

COERCHON_Colin-spectral_clustering

September 6, 2026

1 Spectral Clustering

In this exercise, you will implement spectral clustering and try it on different kinds of data, for instance:

- N-blob: Random points in $\mathbb{R}^2$ according to N Gaussian distributions with different means;
- Two moons: Random points shaped as two intertwined moons;
- Point and circle: random points from a concentrated Gaussian point in the middle and a wide circle around it.

The first step after generating the data is to construct a graph and, for that, we need a similarity function. In this exercise we will use the following similarity:

$$d(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right)$$

Using this similarity, we can compute the weight matrix W representing the graph, compute the graph Laplacian L and finally use its eigenvectors for clustering.

1.1 Setup

```
[1]: from generate_data import worst_case_blob, blobs, two_moons, point_and_circle
from utils import plot_clusters, min_span_tree, plot_graph_matrix, min_span_tree
from utils import plot_clustering_result, plot_the_bend, plot_clusters, \
    plot_edges_and_points
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, confusion_matrix
from scipy.optimize import linear_sum_assignment
import sklearn.metrics as skm
import numpy as np
from skimage import io
import numpy as np
import matplotlib.pyplot as plt
import os
import scipy

import warnings
```

```
warnings.filterwarnings("ignore", category=UserWarning) # To avoid KMeans_
↳ warnings

def compute_best_accuracy(Y_true, Y_pred):
    """ Finds the optimal permutation of labels to maximize accuracy. """
    cm = confusion_matrix(Y_true, Y_pred)
    row_ind, col_ind = linear_sum_assignment(-cm) # Maximizing the diagonal
    Y_pred_aligned = np.zeros_like(Y_pred)

    for i, j in zip(row_ind, col_ind):
        Y_pred_aligned[Y_pred == j] = i # Reassigning labels

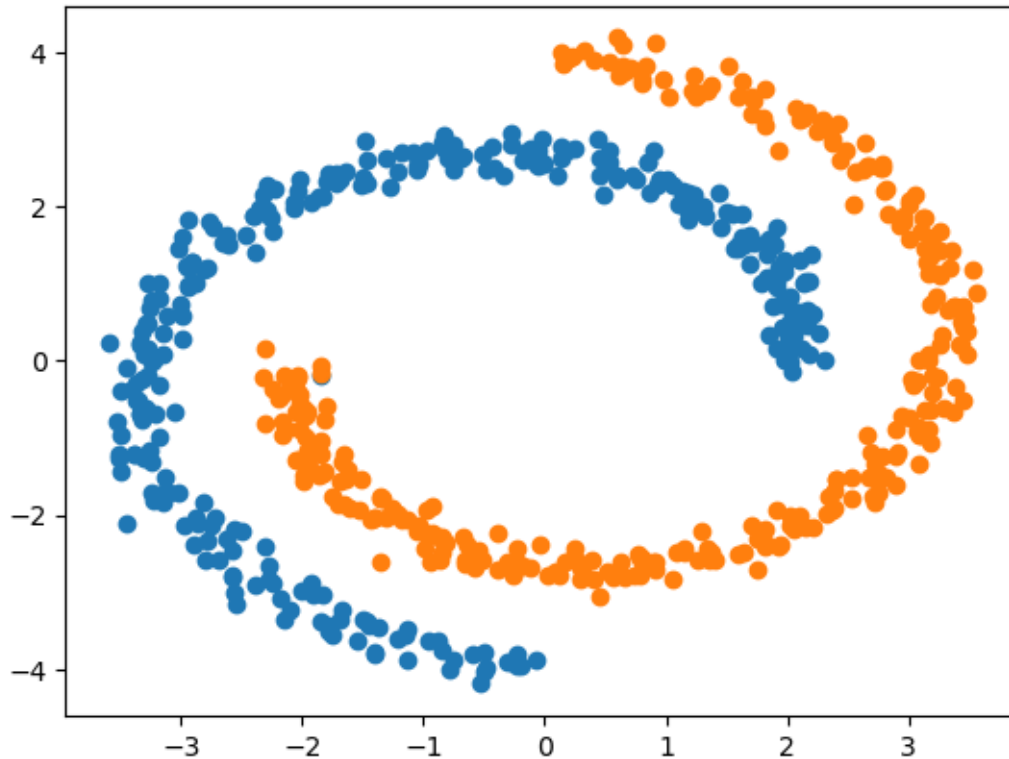
    return accuracy_score(Y_true, Y_pred_aligned)
```

1.2 Generating the data

```
[2]: # Example

for fn in [two_moons]: # try also [worst_case_blob, point_and_circle, blobs,
↳ two_moons]
    X, Y = fn(num_samples=600)
    plot_clusters(X, Y, fignum=str(fn))

# You can check the documentation of each function to see the optional_
↳ parameters, e.g.:
# worst_case_blob?
```



1.3 Building similarity graph

```
[3]: def build_similarity_graph(X, var=1.0, eps=0.0, k=0, knn_mode="OR"):
    """
    Computes the similarity matrix for a given dataset of samples.
    If k=0, builds epsilon graph.
    Otherwise, builds kNN graph.

    Parameters
    -----
    X : numpy array
        (n x m) matrix of m-dimensional samples
    var : double
        the sigma value for the exponential function, already squared
    eps : double
        threshold for epsilon graphs
    k : int
        The number of neighbours k for k-nn. If zero, use epsilon-graph

    Returns
    -----
    W: (n x n) dimensional matrix representing the weight matrix of the graph
```

```

"""
n = X.shape[0]
W = np.zeros((n, n))

"""
Build similarity graph, before threshold or kNN
similarities: (n x n) matrix with similarities between all possible couples
of points.
The similarity function is  $d(x,y)=\exp(-||x-y||^2/(2*var))$ 
"""

similarities = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        if i != j:
            dist_squared = np.sum((X[i] - X[j]) ** 2)
            similarities[i, j] = np.exp(-dist_squared / (2 * var))

# If epsilon graph
if k == 0:
    """
    compute an epsilon graph from the similarities
    for each node  $x_i$ , an epsilon graph has weights
     $w_{ij} = d(x_i, x_j)$  when  $w_{ij} \geq \epsilon$ , and 0 otherwise
    """
    W[similarities >= eps] = similarities[similarities >= eps] # Keep
weights above epsilon threshold
    pass

# If kNN graph
elif k != 0:
    """
    compute a k-nn graph from the similarities
    for each node  $x_i$ , a k-nn graph has weights
     $w_{ij} = d(x_i, x_j)$  for the k closest nodes to  $x_i$ , and 0
    for all the k-n remaining nodes
    Remember to remove self similarity and
    make the graph undirected
    """
    knn_indices = np.argsort(-similarities, axis=1)[: , 1:k+1] # Get indices
of k largest values excluding self-similarity

    if knn_mode == "OR":
        for i in range(n):
            for j in knn_indices[i]:
                W[i, j] = similarities[i, j] # Keep the similarity
    elif knn_mode == "AND":

```

```

        for i in range(n):
            for j in knn_indices[i]:
                if i in knn_indices[j]:
                    W[i, j] = similarities[i, j] # Keep the similarity if  $\rightarrow$  mutual

    # Make the graph undirected by symmetrizing the matrix
    W = np.maximum(W, W.T)
    pass

    return W

```

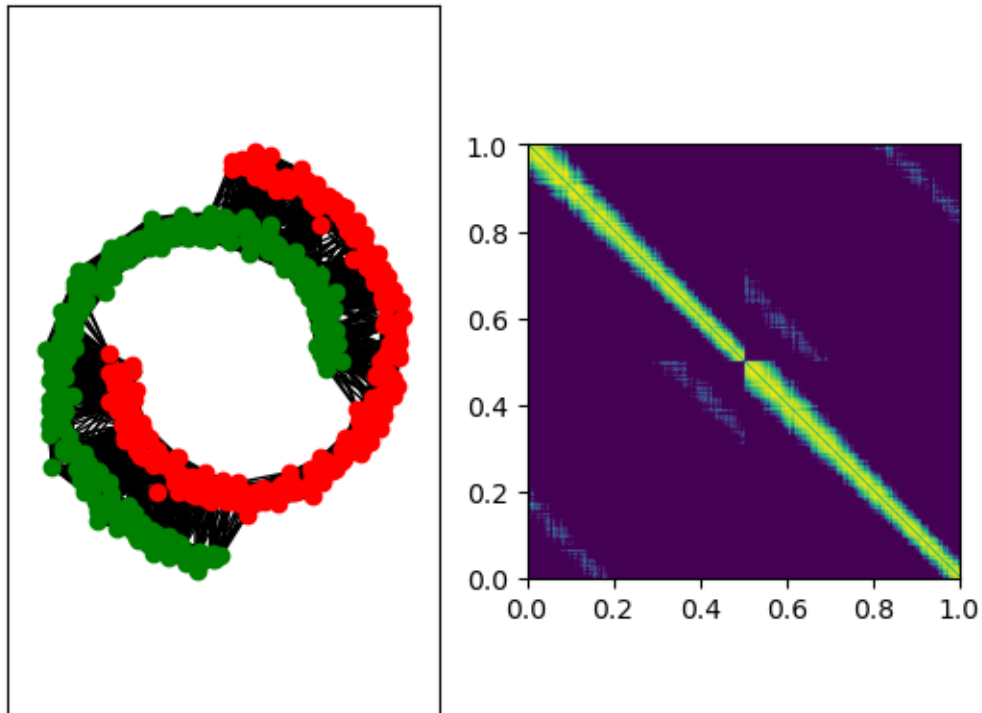
[4]: # Example

```

W = build_similarity_graph(X, eps=0.5, k=0, knn_mode="OR")
# W[0, 599] = 1 # add two placeholder edges to reproduce example picture
# W[10, 500] = 1 #

plot_graph_matrix(X, Y, W) # shows the graph and the visualization of W

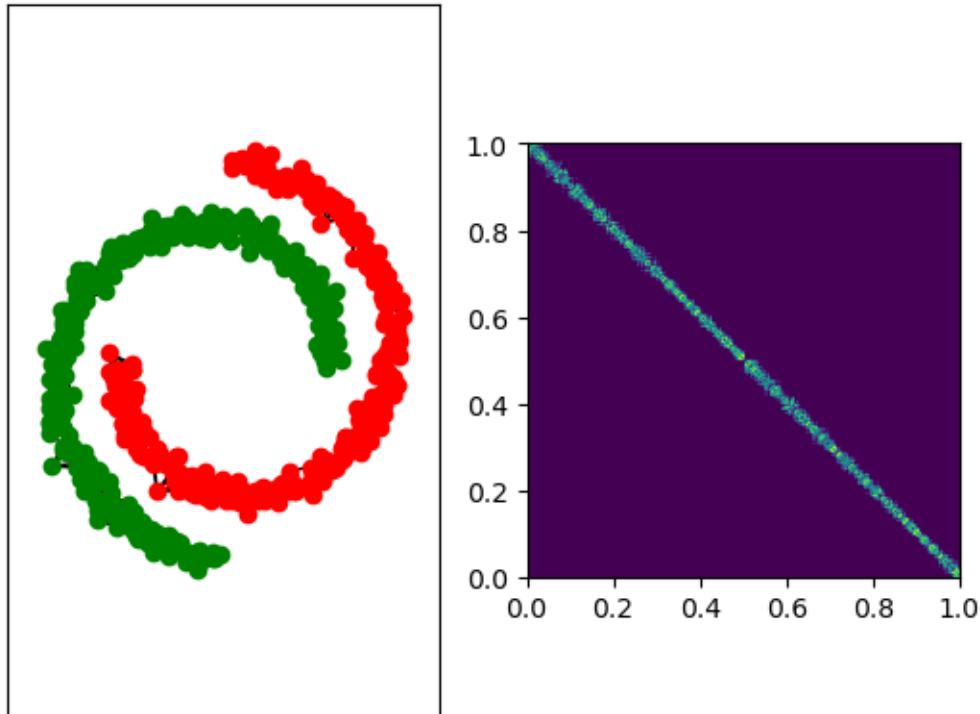
```



```
[5]: # Example

W = build_similarity_graph(X, eps=0, k=8, knn_mode="OR")
# W[0, 599] = 1    # add two placeholder edges to reproduce example picture
# W[10, 500] = 1   #

plot_graph_matrix(X, Y, W) # shows the graph and the visualization of W
```



1.4 Computing the Laplacian

```
[6]: def build_laplacian(W, laplacian_normalization='unn'):
    """
    Compute graph Laplacian.

    Parameters
    -----
    W : numpy array
        Adjacency matrix (n x n)
    laplacian_normalization : str
        String selecting which version of the laplacian matrix to construct.
        'unn': unnormalized,
        'sym': symmetric normalization
```

```

        'rw': random-walk normalization

Returns
-----
L: (n x n) dimensional matrix representing the Laplacian of the graph
"""
L = np.zeros(W.shape)

if laplacian_normalization == 'unn':
    """
    Compute the unnormalized Laplacian.
     $L = D - W$ 
    """
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    pass

elif laplacian_normalization == 'sym':
    """
    Compute the symmetric Laplacian.
     $D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2}$ 
    """
    D = np.diag(np.sum(W, axis=1))
    D_sqrt = np.diag(1 / np.sqrt(np.sum(W, axis=1) + 1e-10))
    L = np.eye(W.shape[0]) - D_sqrt @ W @ D_sqrt
    pass

elif laplacian_normalization == 'rw':
    """
    Compute the random-walk Laplacian.
     $D^{-1} L = I - D^{-1} W$ 
    """
    D = np.diag(np.sum(W, axis=1))
    D_inv = np.diag(1 / (np.sum(W, axis=1) + 1e-10))
    L = np.eye(W.shape[0]) - D_inv @ W
    pass

# for example (unnormalized): L = D - W
return L

```

1.5 Clustering

```

[7]: def spectral_clustering(L, chosen_eig_indices=None, num_classes=2,
    ↪ show_valeurs_propres=False, show_vecteurs_propres=False):
    """
    Parameters
    -----

```

```

L : numpy array
    Graph Laplacian (standard or normalized)
chosen_eig_indices : list or None
    Indices of eigenvectors to use for clustering.
    If None, use an adaptive choice of eigenvectors.
num_classes : int
    Number of clusters to compute (defaults to 2)
show : bool
    If True, displays eigenvalues and the first two eigenvectors.

Returns
-----
Y : numpy array (num_samples, )
    Cluster assignments
"""

"""
Use the function scipy.linalg.eig or scipy.sparse.linalg.eigs to compute:
U = (n x n) eigenvector matrix          (sorted)
E = (n x n) eigenvalue diagonal matrix  (sorted)
"""
E, U = scipy.linalg.eig(L)

idx = np.argsort(np.real(E))
E = np.real(E[idx]) # Ensuring real eigenvalues
U = np.real(U[:, idx]) # Ensuring real eigenvectors

if chosen_eig_indices is None:
    # Compute the gap between successive eigenvalues
    gaps = np.diff(E[:num_classes + 5])

    # If the first jump is between eigenvalue 0 and 1, find the next largest
    ↪ jump
    if np.argmax(gaps) == 0:
        sorted_indices = np.argsort(gaps)[::-1]
        best_index = sorted_indices[1]
    else:
        best_index = np.argmax(gaps)

    chosen_eig_indices = np.arange(0, best_index + 1)

U_selected = U[:, chosen_eig_indices]

if show_valeurs_propres:
    plt.figure(figsize=(8, 5))
    plt.plot(range(10), E[:10], 'bo-', markersize=6, label="First 10
    ↪ eigenvalues")

```

```

        last_eig_index = chosen_eig_indices[-1]
        plt.axvline(x=last_eig_index+0.5, color='r', linestyle='--',
→label="Selected eigenvalue threshold")
        plt.xlabel("Eigenvalue index")
        plt.ylabel("Eigenvalue")
        plt.title("First 10 sorted Laplacian eigenvalues")
        plt.legend()
        plt.grid(True)
        plt.show()
    if show_vecteurs_propres:
        plt.figure(figsize=(8, 5))
        plt.plot(U[:, 0], label="1st eigenvector", linestyle='-', marker='o')
        plt.plot(U[:, 1], label="2nd eigenvector", linestyle='-', marker='s',
→alpha=0.8)
        plt.xlabel("Node index")
        plt.ylabel("Eigenvector value")
        plt.title("Evolution of the first two eigenvectors")
        plt.legend()
        plt.grid(True)
        plt.show()

    """
    Compute the clustering assignment from the eigenvectors
    Y = (n x 1) cluster assignments in [0,1,...,num_classes-1]
    """
    kmeans = KMeans(n_clusters=num_classes)
    Y = kmeans.fit_predict(U_selected)

    return Y

```

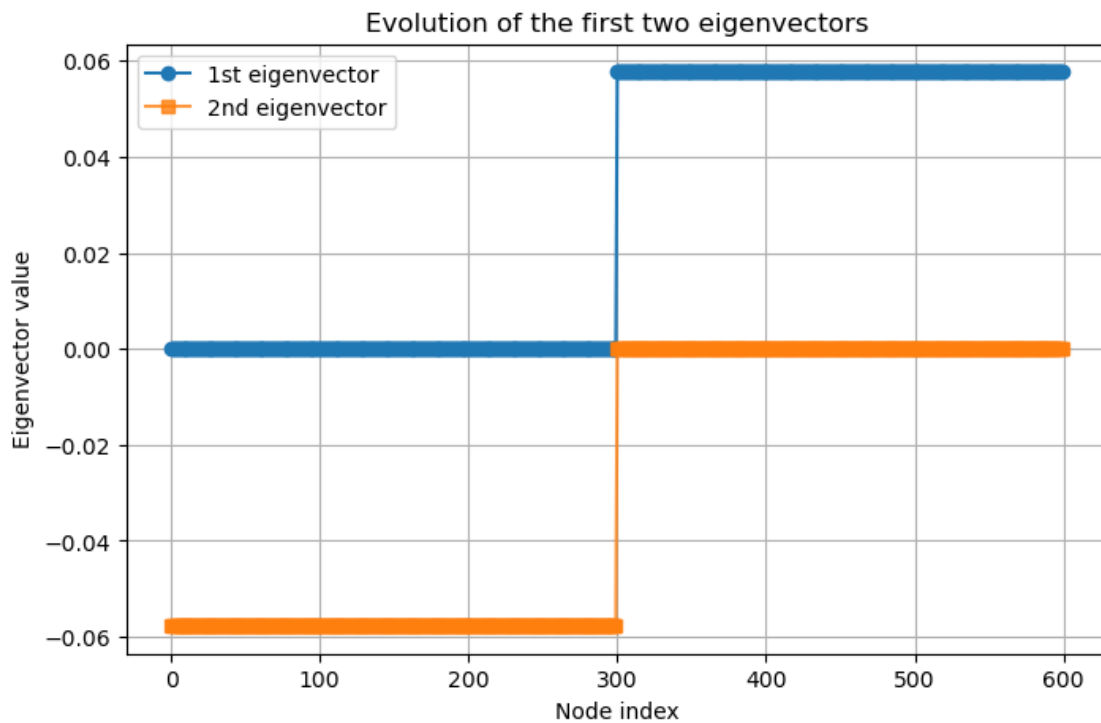
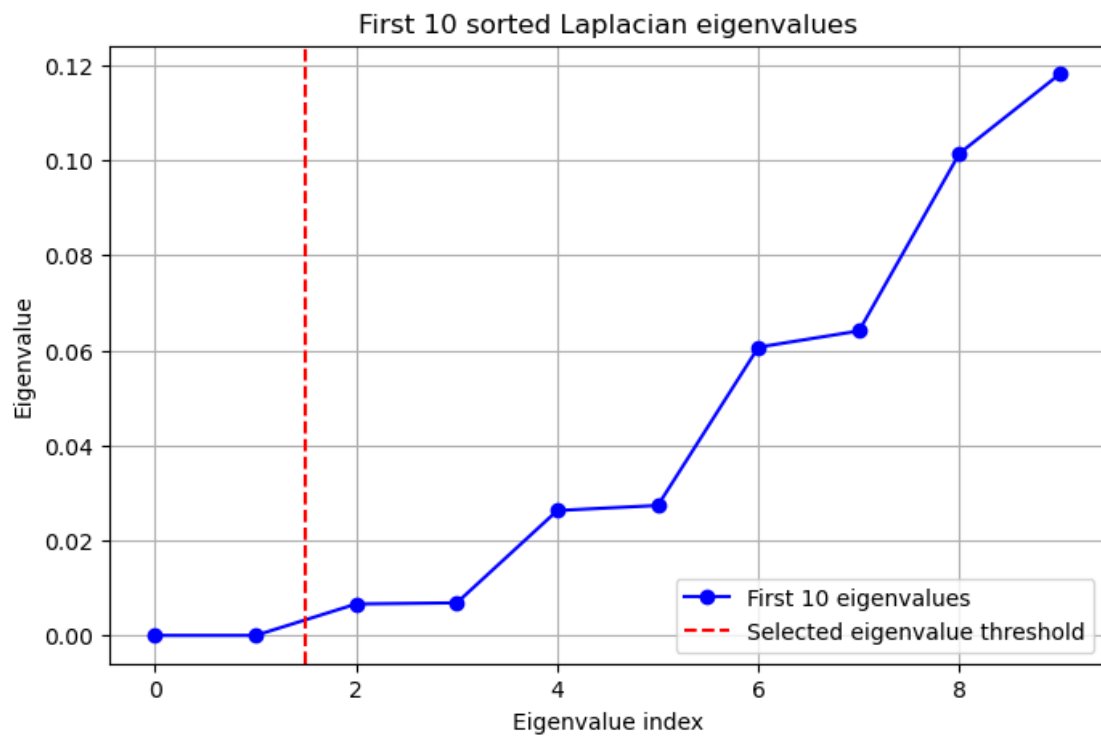
```

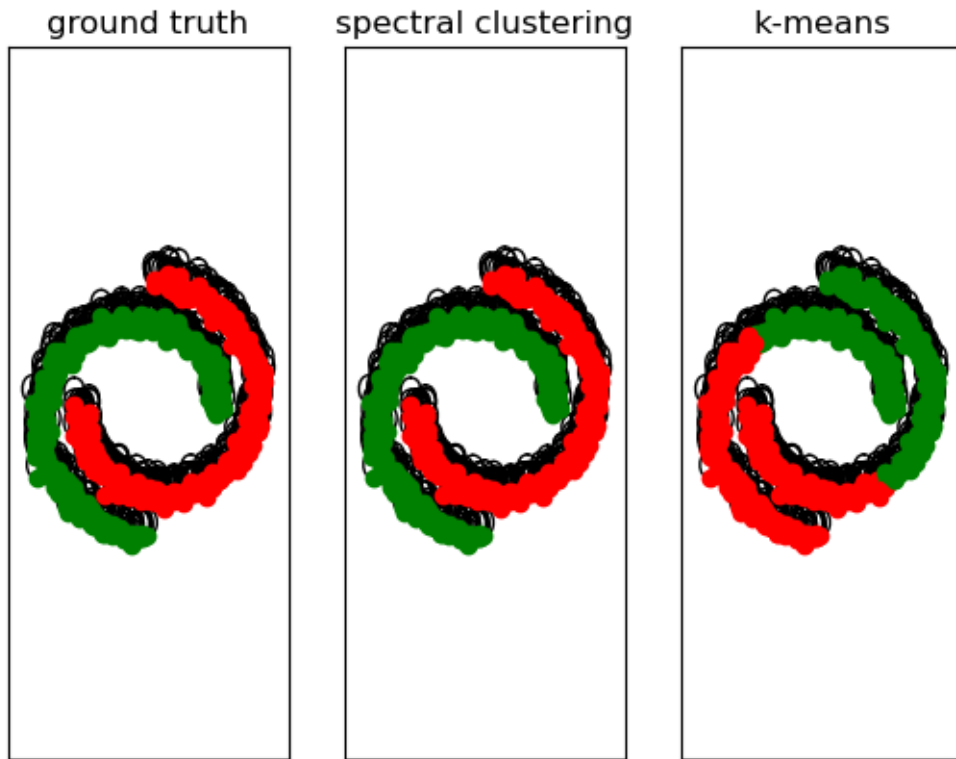
[8]: # Example

num_classes = 2
L = build_laplacian(W)
Y_rec = spectral_clustering(L, chosen_eig_indices=[0, 1],
→num_classes=num_classes, show_valeurs_propres=True, show_vecteurs_propres=True)

# Plot results, comparing to KMeans
plot_clustering_result(X, Y, L, Y_rec, KMeans(num_classes).fit_predict(X))

```





1.6 Parameter sensitivity

```
[9]: def parameter_sensitivity():
    """
    TO BE COMPLETED.

    A function to test spectral clustering sensitivity to parameter choice.
    """
    # The number of samples to generate
    num_samples = 500

    """
    Choose parameters
    """
    var = 1.0 # exponential_euclidean's sigma^2
    laplacian_normalization = 'unn'
    chosen_eig_indices = None

    """
    Choose candidate parameters
    """
```

```

print("\nTesting for the k-NN graph\n")

parameter_type = 'k' # Choose between 'k' or 'eps'
parameter_candidate = [1, 5, 10, 15, 20, 25, 30, 35, 40, 50, 60, 80, 100,
↪150, 200]
parameter_performance = []

for parameter in parameter_candidate:
    # print("Testing parameter k = ", parameter)
    # Generate data
    X, Y = two_moons(num_samples, 1, 0.02)
    num_classes = len(np.unique(Y))

    parameter_wrap = {parameter_type: parameter}

    W = build_similarity_graph(X, **parameter_wrap)
    L = build_laplacian(W, laplacian_normalization)

    Y_rec = spectral_clustering(L, chosen_eig_indices, num_classes)

    parameter_performance += [skm.adjusted_rand_score(Y, Y_rec)]

plt.figure()
plt.plot(parameter_candidate, parameter_performance)
plt.title(f'Parameter sensitivity ({parameter_type})')
plt.show()

print("\nTesting for the epsilon-graph\n")

parameter_type = 'eps' # Choose between 'k' or 'eps'
parameter_candidate = [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.
↪9, 1, 2, 3]
parameter_performance = []

for parameter in parameter_candidate:
    # print("Testing parameter epsilon = ", parameter)
    # Generate data
    X, Y = two_moons(num_samples, 1, 0.02)
    num_classes = len(np.unique(Y))

    parameter_wrap = {parameter_type: parameter}

    W = build_similarity_graph(X, **parameter_wrap)
    L = build_laplacian(W, laplacian_normalization)

    Y_rec = spectral_clustering(L, chosen_eig_indices, num_classes)

```

```

        parameter_performance += [skm.adjusted_rand_score(Y, Y_rec)]

plt.figure()
plt.plot(parameter_candidate, parameter_performance)
plt.title(f'Parameter sensitivity ({parameter_type})')
plt.show()

```

1.7 Image segmentation

```

[10]: def image_segmentation(input_img='four_elements.bmp', num_classes=2, k=10,
    ↪var=1, laplacian_normalization='unn', chosen_eig_indices=None):
    """
    Function to perform image segmentation.

    :param input_img: Nom du fichier image dans le dossier /data
    :param num_classes: Nombre de segments souhaités
    :param k: Nombre de voisins pour le graphe k-NN
    :param var: Paramètre de variance pour la similarité
    :param laplacian_normalization: Type de normalisation du Laplacien ('unn',
    ↪'sym', 'rw')
    """

    filename = os.path.join('data', input_img)

    X = io.imread(filename)
    X = (X - np.min(X)) / (np.max(X) - np.min(X))

    im_side = np.size(X, 1)
    Xr = X.reshape(im_side ** 2, 3)
    """
    Y_rec should contain an index from 0 to c-1 where c is the
    number of segments you want to split the image into
    """

    """
    Choose parameters
    """

    W = build_similarity_graph(Xr, var=var, k=k)
    L = build_laplacian(W, laplacian_normalization)
    Y_rec = spectral_clustering(L, chosen_eig_indices=chosen_eig_indices,
    ↪num_classes=num_classes, show_valeurs_propres=True)

    plt.figure()

    plt.subplot(1, 2, 1)

```

```
plt.imshow(X)

plt.subplot(1, 2, 2)
Y_rec = Y_rec.reshape(im_side, im_side)
plt.imshow(Y_rec)

plt.show()
```

```
[11]: # image_segmentation()
```

2 Questions

2.0.1 Q1.1: What is the purpose of the option parameter in `worst_case_blob`?

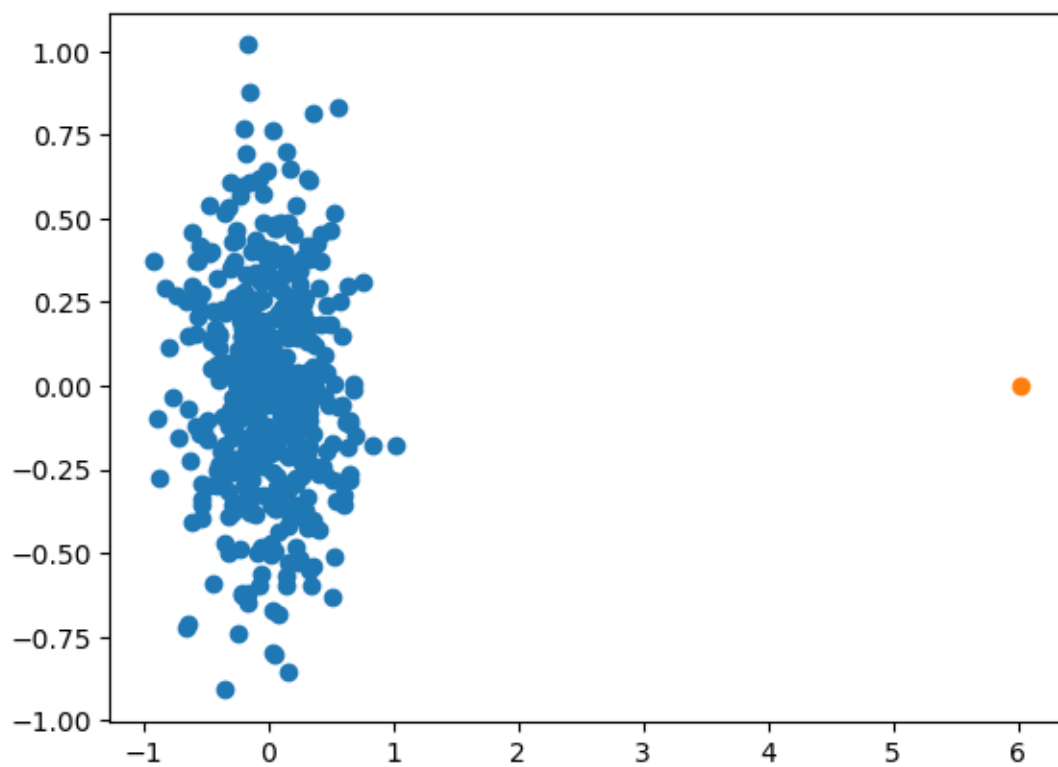
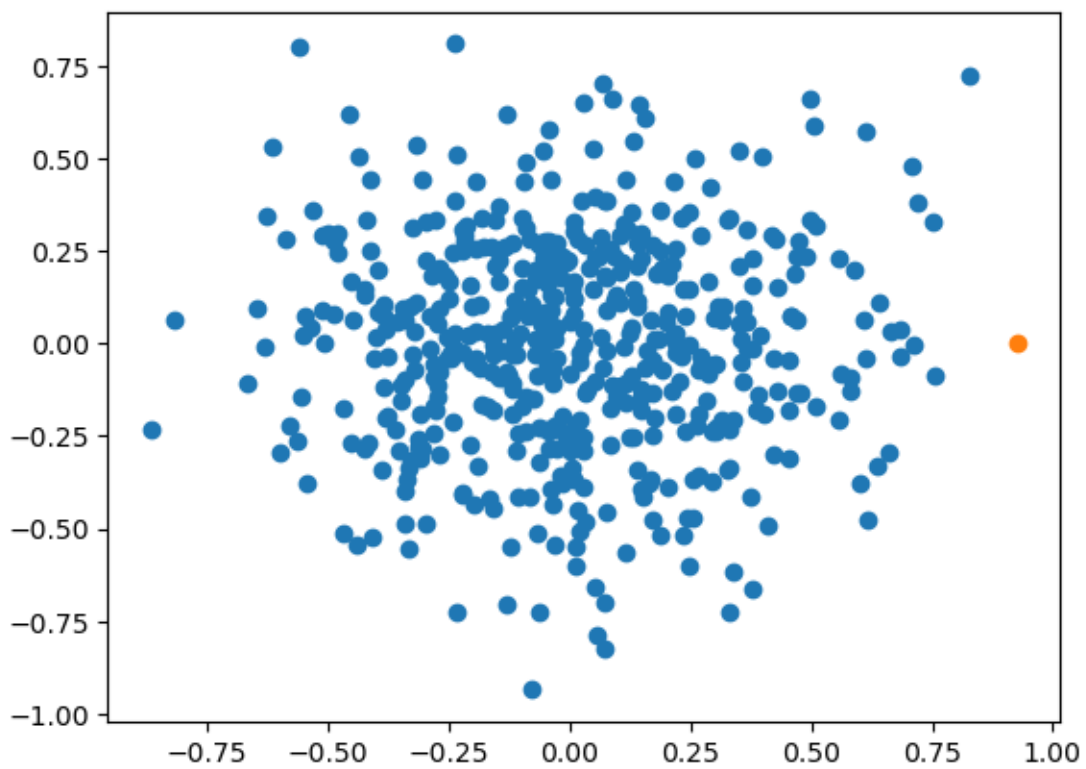
```
[12]: delta = [0.1, 5, 100] # By default, delta = 5 in the worst_case_blob code
      for i in range(len(delta)):

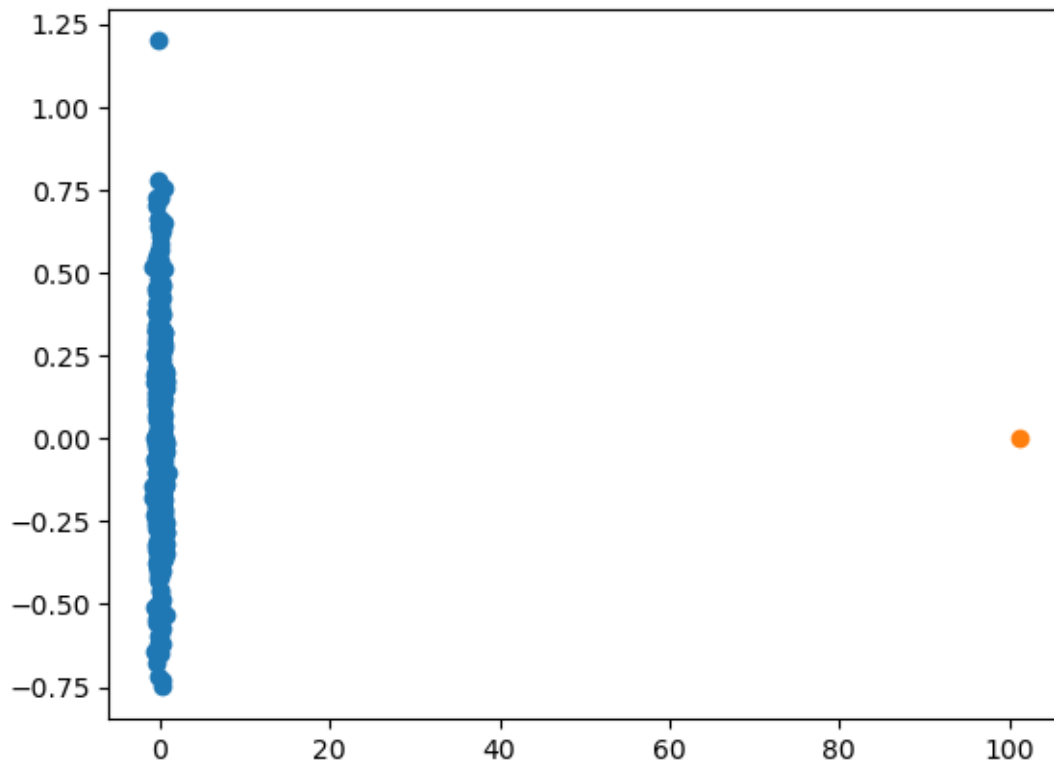
          X, Y = worst_case_blob(500, delta[i])
          Y[-1] = 1
          print(X[-1])

          plot_clusters(X, Y, fignum='delta = ' + str(delta[i]))

      plt.show()
```

```
[0.92597236 0.      ]
[6.01917997 0.      ]
[101.19862038 0.      ]
```





The parameter `delta` in `worst_case_blob` controls the dispersion of points within a square $([-1, 1]^2)$.

- **Large delta** → Points are more spread out.
- **Small delta** → Points are clustered around the segment $(\{(x, y) \in \mathbb{R} \mid x = 0, y \in [-1, 1]\})$.
- An isolated point is always positioned to the right of all others (outside the square).

This affects the difficulty of clustering by creating a “worst-case” scenario where a point is far from the center of mass of all other points.

2.0.2 Q1.2: While varying the parameter of `worst_case_blob`, try to select a value of ϵ that keeps the graph connected (the function `min_span_tree` can help you). What happens? Can you tell when it is a bad idea to use this approach?

```
[13]: delta_values = [0.01, 1, 5]
      num_samples = 500

      for delta in delta_values:
          print(f"- For delta = {delta}")

          X, Y = worst_case_blob(num_samples, delta=delta)
```

```

W = build_similarity_graph(X, eps=0.0)

T = W * min_span_tree(-W)  # Apply the MST mask on W (since the MST is
→binary)

# Find epsilon as the smallest nonzero edge in the MST
eps_opt = np.min(T[np.nonzero(T)])

print(f" Optimal epsilon = {eps_opt:.6f}")

# Build the epsilon graph using this epsilon
W_eps = build_similarity_graph(X, eps=eps_opt)

# Number of edges before and after thresholding
n_edges_full = np.sum(W > 0) // 2
n_edges_eps = np.sum(W_eps > 0) // 2
percent_removed = 100 * (n_edges_full - n_edges_eps) / n_edges_full

print(f" Number of edges (full graph): {n_edges_full}")
print(f" Number of edges (epsilon graph): {n_edges_eps}")
print(f" Percentage of removed edges: {percent_removed:.2f} %")

fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].imshow(W, cmap="hot")
ax[0].set_title("Heatmap of W (full graph)")
ax[1].imshow(W_eps, cmap="hot")
ax[1].set_title(f"Heatmap of W_eps (epsilon={eps_opt:.3f})")
plt.show()

fig, ax = plt.subplots(1, 2, figsize=(10, 4))

plt.sca(ax[0])
plot_edges_and_points(X, Y, W, title="Before thresholding")
plt.sca(ax[1])
plot_edges_and_points(X, Y, W_eps, title="After thresholding")

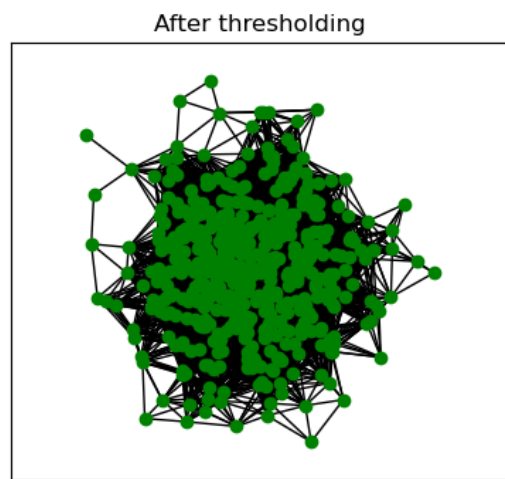
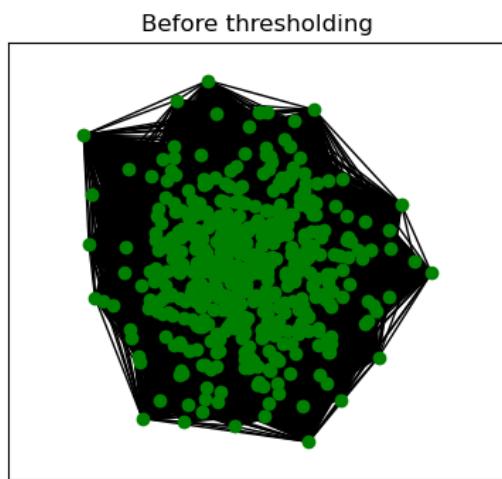
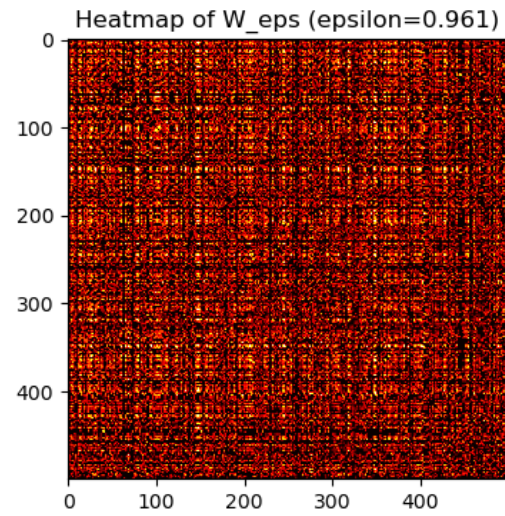
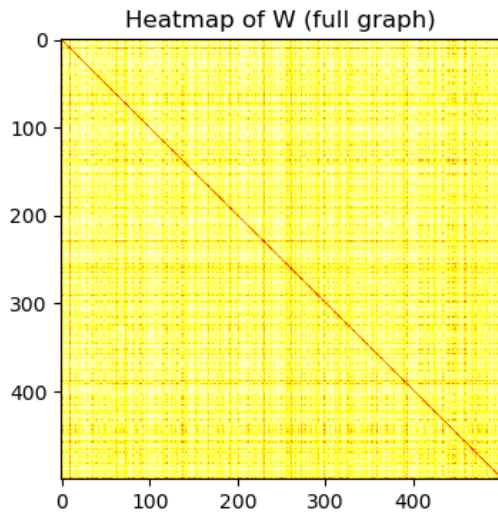
plt.show()

```

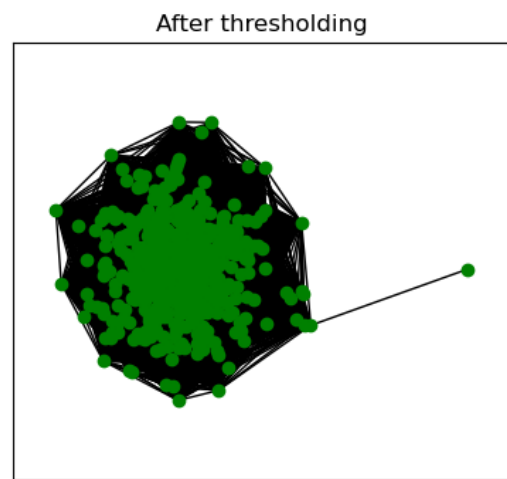
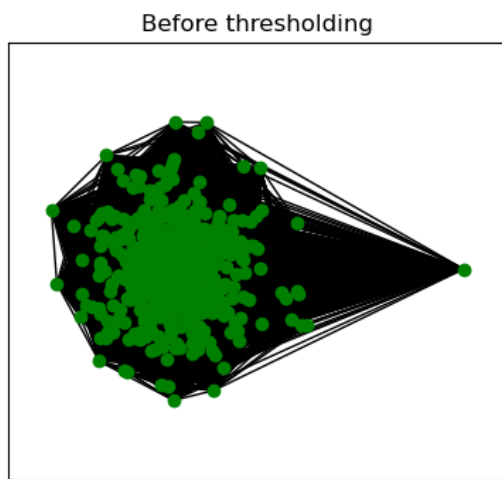
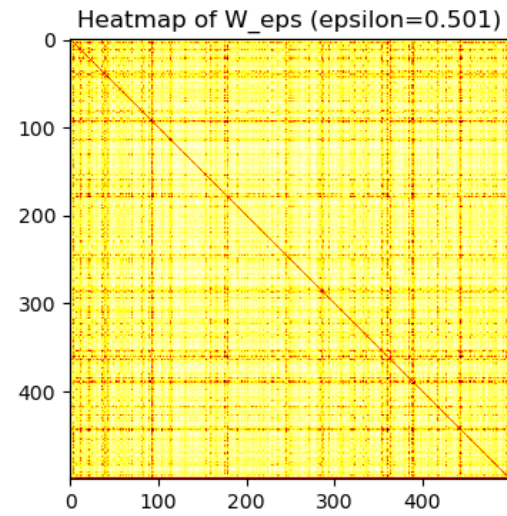
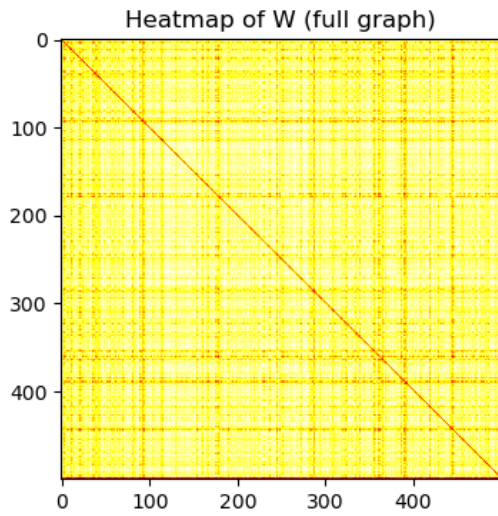
```

- For delta = 0.01
Optimal epsilon = 0.961433
Number of edges (full graph): 124750
Number of edges (epsilon graph): 24102
Percentage of removed edges: 80.68 %

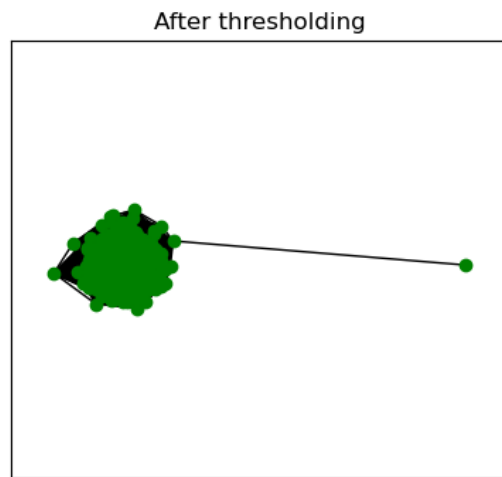
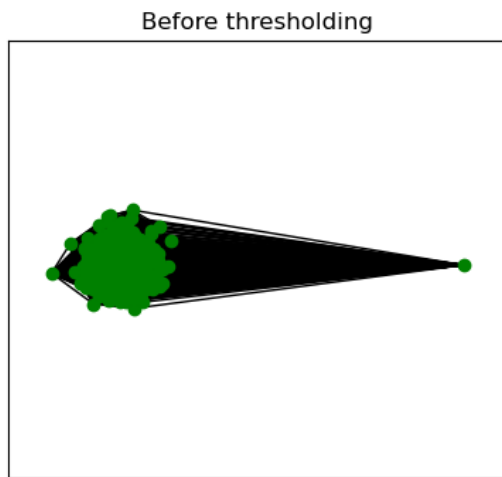
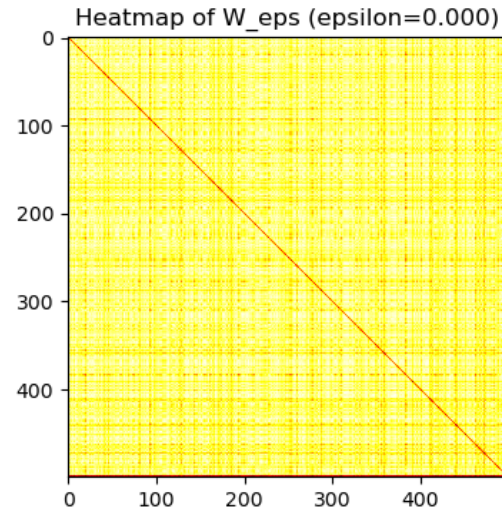
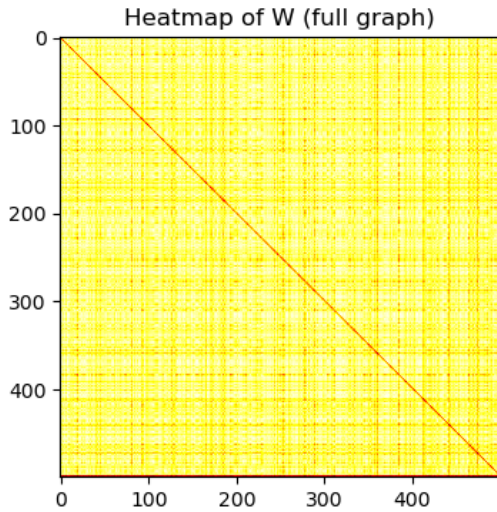
```



- For $\delta = 1$
 Optimal epsilon = 0.501357
 Number of edges (full graph): 124750
 Number of edges (epsilon graph): 121087
 Percentage of removed edges: 2.94 %



- For $\delta = 5$
 Optimal $\epsilon = 0.000002$
 Number of edges (full graph): 124750
 Number of edges (epsilon graph): 124252
 Percentage of removed edges: 0.40 %



The goal is to choose ϵ to keep the graph connected using `min_span_tree`.

Observations:

- **Small delta (0.01) $\rightarrow$ Large ϵ (0.97)**
 - The points are dispersed, but the isolated point is close to the group.
 - ϵ is large because it is easy to connect all points, leading to a strong reduction in edges (82.82%).
- **Medium delta (1) $\rightarrow$ Intermediate ϵ (0.56)**
 - The isolated point starts to move away.
 - ϵ is smaller, and the graph remains dense (only 4.48% of edges are removed).

- **Large delta (5) $\rightarrow$ Very small ϵ (0.000003)**
 - The isolated point is far from the main group.
 - ϵ is tiny since it needs to connect the isolated point.
 - As a result, almost no edges are removed (0.40%).

To conclude, **if delta is too large**, the isolated point forces ϵ to be very small, retaining almost all edges.

Thus, this approach is not ideal because, in a scenario with points too far from the center of mass (such as outliers), too many connections are formed, making **the epsilon-graph nearly useless**.

2.0.3 Q1.3: Is it important to build a connected graph? Compare k-NN to ϵ -graphs with respect to the difficulty of building a connected graph.

```
[14]: # Example 1: Separated Blobs

X, Y = blobs(500, n_blobs=3, blob_var=0.2)
plot_clusters(X, Y)
plt.show()

# k-NN graph
k_list = [5, 80, 250]
fig, ax = plt.subplots(1, 3, figsize=(10, 5))
for i, k in enumerate(k_list):
    W = build_similarity_graph(X, k=k)
    plt.sca(ax[i])
    plot_edges_and_points(X, Y, W, title=f'k = {k}')
plt.show()

# epsilon-graph
eps_list = [0.1, 0.5, 0.9]
fig, ax = plt.subplots(1, 3, figsize=(10, 5))
for i, eps in enumerate(eps_list):
    W = build_similarity_graph(X, eps=eps)
    plt.sca(ax[i])
    plot_edges_and_points(X, Y, W, title=f'eps = {eps}')
plt.show()

# Example 2: Worst-case blob with delta=2.5

X, Y = worst_case_blob(600, delta=2.5)
plot_clusters(X, Y)
plt.show()

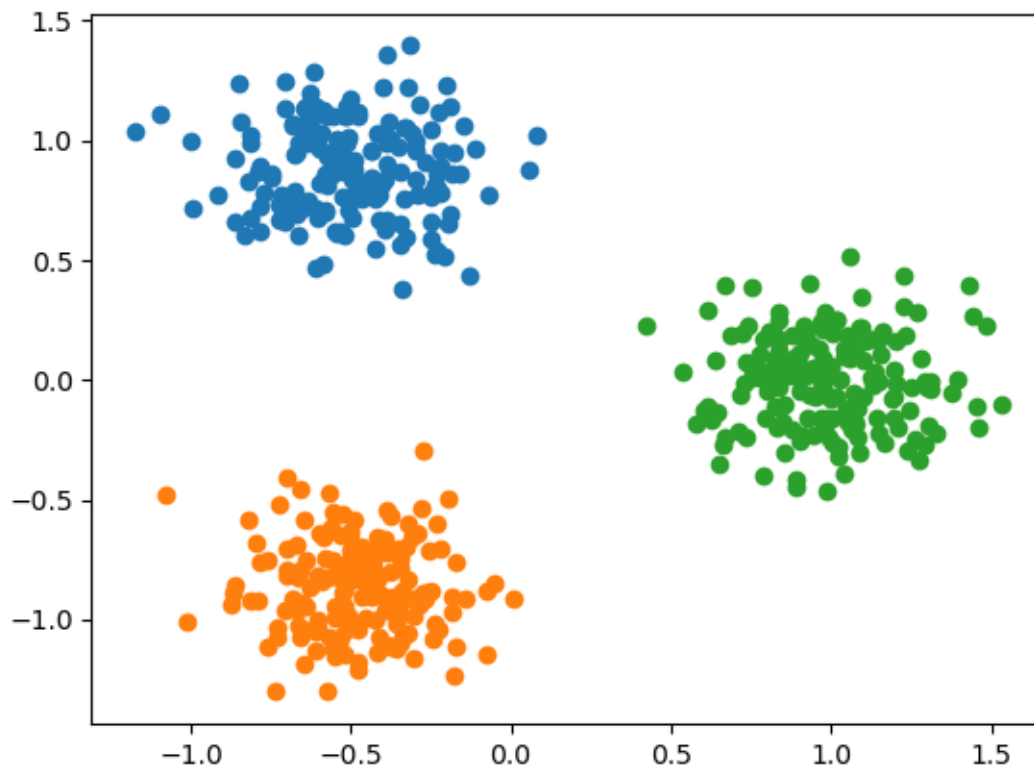
# k-NN graph
k_list = [1, 5, 20]
```

```

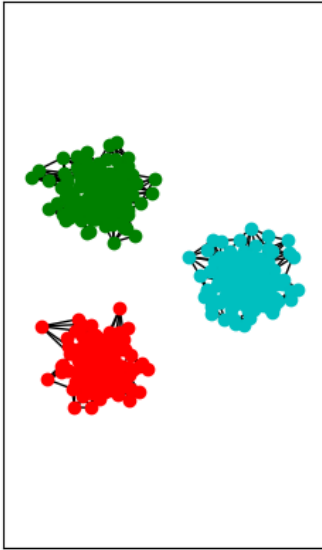
fig, ax = plt.subplots(1, 3, figsize=(10, 4))
for i, k in enumerate(k_list):
    W = build_similarity_graph(X, k=k)
    plt.sca(ax[i])
    plot_edges_and_points(X, Y, W, title=f'k = {k}')
plt.show()

#  $\epsilon$ -graph
eps_list = [0, 0.1, 0.4]
fig, ax = plt.subplots(1, 3, figsize=(10, 4))
for i, eps in enumerate(eps_list):
    W = build_similarity_graph(X, eps=eps)
    plt.sca(ax[i])
    plot_edges_and_points(X, Y, W, title=f'eps = {eps}')
plt.show()

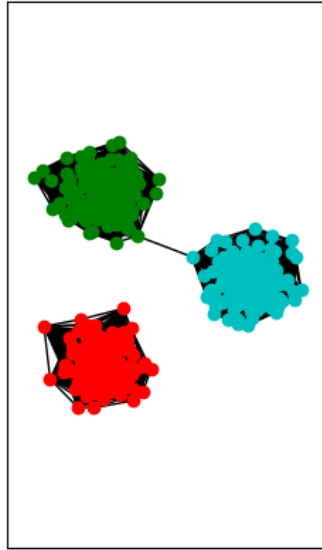
```



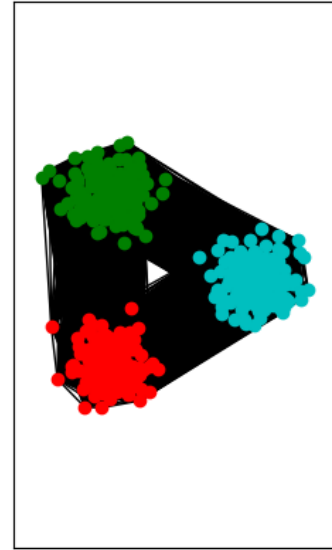
$k = 5$



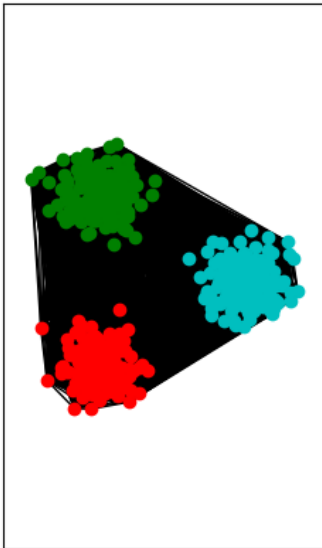
$k = 80$



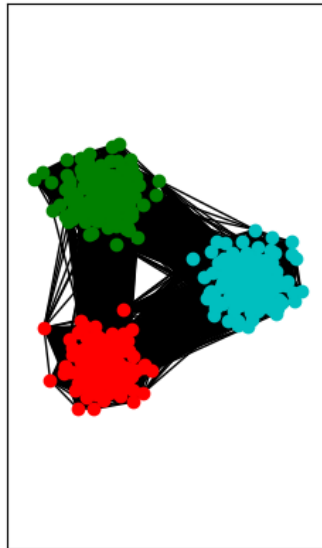
$k = 250$



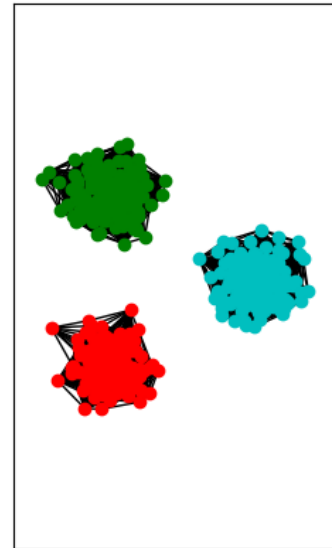
$\text{eps} = 0.1$

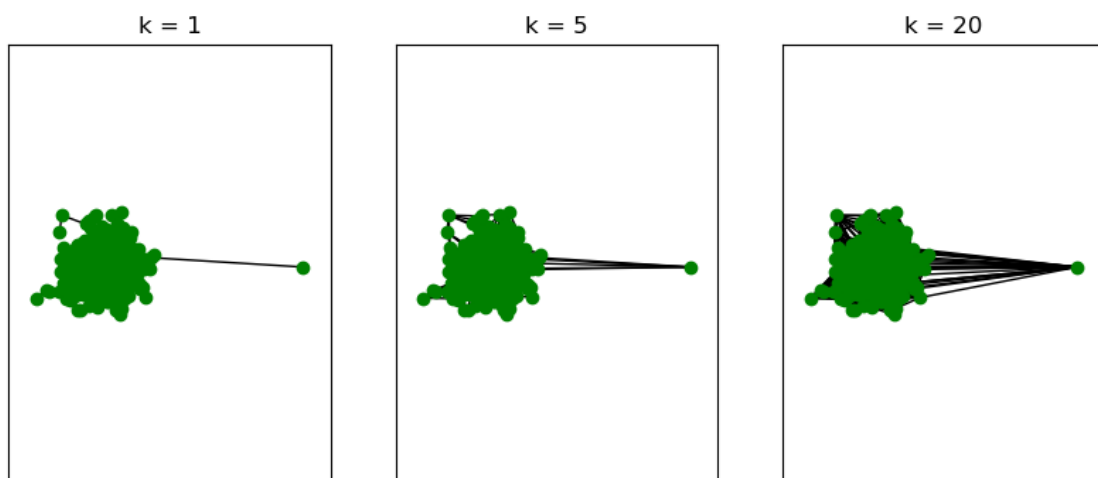
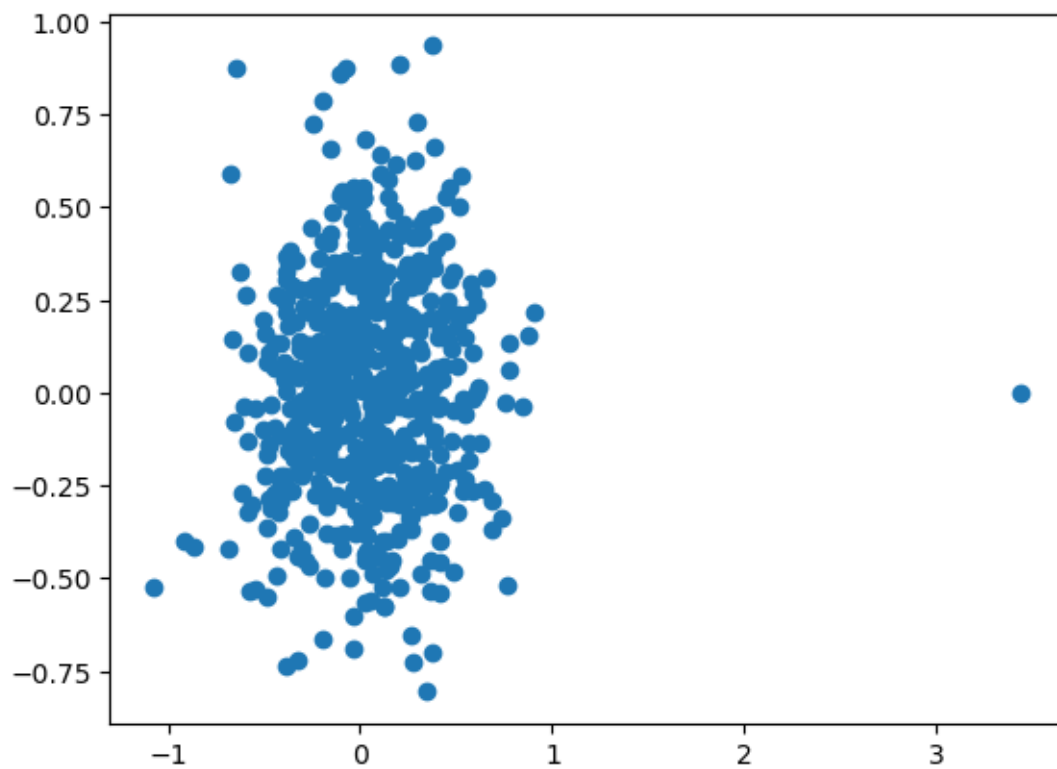


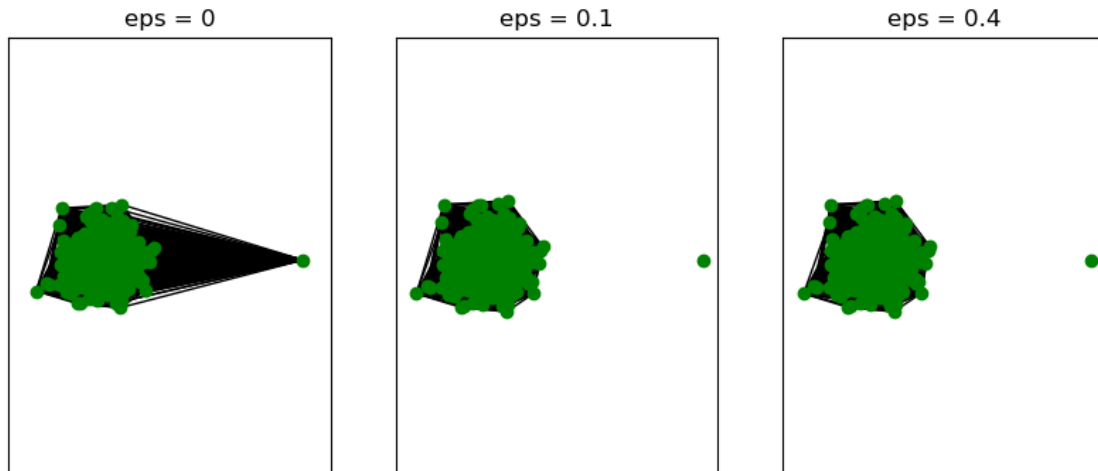
$\text{eps} = 0.5$



$\text{eps} = 0.9$







A connected graph is preferable for our spectral clustering algorithms, but it is not strictly necessary.

Comparison: k -NN vs ϵ -graph

- **Separated blobs:**
 - A k -NN **graph** is often **disconnected** if k is too small ($<$ cluster size).
 - An ϵ -**graph** connects points more easily, but if ϵ is too small, it may create too many connections.
- **Worst-case blob ($\delta = 2.5$):**
 - A k -NN **graph** always connects the isolated point, even with $k = 1$.
 - An ϵ -**graph** may leave the isolated point (or rather, the outlier) disconnected if ϵ is moderate or large, which can be useful in some cases.

Conclusion:

- k -NN **graph**: More reliable for ensuring connectivity but rarely fully connected when k is small.
- ϵ -**graph**: Better for continuous structures with no outliers (otherwise, they may be left out).

The optimal choice depends **on the context and the data structure**.

2.0.4 Q2.1: Build a graph starting from the data generated in blobs, and keep the graph connected. Use `spectral_clustering` to cluster the data. Motivate your choice on which eigenvectors to use and how you computed the clustering assignments from the eigenvectors. Now compute a similar clustering using the built-in `k-means` and compare the results.

```
[15]: X, Y = blobs(num_samples=600)
      plot_clusters(X, Y)
      plt.show()

      # Constructing the graph with a well-chosen  $k$ -NN
```

```

k = 320
W = build_similarity_graph(X, k=k)

plot_edges_and_points(X, Y, W, title=f'k = {k}')

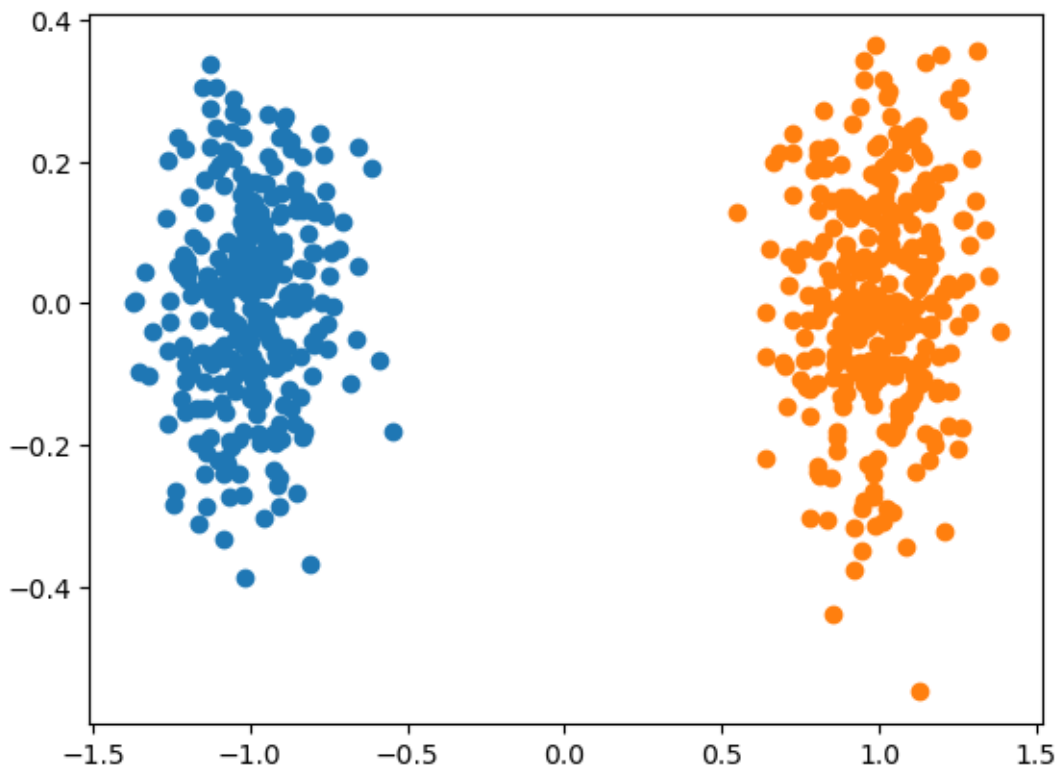
L = build_laplacian(W)

num_classes = 2 # We know the blobs form 2 groups
chosen_eig_indices = [1] # Selecting the 2nd eigenvector
Y_rec = spectral_clustering(L, chosen_eig_indices=chosen_eig_indices,
    ↪ num_classes=num_classes, show_valeurs_propres=True, show_vecteurs_propres=True)

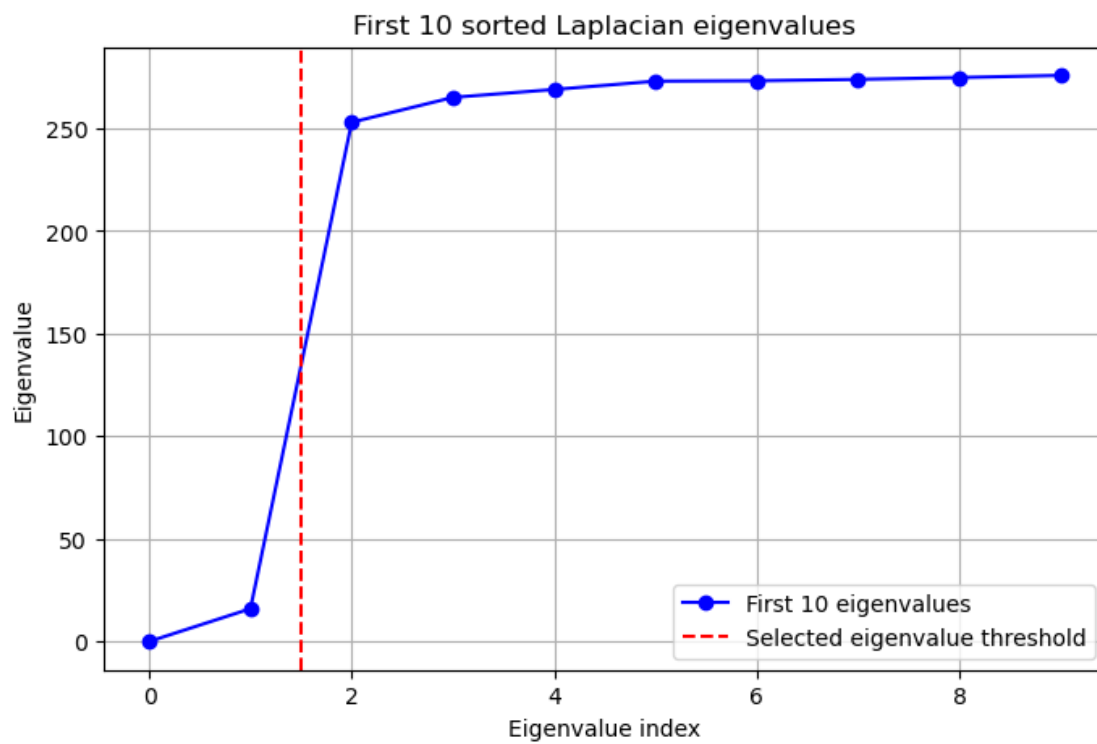
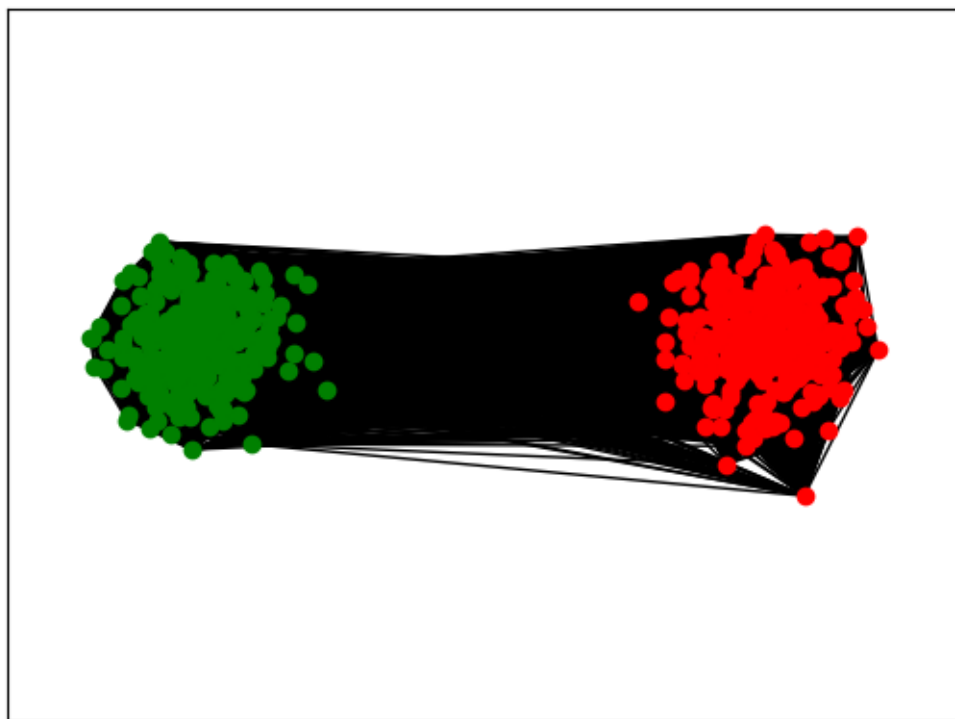
Y_kmeans = KMeans(num_classes, n_init=10, random_state=42).fit_predict(X)

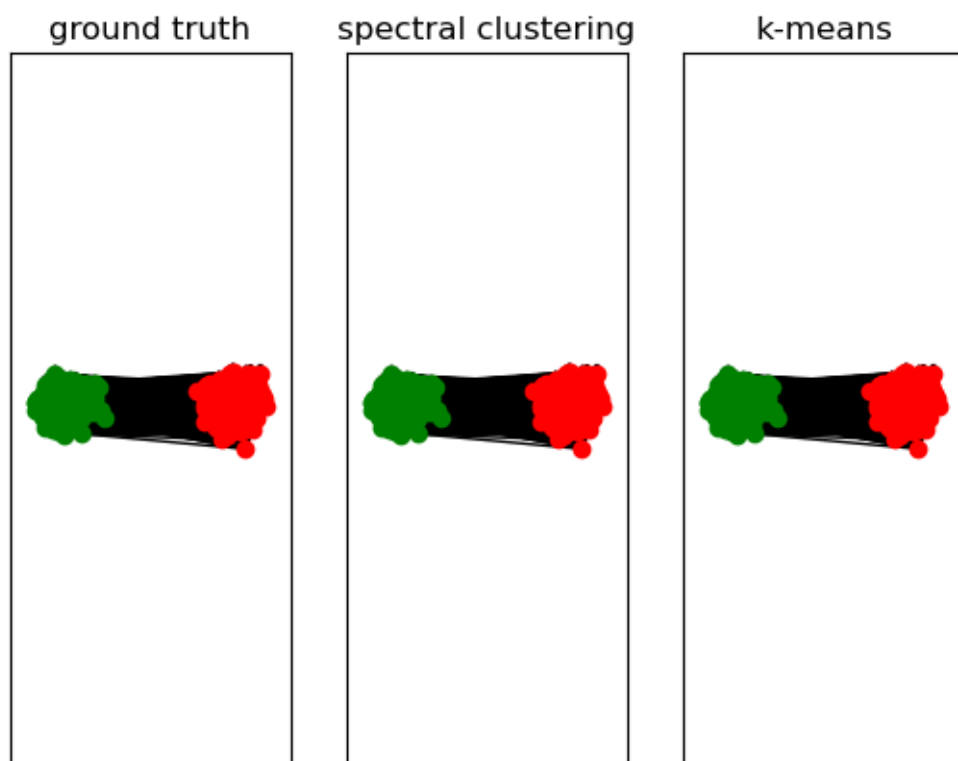
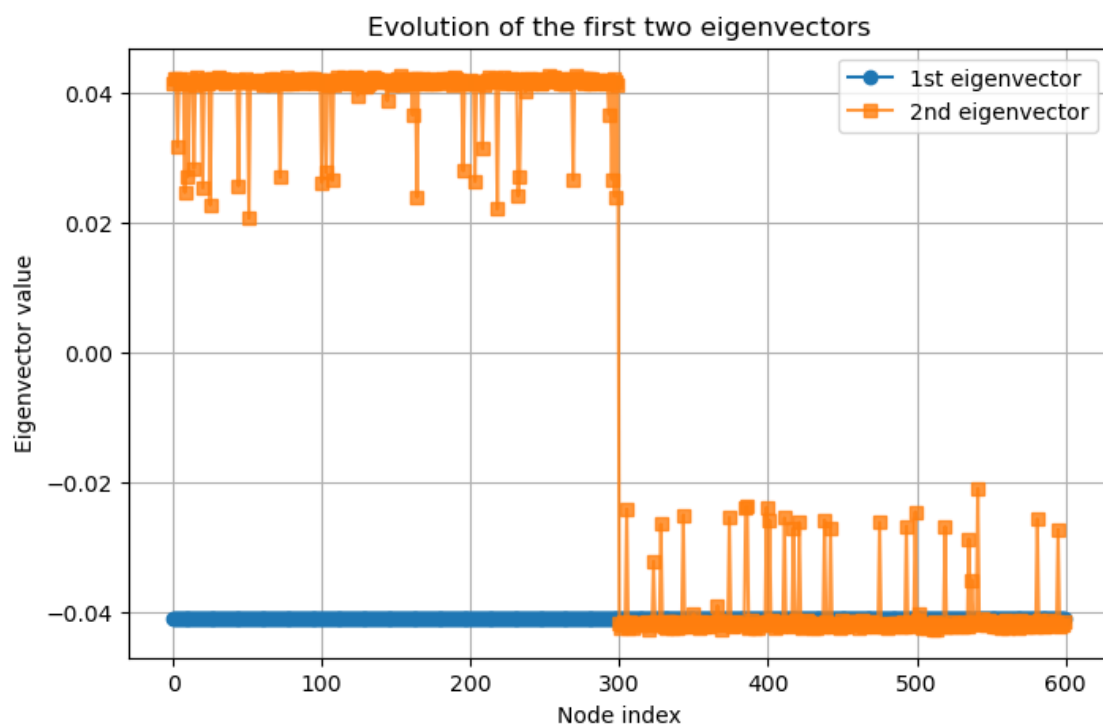
# Comparing results
plot_clustering_result(X, Y, L, Y_rec, Y_kmeans)

```



k = 320





Since each blob contains **300 points**, I chose to build a **k -NN graph** with $k = 320 > 300$ to ensure a **connected graph** while limiting unnecessary connections.

- Since the **graph is connected**, the smallest eigenvalue of the Laplacian is 0, with a **multiplicity of 1**. Its associated eigenvector is constant, making it useless for clustering.
- Therefore, we use **the second smallest eigenvalue** to separate the clusters.
- Both algorithms (Spectral Clustering and K-Means) produce the same result.

2.0.5 Q2.2: Build a graph starting from the data generated in blobs, but this time make it so that the two components are separated. How do you choose which eigenvectors to use in this case? Motivate your answer.

```
[16]: X, Y = blobs(num_samples=600)

# Constructing the graph with a well-chosen k-NN
k = 100
W = build_similarity_graph(X, k=k)

plot_edges_and_points(X, Y, W, title=f'k = {k}')

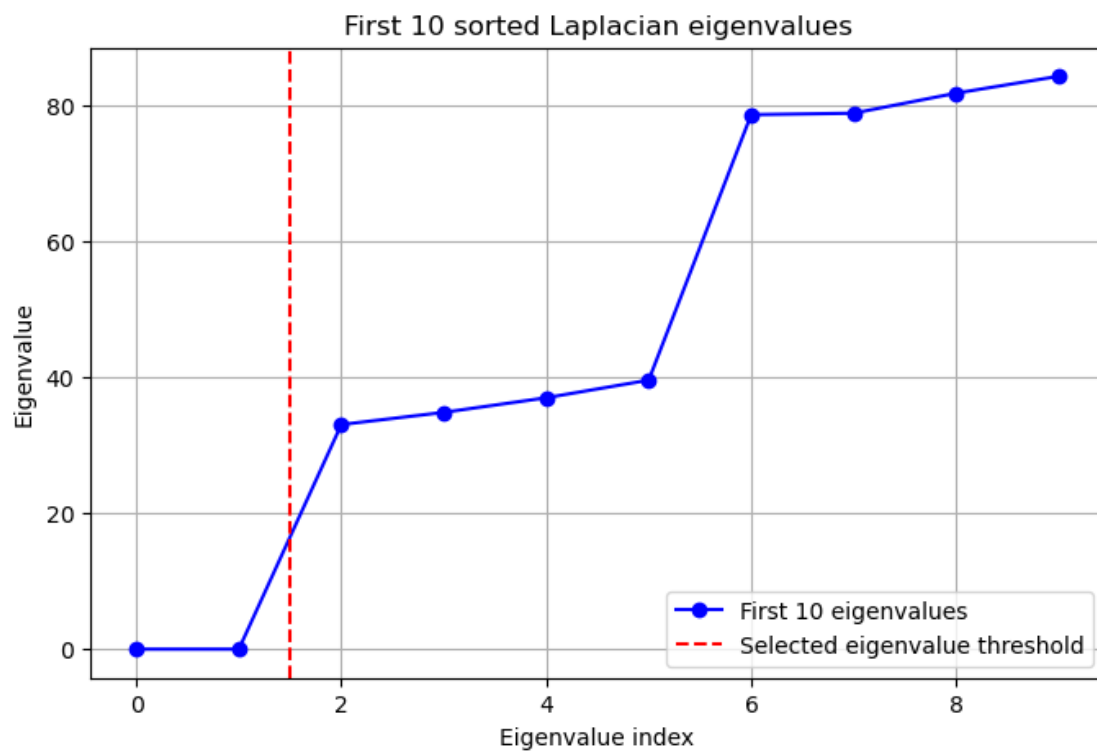
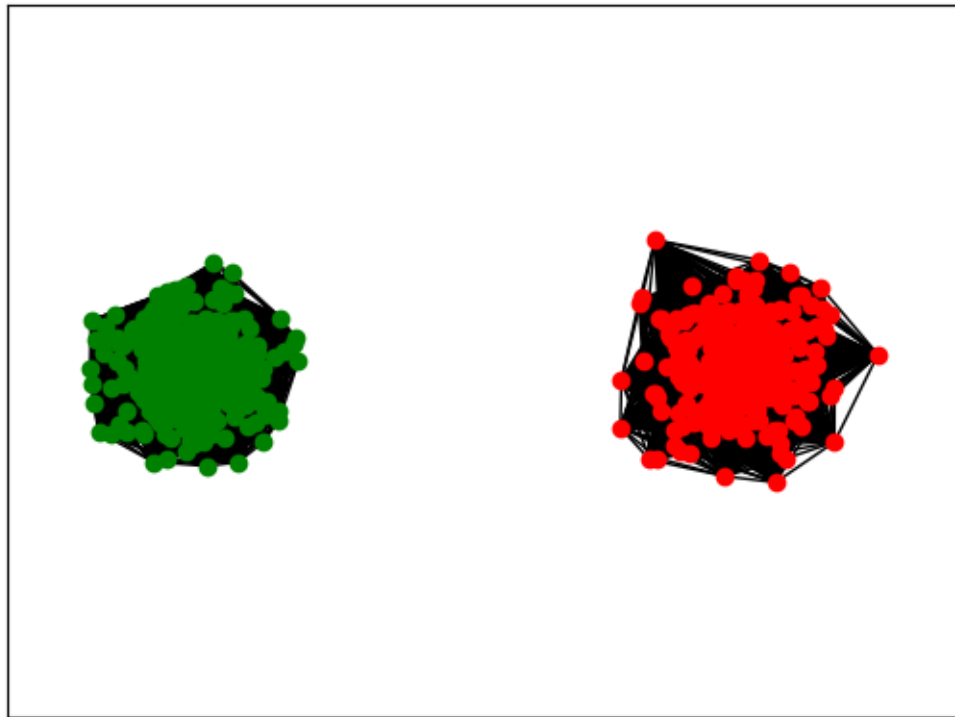
L = build_laplacian(W)

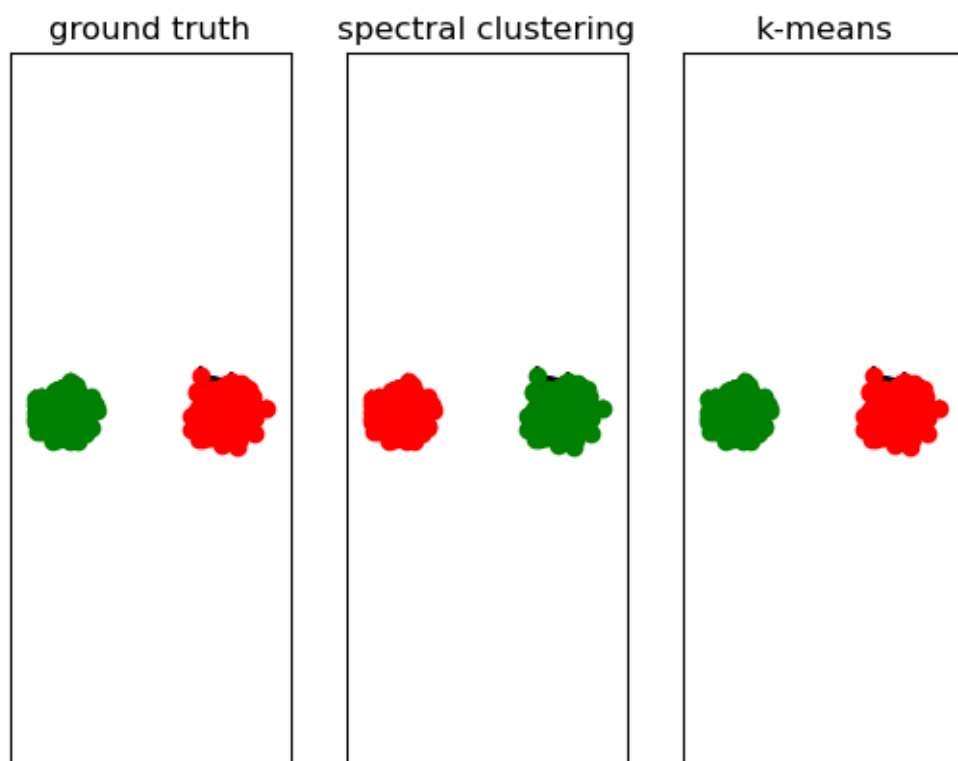
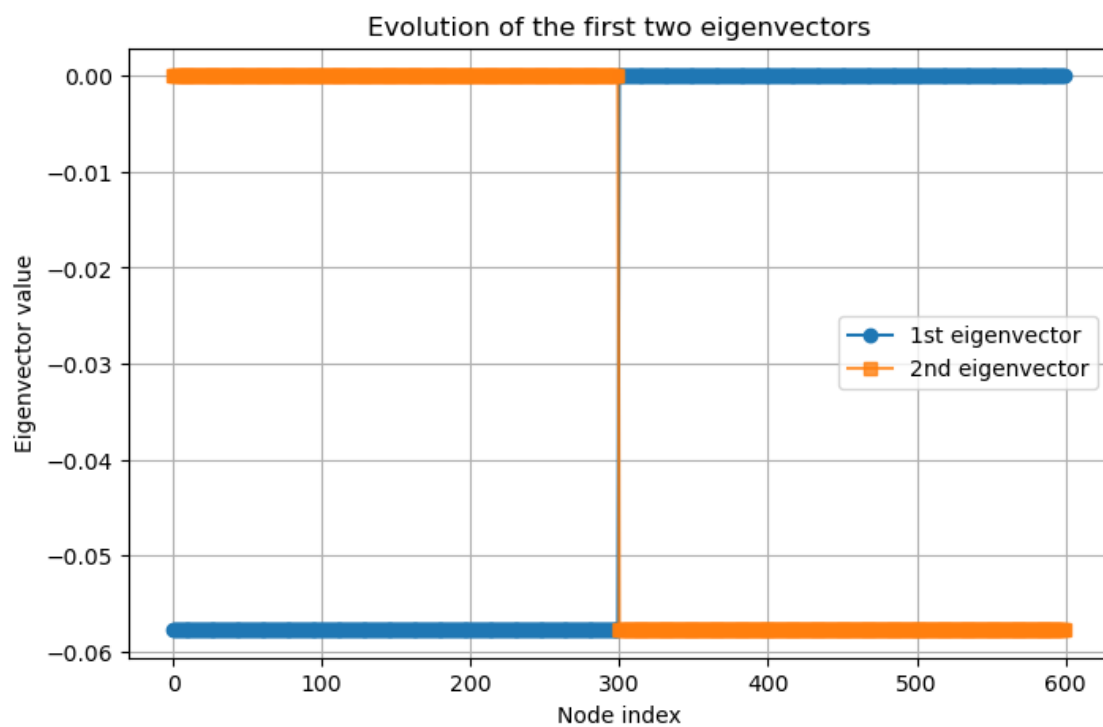
num_classes = 2 # We know the blobs form 2 groups
chosen_eig_indices = [0,1] # Selecting the 1st eigenvector
Y_rec = spectral_clustering(L, chosen_eig_indices=chosen_eig_indices,
    ↪ num_classes=num_classes, show_valeurs_propres=True, show_vecteurs_propres=True)

Y_kmeans = KMeans(num_classes, n_init=10, random_state=42).fit_predict(X)

# Comparing results
plot_clustering_result(X, Y, L, Y_rec, Y_kmeans)
```

k = 100





This time, **the two components are disconnected**. This means the graph will have **two connected components**.

- Since the graph has $p = 2$ connected components, the **eigenvalue 0** of the Laplacian has a **multiplicity of 2**.
- The **first two eigenvectors** correspond to the indicators of the **two components**. These are the ones we use for spectral clustering.
- We can even note that in the case of $p = 2$ connected components, a single eigenvector would be sufficient (each component has a constant value of 0 or 1).

2.0.6 Q2.3: In the function `spectral_clustering`, implement an automatic strategy to choose the eigenvectors to use for clustering, when `chosen_eig_indices=None`. Explain your strategy. Generate a dataset with 4 blobs and test your function. What happens when the variance of the blobs increase?

```
[17]: variances = [0.05, 0.15, 0.5]
num_classes = 4

for var in variances:
    print(f"\n- Test with variance {var}")

    X, Y = blobs(num_samples=600, n_blobs=num_classes, blob_var=var)
    plot_clusters(X, Y)
    plt.show()

    k = 200 # Using a safe value for connectivity
    W = build_similarity_graph(X, k=k)
    L = build_laplacian(W)

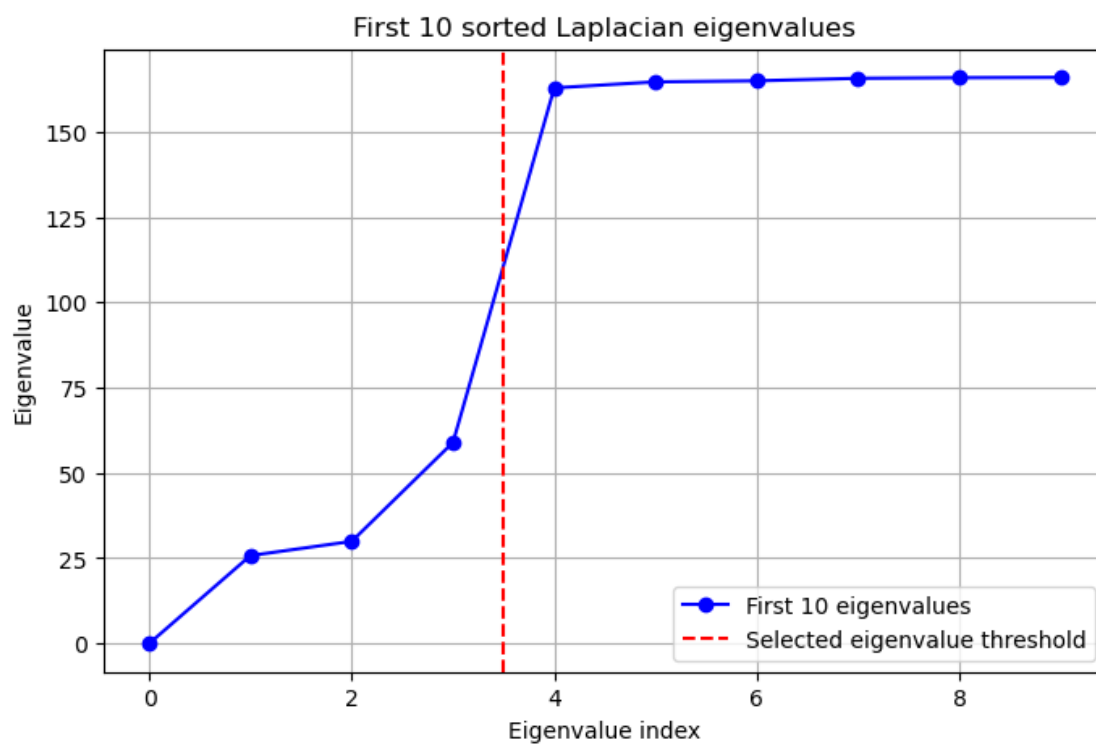
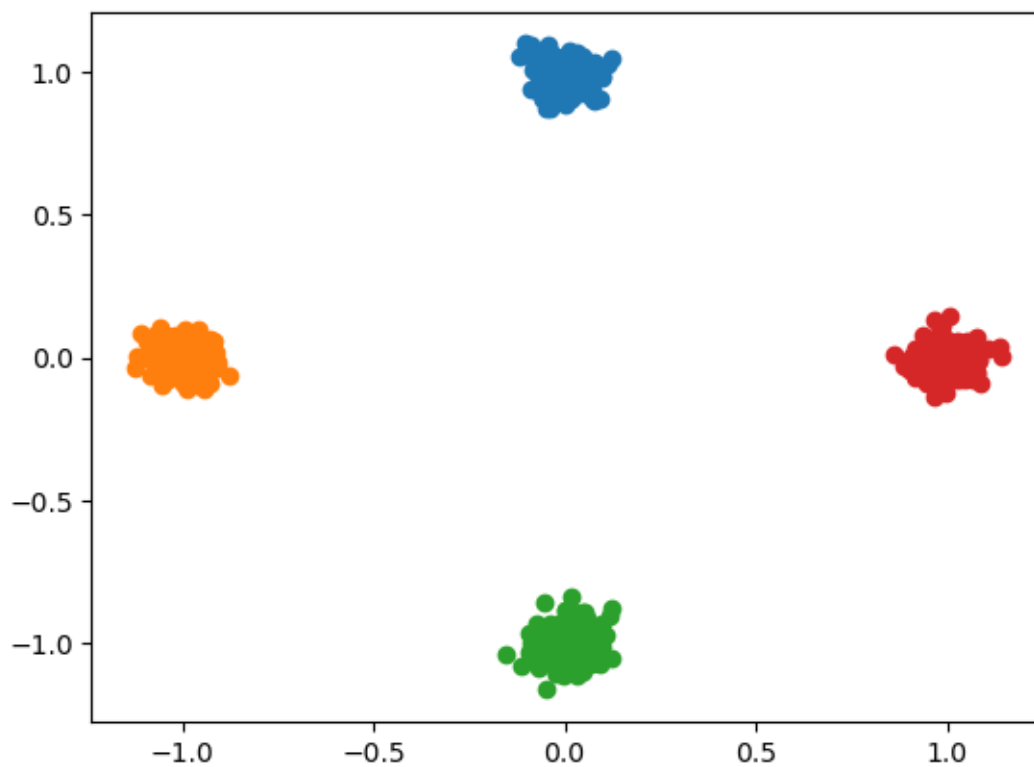
    # Spectral clustering with automatic eigenvector selection
    Y_rec = spectral_clustering(L, chosen_eig_indices=None,
    ↪ num_classes=num_classes, show_valeurs_propres=True)

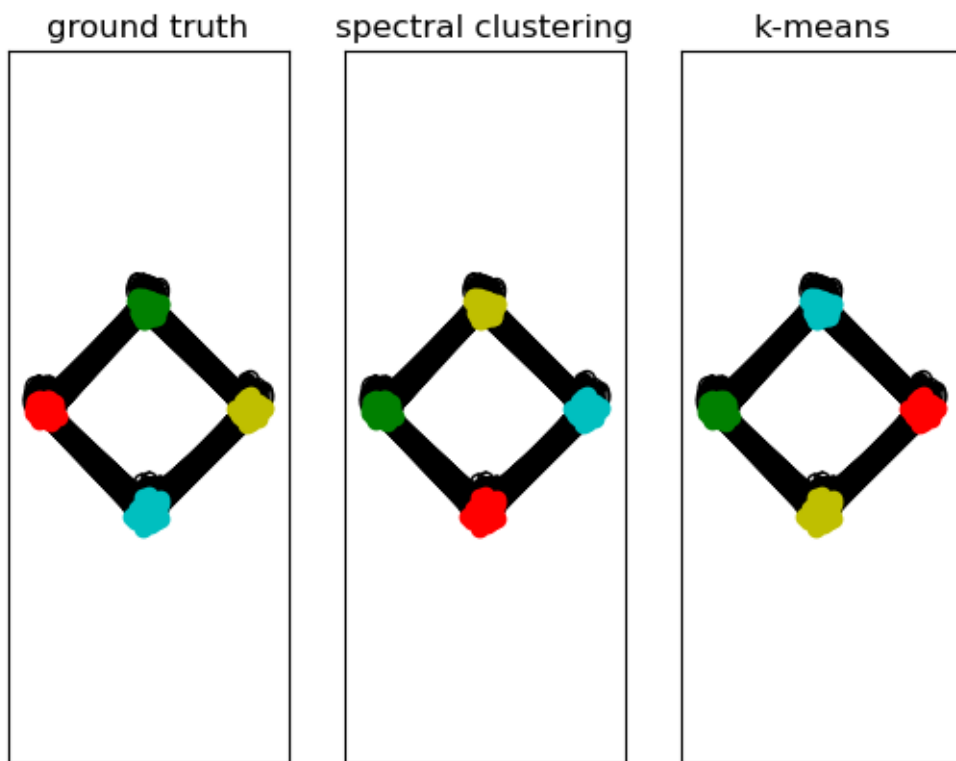
    # K-Means clustering for comparison
    Y_kmeans = KMeans(num_classes, n_init=10, random_state=42).fit_predict(X)

    plot_clustering_result(X, Y, L, Y_rec, Y_kmeans)

    acc_spectral = compute_best_accuracy(Y, Y_rec)
    acc_kmeans = compute_best_accuracy(Y, Y_kmeans)
    print(f" Accuracy Spectral Clustering: {acc_spectral:.4f}")
    print(f" Accuracy K-Means: {acc_kmeans:.4f}")
```

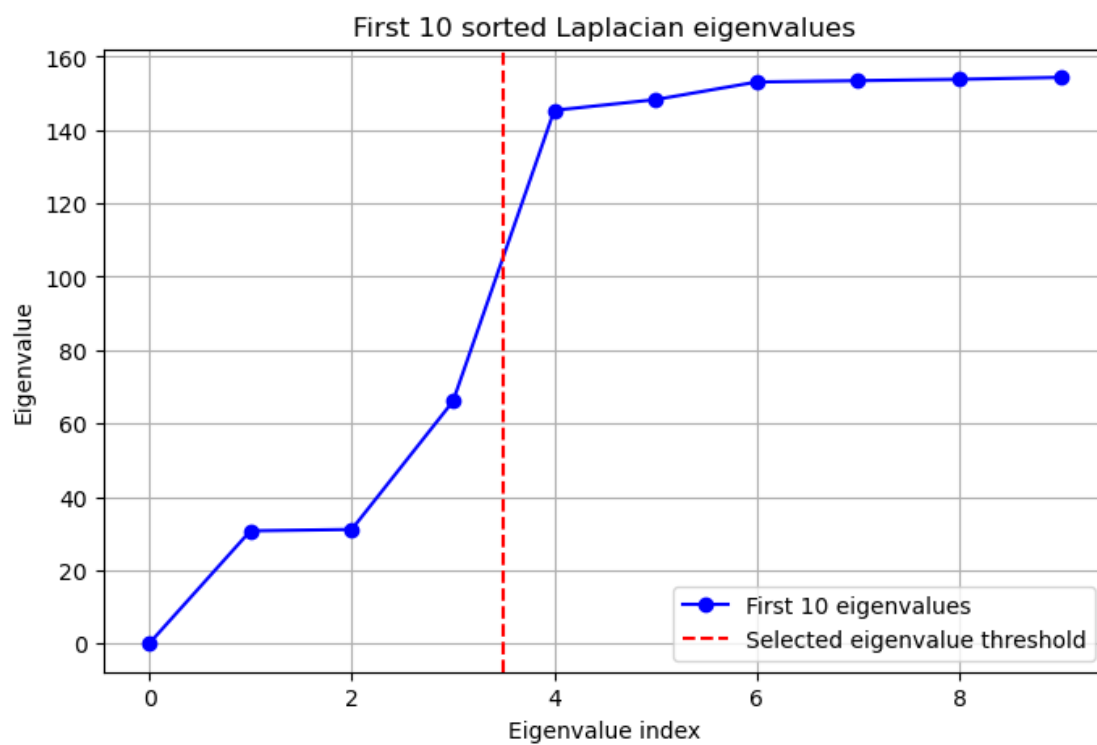
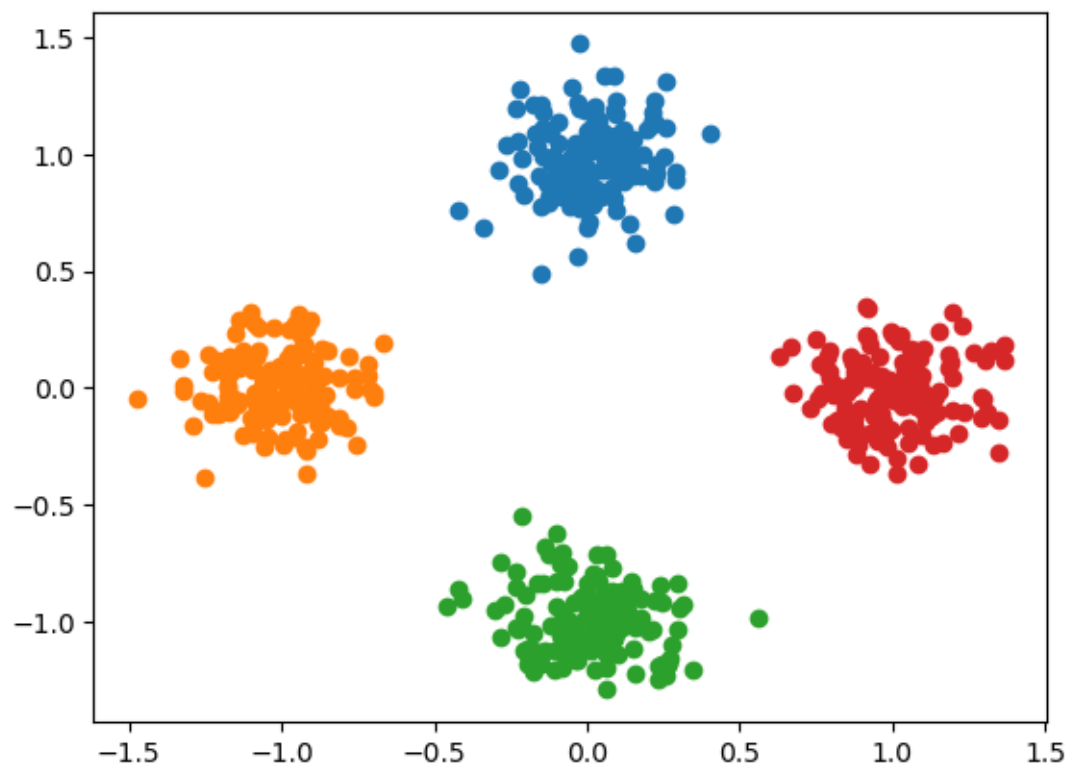
- Test with variance 0.05

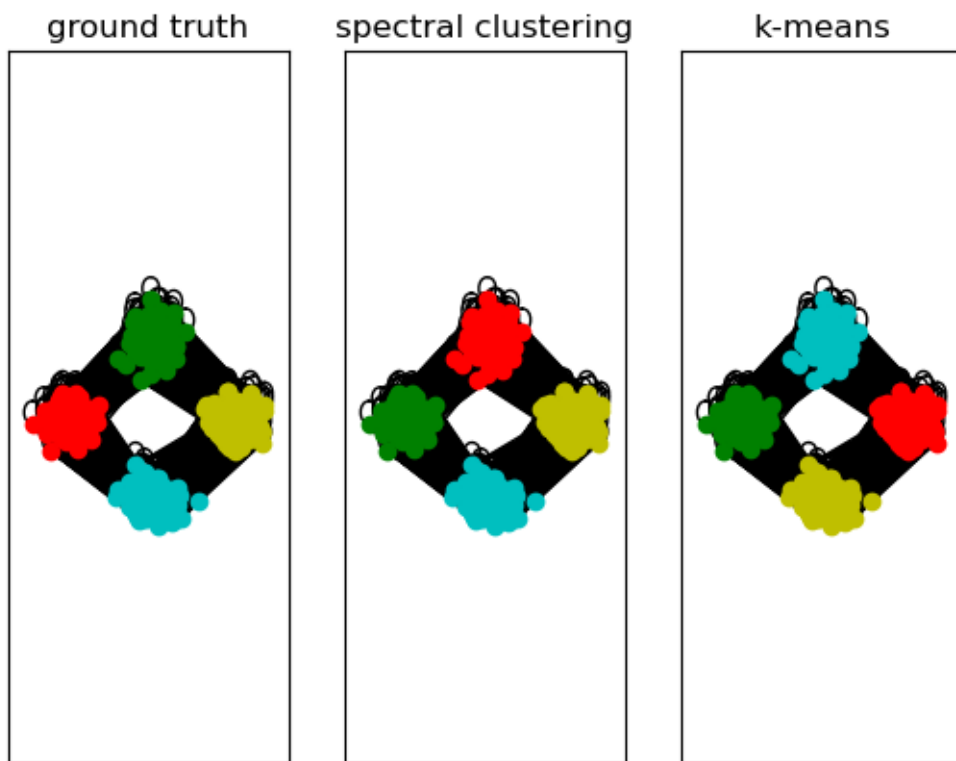




Accuracy Spectral Clustering: 1.0000
Accuracy K-Means: 1.0000

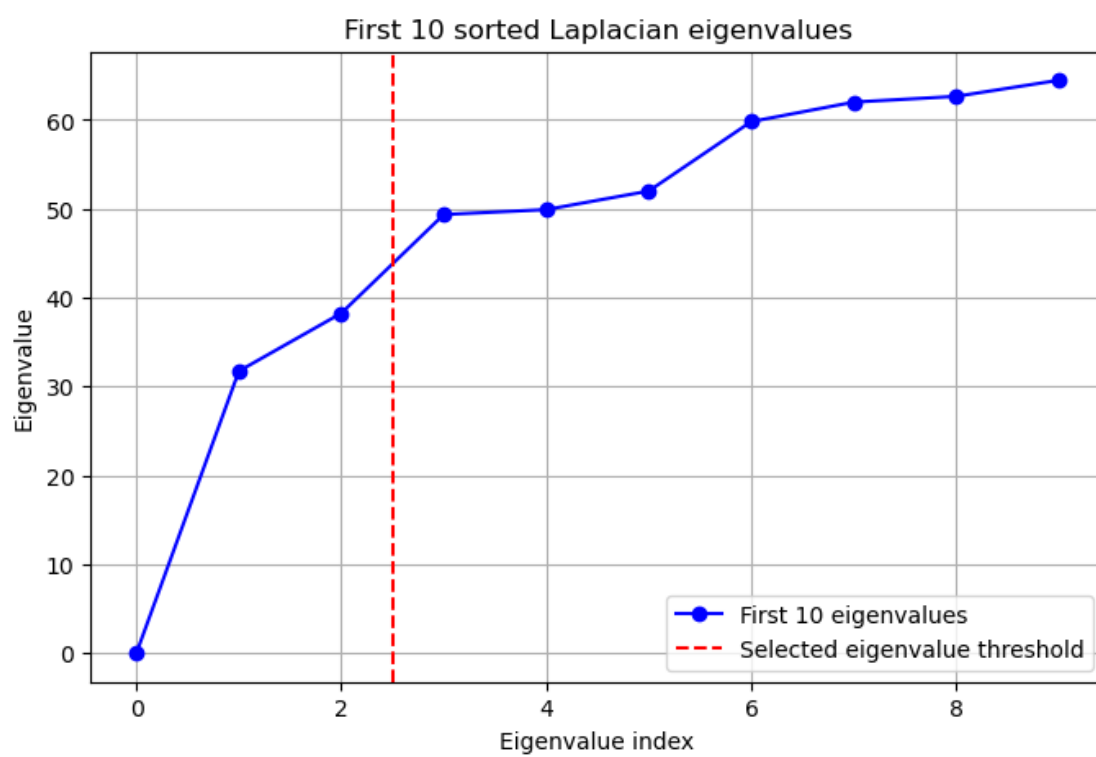
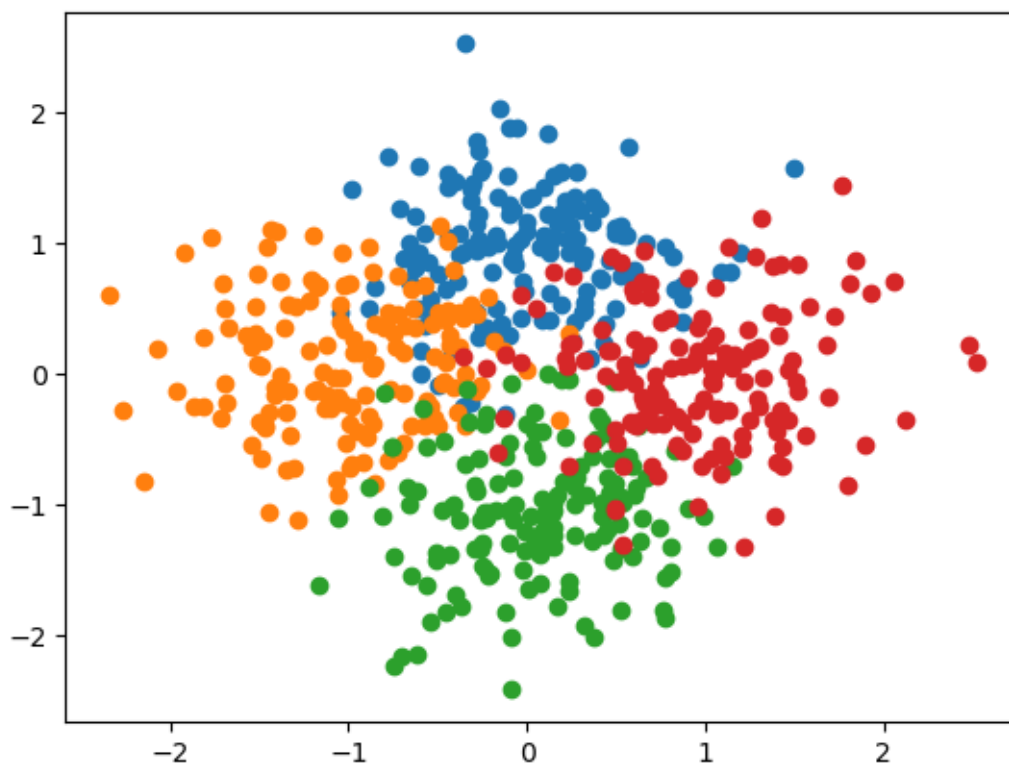
- Test with variance 0.15

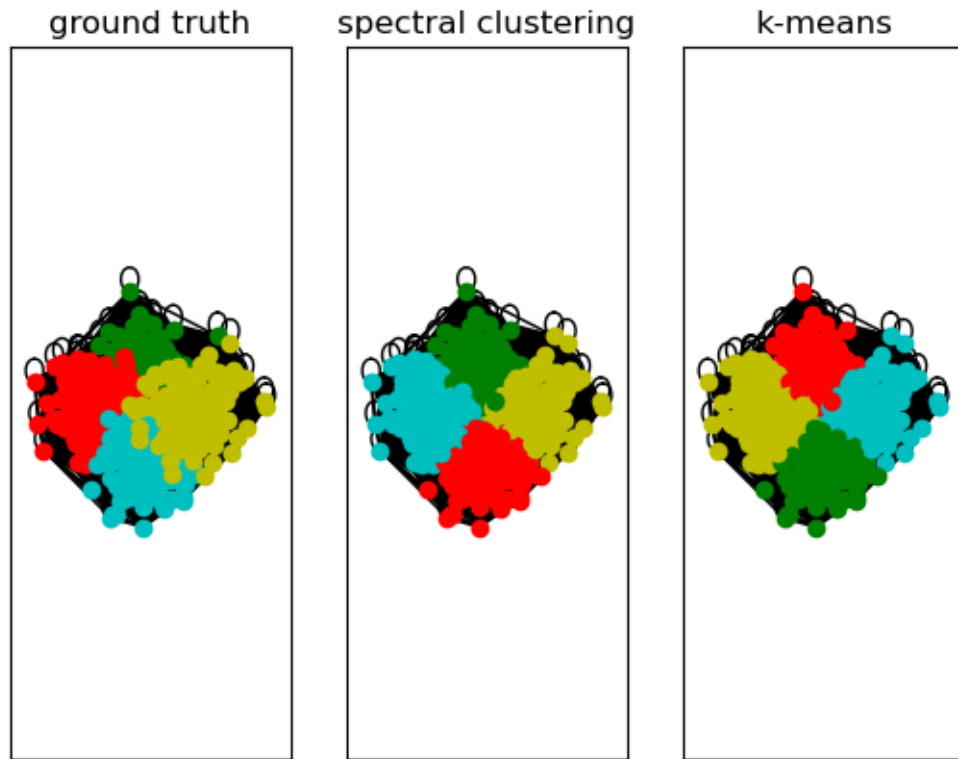




Accuracy Spectral Clustering: 1.0000
Accuracy K-Means: 1.0000

- Test with variance 0.5





Accuracy Spectral Clustering: 0.8450

Accuracy K-Means: 0.8433

The idea is to implement an automatic strategy for selecting eigenvectors in `spectral_clustering`.

My Methodology: “Elbow Method”

- We analyze the gaps between successive eigenvalues (`gaps = np.diff(E)`).
- We select the eigenvectors up to the largest gap.
- If the largest gap is between λ_0 and λ_1 , we take the next one (the second-largest gap).

Obtained Results:

Variance	Accuracy Spectral Clustering	Accuracy K-Means
0.05	1.0000	1.0000
0.15	1.0000	1.0000
0.5	0.6500	0.8483

- **Low variance (0.05 & 0.15):** The clusters are well separated, and both methods achieve 100% accuracy.
- **High variance (0.5):** The clusters overlap, Spectral Clustering fails (65%), detecting 3 clusters instead of 4, while K-Means performs slightly better (84.83%).

This can be explained by a key argument: the eigenvalues become closer together, making the elbow method less effective. Thus, the “**elbow method**” works well for well-separated clusters, but when variance is too high, Spectral Clustering becomes unstable (it may reduce the number of detected clusters, for example).

2.0.7 Q2.4: When you built the cluster assignment, did you use thresholding, k-means or both? Do you have any opinion on when to use each?

In my implementation of Spectral Clustering, I use **K-means** on the matrix of selected eigenvectors.

The **thresholding method** can be efficient and fast, especially in simple cases with **two classes**, where separating the values of the first eigenvector is enough.

However, **K-means is more general** and works well even when the cluster structure is more complex.

2.0.8 Q2.5: Generate two_moons data and compare spectral clustering to k-means. Do you notice any difference? Taking into consideration the graph structure, can you explain them?

```
[18]: X, Y = two_moons(num_samples=600)
      plot_clusters(X, Y)
      plt.show()

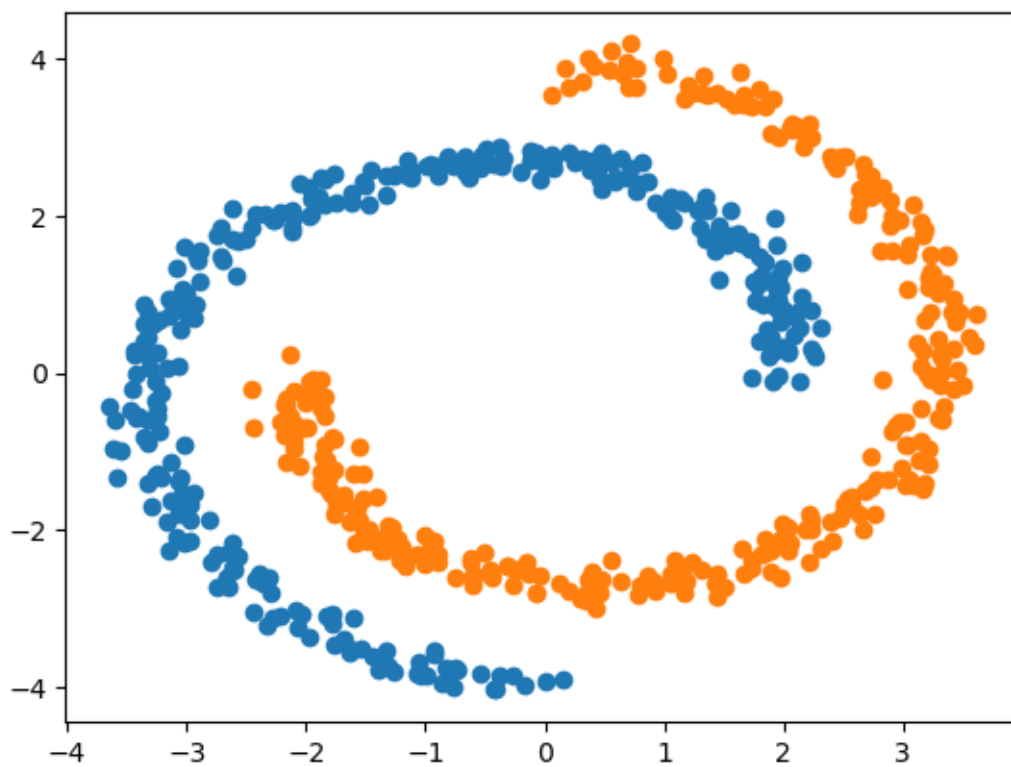
      # Constructing the k-NN graph with k=10, then applying the standard method seen
      ↪ earlier
      k = 10
      W = build_similarity_graph(X, k=k)
      plot_edges_and_points(X, Y, W, title=f'k = {k}')

      L = build_laplacian(W)

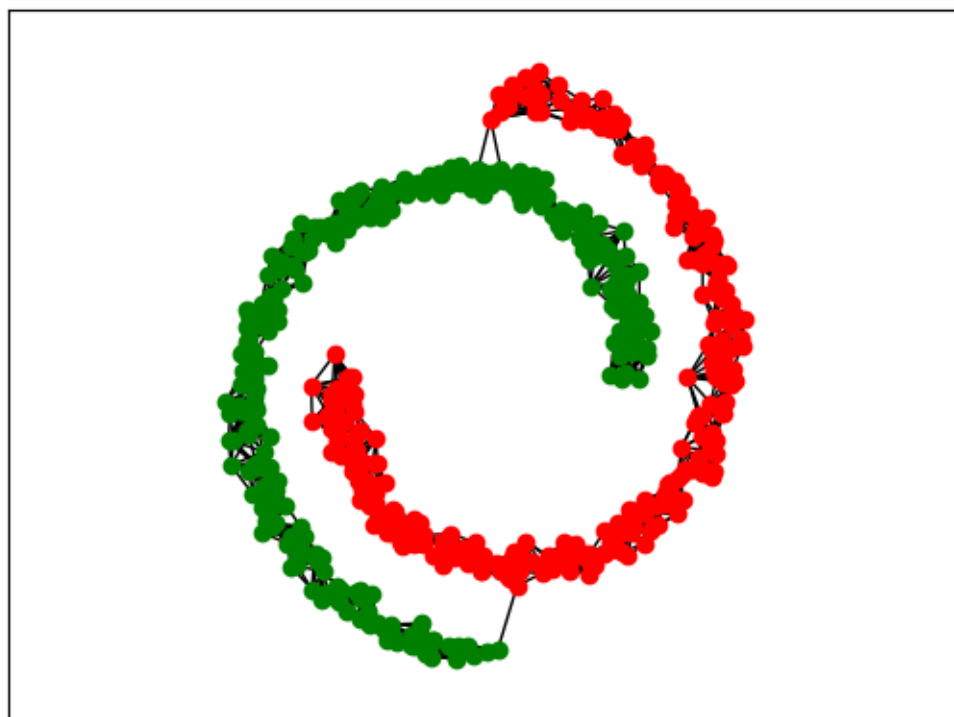
      num_classes = 2
      chosen_eig_indices = [1] # Selecting the second smallest eigenvalue
      Y_rec = spectral_clustering(L, chosen_eig_indices=chosen_eig_indices,
      ↪ num_classes=num_classes)

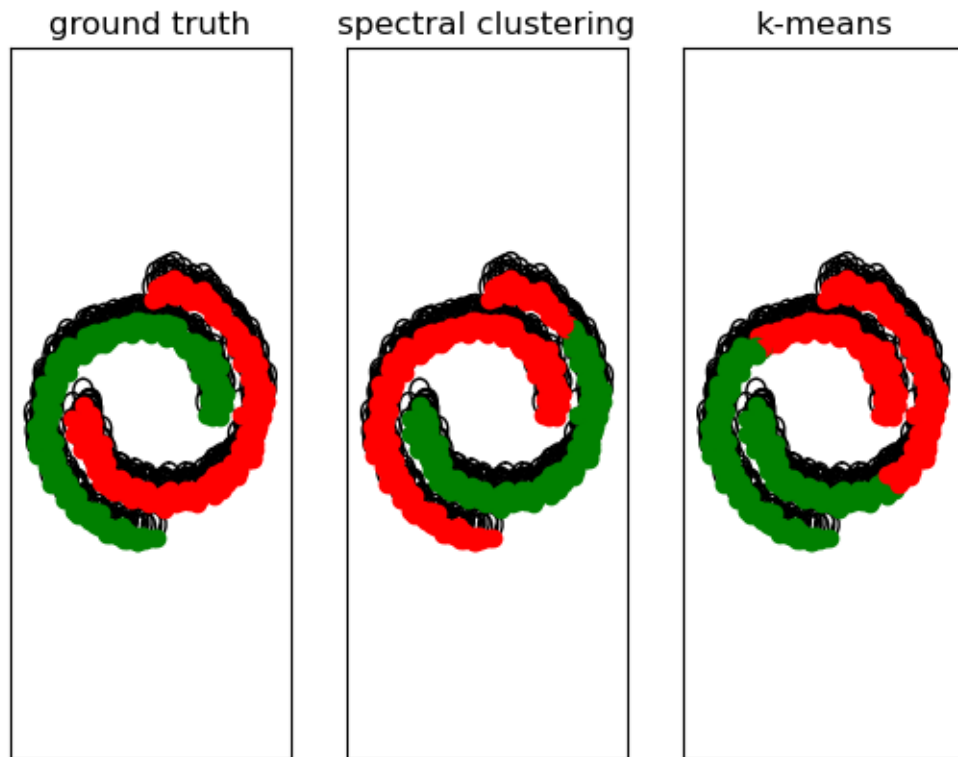
      Y_kmeans = KMeans(num_classes, n_init=10, random_state=42).fit_predict(X)

      plot_clustering_result(X, Y, L, Y_rec, Y_kmeans)
```



$k = 10$





With this dataset, we observe a **clear difference between the results of Spectral Clustering and K-Means**.

K-Means fails to capture the underlying moon-shaped structure and incorrectly splits the clusters. In contrast, by constructing a k -NN graph with a moderate k , we preserve the local structure of the data. Spectral Clustering then leverages this representation to correctly separate the two groups.

Thanks to the information contained in the graph, **Spectral Clustering produces significantly better results than K-Means**, which remains limited to a “convex” view of the dataset.

2.0.9 Q2.6: Generate point_and_circle data and compare spectral clustering using the normal Laplacian and the random-walk regularized Laplacian. Do you notice any difference? Taking into consideration the graph structure, can you explain them?

```
[19]: X, Y = point_and_circle(num_samples=600)
      plot_clusters(X, Y)
      plt.show()

      k_list = [20, 110, 200]
```

```

laplacian_types = ['unn', 'rw'] # Unnormalized Laplacian and Random-Walk
↳Laplacian

for k in k_list:
    print(f"\nTest with k = {k}")

    # Constructing the k-NN graph
    W = build_similarity_graph(X, k=k)

    # Loop over both types of Laplacians
    for laplacian_norm in laplacian_types:
        print(f"\n- Laplacian used: {laplacian_norm}")

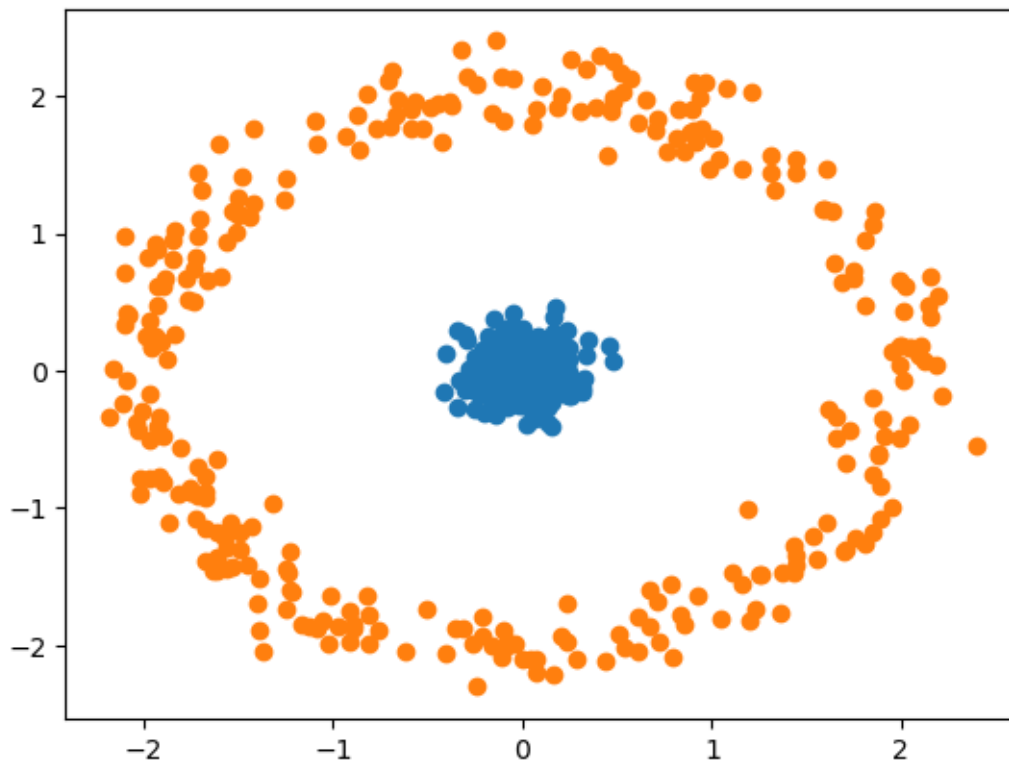
        L = build_laplacian(W, laplacian_normalization=laplacian_norm)

        num_classes = 2
        chosen_eig_indices = [1] # Selecting the second smallest eigenvalue
        Y_rec = spectral_clustering(L, chosen_eig_indices=chosen_eig_indices,
↳num_classes=num_classes)

        Y_kmeans = KMeans(num_classes, n_init=10, random_state=42).fit_predict(X)

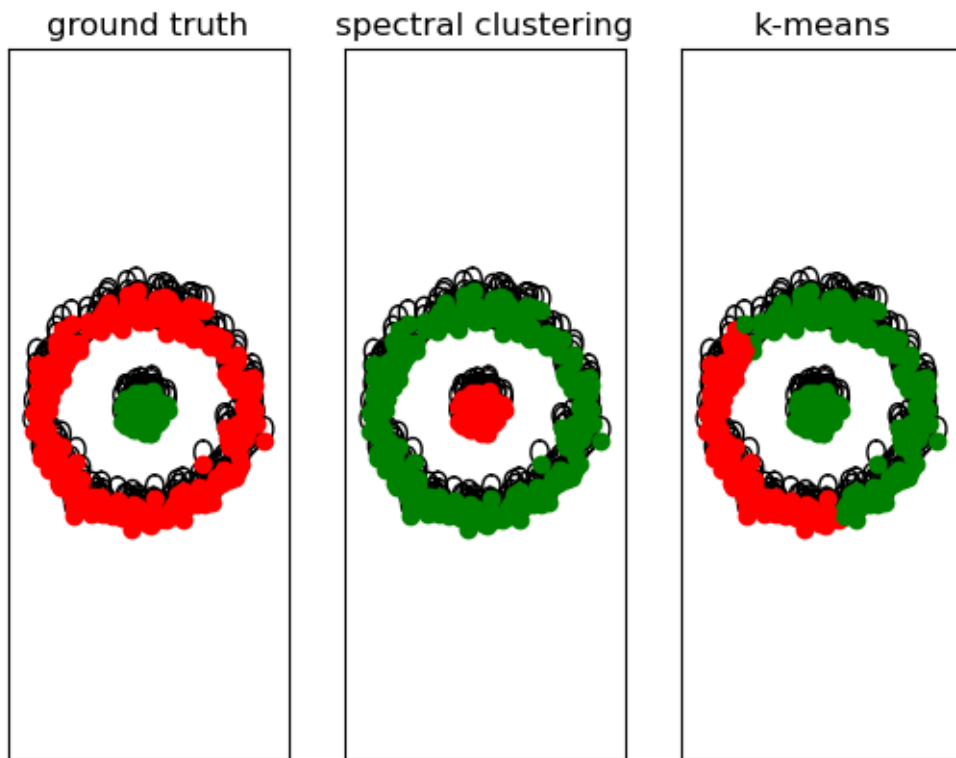
        plot_clustering_result(X, Y, L, Y_rec, Y_kmeans)

```

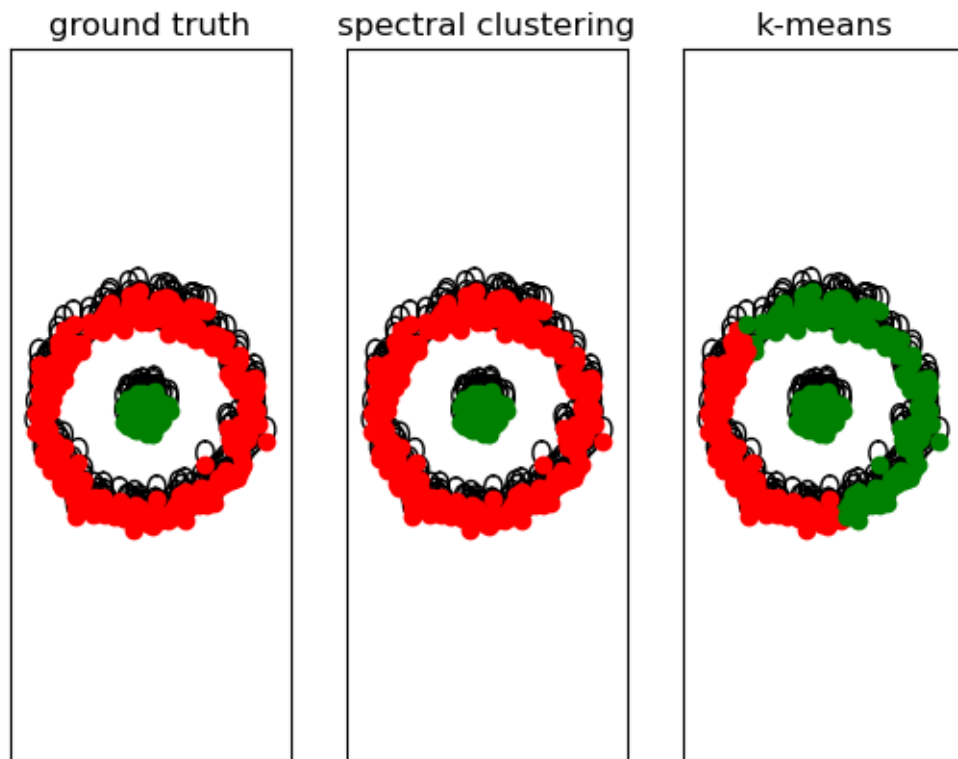


Test with $k = 20$

- Laplacian used: unn

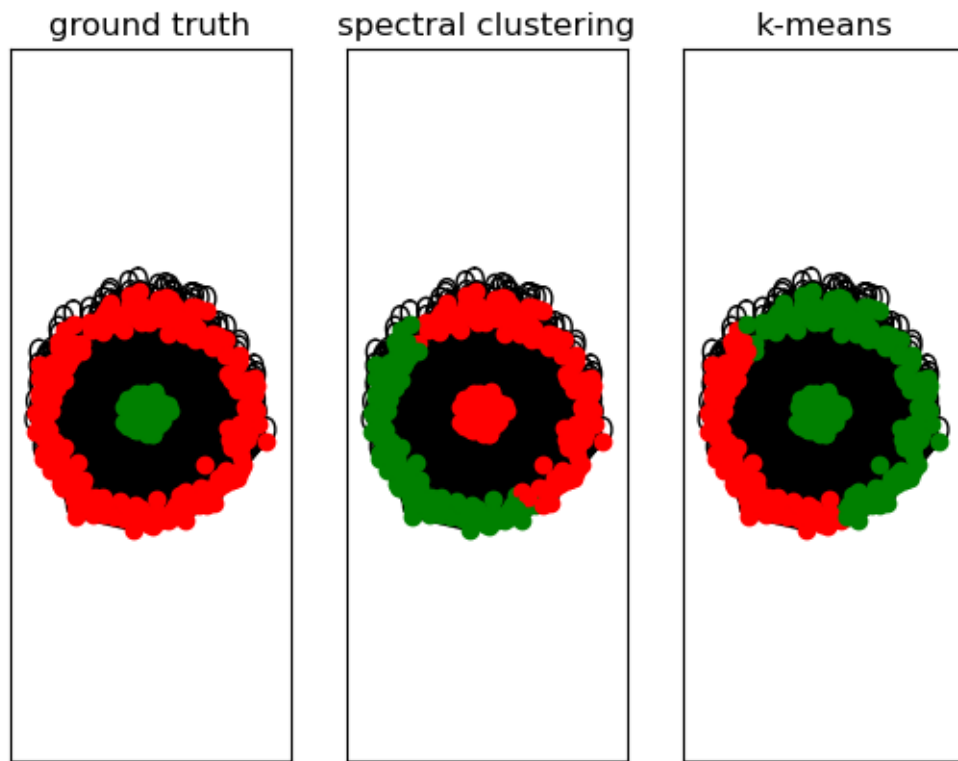


- Laplacian used: rw

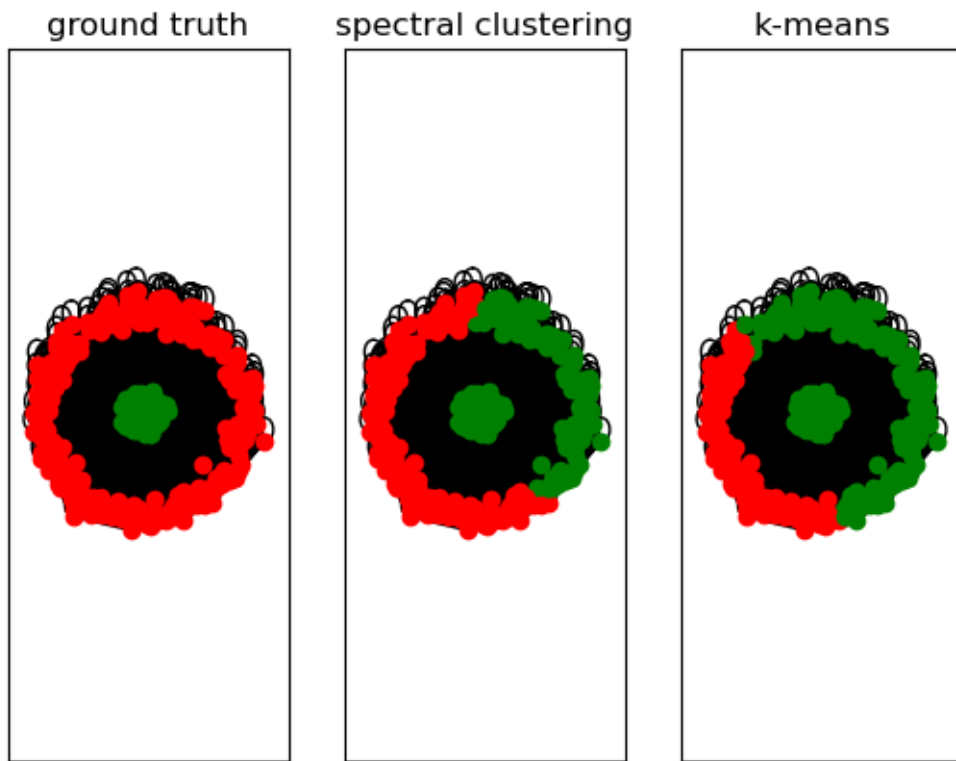


Test with $k = 110$

- Laplacian used: unn

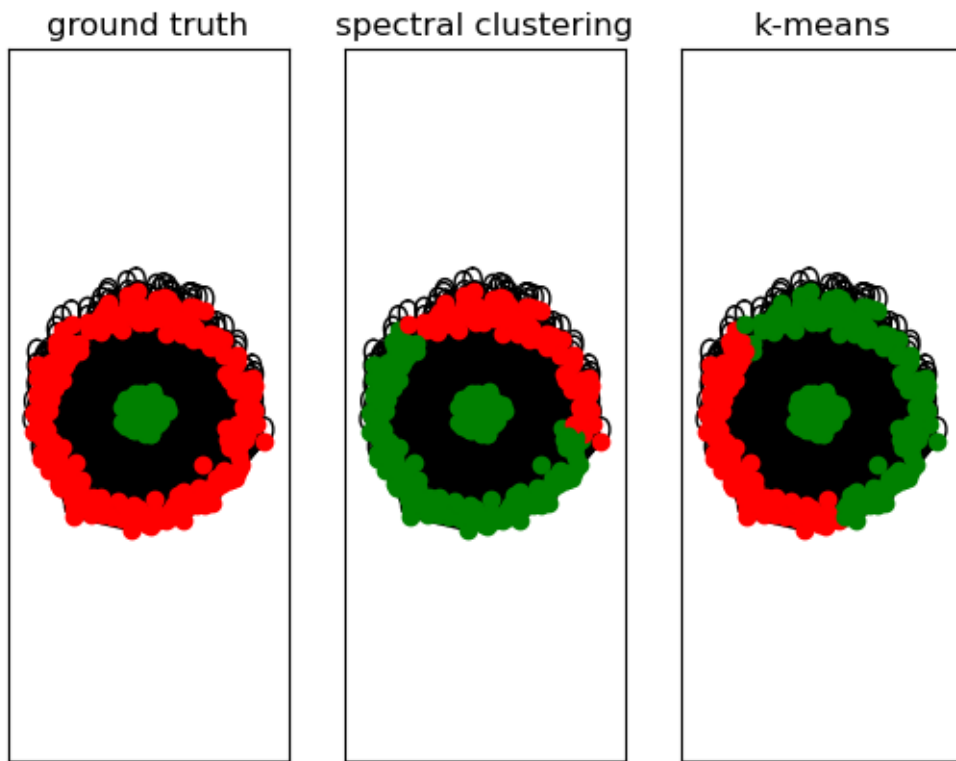


- Laplacian used: rw

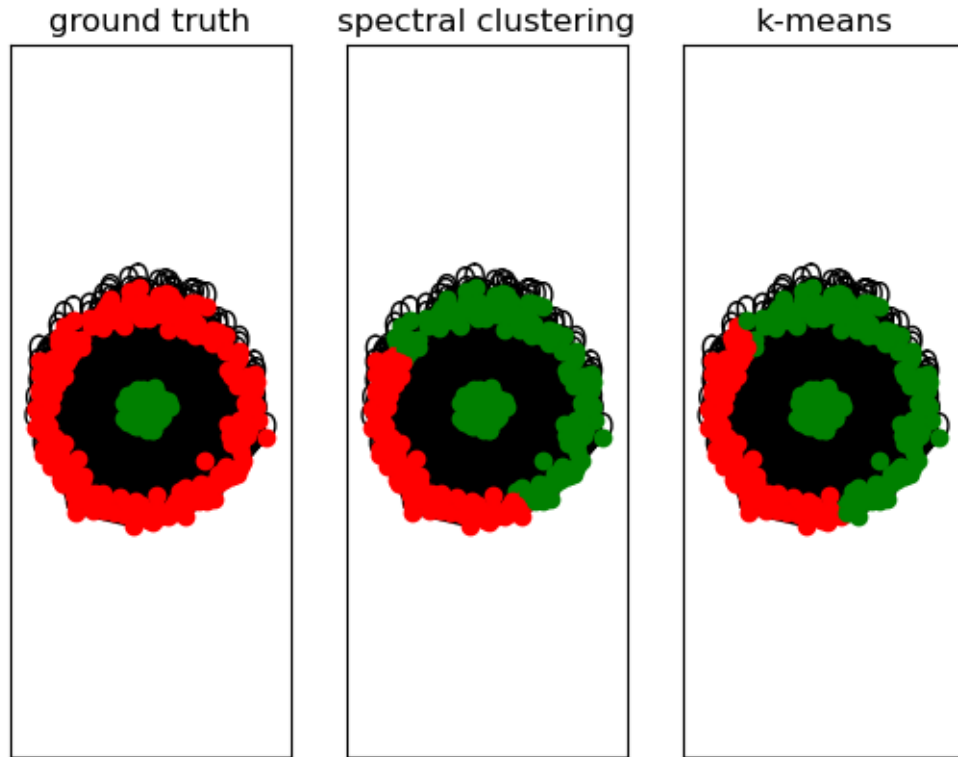


Test with $k = 200$

- Laplacian used: unn



- Laplacian used: rw



With this dataset, we observe three distinct behaviors depending on the choice of k in the k -NN graph.

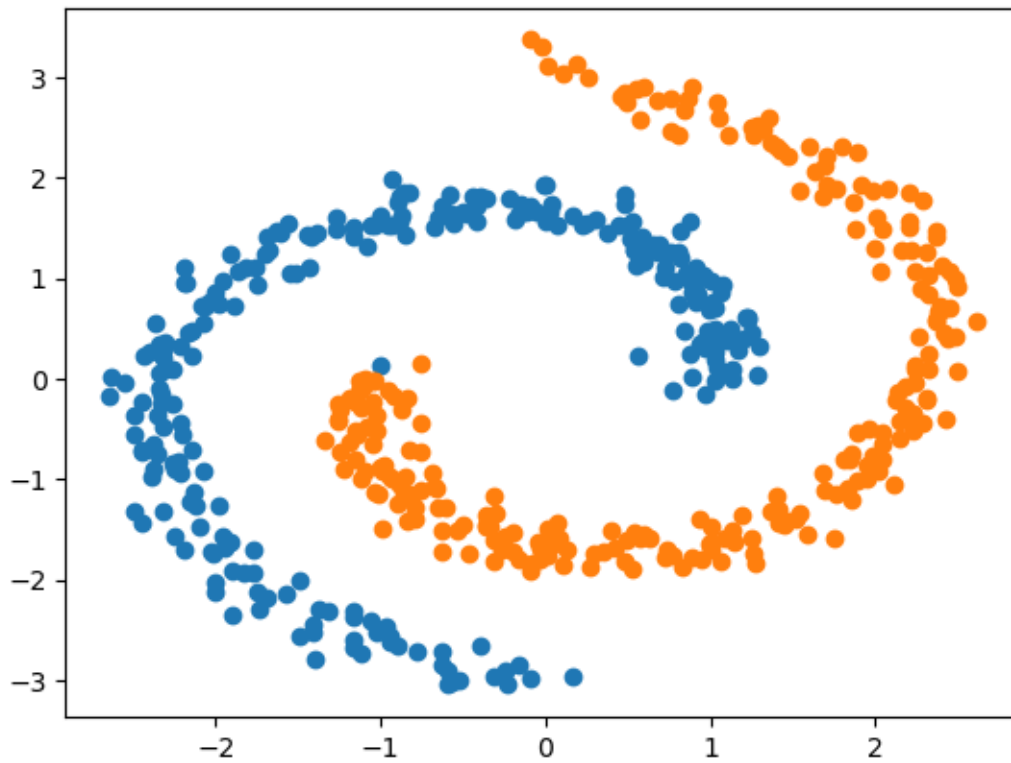
- $k = 20$: The graph naturally forms **two connected components**, making the task easier for Spectral Clustering.
- $k = 100$: Only **Spectral Clustering with the Laplacian L_{rw} (random-walk)** correctly clusters the data. This is due to a property of Spectral Clustering. In fact, with L_{rw} , we optimize the **Ncut (Normalized Cut)**, which helps better separate the groups.
- $k = 200$: Neither algorithm works correctly. The graph is **too dense**, reducing the effect of local structure and preventing **Spectral Clustering and K-Means from recovering the natural cluster separation**.

These results show that **the choice of k in the graph and the type of Laplacian significantly impact the performance of Spectral Clustering**.

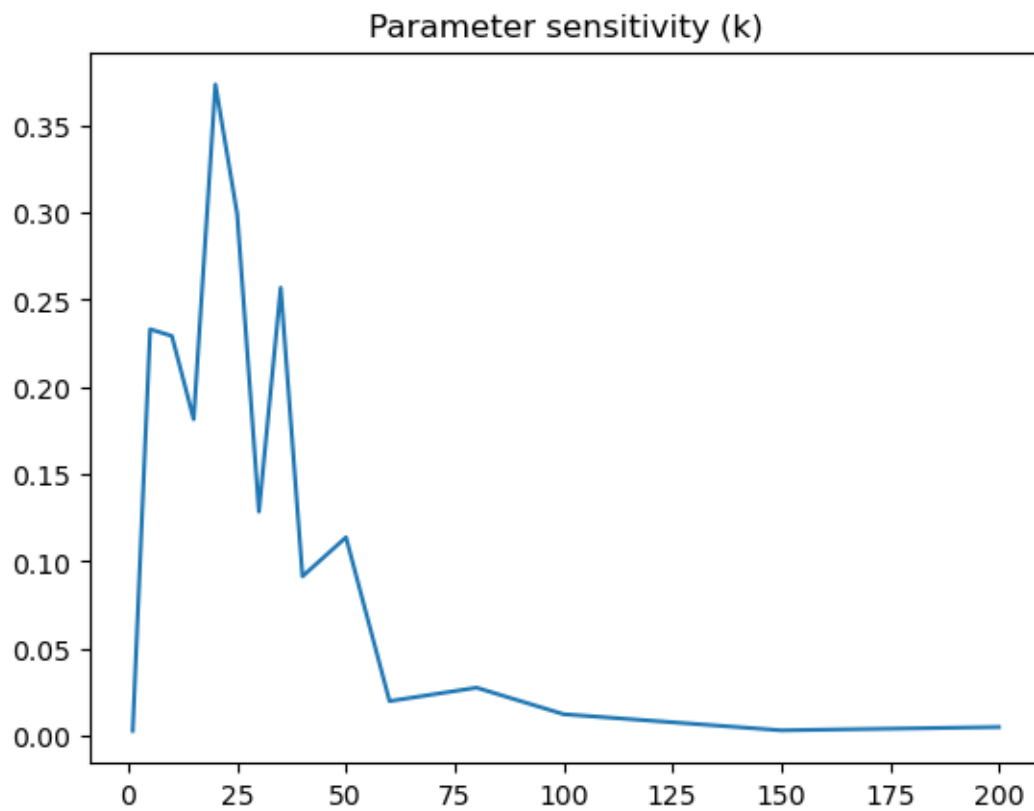
2.0.10 Q2.7: Using the function `parameter_sensitivity`, comment on the stability of spectral clustering when varying ϵ or k .

```
[20]: X, Y = two_moons(500, 1, 0.02)
      plot_clusters(X, Y)
      plt.show()
```

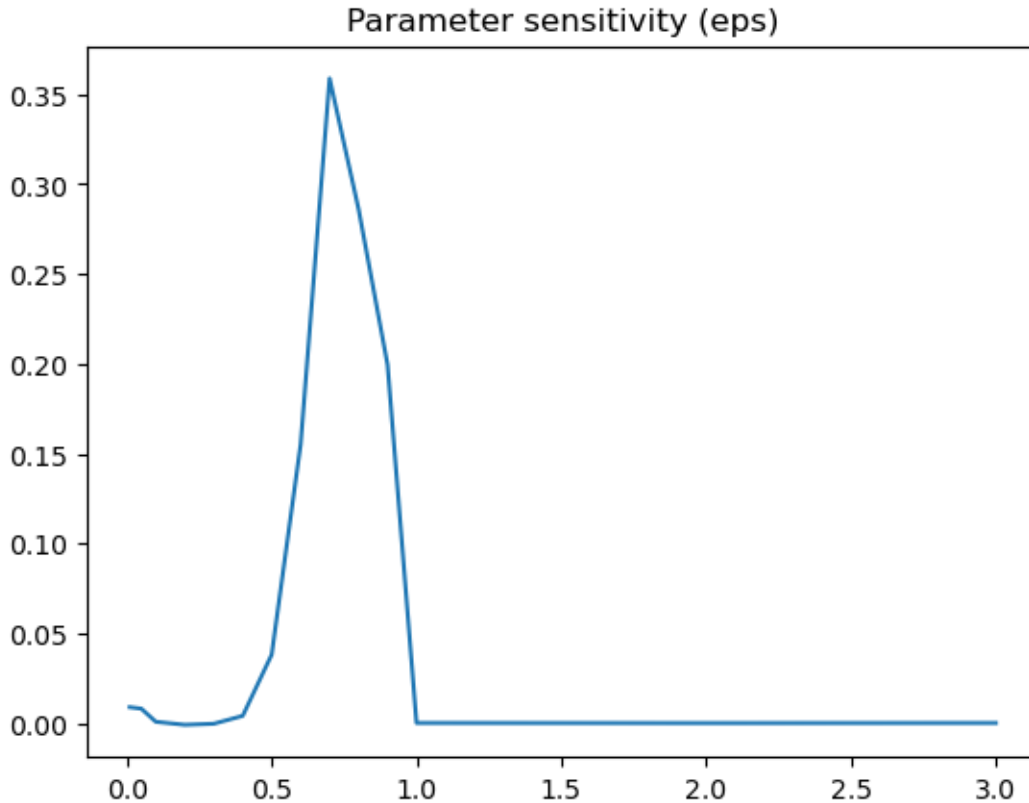
```
parameter_sensitivity()
```



Testing for the k-NN graph



Testing for the epsilon-graph



The results vary significantly depending on k and ε . As observed in this sensitivity analysis, a small change in k or ε can drastically impact the results.

For the `two_moons` dataset (with a variance of 0.02 and a radius of 1), the best results appear to be around $k \approx 15$ and $\varepsilon \approx 0.8$.

2.0.11 Q2.8: If we did not have access to “true” labels, how could we evaluate the clustering result (or what should we not use as evaluation)?

If we do not have access to the “true” labels, it is necessary to rely on mathematical metrics to evaluate clustering performance. Some commonly used metrics include:

- **Dunn Index** ([Wikipedia](#))
- **Davies–Bouldin Index** ([Wikipedia](#))
- **Silhouette Score** ([Wikipedia](#))
- **Modularity**, which is more commonly used for graphs ([Wikipedia](#)).

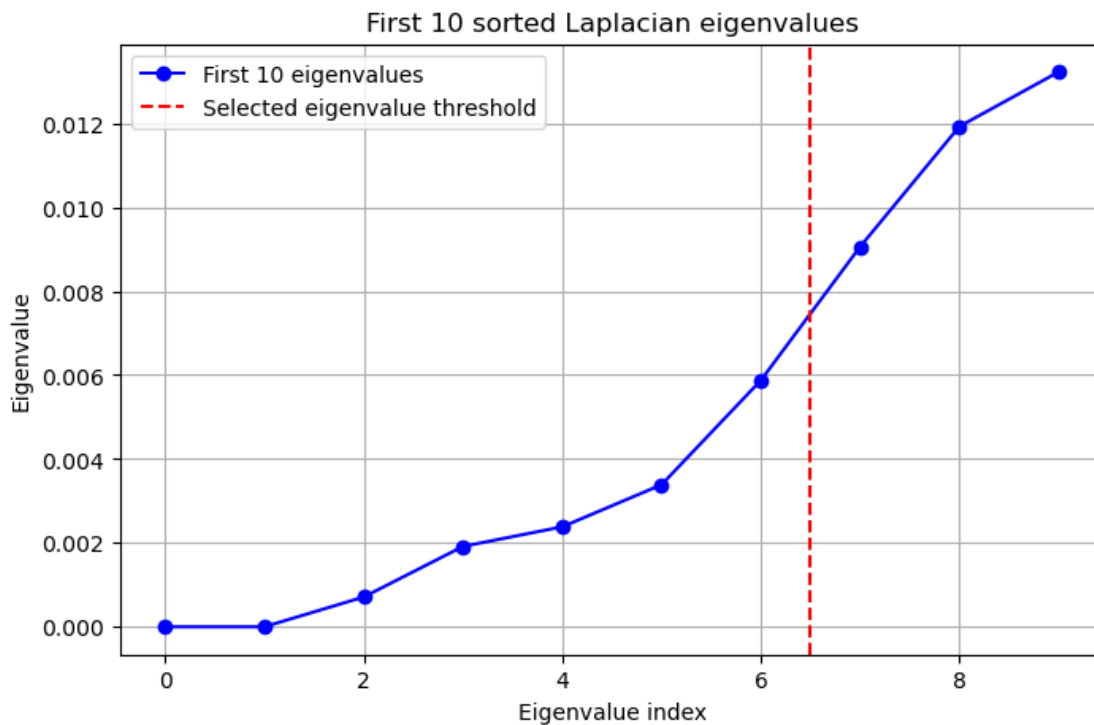
For simple datasets, visually comparing the results can be faster and more intuitive. However, for more complex cases, this method remains anecdotal at best.

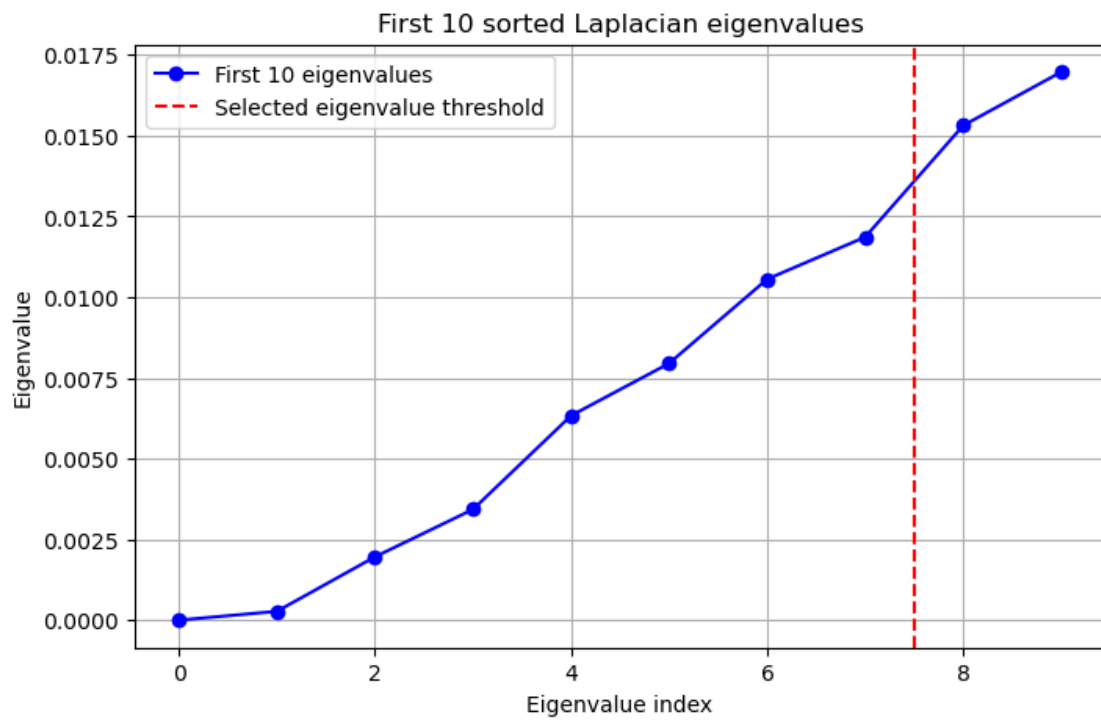
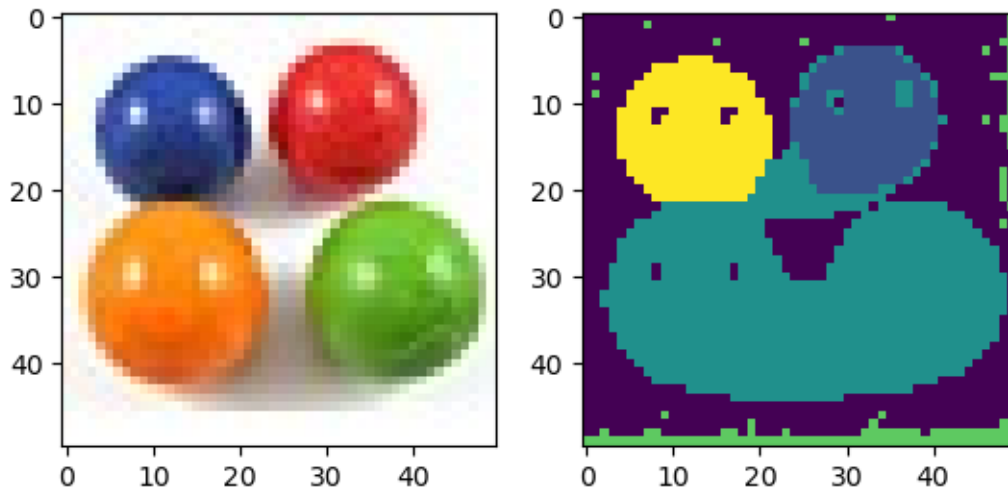
2.0.12 Q3.1: Complete all the code necessary, and cluster the four_elements and fruit_salad images in the data folder and include here some images of the clustering. Explain your design choices in the overall pipeline (e.g. which graph, which pre-processing...) and how you think they impact the results you are seeing. Report also at least one failure mode for the algorithm.

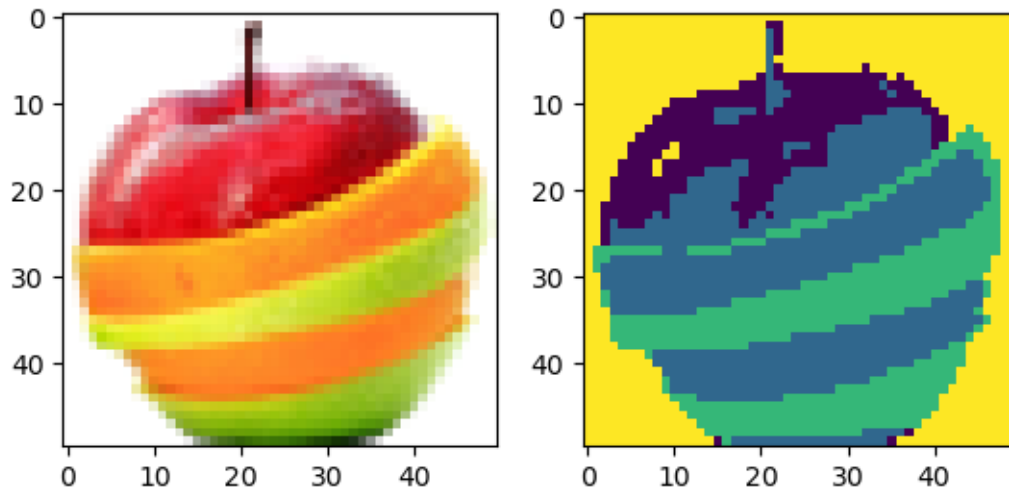
```
[21]: print("\nFirst test with k=10:\n")

image_segmentation('four_elements.bmp', num_classes=5, k=10, var=0.5,
    ↪laplacian_normalization='rw')
image_segmentation('fruit_salad.bmp', num_classes=4, k=10, var=0.5,
    ↪laplacian_normalization='rw')
```

First test with k=10:



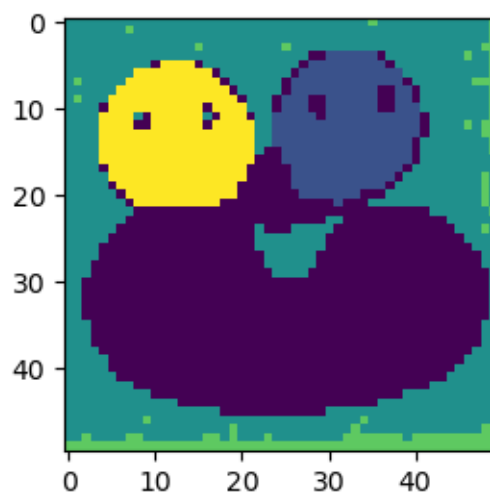
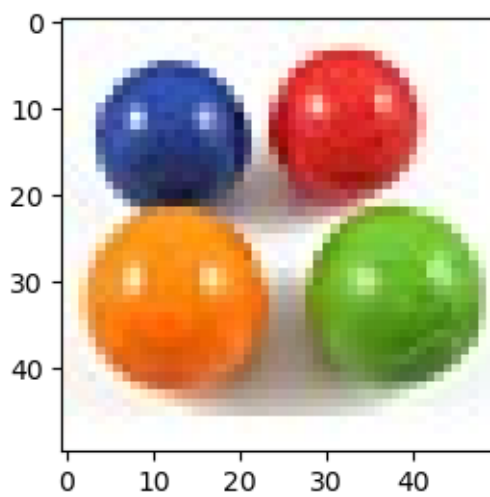
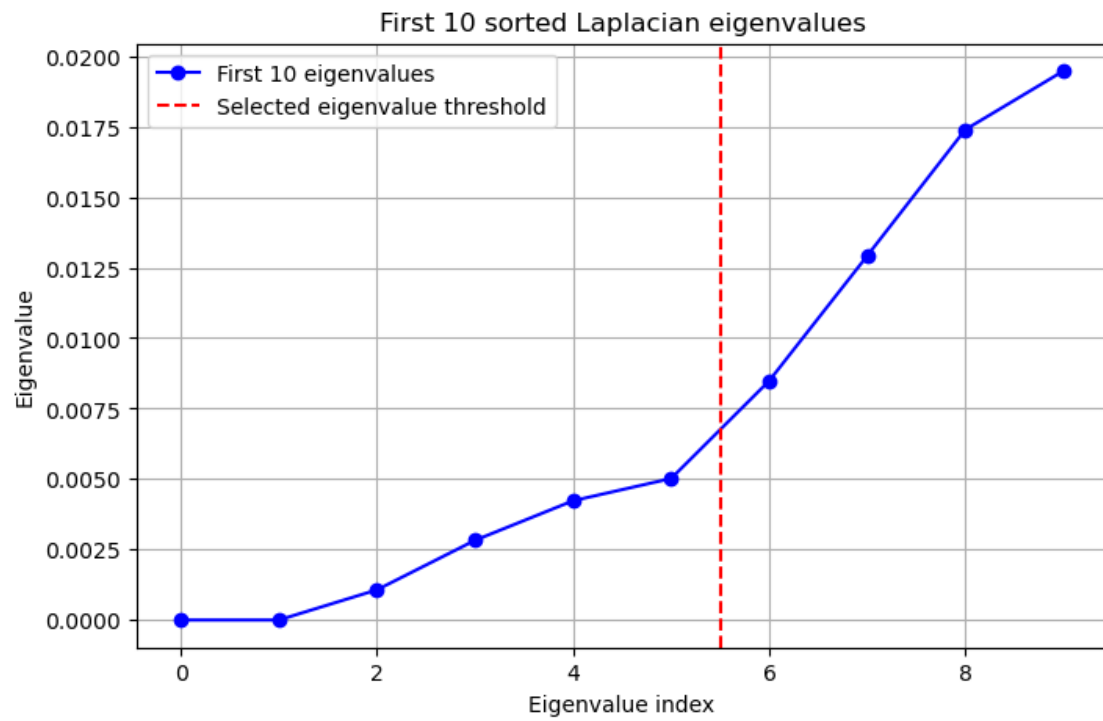


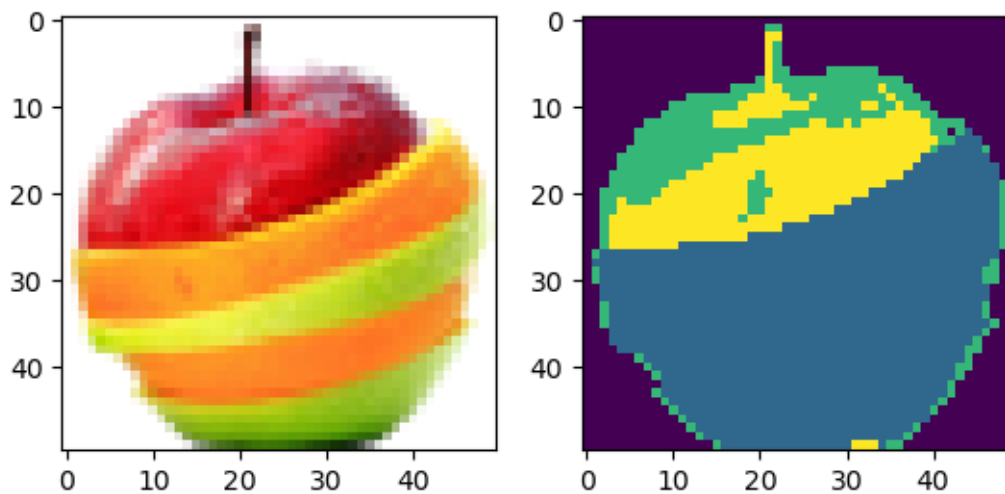
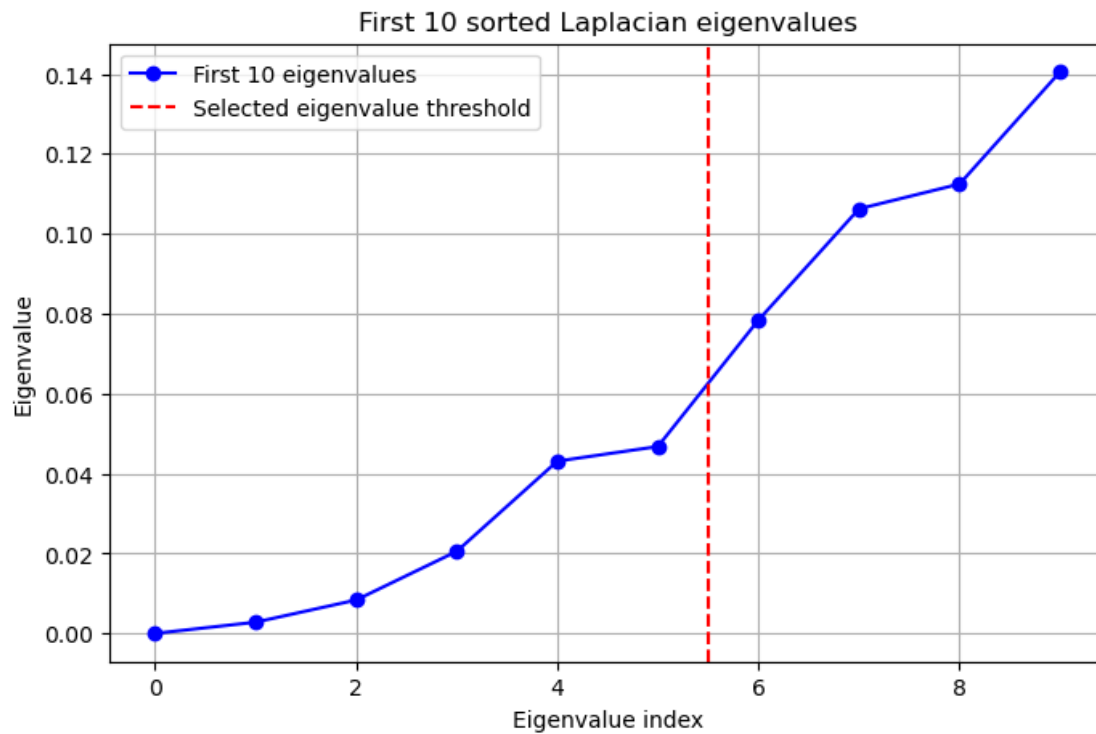


```
[22]: print("\nOther tests:\n")

image_segmentation('four_elements.bmp', num_classes=5, k=15,
    ↪chosen_eig_indices=[0, 1, 2, 3, 4, 5], laplacian_normalization='sym')
image_segmentation('fruit_salad.bmp', num_classes=4, k=80,
    ↪chosen_eig_indices=[0, 1, 2, 3, 4, 5], laplacian_normalization='sym')
```

Other tests:



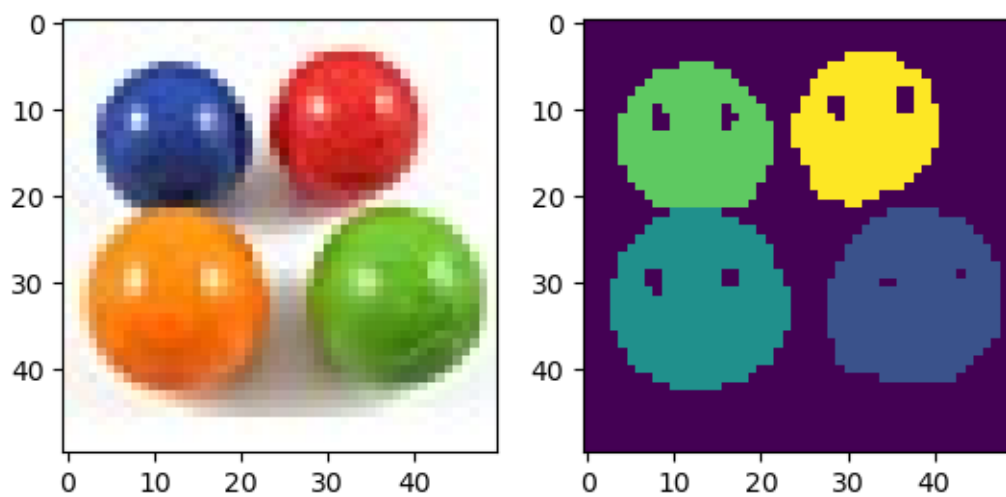
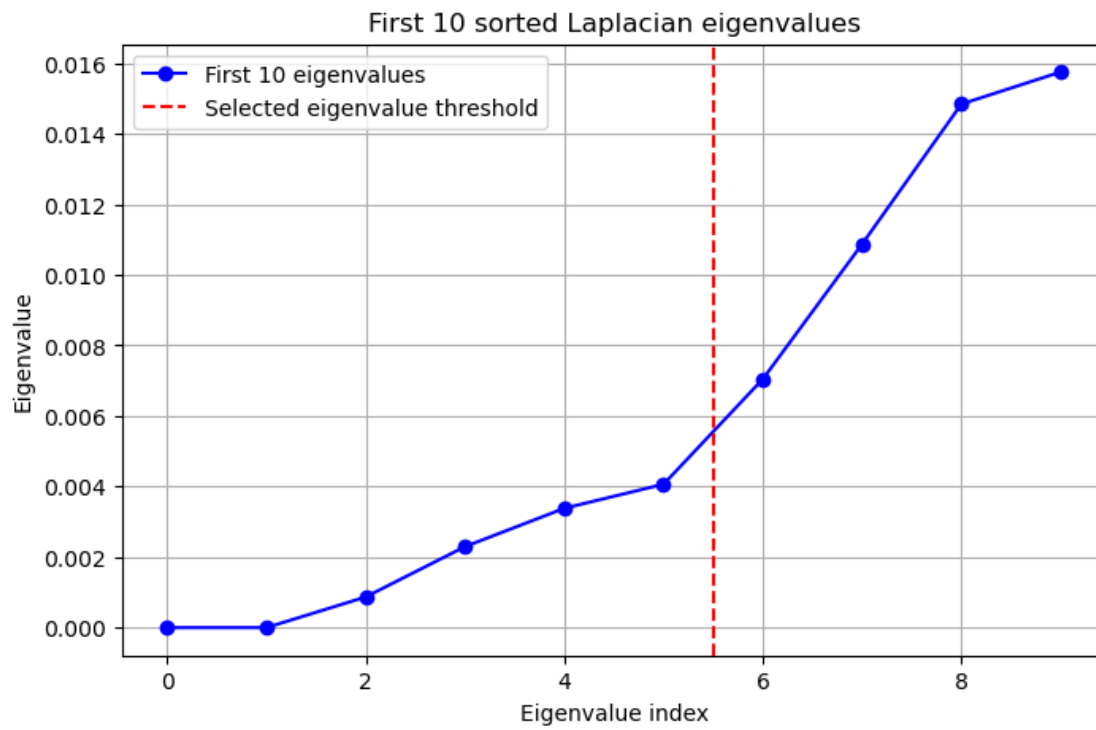


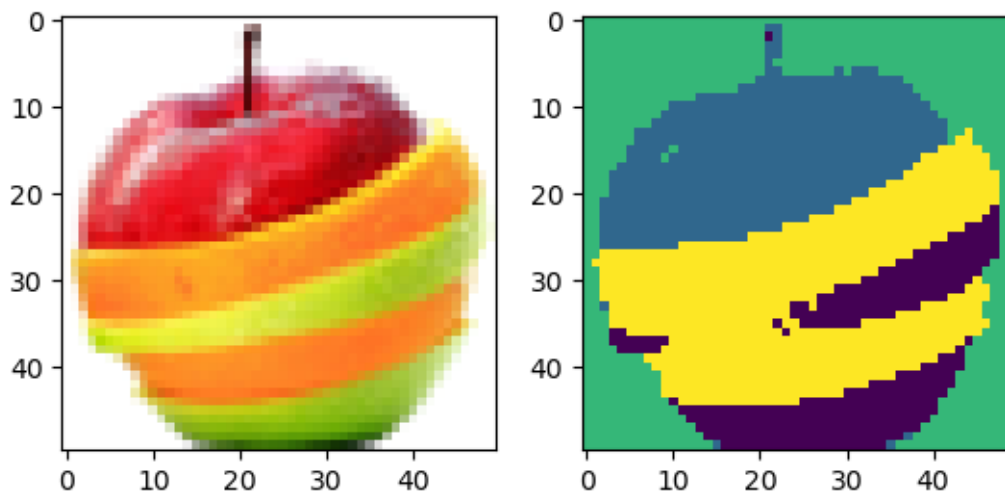
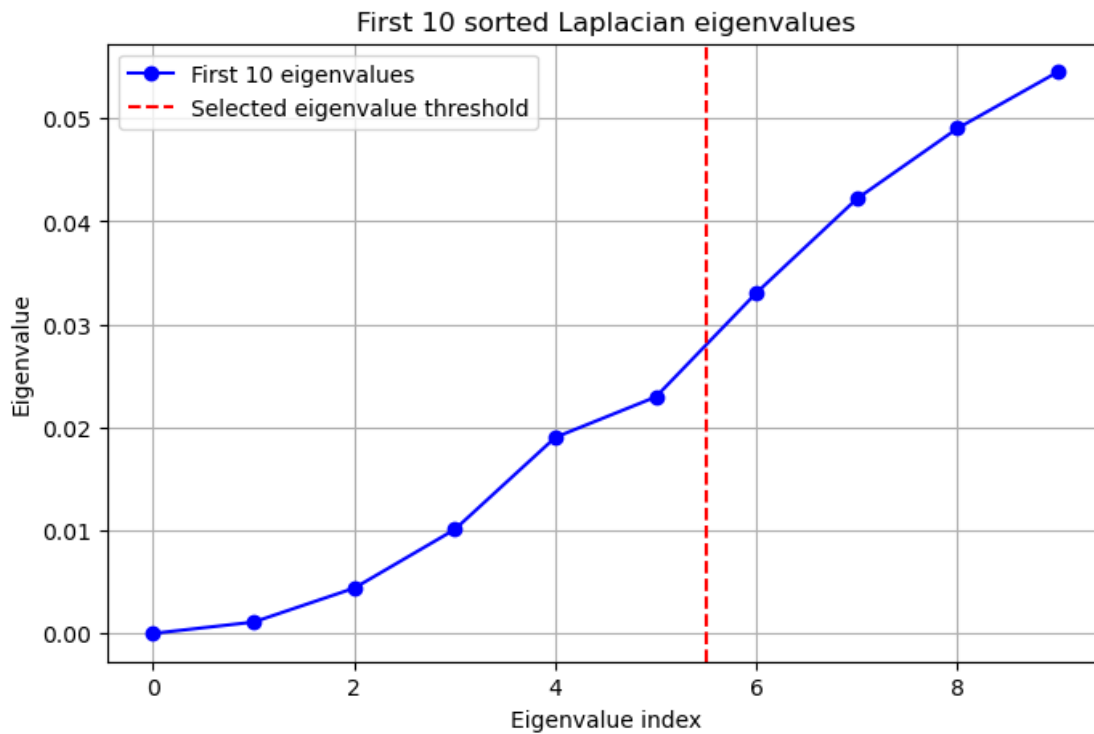
```
[29]: print("\nFinal test:\n")

image_segmentation('four_elements.bmp', num_classes=5, k=12,
↪chosen_eig_indices=[0,1,2,3,4,5], laplacian_normalization='sym')
```

```
image_segmentation('fruit_salad.bmp', num_classes=4, k=35,   
↳laplacian_normalization='rw')
```

Final test:





2.0.13 Q3.1: Clustering four_elements and fruit_salad images

After extensive testing, I have noticed that k is a **highly sensitive parameter**. Even small changes in its value can significantly impact the segmentation results.

Additionally, L_{sym} and L_{rw} seem to be **more stable choices** for the Laplacian normalization.

However, I am unsure why one performs better than the other in certain cases.

Based on my experiments, I found the following parameters to yield the best results:

```
image_segmentation('four_elements.bmp', num_classes=5, k=12, chosen_eig_indices=[0,1,2,3,4,5], 1
image_segmentation('fruit_salad.bmp', num_classes=4, k=24, laplacian_normalization='sym')
```

2.0.14 Q3.2: A full graph built between the pixels of a 50×50 image corresponds to 50^2 nodes. Solving the full eigenvalue problem in this case would scale in the order of 2^{34} . Even on weak hardware this takes only seconds to minutes. Segmenting a Full HD picture of 1920×1080 would scale in the order of 2^{64} (about a month on a decent machine). Beyond that, the large picture would require to store in memory a graph over millions of nodes. A full graph on that scale requires about 1TB of memory. Can you think two simple techniques to reduce the computational and occupational cost of Spectral Clustering?

For Full HD images, it quickly becomes clear that computational time will be a major issue. To address this, we can consider two simple strategies:

- **Reduce the image size** by decreasing the number of pixels. This naturally lowers the computational cost.
- **Use a non-complete graph**, such as a k -NN or an ϵ -graph. This drastically reduces the number of nodes, which in turn speeds up the Spectral Clustering process.

These two approaches allow Spectral Clustering to be applied to large images while still achieving effective segmentation.

2.0.15 Q3.3: Choose a larger image of your choice, and try to implement some of the scalability modifications to run spectral clustering on it. Report some examples of the results you achieve.

[]: