference in complexity between them.



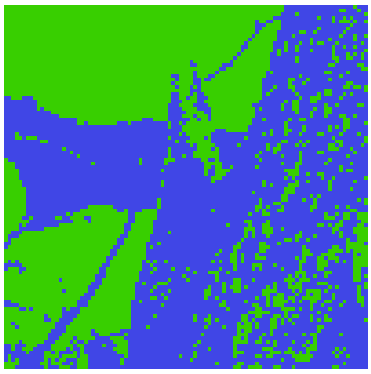
**(a)** Image 1 : 2 clusters (3 iterations)



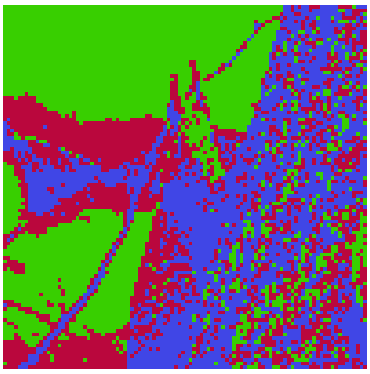
**(b)** Image 1 : 3 clusters (5 iterations)



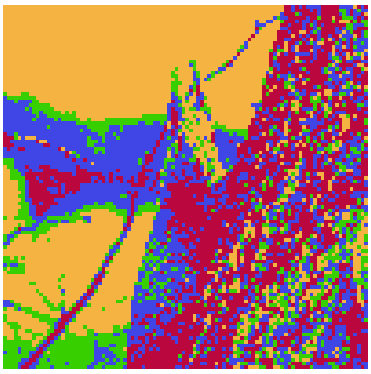
**(c)** Image 1 : 4 clusters (17 iterations)



**(d)** Image 2 : 2 clusters (10 iterations)



**(e)** Image 2 : 3 clusters (23 iterations)

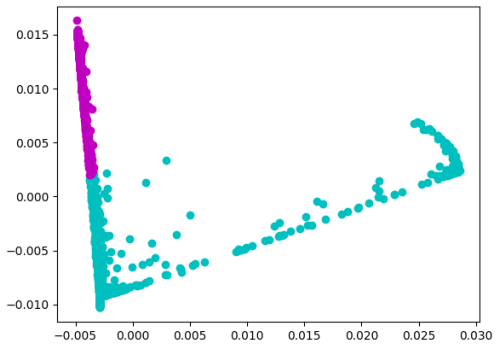


**(f)** Image 2 : 4 clusters (24 iterations)

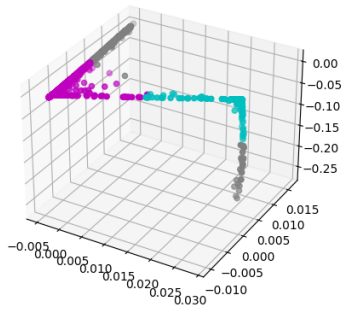
**Figure 3:** Comparison of clustering with random initialization

In terms of eigenspace, we can see for the second image that it separates fairly well to

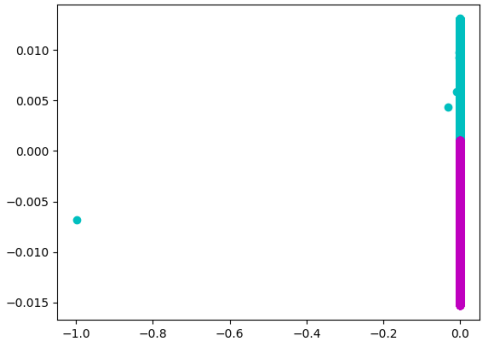
represent each class, which confirms the previous groupings. The sum in each dimension of eigenvector is 0.



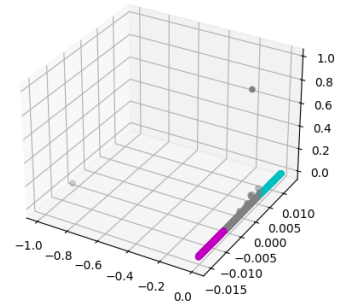

**(a)** Image 1 : 2 clusters




**(b)** Image 1 : 3 clusters




**(c)** Image 2 : 2 clusters



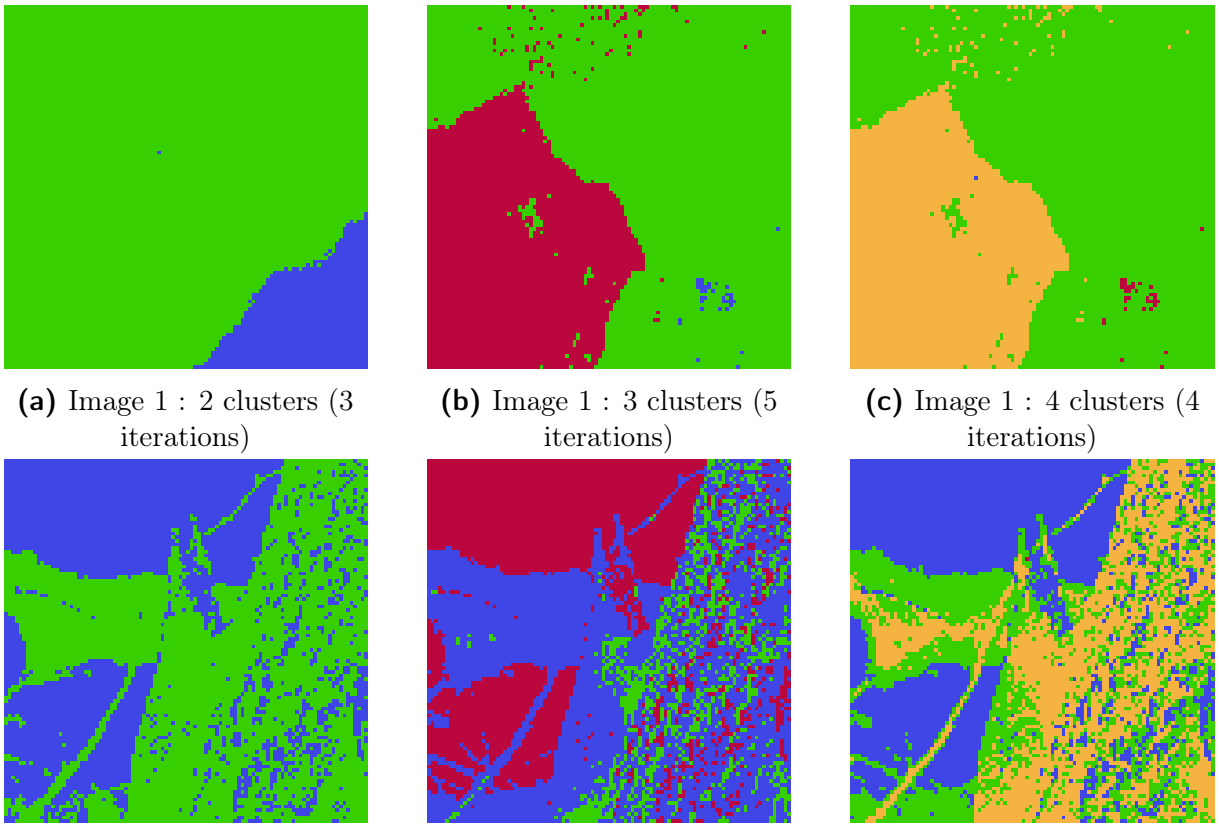

**(d)** Image 2 : 3 clusters

**Figure 4:** Comparison of eigenspace with random

**k-means++ :**

For the first image, except with 2 clusters, the results are similar to the random method, both visually and in the number of iterations required to converge.

However, for the second image, it's trickier. The 3 results are exactly the same with 2, 3, or 4 clusters because, with 3 and 4 clusters, the algorithm predicts only 1 point for each cluster with a $\gamma_c$ of 0.0001. For this reason, $\gamma_c$ has been set to 0.001 for this case. And, it's difficult to see in the 4 clusters case, but there are 4 clusters (blue, green, yellow, and purple). Purple is mainly on the trunk of the tree. However, when the eigenspaces are used, the vectors cancel out at 0.



**(a)** Image 1 : 2 clusters (3 iterations)

**(b)** Image 1 : 3 clusters (5 iterations)

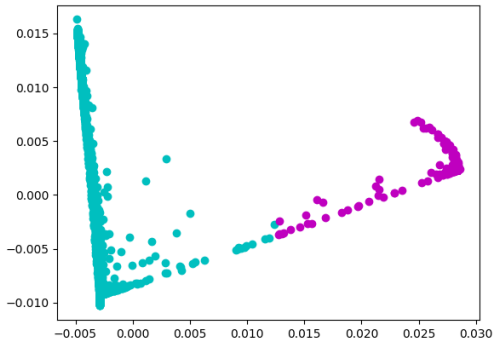
**(c)** Image 1 : 4 clusters (4 iterations)

**(d)** Image 2 : 2 cluster (8 iterations)s

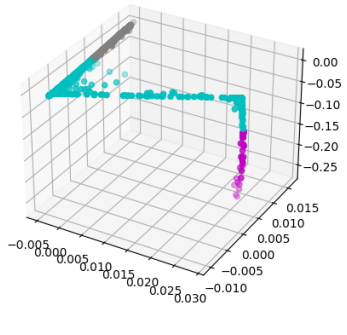
**(e)** Image 2 : 3 clusters (6 iterations)

**(f)** Image 2 : 4 clusters (11 iterations)

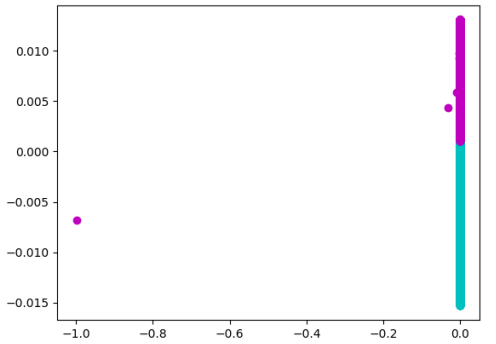
**Figure 5:** Comparison of clustering with k-means



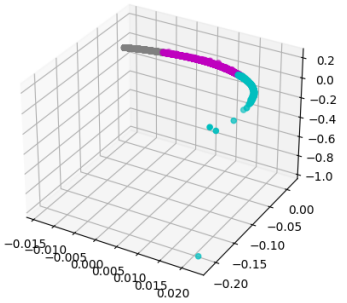

**(a)** Image 1 : 2 clusters




**(b)** Image 1 : 3 clusters




**(c)** Image 2 : 2 clusters




**(d)** Image 2 : 3 clusters

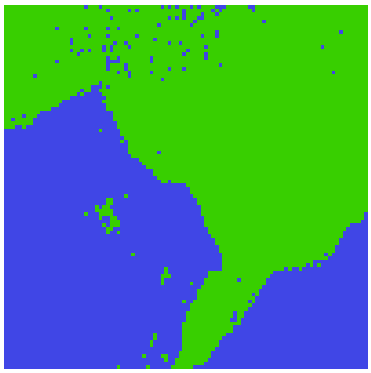
**Figure 6:** Comparison of eigenspace with k-means

## 4.2 Spectral clustering : Normalized cut

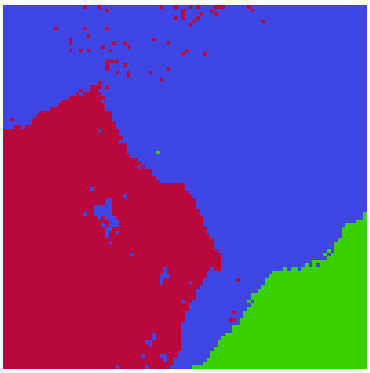
**Random :**

This time, the results are much better than with the ratio cut. Indeed, using the normalized cut seems to generate the clusters much more accurately, especially noticeable in the first image, regardless of the number of clusters. The part of the sea at the bottom right is clearly discernible from the rest, and all the details are well highlighted even though there are only 4 clusters.

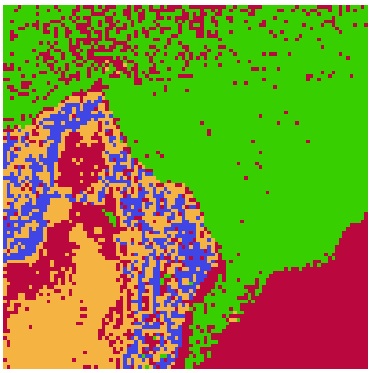
For the second image, due to its complexity, it's more challenging to determine if this method is better than the ratio cut overall. However, initially, the outlines of some shapes, especially the background in the bottom left, are not represented. Additionally, it converges slower with 4 clusters, so it probably has shifted towards overfitting.



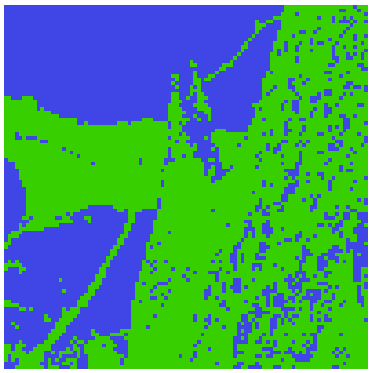
**(a)** Image 1 : 2 clusters (6 iterations)



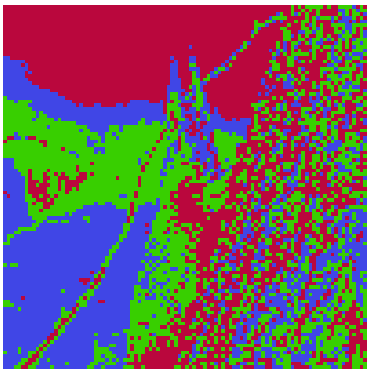
**(b)** Image 1 : 3 clusters (6 iterations)



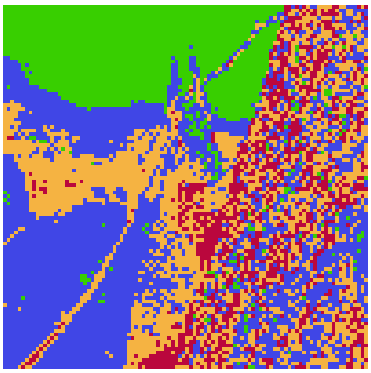
**(c)** Image 1 : 4 clusters (17 iterations)



**(d)** Image 2 : 2 clusters (8 iterations)

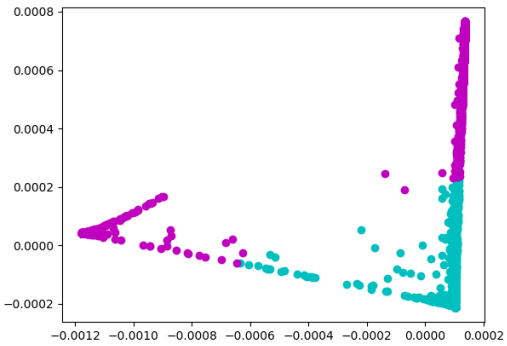


**(e)** Image 2 : 3 clusters (15 iterations)

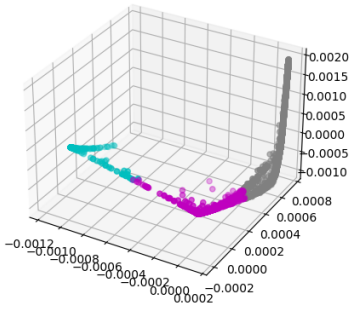


**(f)** Image 2 : 4 clusters (32 iterations)

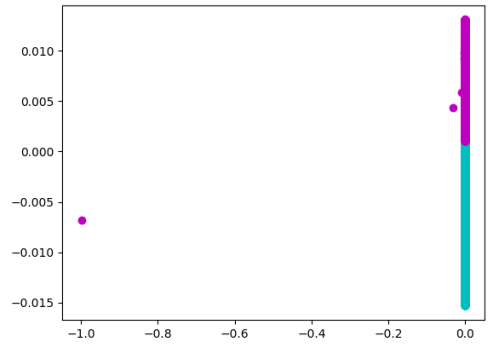
**Figure 7:** Comparison of clustering with random



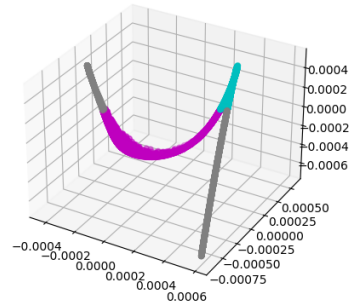

**(a)** Image 1 : 2 clusters




**(b)** Image 1 : 3 clusters




**(c)** Image 2 : 2 clusters

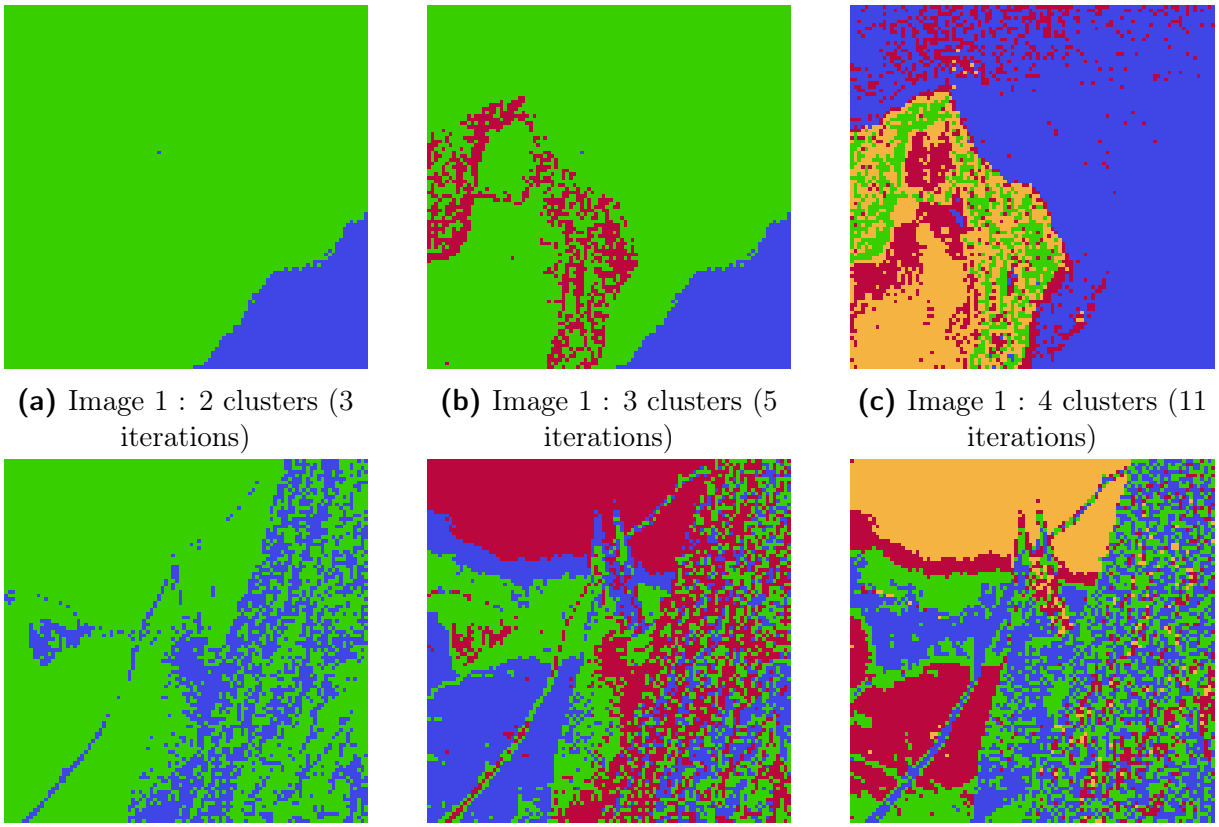



**(d)** Image 2 : 3 clusters

**Figure 8:** Comparison of eigenspace with random

**k-means++ :**

Unlike the random method, to obtain relevant results with k-means, it is better to increase the number of clusters. This is probably because it converges faster. However, for the first image, with 4 clusters, even though the sea at the bottom right is not distinguished, on the mainland, we can see many different shapes, but still fewer than with random. On the other hand, for image 2, the results are better with this method. The contours of the shapes in the background of the image are distinguishable. However, the complexity of this image poses a problem again, especially in the foreground with the squirrel and the tree trunk.



**(a)** Image 1 : 2 clusters (3 iterations)

**(b)** Image 1 : 3 clusters (5 iterations)

**(c)** Image 1 : 4 clusters (11 iterations)

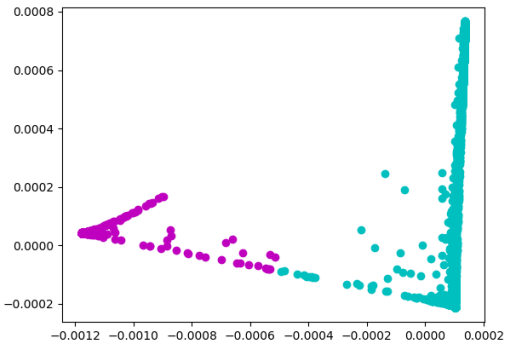
**(d)** Image 2 : 2 clusters (9 iterations)

**(e)** Image 2 : 3 clusters (14 iterations)

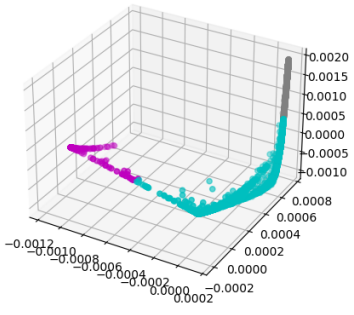
**(f)** Image 2 : 4 clusters (32 iterations)

**Figure 9:** Comparison of clustering with k-means

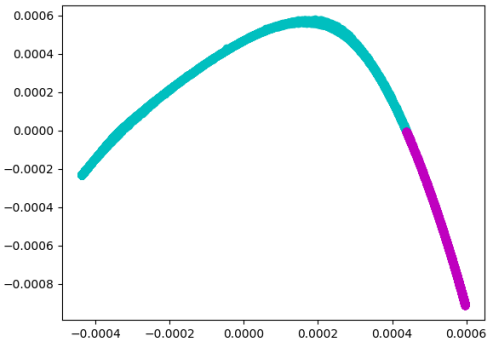
Unlike random initialization, for image 1, the clustering result in eigenspace is excellent.



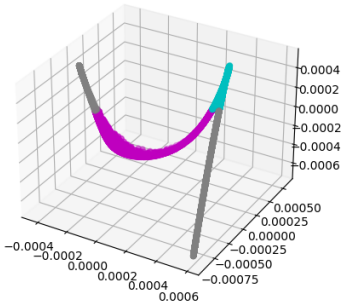

**(a)** Image 1 : 2 clusters




**(b)** Image 1 : 3 clusters




**(c)** Image 2 : 2 clusters




**(d)** Image 2 : 3 clusters

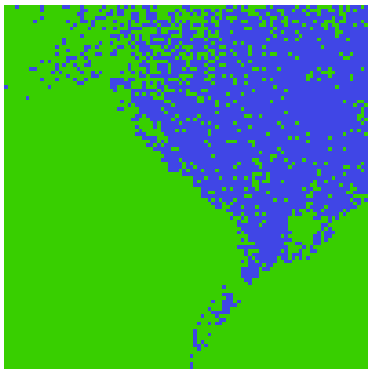
**Figure 10:** Comparison of eigenspace with k-means

## 4.3 K-means clustering

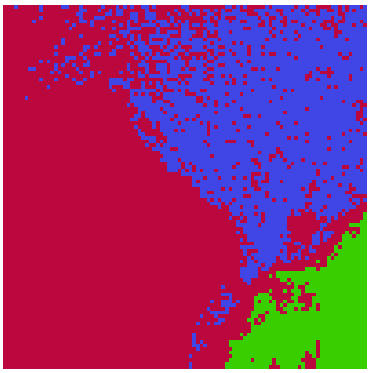
**Random :**

The results obtained by this method are mixed. With a low number of clusters, the results are correct, but when the number of clusters increases, they are worse than the previous methods. For both images, with 3 clusters, the red space is overrepresented to the point of containing only red points.

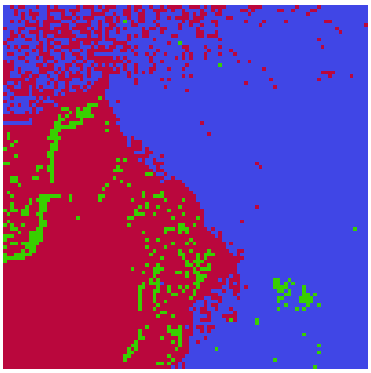
However, for the cases with 4 clusters, the method does not converge, even with parameters set to 0.01. Therefore, only the first iterations of each are shown here. From the second iteration, the generated image was monochrome.



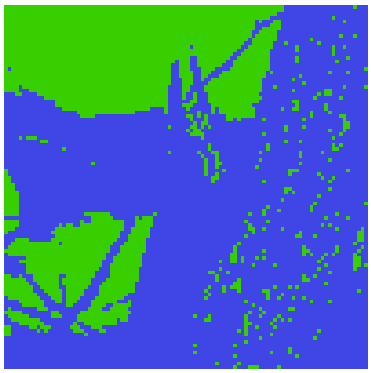
**(a)** Image 1 : 2 clusters (20 iterations)



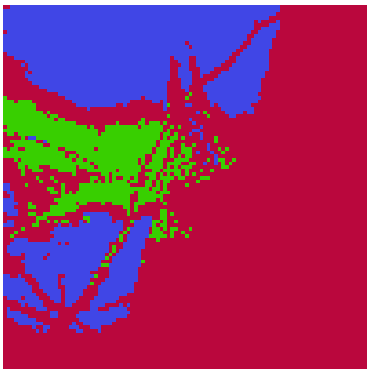
**(b)** Image 1 : 3 clusters (16 iterations)



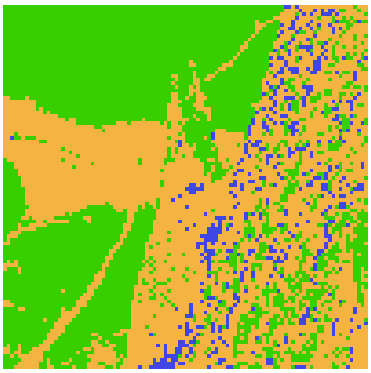
**(c)** Image 1 : 4 clusters (1 iteration)



**(d)** Image 2 : 2 clusters (15 iterations)



**(e)** Image 2 : 3 clusters (29 iterations)

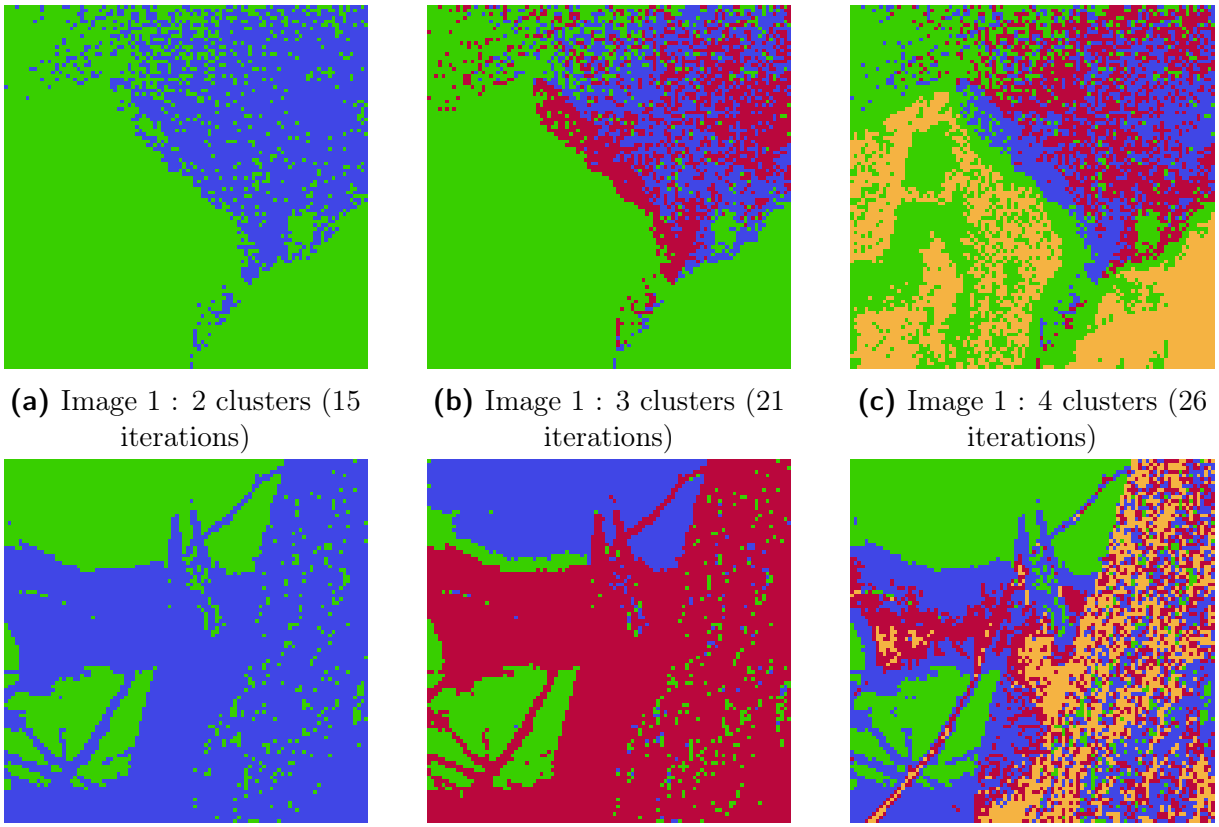


**(f)** Image 2 : 4 clusters (1 iteration

**Figure 11:** Comparison of clustering with random

**k-means++ :**

It is surprising to observe that k-means has a worse result than random for 2 and 3 clusters on image 1 and for 2 clusters on image 2. But for 4 clusters, the results are much better. We can easily distinguish a squirrel in the second image, and in the first image, the relief and the borders of the elements are greatly emphasized.



**(a)** Image 1 : 2 clusters (15 iterations)

**(b)** Image 1 : 3 clusters (21 iterations)

**(c)** Image 1 : 4 clusters (26 iterations)

**(d)** Image 2 : 2 clusters (6 iterations)

**(e)** Image 2 : 3 clusters (20 iterations)

**(f)** Image 2 : 4 clusters (34 iterations)

**Figure 12:** Comparison of clustering with k-means

# 5 Conclusion

In this study, we explored and compared different clustering methods applied to images, highlighting their performance and respective limitations. The comparative analysis between the ratio cut and normalized cut methods revealed significant differences in the accuracy and relevance of the generated clusters.

Spectral clustering yields better results than kernel k-means. Specifically, the use of normalized cut proved to be more effective in generating precise clusters. This can be attributed to the fact that we do not update the cluster means in kernel k-means. It relies more on the initial state, and regardless of the initialization method, it contains random properties. This is why the clustering results are different each time we run the program, even if all hyperparameters are identical, due to different initial states.

This is particularly evident in the first image, where the details and boundaries of the elements are better highlighted. However, for the more complex second image, although the normalized cut showed promising initial results, it tended to converge more slowly and exhibited signs of overfitting with a higher number of clusters. Therefore, hyperparameters are very important. If we set incorrect hyperparameters, the result can be absurd. We even need to vary them to achieve results (avoiding division by zero).

On the other hand, the k-means method, although it converges faster, requires a higher number of clusters to obtain relevant results, which can be a disadvantage depending on the complexity of the analyzed images. For image 1, although the results are less detailed than with the random method, increasing the number of clusters revealed more distinct shapes. For image 2, k-means showed superior results by clearly distinguishing the contours of the shapes in the background, despite the increased complexity of the image.

In conclusion, each clustering method has its advantages and disadvantages depending on the context and complexity of the images being processed. The choice of the most appropriate method depends on the specifics of the image and the objectives of the analysis. The normalized cut method offers better accuracy for less complex images, while k-means is more effective for complex images requiring detailed classification. A thorough understanding of the characteristics of each method thus allows for the optimization of the clustering process for various applications in image processing.