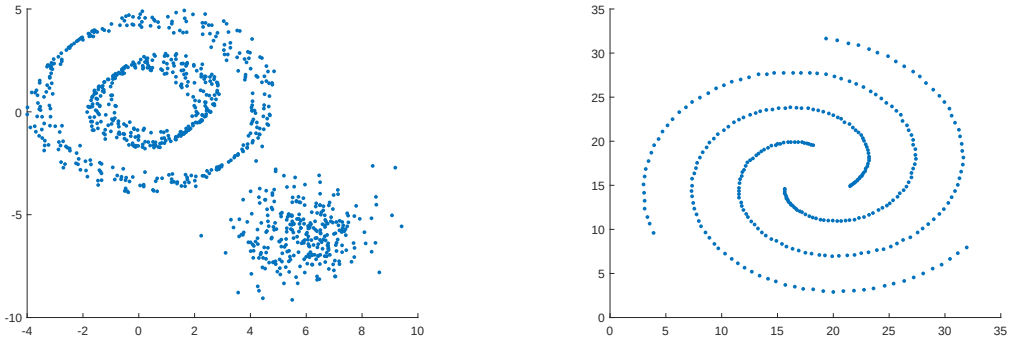
Computational Linear Algegbra For Large Scale Problems

Nenna Giulio, Ornella Elena Grassi

Spectral Clustering Homework

The aim of this homework is to implement and apply **Spectral Clustering** to two different sets of datapoints in $\mathbb{R}^2$. The two sets are shown in Figure 1



(a) Scatterplot of the data stored in `Circle.mat`

(b) Scatterplot of the data stored in `Spiral.mat`

Figure 1: Scatterplot of the two Datasets

As it is clearly visible through visual inspection, both datasets contain 3 different shapes that can be classified as different clusters. In the `Circle` dataset there are two concentrical circles and a cloud of points in the bottom right while in the `Spiral` dataset there are 3 spirals. Traditional clustering algorithms, that mainly rely on euclidean distance, may fail in recognizing the presence of shapes in our data hence our need to rely on a different technique called **Spectral clustering**.

1 K-Nearest Neighborhood Graph

First, we need to define a similarity function that measures "how much our points are similar to each other". Let X_i and X_j be two points in our data, then we will use a similarity measure defined as:

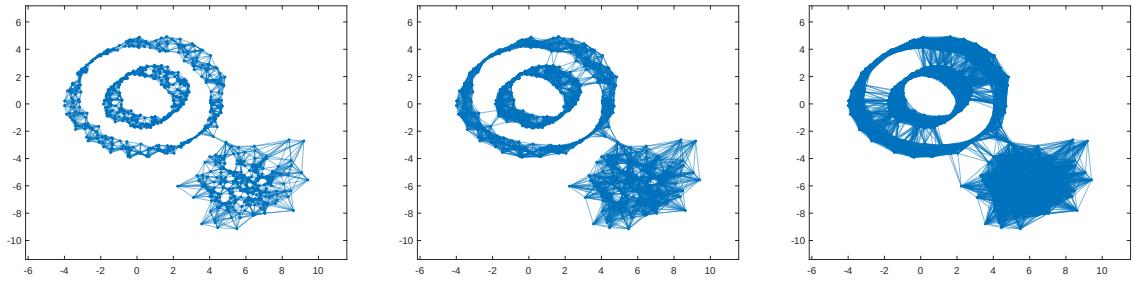
$$s_{i,j} = \exp\left(-\frac{\|X_i - X_j\|^2}{2\sigma^2}\right) \tag{1.1}$$

Then, a *K-Nearest Neighborhood* similarity graph is a Graph $G = (V, E)$ where each vertex $v_1, \dots v_n$ represents a point and two vertices v_i and v_j are connected by an undirected edge $e_{i,j}$ if the similarity between v_i and v_j is among the K-th highest similarities between v_i and other vertices in V. For such graph we can define the relative adjacency matrix as $W_{i,j} = s_{i,j}$ where each entry $W_{i,j}$ is nonzero only if there exists an edge between v_i and v_j. W has zero-values on the diagonal by definition.

The following MATLAB code was use to generate the K-NN similarity graph of our data:

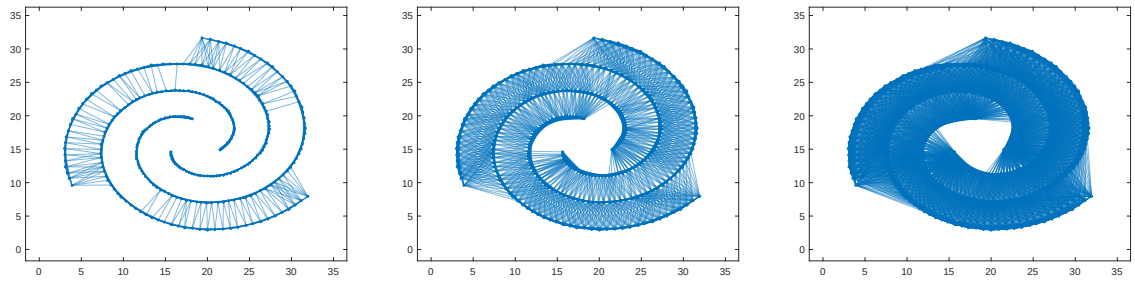
```
function [G, W] = knn_graph(X, K, sigma)
%An adjacency matrix of a K-nn graph has, for each column and row, only K
%nonzero elements that correspond to the nearest neighbors of each point.

N = length(X);
X = X(:,1:2);

W = sparse(N, N);
for i = 1:N %for each point
    %Compute the similarity for every other point
    sim = exp(-((X(i, 1)- X(:, 1)).^2 + (X(i, 2)- X(:, 2)).^2)/(2*sigma^2));
    sim(i) = 0; %set similarity with itself to 0
    [sim, idx] = maxk(sim, K); %Compute the K most similar points and its indices
    W(i, idx) = sim; %set values of the adjacency matrix
    W(idx, i) = sim;
end
G = graph(W);
```

Running `knn_graph` on both dataset using $\sigma = 1$ and testing with $K = 10, 20, 40$ gave the following results:



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 2: K-NN graphs plots for `Circle` data with different values of K



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 3: K-NN graphs plots for `Spiral` data with different values of K

Our objective is to find a K-NN graph that "separates well" the shapes, meaning that in the best case each shape is a connected component of the graph. As we can see, for both `Circle` and `Spiral` data, the value $K = 10$ seems to separate the shapes well. We will quantify the concept of *good cluster separation* in the next section.

2 Degree and Laplacian Matrices

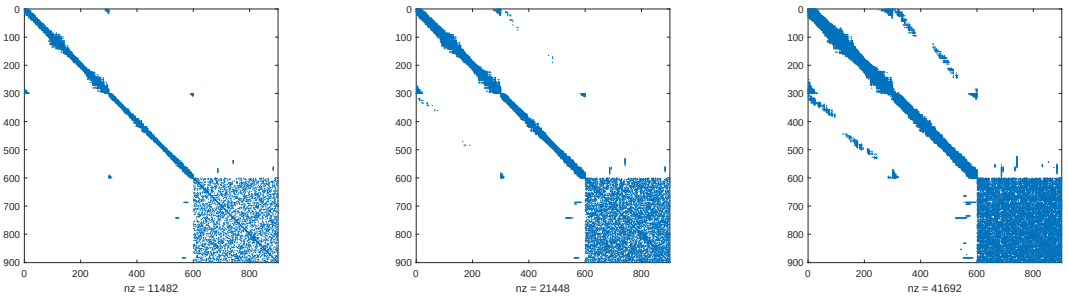
The *Degree matrix* D is a diagonal matrix that has the degree of each node on the diagonal where the degree of a node v_i is the sum of the weights of all the edges that are connected to such node. Since in our case edge weights correspond to similarities, the degree of each node is the sum of the similarities with all of its neighbors. This can be simply achieved with the following:

$$D = \text{diag}(W\mathbb{1}) \tag{2.1}$$

Since the Laplacian matrix is defined as $L = D - W$, we use the following code to compute both D and L:

```
function [L, D] = graph_laplacian(W)
    D = sparse(diag(sum(W)));
    L = D-W;
end
```

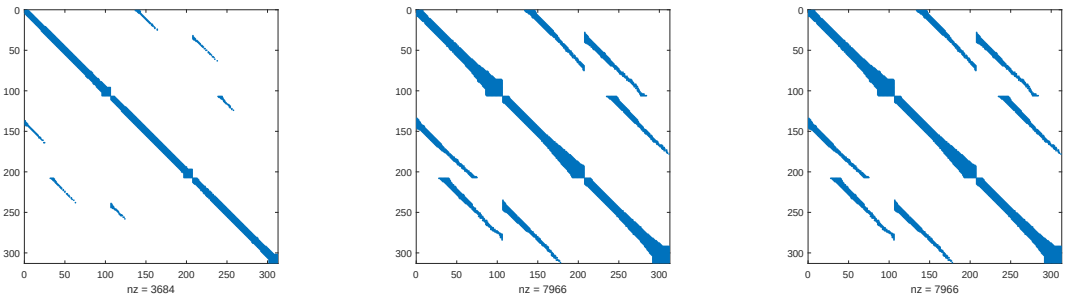
After computing both matrices for each dataset and for each value of K we are testing for the K-NN graph we can inspect each Laplacian matrix using the `spy` command in MATLAB:



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 4: `spy` plots of the Laplacian matrix for `Circle` data with different values of K

In Figure 4 we can see that, for each Laplacian matrix inspected, there are 3 sections. Each section defines a "*shape cluster*" in the `Circle` dataset: the first one is clearly visible in the bottom right corner of each matrix and it is relative to the point cloud. The other two sections are along the diagonal in the top left of the matrix, they are somewhat visible for $K = 10$ and $K = 20$ and they are relative to the two "circles" visible in the scatterplot in Figure 1. The Laplacian matrix inspected for $K = 40$ clearly shows a sort of **data pollution** from one shape cluster to another, this might suggest that the value $K = 40$ may be too high for deploying the clustering.



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 5: `spy` plots of the Laplacian matrix for `Spiral` data with different values of K

The same concept can be applied for Figure 5. Along the diagonal, three sections are visible and they represent the three different spirals seen in Figure 1. As for `Circle` data, the KNN graph

computed on the `Spiral` data with $K = 40$ shows pollution between each spiral and this is visibile since there are a lot of non-zero elements away from the diagonal.

The recurrent theme for both dataset is that the more K is higher, the more each shape will "pollute" another. This means that the corresponding Laplacian matrix will show a **lot of non-zero elements away from the diagonal** and this is a phenomenon that can (and will) impact computational costs.

3 Connected components computation

In order to compute the number of connected components in each graph we use the following result:

Theorem 3.1. *Let $G = (\mathcal{V}, W)$ be a finite graph and let L be its laplacian matrix. Then L has $\lambda = 0$ as an eigenvalue and its algebraic multiplicity correspond to the number of connected components in G.*

Using the Matlab function `eigs` we can compute the smallest $C = 6$ eigenvalues of both graphs laplacian matrices L and for each value of K obtaining the following results:

$K = 10$	$K = 20$	$K = 40$
8.5482e-17	2.4854e-16	8.1617e-16
3.7050e-16	0.0111	0.0482
0.0048	0.0652	0.7028
0.0286	0.1620	0.7797
0.0425	0.1724	0.9065
0.0429	0.3220	1.376

Table 1: Smallest 6 eigenvalues of the Laplacian matrix for the `circle` dataset

$K = 10$	$K = 20$	$K = 40$
1.0496e-16	1.7853e-16	4.0554e-16
1.9667e-04	0.0018	0.0023
2.7219e-04	0.0020	0.0025
0.0041	0.0048	0.0049
0.0044	0.0054	0.0062
0.0046	0.0056	0.0067

Table 2: Smallest 6 eigenvalues of the Laplacian matrix for the `spiral` dataset

From a visual inspection of both dataset we would expect to find 3 connected components and consequently an eigenvalue $\lambda = 0$ of both laplacian matrices with algebraic multiplicity $M = 3$.

Considering the case $K = 10$ for both datasets we can see that the first 3 eigenvalues are "almost" equal to 0, and we can see a jump in order of magnitude from the fourth eigenvalue. On the other hand the cases $K = 20$ and $K = 40$ show that only the first eigenvalue could be considered equal to 0 (which is always true) but the second and third eigenvalues are somewhat smaller than the others.
This behaviour is perfectly in line with the "data pollution" concept seen graphically in 2. We can in fact interpret an "almost null" eigenvalue as an eigenvalue corresponding to an "almost connected component". The less the component is "isolated", the greater its eigenvalue will be. In the case $K = 10$ we clearly have 3 connected components that are isolated fairly well, hence the value of the first 3 eigenvalues are almost 0 for both graphs. In the cases $K = 20$ and $K = 40$, the 3 connected

components that we expected are not well isolated and present a lot of edges from one to another, hence the first 3 eigenvalues are not small enough to be considered null.

In any case we consider $M = 3$ the number of connected components and construct the matrix $U \in \mathbb{R}^{N\times 3}$ using the $M = 3$ eigenvectors u_1, u_2, u_3 corresponding to the first 3 eigenvalues as columns.

The following Matlab code was used for the computation in this section:

```
%% --- TASK 3-4-5 ---
n_eigs=6;
U_circle={};
U_spiral={};
eigs_circle={};
eigs_spiral={};

for i=1:length(K) %for each value of K tested
    [U_circle{i}, eigs_circle{i}] = eigs(L_circle{i}, n_eigs, 'smallestabs'); %compute the n_eigs
     smallest eigenvalues for circle
    [U_spiral{i}, eigs_spiral{i}] = eigs(L_spiral{i}, n_eigs, 'smallestabs'); %same for spiral
end

for i = 1:length(K)
    %simple code to generate a matrix where in each column the eigenvalues
    %are listed for each value of K tested
    eigs_circle_tot(:,i) = diag(eigs_circle{i});
    eigs_spiral_tot(:,i) = diag(eigs_spiral{i});
end
M=3; %using the first 3 eigenvectors
for i = 1:length(K)
    %trim the matrix U to the first M columns
    U_circle{i} = U_circle{i}(:, 1:M);
    U_spiral{i} = U_spiral{i}(:, 1:M);
end
```

4 Performing Spectral Clustering

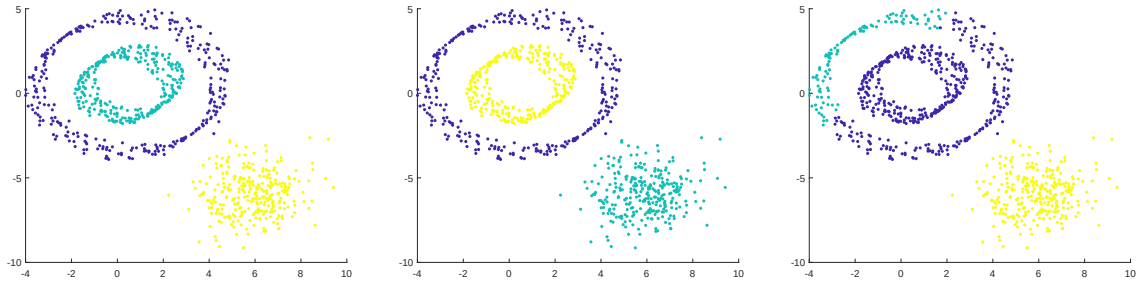
After computing the matrix $U \in \mathbb{R}^{N\times 3}$ for both the `circle` and the `spiral` datasets and for each $K \in \{10, 20, 40\}$ we can now finally deploy a clustering algorithm.

Let $y_i \in \mathbb{R}^M$ be the vector corresponding to the i-th row of the matrix U. Then we can use the *K-Means* algorithm to cluster the points y_i, $i \in \{1, \dots, N\}$ into M clusters and assign the original datapoints to the same cluster as their corresponding rows in U.

The following code was used in order to perform clustering:

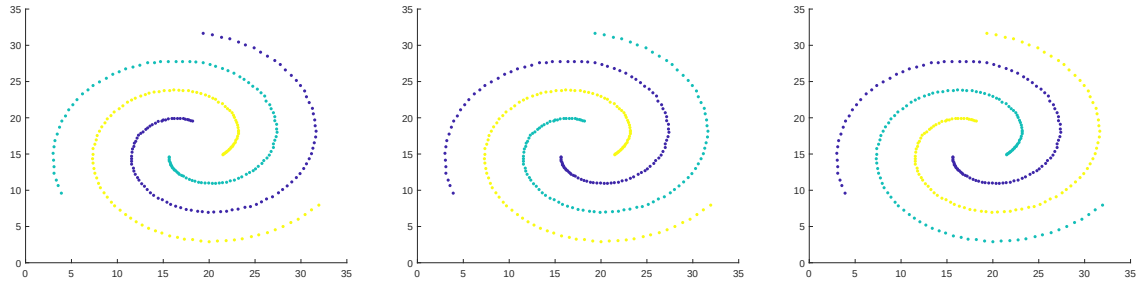
```
%% --- TASK 6-7-8---
M=3;
idx_circle={}; %cluster labels for circle dataset
idx_spiral={}; %same for spiral


for i = 1:length(K)
    idx_circle{i} = kmeans(U_circle{i}, M);
    idx_spiral{i} = kmeans(U_spiral{i}, M);
end
```

Hence for both dataset we obtain 3 different clusterings, one for each value of K tested. Plotting the results we obtain:



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 6: Spectral clustering for `circle` data with different values of K



(a) $K = 10$ (b) $K = 20$ (c) $K = 40$

Figure 7: Spectral clustering for `spiral` data with different values of K

As it is clearly visible in Figure 6, performing spectral clustering on the `circle` data yields good results both in the case with $K = 10$ and $K = 20$. The same couldn't be said about the case $K = 40$ where the effect of pollution from one shape to another leads to poor results in terms of the proficiency of the algorithm to distinguish shapes.

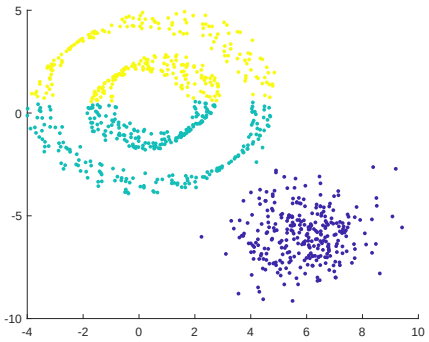
Although not every value of K yields good results for the `circle` data, the same isn't true for the `Spiral` data, where the presence of edges between spirals in the adjacency graph seems to have no effects on the fitness of the clustering, even with $K = 40$.

5 Comparison with other clustering algorithms

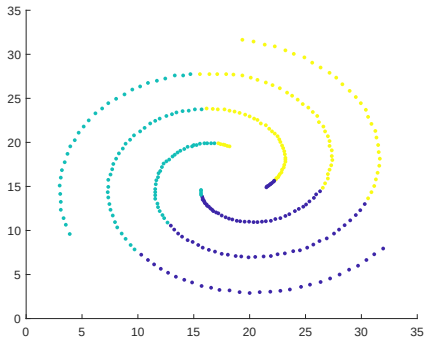
In this section we will test other clustering algorithm on the data, visualizing how they perform in comparison with the spectral clustering performed above.

5.1 Plain *Kmeans* Algorithm

Performing the plain *K-means* algorithm on our data yields poor results since the algorithm is based on euclidean distance. This is the reason it fails to recognize shapes other than balls.



(a) *K-means* algorithm performed on `circle`



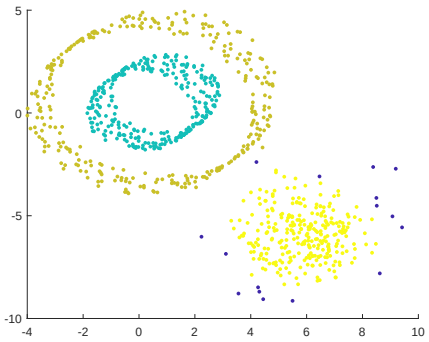
(b) *K-means* algorithm performed on `spiral`

Figure 8: *K-means* algorithm performed onto the datasets

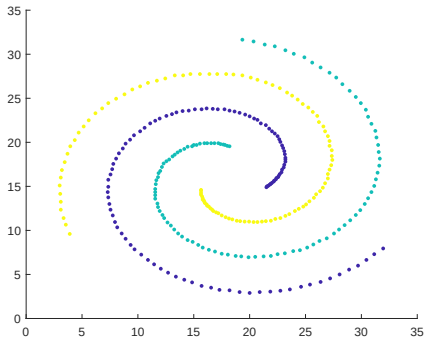
As it is clearly visible in figure 8, the *K-means* algorithm performed on spatial data is only able to recognize the cloud of points in the `circle` data since is euclidean-ball shaped. It clearly fails to recognize any other shape and this results in a very poor clustering with regard to shaped data.

5.2 DBSCAN algorithm

The DBSCAN algorithm is an "explorative" approach to clustering. Without going into details, at its core the DBSCAN algorithm populate each cluster by searching the neighborhood of each point in the cluster hence performs much better at recognizing shapes than the plain *K-means* algorithm.



(a) DBSCAN algorithm performed on `circle`



(b) DBSCAN algorithm performed on `spiral`

Figure 9: DBSCAN algorithm performed onto the datasets

The results of the clustering are shown in Figure 9. Those are obtained using the `dbscan` function in MATLAB performing some minor tuning of the parameters. In particular the clustering for the `circle` data was obtained using the default metric of the algorithm which is the euclidean distance while for the `spiral` data the Mahalanobis distance was used.

As it is visible in Figure 9 the DBSCAN algorithm is perfectly capable of recognizing shapes and correctly classifies all points of the two datasets with the only exception of some points in the `circle` dataset.

6 Using the Normalized Laplacian Matrix

When using a graph $G = (\mathcal{V}, W)$ with weight matrix W and Laplacian matrix L as defined in 2 we can introduce the notion of *Normalized Symmetric Laplacian Matrix:*

$$L_{\text{sym}} := D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}. \tag{6.1}$$

We can easily prove that if λ and x are respectively eigenvalue and eigenvector that solve the *Generalized Eigenvalue problem*:

$$Lx = \lambda D x \tag{6.2}$$

then *lambda* and $y := D^{\frac{1}{2}} x$ are the solution to the canonical eigenvalue problem of L_{sym}, in fact:

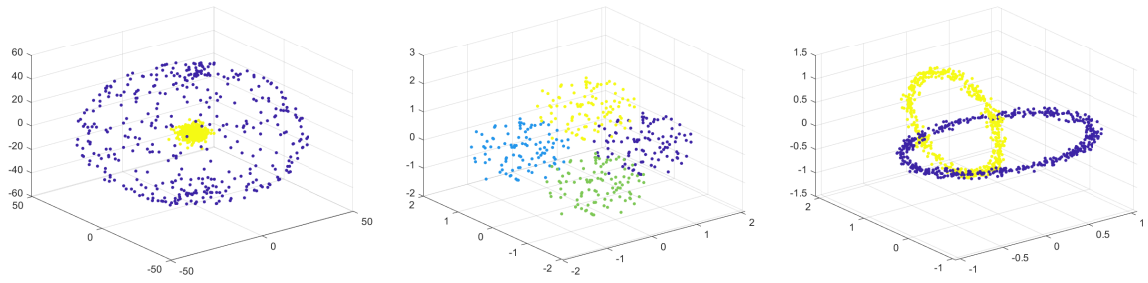
$$\begin{aligned} Lx &= \lambda D x \\ D^{-\frac{1}{2}} L x &= \lambda D^{\frac{1}{2}} x \\ D^{-\frac{1}{2}} L D^{-\frac{1}{2}} \underbrace{D^{\frac{1}{2}} x}_{y} &= \lambda \underbrace{D^{\frac{1}{2}} x}_{y} \end{aligned}$$

This means that L and L_{sym} have the same eigenspace relative to the eigenvalue 0 hence we can use L_{sym} for our spectral clustering purposes.

Testing the spectral clustering algorithm onto our datasets using the normalized laplacian matrix yielded the exact same results in terms of accuracy and performance, hence for our 2D data there is no tangible benefit in using one matrix over the other.

7 Testing the algorithm on other datasets

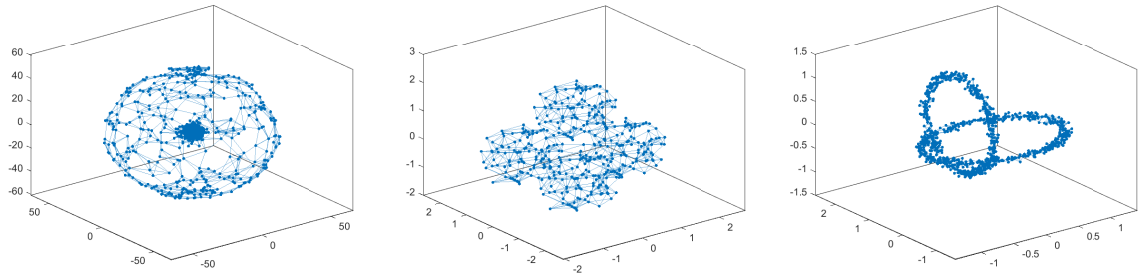
We tried to use the spectral clustering algorithm on other datasets, focusing on 3-dimensional clustering problems. Three datasets have been chosen to benchmark the algorithm: the `tetra`, `chainlink` and `atom` datasets. They were downloaded from this link.



(a) `atom` dataset (b) `tetra` dataset (c) `chainlink` dataset

Figure 10: 3D-scatterplots of the three datasets used for benchmarking

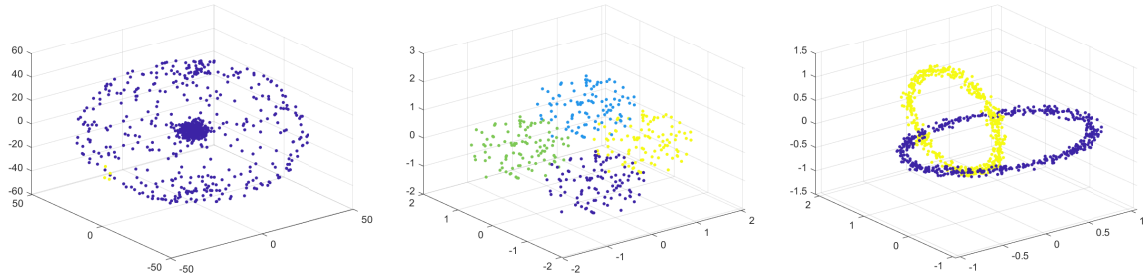
Different values of K were used for these datasets. In particular, we experimented with values $K \in \{5, 10, 20\}$. Plots of the adjacency graphs are shown for $K = 5$:



(a) `atom` dataset (b) `tetra` dataset (c) `chainlink` dataset

Figure 11: Adjacency graph plot for the three datasets, computed with $K = 5$

For every dataset and every value of K we hence have an adjacency graph from which we can recover the relative Laplacian matrix L. From a quick estimation of the conditioning number k we notice that each Laplacian matrix computed is **ill conditioned**, conditioning numbers span in fact from $k \sim 10^{17}$ for the `tetra` dataset to $k \sim 10^{61}$ `atom` dataset. The Laplacian matrix of the `atom` dataset is extremely ill conditioned and this will result in very inaccurate computation of its eigenvectors.
As in 3 we have chosen the number of cluster from evaluation of the eigenvalues of each Laplacian matrix L and used $M = 2$ for the `atom` dataset, $M = 4$ for `tetra` and $M = 2$ for `chainlink`.
Performing the *K-means* algorithm on the truncated eigenvector matrix as in 4 produced the following results:



(a) `atom` dataset (b) `tetra` dataset (c) `chainlink` dataset

Figure 12: Spectral clustering performed on three datasets, best results of K are shown

As it is clearly visible from figure 12 we can see that the Spectral Clustering Algorithm performs well on both `atom` and `chainlink` datasets while it is not able to correctly cluster points from the `atom` dataset. This is due to the fact that the computation of the eigenvalues for the Laplacian matrix L is very inaccurate due to its bad conditioning number.

8 Implementing a method for the eigenvectors and eigenvalues computation

In order to implement the Spectral clustering algorithm we heavily relied on the computation of eigenvectors and eigenvalues. In particular, we have computed eigenvetors and eigenvalues of the Laplacian matrix L relative to the neighborhood graph in order to perform the clustering in the space of the eigenvectors where points are well separated (in an euclidean sense).

In this section we focus on implementing an iterative algorithm to compute eigenvalues and eigenvectors of a matrix starting with the smallest eigenvalue λ_1 and relative eigenvector u_1 and then

continuing with the computation of λ_2, λ_3 ... λ_K until a desired number K of eigenvalues has been computed.

The algorithm is based on the **Power Method** and the **Deflation** principle. Let's start with a brief explanation of what the power method does.

The Power Method

The power method is an iterative algorithm that, given a matrix $A \in \mathbb{R}^{n\times n}$, gives as output the greatest eigenvalue in modulus λ_1 and its relative eigenvalue u_1.

Let $v_0 \in \mathbb{R}^n$ be an initial vector and let $\sigma(A) = \{\lambda_2, \lambda_2, ...\lambda_n\}$ be the spectrum of A with eigenvalues sorted in a descending order of magnitude. Let $u_1, u_2, ...u_n$ be the relative eigenvectors of each eigenvalue. if the matrix A is diagonalizable, then its eigenvectors form a basis for $\mathbb{R}^n$, hence:

$$v_0 = \alpha_1 u_1 + \alpha_2 u_2 + ... + \alpha_n u_n$$

Furthermore:

$$\begin{aligned} v_1 := Av_0 =& \alpha_1 Au_1 + \alpha_2 Au_2 + ... + \alpha_n Au_n = && (8.1)\\ & \alpha_1\lambda_1 u_1 + \alpha_2\lambda_2 u_2 + ... + \alpha_n\lambda_n u_n && (8.2)\\ & \lambda_1\left(\alpha_1 u_1 + \alpha_2\frac{\lambda_2}{\lambda_1}u_2 + ... + \alpha_n\frac{\lambda_n}{\lambda_1}u_n\right) && (8.3)\end{aligned}$$

Then if we define $v_k := A^k v_0$, using 8.1 iteratively we find that:

$$v_k = \lambda_1^k\left(\alpha_1 u_1 + \alpha_2\left(\frac{\lambda_2}{\lambda_1}\right)^k u_2 + ... + \alpha_n\left(\frac{\lambda_n}{\lambda_1}\right)^k u_n\right) \tag{8.4}$$

Since $\lim_{k\to\infty}\left(\frac{\lambda_i}{\lambda_1}\right)^k = 0$ for all $i \in \{2, 3, ...n\}$, we find out that:

$$\lim_{k\to\infty}\frac{1}{\lambda_1^k}v_k = \alpha_1 u_1 \qquad \text{and} \qquad \lim_{k\to\infty}\frac{v_{k+1}}{v_k} = \lambda_1 \tag{8.5}$$

where the limit operator used in conjunction with vectors in $\mathbb{R}^n$ is to be intended *component-wise*.

Given v_0 and v_k defined before, let us now define the *Rayleigh quotient* as follows:

$$\mathcal{R}_A^{(k)} := \frac{v_k^T A v_k}{v_k^T v_k} = \frac{v_k^T v_{k+1}}{v_k^T v_k} \tag{8.6}$$

Using 8.4, after some computation, it can be proved that:

$$\lim_{k\to\infty}\frac{v_k^T v_{k+1}}{v_k^T v_k} = \lim_{k\to\infty}\left(\frac{\lambda_1^{2k+1}}{\lambda_1^{2k}}\right)\left(\frac{\alpha_1^2 u_1^T u_1 + \mathcal{O}\left(\frac{\lambda_2}{\lambda_1}\right)^{k+1} + ... + \mathcal{O}\left(\frac{\lambda_n}{\lambda_1}\right)^{k+1}}{\alpha_1^2 u_1^T u_1 + \mathcal{O}\left(\frac{\lambda_2}{\lambda_1}\right)^{k} + ... + \mathcal{O}\left(\frac{\lambda_n}{\lambda_1}\right)^{k}}\right) = \lambda_1 \tag{8.7}$$

Given the results shown above, we are now ready to define the Power Method Algorithm based on the Rayleigh Quotient convergence criterium. The following algorithm returns both the greatest eigenvalue in modulus of the matrix A and its corresponding eigenvector.

Through some algebraic estimation of 8.7, it is possible to prove that the speed of convergence is controlled by the factor $\left(\frac{\lambda_2}{\lambda_1}\right)^k$ hence the speed of convergence depends on how much the greatest eigenvector λ_1 is far from λ_2 in the spectrum space. Moreover, for symmetric matrices, since eigenvectors form an orthonormal basis, the speed of convergence of the algorithm **doubles**.

Algorithm 1 Power Method with Rayleigh Quotient

Given $v_{\text{old}} \neq 0$, $\lambda_1^{\text{old}} = 1$
$v_{\text{old}} \leftarrow \frac{v_{\text{old}}}{\|v_{\text{old}}\|}$
$K \leftarrow 0$
while $K \leq$ `MaxIter` and $\frac{|\lambda_1^{\text{new}} - \lambda_1^{\text{old}}|}{\lambda_1^{\text{new}}} >$ `tol` **do**
  $\lambda_1^{\text{old}} \leftarrow \lambda_1^{\text{new}}$
  $v_{\text{new}} \leftarrow A v_{\text{old}}$
  $\lambda_1^{\text{new}} \leftarrow v_{\text{old}}^T v_{\text{new}}$
  $K \leftarrow K + 1$
end while
return λ_1^{new} , v_{new}

Power Method for computing the Smallest Eigenvalue

Our original objective was to compute the eigenvalues and eigenvectors of the laplacian matrix starting from the smallest. This is possible through the power method even if the output of the algorithm is the greatest eigenvalue of a given matrix.

From now on let's consider that we are always working with the laplacian matrix L, which is symmetric and positive semidefinite. Let:

$$\sigma(L) = \{\lambda_1, \lambda_2, ...\lambda_n\} \quad \text{with} \quad \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n = 0$$

Using the *Power Method* algorithm we are able to compute λ_1. Hence if we define the matrix $\tilde{L} := (\lambda_1 + \epsilon)I - L$ with $\epsilon > 0$ we have that, if λ is an eigenvalue of L and v its relative eigenvector:

$$\tilde{L}v = ((\lambda_1 + \epsilon)I - L)\, v = (\lambda_1 + \epsilon)v - \lambda v = (\lambda_1 + \epsilon - \lambda)v \tag{8.8}$$

From which we obtain that if λ is an eigenvalue of L and v is its relative eigenvector, then $\tilde{\lambda} = (\lambda_1 + \epsilon - \lambda)$ is an eigevalue of $\tilde{L}$ with v as its relative eigenvector. Moreover, since $\epsilon > 0$ all eigenvalues of $\tilde{L}$ are strictly greater than 0 hence $\tilde{L}$ is symmetric positive definite. If we now take a look at the spectrum of $\tilde{L}$:

$$\sigma(\tilde{L}) = \{\epsilon, \tilde{\lambda}_2, ...\tilde{\lambda}_n\} \tag{8.9}$$

we now find out that $\tilde{\lambda}_n = \lambda_1 + \epsilon - \lambda_n$ is the greatest eigenvalue of $\tilde{L}$. Hence we can compute $\tilde{\lambda}_n$ using the *Power Method* on $\tilde{L}$ and retrieve $\lambda_n = \lambda_1 + \epsilon - \tilde{\lambda}_n$. In this way we computed the smallest eigenvector of L using the power method.

Deflation principle for eigenvalue computation

Let L be a matrix and:

$$\sigma(L) = \{\lambda_1, \lambda_2, ...\lambda_n\} \quad \text{with eigenvectors} \quad \{u_1, u_2, ..., u_n\}$$

If we define rank 1 matrices as $U_i = u_i u_i^T$, then it can be shown that the original matrix L can be decomposed as:

$$L = \sum_{i=1}^{n} \lambda_i U_i.$$

Furthermore, for every $k \in \{1, 2, ..., n\}$, if we define $L_k := L - \lambda_k U_k$ then we say that the matrix L has been *deflated* of the eigenvalue λ_k. In fact:

$$\sigma(L_k) = \{\lambda_1, \lambda_2, ..., \underbrace{\lambda_k}_{=0}, ..., \lambda_n\}$$

This means that, given the matrix L, we can find its greatest eigenvalue λ_1 and relative eigenvector u_1, then deflate L of λ_1 obtaining $L_1 = L - \lambda_1 u_1 u_1^T$. At this point the greatest eigenvector of L_1 will be λ_2 hence we can compute λ_2 and u_2 simply running the *Power Method* on L_1 and so on.

At this point we are ready to combine everything we presented until now to formulate an algorithm that will return the K **smallest** eigenvalues of a matrix L and its relative eigenvalues.

Given a matrix L we can compute its greatest eigenvector $\lambda_{\max}$ and define the matrix $\tilde{L} = (\lambda_{\max} + \epsilon)I - L$. Performing the Power method onto $\tilde{L}$ will return $\tilde{\lambda}_1$ from which we can retrieve the smallest eigenvalue of L that we now call λ_1. Deflating $\tilde{L}$ of $\tilde{\lambda}_1$ we obtain $\tilde{L}_1$. Performing the power method onto $\tilde{L}_1$ will return $\tilde{\lambda}_2$ from which we can retrieve the second smallest eigenvalue of L, λ_2. Performing this basic algorithm for K steps we obtain the K smallest eigenvalues and relative eigenvectors of the matrix L. This can be summarized in the following algorithm:

Algorithm 2 Inverse Power Method for computing K smallest eigenvalues

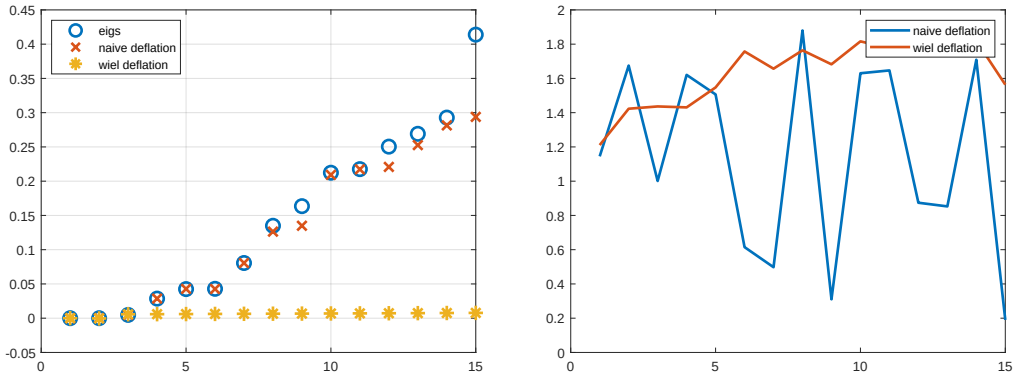
Given L, K, ϵ, `Maxiter`, *tol*
$\lambda_{\max}, v_{\max} \leftarrow$ *Power Method*$(L,$ `tol, Maxiter`$)$
$\mu = \lambda_{\max} + \epsilon$
$B \leftarrow \mu I - L$
$k \leftarrow 0$
for $k \in \{1, 2, ..., K\}$ **do**
  $\lambda, v \leftarrow$ *Power Method*$(B,$ `tol, Maxiter`$)$
  $\lambda_k \leftarrow \mu - \lambda$
  $u_k \leftarrow v$
  $B \leftarrow$ *Deflate*$(B,$ `type`$)$
end for
return $\{\lambda_1, ..., \lambda_K\}, \{u_1, ...u_k\}$

This algorithm has been implemented in the a MATLAB function as shown in 10.1

9 Testing the Inverse Power Method with Deflation

In this section we will now test the accuracy and the performance of our Inverse Power Method with Deflation algorithm against the `eigs` function that is natively implemented in MATLAB. As perviously stated our objective is to compute the K smallest eigenvalues of a symmetric positive semidefinite matrix L.

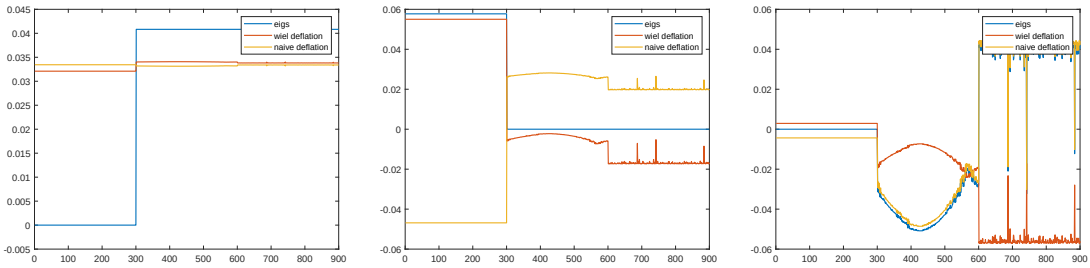
As benchmark matrix we now use the Laplacian matrix relative to the `circle` K-NN graph with $K = 10$. For the first benchmark run we computed the 15 smallest eigenvalues and eigenvector using the `eigs` function in MATLAB and the Inverse power method implemented by us using both the naive deflation method and the Wielandt's deflation method.



(a) Eigenvalues comparison (b) Partial times to compute eigenvalues in seconds

Figure 13: On the left the comparison between the values of the eigenvalues computed using all methods shown. On the right a plot of the partial times to compute each eigenvalue by using the inverse power method with deflation.

As we can see clearly in figure 13, Wielandt deflation performs very badly with respect to the `eigs` function natively implemented in Matlab for eigenvalue computation. On the other hand, using the naive deflation yielded good results, especially for the first 7 eigenvalues. After some iterations we see that our method for computing eigenvalues is not accurate with respect to the `eigs` function and this can be attributed to the poor stability of the deflation procedure: at each iteration we are subtracting a rank 1 matrix computed based on an iterative method, and we are propagating potential errors throughout all iterations.



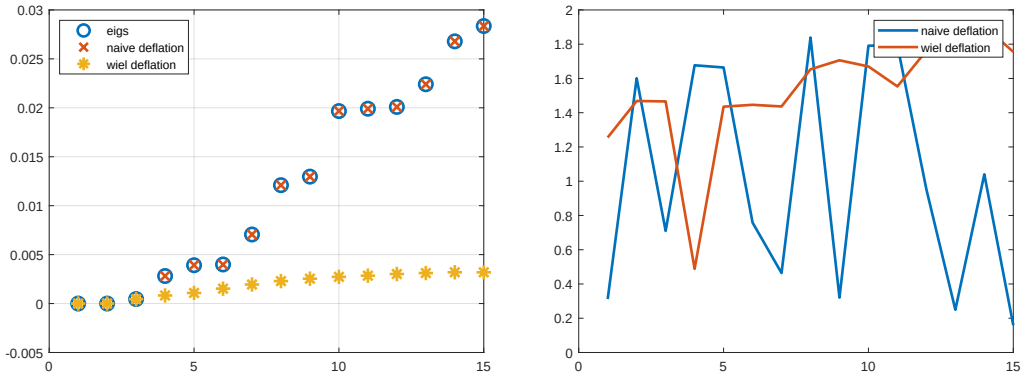
(a) u_1 (b) u_2 (c) u_3

Figure 14: A plot of the first three eigenvectors computed using the three methods

In figure 14 we plotted the first three eigenvectors computed using the three methods. We can see how each method is able to compute eigenvectors that separate well the clusters of points. As it is clearly visible in the figure, the three clusters of points are well distinguishable using eigenvectors computed with all methods used. This is enforced by the fact that, performing the spectral clustering algorithm using our eigenvalues computation methods, yielded the same results as using the `eigs` function meaning that we are perfectly able to cluster the points in our benchmark data with our algorithm.

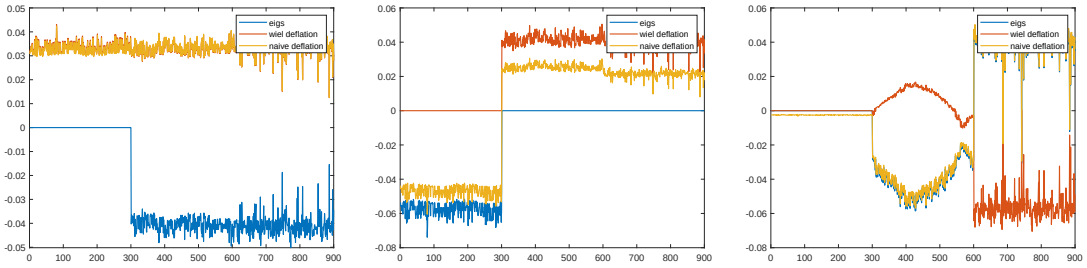
Using the normalized Laplacian matrix

Using the normalized Laplacian matrix $L_{\text{sym}} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$ means that we are shrinking the spectrum of our matrix. We hence expect different results in terms of accuracy of the eigenvalues due to a smaller numerical error since we are comparing at each step numbers in the same order of magnitude.



(a) Eigenvalues comparison (b) Partial times to compute eigenvalues in seconds

Figure 15: On the left the comparison between the values of the eigenvalues computed using all methods shown. On the right a plot of the partial times to compute each eigenvalue by using the inverse power method with deflation.



(a) u_1 (b) u_2 (c) u_3

Figure 16: A plot of the first three eigenvectors computed using the three methods

In this case the naive deflation technique is much more stable than the unnormalized case. In fact, as we can see in 15, eigenvalues computed using the naive deflation technique in the inverse power method algorithm are the same as the one computed with the `eigs` function. As before, Wielandt delfation performs poorly in terms of accuracy with respect to `eigs`. Nevertheless, all three methods used are able to compute eigenvectors that separate well the cluster of points in our benchmark data, as it is visible in figure 16.

10 Appendix

10.1 MATLAB implementation of the Inverse Power Method with Deflation

```
function [D, U, times_partial] = inverse_power_method_deflation(A, K, tol, maxiter, deflation_type)
%INVERSE_POWER_METHOD_DEFLATION computes the K smallest eigenvalues and eigenvectors of a
% A using the power method algorithm
%   A = SYIMMETRIC POSITIVE SEMIDEFINITE matrix
%   K = number of eigenvectors and eigenvalues to be computed
%   deflation = type of deflation to be used (naive or wiel)
%
N = length(A);
D = spalloc(K, K, K);
U = zeros([N, K]);

% Compute the greatest eigenvalue of the matrix A
[lambda_max, v_max] = power_method(A, tol, maxiter);

% Compute the new matrix on which the power method will be applied
% (spectrum shift)
mu = lambda_max+1e3*tol;
B = mu*eye(N)-A;
times_partial = zeros(K,1);
if strcmp(deflation_type, 'naive')
    for eig_iter = 1:K %for each eigenvalue to be computed
        tic
        [lambda, v] = power_method(B, tol, maxiter); %compute the greatest eigenvector of B
        times_partial(eig_iter) = toc;
        D(eig_iter, eig_iter) = mu-lambda; %save eigenvalue of A from the back of the spectrum
        U(:, eig_iter) = v; %save eigenvector of A (is the same of B)
        % Deflate B
        B = B - lambda* (v)*(v');
    end
elseif strcmp(deflation_type, 'wiel')
    % performing eigenvalues computation using Wielandt deflation
    for eig_iter = 1:K %for each eigenvalue to be computed
        tic
        [lambda, v] = power_method(B, tol, maxiter); %compute the greatest eigenvector of B
        times_partial(eig_iter) = toc;
        D(eig_iter, eig_iter) = mu-lambda; %save eigenvalue of A from the back of the spectrum
        U(:, eig_iter) = v; %save eigenvector of A (is the same of B)
        % Deflate B
        [~, idx] = max(v);
        idx = min(idx);
        x = (1/lambda*v(idx))*(B(idx,:)');
        B = B - lambda* (v)*(x');
    end
else
    disp('Specify a valid deflation type')
end


end


function [lambda, v] = power_method(A, tol, maxiter)
    N = length(A);
    v_0 = rand(N,1);
    v_old = v_0 / norm(v_0);
    lambda_old = 0;
    lambda_new = 2;
    k=0;
    while k<maxiter && abs((lambda_old - lambda_new)/lambda_old)>tol
        lambda_old = lambda_new;
        v_new = A*v_old;
        lambda_new = v_old'*v_new;
        v_old = v_new/norm(v_new);
        k = k+1;
```

```
    end
    lambda = lambda_new;
    v = v_new/norm(v_new);
end
```