# SF2526 - HW2 - Group 7

David Ahnlund, Ludvig Wärnberg Gerdin

February 17, 2023

## Exercise 1

### (a)

The R-matrix as well as the indicator vectors for each iteration is displayed in the notebook `ex1.ipynb`.

Let the starting matrix $R_0$ be,

$$R_0 = \begin{bmatrix} 3 & 3 & 3 \\ -1 & -1 & 0 \end{bmatrix}$$

The first indicator vectors are

$$1_A = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

giving us that $R_1$ is

$$R_1 = \begin{bmatrix} (1-1/2+0)/3 & (1+0-1/2)/3 & (1-3-1)/3 \\ 1.5 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1/6 & 1/6 & -1 \\ 1.5 & 2 & 0 \end{bmatrix}.$$

The second indicator vectors are

$$1_A = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

giving us that $R_2$ is

$$R_2 = \begin{bmatrix} (1+3/2)/2 & (1+2)/2 & (1+0)/2 \\ (-1/2+0)/2 & (0-0.5)/2 & (-3-1)/2 \end{bmatrix} = \begin{bmatrix} 5/4 & 3/2 & 1/2 \\ -1/4 & -1/4 & -2 \end{bmatrix}.$$

The third indicator vectors are

$$1_A = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

giving us that $R_3$ is

$$R_3 = \begin{bmatrix} (-0.5+0)/2 & (0-0.5)/2 & (-3-1)/2 \\ (1+3/2)/2 & (1+2)/2 & (1+0)/2 \end{bmatrix} = \begin{bmatrix} -1/4 & -1/4 & -2 \\ 5/4 & 3/2 & 1/2 \end{bmatrix}.$$

The third indicator vectors are

$$1_A = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

giving us that $R_3$ is

$$R_3 = \begin{bmatrix} (-0.5+0)/2 & (0-0.5)/2 & (-3-1)/2 \\ (1+3/2)/2 & (1+2)/2 & (1+0)/2 \end{bmatrix} = \begin{bmatrix} -1/4 & -1/4 & -2 \\ 5/4 & 3/2 & 1/2 \end{bmatrix}.$$

We stop here since we have reached a point since we are in a loop. That means that the first two data points belong to cluster B and the second two points belong to cluster A. This is confirmed by running K-means from `scikit-learn`, where the algorithm requires 3 iterations to arrive at the same result.

## B

Let the starting matrix $R_0$ be,

$$R_0 = \begin{bmatrix} -0.5 & 0 & -3 \\ -1 & -1 & 1 \end{bmatrix}$$

The first indicator vectors are

$$1_A = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix},$$

giving us that $R_1$,

$$R_1 = \begin{bmatrix} (1+3/2)/2 & (1+2)/2 & (1+0)/2 \\ (-0.5+0)/2 & (0-0.5) & (-3-1)/2 \end{bmatrix} = \begin{bmatrix} 5/4 & 3/2 & 1/2 \\ -1/4 & -1/4 & -2 \end{bmatrix}.$$

The second indicator vectors are

$$1_A = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

giving us that,

$$R_2 = \begin{bmatrix} (-0.5+0)/2 & (0-0.5)/2 & (-3-1)/2 \\ (1+1.5)/2 & (1+2)/2 & (1+0)/2 \end{bmatrix} = \begin{bmatrix} -1/4 & -1/4 & -2 \\ 5/4 & 3/2 & 1/2 \end{bmatrix}.$$

The third indicator vectors are

$$1_A = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad 1_B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}.$$

We stop here since we have reached a point since we are in a loop. That means that the first two data points belong to cluster A and the second two points belong to cluster B. This is confirmed by running K-means from `scikit-learn`, where the algorithm requires 2 iterations to arrive at the same result.

The similarity between the two starting points is that the centroids after a couple iterations become the same. However, the resulting clusters are different.

The second starting matrix $R$ is better in terms of computation as it requires fewer iterations to arrive at the final clusters.

In [1]:

```
import numpy as np
```

# Exercise 1

In [2]:

```
X = np.array(
    [[1, 1, 1],
     [1.5, 2, 0],
     [-0.5, 0, -3],
     [0, -0.5, -1]]
)
```

## Sub-question (a)

Using $n$ data points, given by the rows in the matrix $X \in \mathbb{R}^{nxp}$, we want to determine $k$ clusters. The cent roids of those clusters are the rows in the matrix $R \in \mathbb{R}^{kxn}$.

In [3]:

```
def f(R):
    indicators = np.zeros((4, 2))
    for i, row in enumerate(X):
        mat = np.tile(row, (R.shape[0], 1))
        norms = np.linalg.norm(R - mat, axis=1)
        indicators[i, :] = (norms == norms.max()).astype(int)

    return indicators
```

In [4]:

```
R0 = np.array(
    [[3, 3, 3],
     [-1, -1, 0]]
)
```

In [5]:

```
R1 = np.array(
    [[1 / 6, 1 / 6, -1],
     [1.5, 2, 0]]
)
```

In [6]:

```
R2 = np.array(
    [[2.5 / 2, 3 / 2, 1 / 2],
     [-1 / 4, -1 / 4, -2]]
)
```

In [7]:

```
R3 = np.array(
    [[-1 / 4, -1 / 4, -2],
     [2.5 / 2, 3 / 2, 1 / 2]]
)
```

In [8]:

```
print(f"""
Indicator vectors #1:

{f(R0)}

Indicator vectors #2:

{f(R1)}

Indicator vectors #3:

{f(R2)}

Indicator vectors #4:

{f(R3)}
"""
      )
```

```
Indicator vectors #1:

[[1. 0.]
 [0. 1.]
 [1. 0.]
 [1. 0.]]

Indicator vectors #2:

[[1. 0.]
 [1. 0.]
 [0. 1.]
 [0. 1.]]

Indicator vectors #3:

[[0. 1.]
 [0. 1.]
 [1. 0.]
 [1. 0.]]

Indicator vectors #4:

[[1. 0.]
 [1. 0.]
 [0. 1.]
 [0. 1.]]
```

Sub-question (b)

In [9]:

```
R0 = np.array(
    [[-0.5, 0, -3],
     [-1, -1, 1]]
)
```

In [10]:

```
R1 = np.array(
    [[5 / 4, 3 / 2, 1 / 2],
     [-1 / 4, -1 / 4, -2]]
)
```

In [11]:

```
R2 = np.array(
    [[-1 / 4, -1 / 4, -2],
     [5 / 4, 3 / 2, 1 / 2]]
)
```

In [12]:

```
print(f"""
Indicator vectors #1:

{f(R0)}

Indicator vectors #2:

{f(R1)}

Indicator vectors #3:

{f(R2)}
"""
      )
```

```
Indicator vectors #1:

[[1. 0.]
 [1. 0.]
 [0. 1.]
 [0. 1.]]

Indicator vectors #2:

[[0. 1.]
 [0. 1.]
 [1. 0.]
 [1. 0.]]

Indicator vectors #3:

[[1. 0.]
 [1. 0.]
 [0. 1.]
 [0. 1.]]
```

Verify with scikit-learn

In [13]:

```
from sklearn.cluster import KMeans
```

In [14]:

```
def _get_kmeans_labels(X, init):
    km = KMeans(n_clusters=2, init=init, n_init=1).fit(X)
    return km.labels_, km.n_iter_
```

In [15]:

```
R0 = np.array(
    [[3, 3, 3],
     [-1, -1, 0]]
)
labels, iters = _get_kmeans_labels(X, R0)
print(f"""
Clusters: {labels}
N iterations to get there: {iters}
""")
```

```
Clusters: [0 0 1 1]
N iterations to get there: 3
```

In [16]:

```
R0 = np.array(
    [[-0.5, 0, -3],
     [-1, -1, 1]]
)
_get_kmeans_labels(X, R0)
labels, iters = _get_kmeans_labels(X, R0)
print(f"""
Clusters: {labels}
N iterations to get there: {iters}
""")
```

```
Clusters: [1 1 0 0]
N iterations to get there: 2
```

In [1]:

```
%matplotlib inline
from helpers import eig
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
```

# Exercise 2

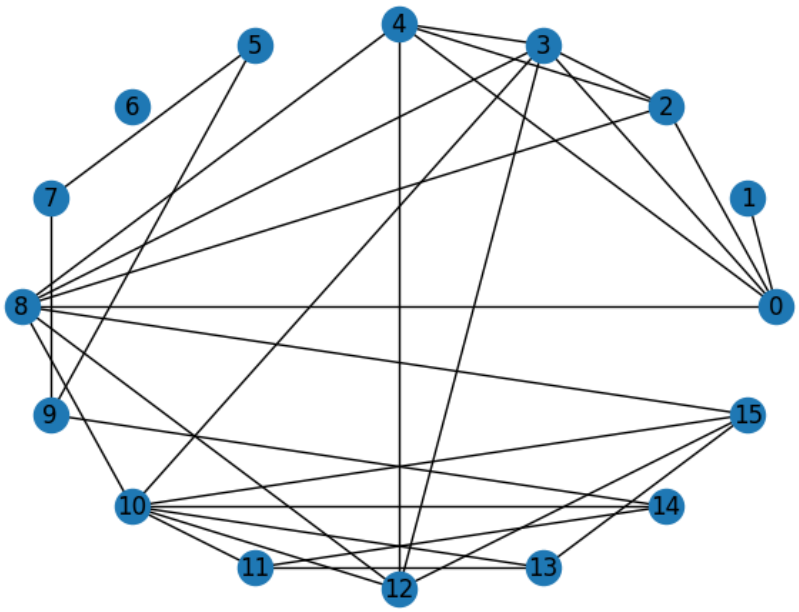
In [2]:

```
def generate_data(n0 = 5, p = 3, c = 2):
    np.random.seed(0)
    A1 = np.random.randn(n0,p) + c * np.array([1,0,0])
    A2 = np.random.randn(n0,p) + c * np.array([0,1,0])
    A3 = np.random.randn(n0,p) + c * np.array([0,0,1])
    A4 = np.array([[0,0,0]])
    A = np.concatenate((A1,A2,A3,A4))
    return A
```

### A)

In [3]:

```
def create_weight_matrix(A, epsilon = 2.5):
    Dist = np.sum(A.T**2, axis=0, keepdims=True)
    Dist = np.sqrt(np.maximum(0,Dist + Dist.T - 2 * A.dot(A.T)))  #maximum just to silence warning message of possible
    np.fill_diagonal(Dist, 0)
    W =  1*(Dist <= epsilon)  - np.eye(Dist.shape[0], Dist.shape[1])
    return W

def show_graph(W):
    G = nx.from_numpy_matrix(W)
    nx.draw(G, with_labels=True, pos=nx.circular_layout(G))
    # Show the plot
    plt.show()
    return G

A = generate_data()
W = create_weight_matrix(A)
Ga = show_graph(W)
```




All but node 6 are connected to each other. Thereby, the graph has 2 connected components. This can be confirmed by using the function `number_connected_components` from the networkx package.

In [4]:

```
print(f"Number of connected components: {nx.number_connected_components(Ga)}")
```

```
Number of connected components: 2
```

### B)

In [5]:

```
D = np.diag(np.sum(W, axis = 1))
L = D-W

print(f"""
Eigenvalues:

{eig(L)[0]}
""")
```

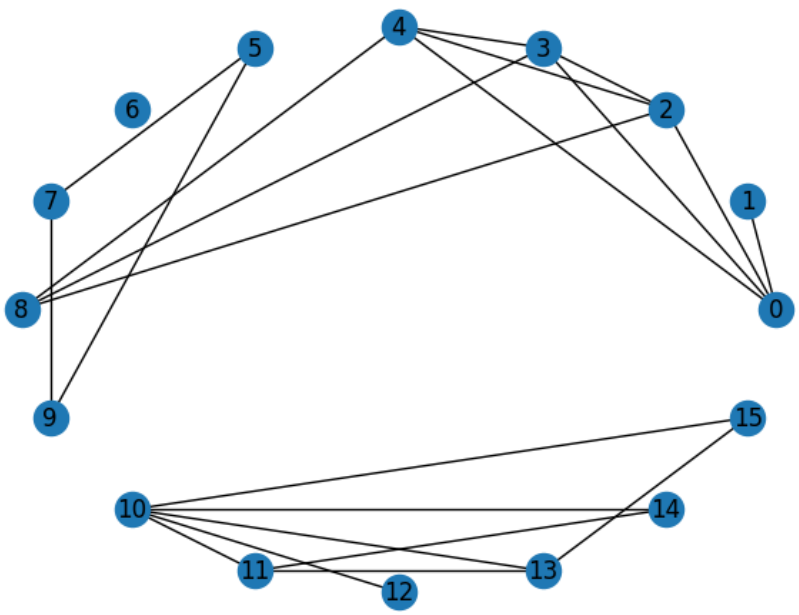
```
Eigenvalues:

[-5.12200472e-16  0.00000000e+00  2.08416978e-01  7.56407481e-01
  1.33066231e+00  2.38389460e+00  3.00000000e+00  3.30269175e+00
  3.94463855e+00  4.67304621e+00  5.10697986e+00  5.73802581e+00
  6.13399942e+00  7.05278617e+00  7.80444226e+00  8.56400861e+00]
```

We see that $\lambda \approx 0$ appears 2 times, so $k = 2$. According to the Lemma 2.3.3 gives us the number of connected components - aligned with what we found in (a).

### C)

In [6]:

```
A = generate_data(c = 3)
W = create_weight_matrix(A)
Gc = show_graph(W)
```




### D)

Here,

- Node 6 is not connected to any other nodes
- Node 5, 9, 7 are connected
- Node 4, 3, 2, 1, 0, 8 are connected
- Node 15, 14, 13, 12, 11, 10 are connected

Giving us that the graph has 4 connected components

In [7]:

```
print(f"Number of connected components: {nx.number_connected_components(Gc)}")
```

```
Number of connected components: 4
```

In [8]:

```
D = np.diag(np.sum(W, axis = 1))
L = D-W

print(f"""
Eigenvalues:

{eig(L)[0]}
""")
```

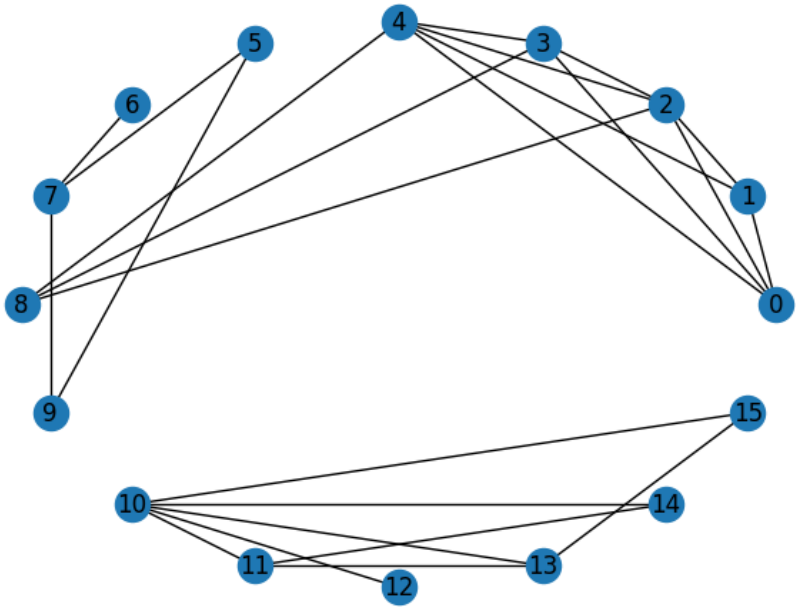
```
Eigenvalues:

[-2.22044605e-16  0.00000000e+00  0.00000000e+00  3.33066907e-16
  9.13869802e-01  1.00000000e+00  1.58578644e+00  3.00000000e+00
  3.00000000e+00  3.00000000e+00  3.57199327e+00  4.41421356e+00
  5.00000000e+00  5.00000000e+00  5.51413693e+00  6.00000000e+00]
```

We see that $\lambda \approx 0$ appears 4 times, so $k = 2$. According to the Lemma 2.3.3 gives us the number of connected components - aligned with the number of connected components found in the graph.

### E)

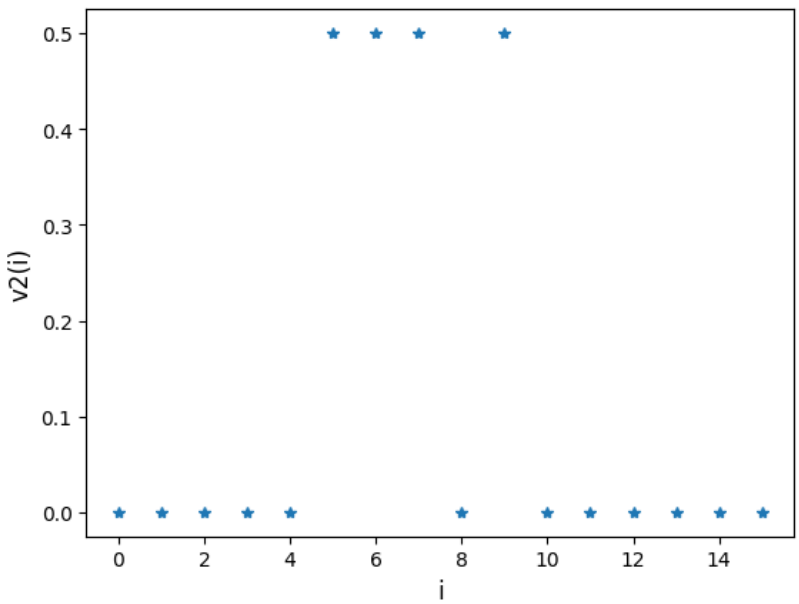
In [9]:

```
A = generate_data(c = 3.1)
W = create_weight_matrix(A, epsilon = 2.8)
Ge = show_graph(W)

D = np.diag(np.sum(W, axis = 1))
L = D-W
eigvals, eigvectors = eig(L)
plt.plot(eigvectors[:,1], '*')
plt.ylabel('v2(i)', fontsize=12)
plt.xlabel('i', fontsize=12)
```




Out[9]:

```
Text(0.5, 0, 'i')
```




The clustering can be read out by looking at the two levels of the points in the graph - one cluster containing the nodes that are placed on the top level and the other containing nodes placed on the bottom level.

In [10]:

```
nx.number_connected_components(Ge)
```

Out[10]:

```
3
```

In [4]:

```
from typing import List

import scipy.io
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
import networkx as nx
from helpers import eig
from sklearn.cluster import KMeans
```

# Exercise 3

In [5]:

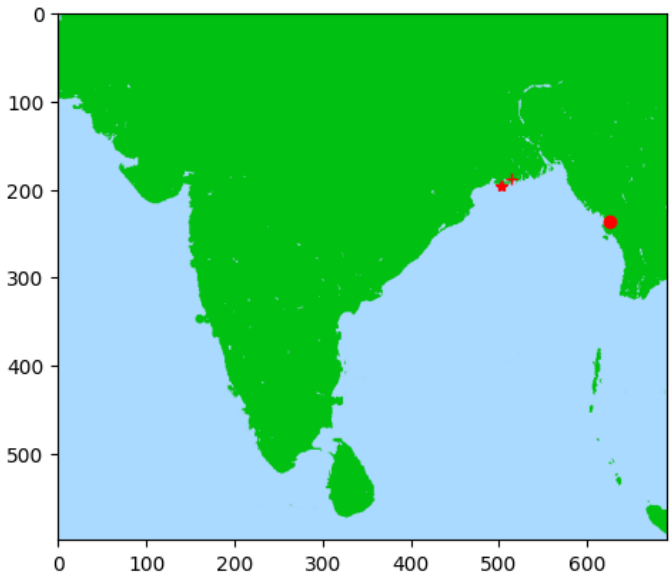
```
def plot_station_map(station_numbers: List[int], xs: List[int], ys: List[int]):
    """ Wrapper for plotting stations given their numbers """
    # Since Python index from 0 we subtract 1 on every element
    jv = [sn - 1 for sn in station_numbers]

    # Plot the stations
    plt.imshow(A)
    for i in range(len(jv)):
        icons = ['r+', 'r*', 'ro']
        plt.plot(ys[jv[i]],xs[jv[i]], icons[i])

    plt.figure()
```

In [6]:

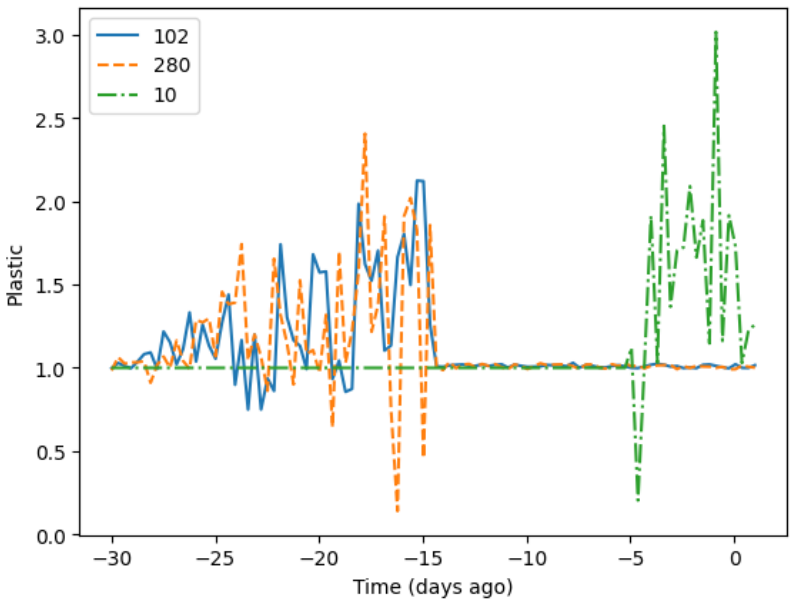
```
def plot_plastic_timeseries(station_numbers: List[str], mat, timeseries):
    """ Wrapper for plotting plastic measurements at stations. """
    # Since Python index from 0 we subtract 1 on every element
    jv = [sn - 1 for sn in station_numbers]

    # Subset the timeseries
    tv = mat['tv'][0]

    # Plot the time series
    plt.figure()
    plt.plot(tv, timeseries[jv[0],:], '-', label = str(jv[0] + 1))
    plt.plot(tv, timeseries[jv[1],:], '--', label = str(jv[1] + 1))
    plt.plot(tv, timeseries[jv[2],:], '-.', label = str(jv[2] + 1))
    plt.ylabel("Plastic")
    plt.xlabel("Time (days ago)")
    plt.legend()
```

In [7]:

```
mat = scipy.io.loadmat('bengali_cleanup.mat')
timeseries = mat['timeseries']
x_coords = mat['x_coords']
y_coords = mat['y_coords']

station_numbers = [102, 280, 10]

# Read the map
A = plt.imread('bengali_map.png')

plot_station_map(station_numbers, x_coords, y_coords)
plot_plastic_timeseries(station_numbers, mat, timeseries)
```



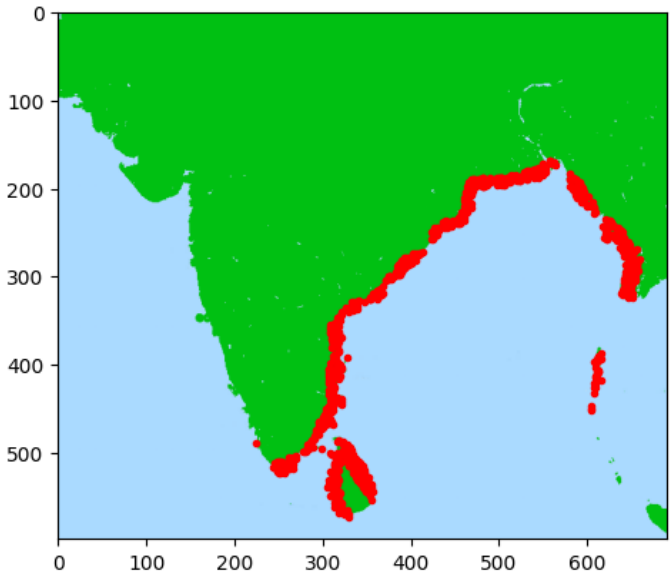

```
<Figure size 640x480 with 0 Axes>
```



## A)

In [8]:

```
plt.figure()
plt.imshow(A)
for i in range(937):
    plt.plot(y_coords[i],x_coords[i], 'r.')
```




## B)

In [9]:

```
# Compute distance matrix in a vectorized way
def build_distance(M):
    S = np.sum(M.T**2, axis=0, keepdims=True)
    S = np.sqrt(np.maximum(0,S + S.T - 2 * M.dot(M.T)))
    np.fill_diagonal(S, 0)
    return S
```

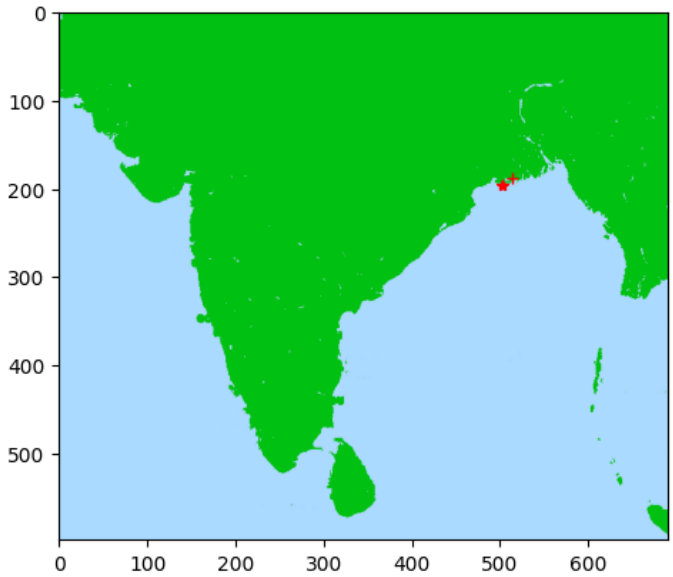
In [10]:

```
Dist = build_distance(timeseries)
```

In [12]:

```
print(f"""
Time series differences between 280 and 102: {round(Dist[101, 279], 2)}
""")
s1 = [102, 280]
plot_station_map(s1, x_coords, y_coords)
```

```
Time series differences between 280 and 102: 3.35
```

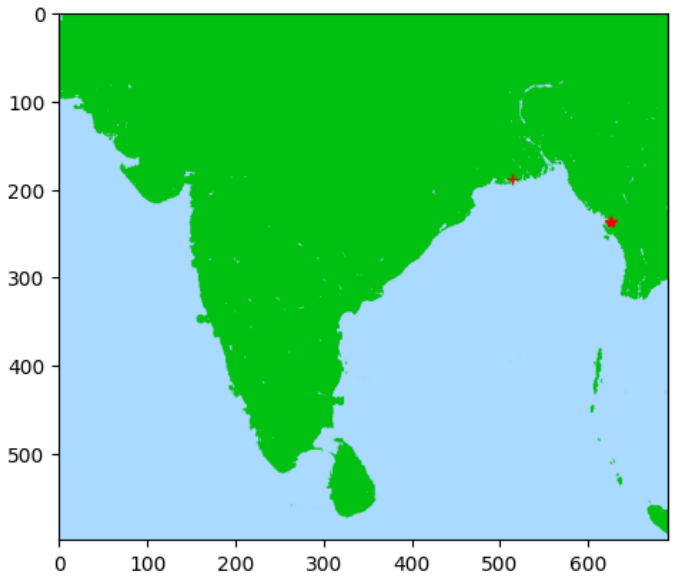



```
<Figure size 640x480 with 0 Axes>
```

In [13]:

```
print(f"""
Time series differences between 280 and 102: {round(Dist[101, 9], 2)}
""")
s1 = [102, 10]
plot_station_map(s1, x_coords, y_coords)
```

```
Time series differences between 280 and 102: 4.61
```

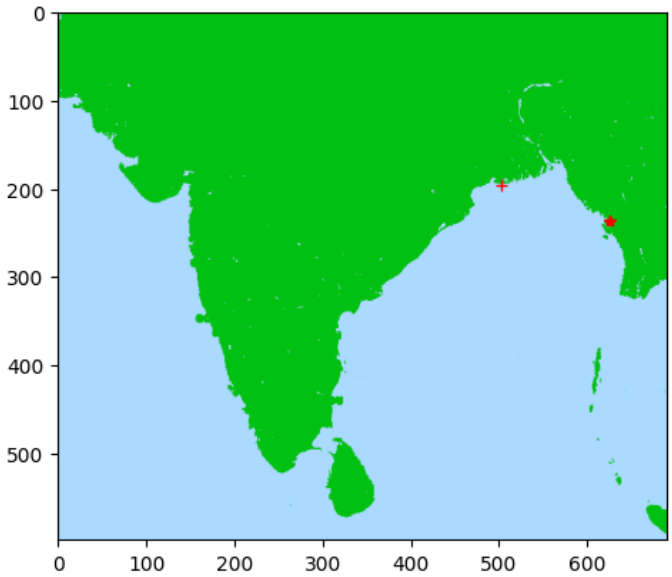



```
<Figure size 640x480 with 0 Axes>
```

In [14]:

```
print(f"""
Time series differences between 280 and 10: {round(Dist[279, 9], 2)}
""")
s1 = [280, 10]
plot_station_map(s1, x_coords, y_coords)
```

```
Time series differences between 280 and 10: 4.84
```



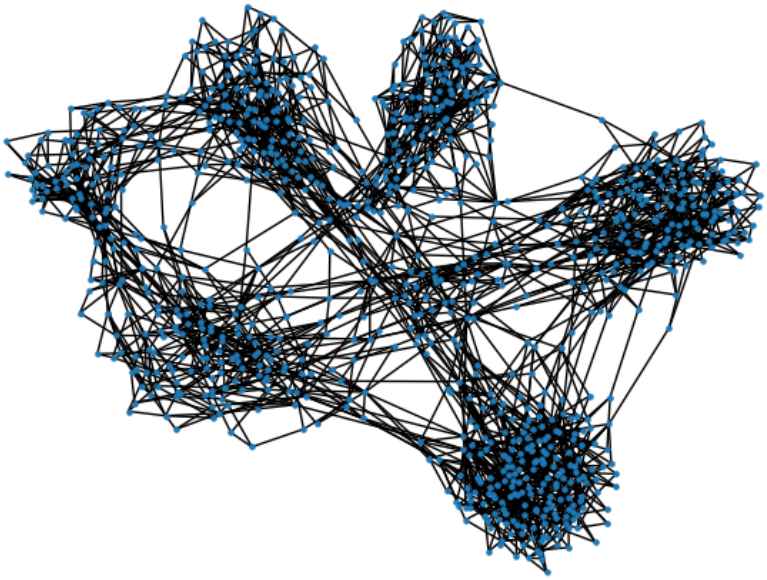

```
<Figure size 640x480 with 0 Axes>
```

To some extent, the results do make sense. The difference in amount waste between nearby stations is more alike than that of pollution of stations being further away.

## C)

In [17]:

```
def knn(Dist, k):
    n = Dist.shape[0]
    W = np.zeros((n, n))
    knn_indices = np.argsort(Dist, axis=1)[:, :k+1]

    for i in range(n):
        W[i, knn_indices[i, 1:]] = 1
        W[knn_indices[i, 1:], i] = 1
    return W


W = knn(Dist, 3)
G = nx.from_numpy_array(W)
nx.draw(G, node_size = 5)
```

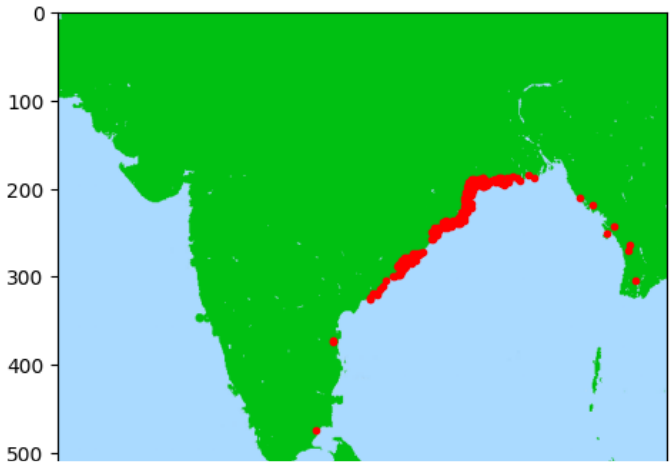


It is indeed hard to distinguish the 7 clusters from above

## D)

In [18]:

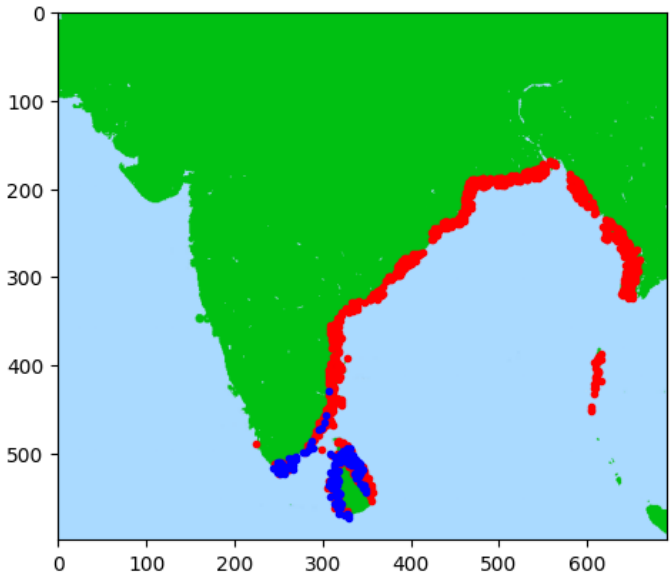
```
def cluster(W, k):
    D = np.diag(np.sum(W, axis = 1))
    L = D - W
    _, eigvectors = eig(L)
    U = eigvectors[:,:k]
    kmeans = KMeans(n_clusters = k).fit(U)
    return kmeans.labels_

clusters = cluster(W, 7)

def plot_cluster(image, clusters, cluster_nr):
    group = []
    for i in range(len(clusters)):
        if clusters[i] == cluster_nr:
            group.append(i)
    plt.figure()
    plt.imshow(image)
    for i in group:
        plt.plot(y_coords[i],x_coords[i], 'r.')

plot_cluster(A, clusters, 0)
plot_cluster(A, clusters, 1)
plot_cluster(A, clusters, 2)
plot_cluster(A, clusters, 3)
plot_cluster(A, clusters, 4)
plot_cluster(A, clusters, 5)
plot_cluster(A, clusters, 6)
```



### E)

In [19]:

```
clusters = cluster(W, 2)

group0 = []
group1 = []
for i in range(len(clusters)):
    if clusters[i] == 0:
        group0.append(i)
for i in range(len(clusters)):
    if clusters[i] == 1:
        group1.append(i)
plt.figure()
plt.imshow(A)
for i in group0:
    plt.plot(y_coords[i],x_coords[i], 'r.')
for i in group1:
    plt.plot(y_coords[i],x_coords[i], 'b.')
```




The general conclusions are still that areas that are similar. The amount of waste in Sri Lanka is similar to that of the southern bay, given that the waste around those areas are emphasised by each other. In all other areas along the coast, the amount of waste is similar, given that no nearby island is emphasising the waste.

In [2]:

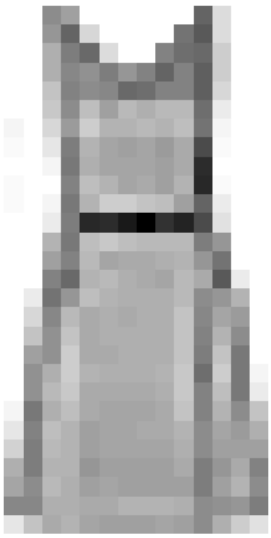
```
import random
import scipy.io
import numpy as np
import matplotlib.pyplot as plt

import networkx as nx
from helpers import eig
from sklearn.cluster import KMeans
```

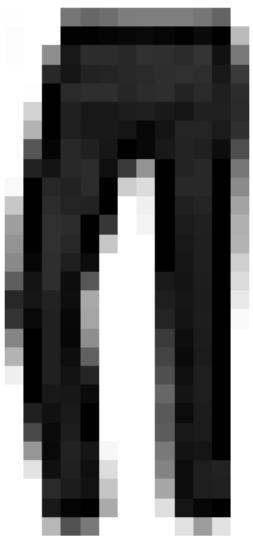
# Exercise 4

In [3]:

```
mat = scipy.io.loadmat('hw2_zalando_new.mat')

correct = mat['correct']
items = mat['items']

print(correct.shape)
print(items.shape)


def zalando_plot(z):
    n = 28  # Image size
    A = np.reshape(z, (n, n))

    # Normalize it
    I = np.argmax(np.abs(z))
    za = z[I]
    A = A / za

    B = 1 - A # It looks nicer with a white background.

    # Plot
    plt.imshow(B, cmap='gray', origin='upper')
    plt.axis('off')
    plt.show()

zalando_plot(items[:,0])
zalando_plot(items[:,1])
```

```
(1000, 1)
(784, 1000)
```





Clearly pants and dresses are the two items in the dataset

## B)

In [4]:

```
def build_distance(items, w):  #Vectorized way of computing distance
    w = np.array([w]).T
    items = w * items
    S = np.sum(items**2, axis=0, keepdims=True)
    S = np.sqrt(np.maximum(0,(S + S.T - 2 * items.T.dot(items))))   #maximum just to silence the warning message of po
    np.fill_diagonal(S, 0)
    return S

# Here is a slower but more intuitive way of computing the distances:
"""
def build_distance(items, w):
    num_points = items.shape[1]
    S = np.zeros((num_points, num_points))
    for i in range(num_points):
        for j in range(i, num_points):
            dist = np.linalg.norm(w * (items[:, i] - items[:, j]))
            S[i, j] = dist
            S[j, i] = dist
    return S
"""

w = np.ones(len(items))  # where len(items) is the number of rows
S = build_distance(items, w)
```

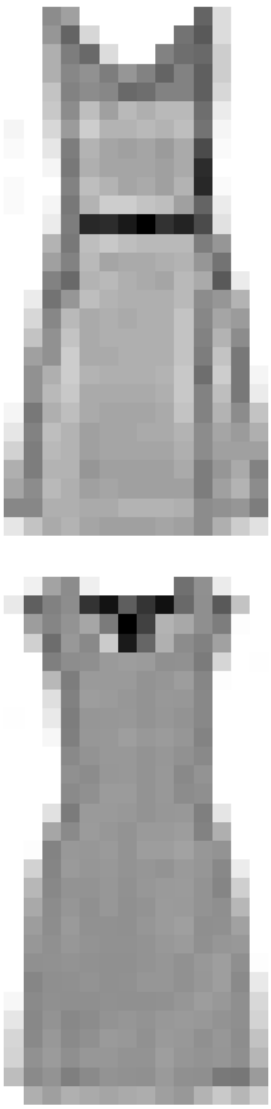
In [5]:

```
print(S[:, :5])
```

```
[[ 0.          9.92163063  9.09682463  7.40750578  8.137665  ]
 [ 9.92163063  0.          5.65675502  6.17920014  5.72210348]
 [ 9.09682463  5.65675502  0.          6.27511884  4.17651662]
 ...
 [ 5.71007595  8.3513411   7.45750901  6.50282167  7.29010115]
 [ 8.31472987  8.28070579  6.44543775  8.10027126  6.59546639]
 [10.73040071  3.14619804  6.1964197   6.7770175   6.57853043]]
```

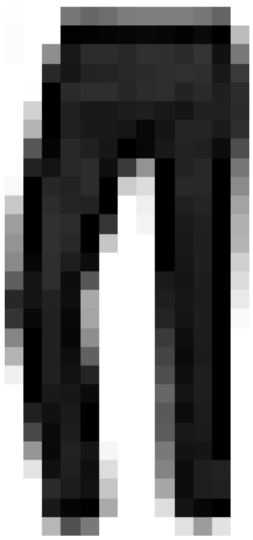
In [6]:

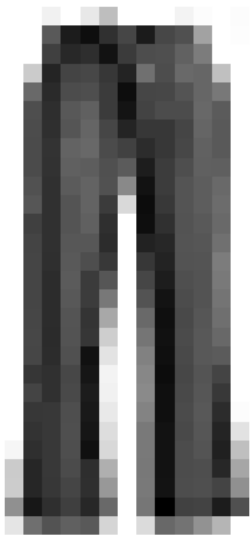
```
# First item:
zalando_plot(items[:,0])

# Closest to first item:
index = np.argmin(S[0,1:]) + 1
zalando_plot(items[:,index])
```



In [6]:

```
# Second item:
zalando_plot(items[:,1])

# Closest to second item
index = np.argmin(S[1,2:]) + 1
zalando_plot(items[:,index])
```





## C)

In [7]:

```
def build_weights(S):  # Quicker vectorized version
    alpha = 0.5
    W = np.zeros(S.shape)
    sigma = np.std(S, axis = 1)
    W = np.exp(-alpha*S**2/sigma**2)
    W -= np.diag(np.diag(W))
    W = (W+W.T)/2
    return W

def build_weights_slow(S): # Slow but intuitive version
    W = np.zeros(S.shape)
    alpha = 0.5
    for i in range(S.shape[0]):
        sigma = np.std(S[:,i])
        for j in range(S.shape[0]):
            W[i,j] = np.exp((-alpha*S[i,j]**2)/(sigma**2))
    W -= np.diag(np.diag(W))
    W = (W+W.T)/2
    return W

import time

start_time = time.time()
W = build_weights_slow(S)
end_time = time.time()

print(W)
print("Elapsed time slow version: {:.2f} seconds".format(end_time - start_time))
```

```
[[0.00000000e+00 2.68312481e-07 1.36933751e-07 ... 6.08580244e-04
  3.25116019e-07 1.76428491e-07]
 [2.68312481e-07 0.00000000e+00 5.93980936e-03 ... 1.80088118e-05
  2.13960027e-05 2.56437328e-01]
 [1.36933751e-07 5.93980936e-03 0.00000000e+00 ... 1.75153947e-05
  2.34822478e-04 3.94580792e-03]
 ...
 [6.08580244e-04 1.80088118e-05 1.75153947e-05 ... 0.00000000e+00
  3.83932734e-06 1.15323883e-05]
 [3.25116019e-07 2.13960027e-05 2.34822478e-04 ... 3.83932734e-06
  0.00000000e+00 1.60736469e-05]
 [1.76428491e-07 2.56437328e-01 3.94580792e-03 ... 1.15323883e-05
  1.60736469e-05 0.00000000e+00]]
Elapsed time slow version: 1.73 seconds
```

In [8]:

```
start_time = time.time()
W = build_weights(S)
end_time = time.time()

print(W)
print("Elapsed time fast version: {:.2f} seconds".format(end_time - start_time))
```

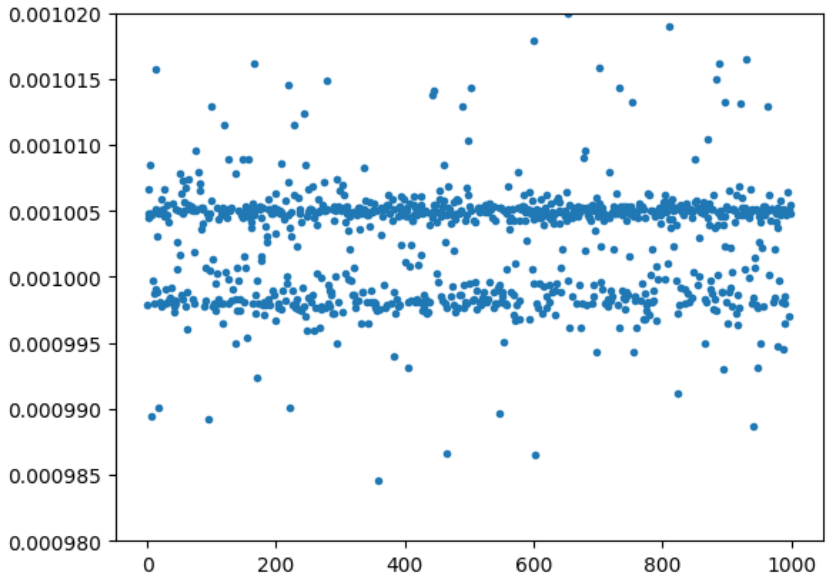
```
[[0.00000000e+00 2.68312481e-07 1.36933751e-07 ... 6.08580244e-04
  3.25116019e-07 1.76428491e-07]
 [2.68312481e-07 0.00000000e+00 5.93980936e-03 ... 1.80088118e-05
  2.13960027e-05 2.56437328e-01]
 [1.36933751e-07 5.93980936e-03 0.00000000e+00 ... 1.75153947e-05
  2.34822478e-04 3.94580792e-03]
 ...
 [6.08580244e-04 1.80088118e-05 1.75153947e-05 ... 0.00000000e+00
  3.83932734e-06 1.15323883e-05]
 [3.25116019e-07 2.13960027e-05 2.34822478e-04 ... 3.83932734e-06
  0.00000000e+00 1.60736469e-05]
 [1.76428491e-07 2.56437328e-01 3.94580792e-03 ... 1.15323883e-05
  1.60736469e-05 0.00000000e+00]]
Elapsed time fast version: 0.04 seconds
```

In [9]:

```
D = np.diag(np.sum(W, axis = 1))
L = D - W

eigvals, eigvectors = eig(L)

xvals = np.zeros((eigvectors.shape[0],1))
plt.plot(eigvectors[:, 1], '.')
plt.ylim(bottom = 0.00098, top = 0.00102)
```

Out[9]:

```
(0.00098, 0.00102)
```




Inspecting the above chart, we realise that the clustering does work relatively well. There are indeed two general levels for the points. Given that there are many points, the plot becomes diluted, but the general levels are visible.

### D)

In [10]:

```
tau = np.median(eigvectors[:,1])
predictions = 1 * (eigvectors[:,1] < tau)
correct_adj = (correct - 1).T[0]

daccuracy = np.mean(predictions == correct_adj)
print(f'Accuracy: {daccuracy*100}%')
```

```
Accuracy: 82.0%
```

If we do the same procedure with an eigenvector such as the 5:th one, we get a better accuracy:

In [11]:

```
tau5 = np.median(eigvectors[:,4])
predictions5 = 1*(eigvectors[:,4] > tau5)
accuracy5 = np.mean(predictions5 == correct_adj)
print(f'Accuracy: {accuracy5*100}%')
```

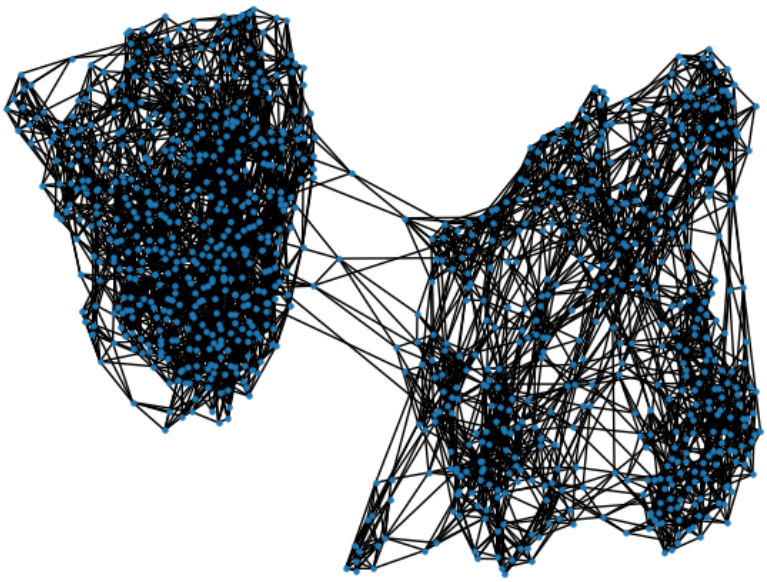
```
Accuracy: 96.2%
```

We could also compare the results above with a kNN + K-means approach, such as in exercise 3. In that case we get an even better accuracy:

In [13]:

```
def knn(Dist, k):
    n = Dist.shape[0]
    W = np.zeros((n, n))
    knn_indices = np.argsort(Dist, axis=1)[:, :k+1]

    for i in range(n):
        W[i, knn_indices[i, 1:]] = 1
        W[knn_indices[i, 1:], i] = 1
    return W

W = knn(S, 6)    # Here we choose a neighbourhood of 6, but could be changed to another number of course
G = nx.from_numpy_array(W)
nx.draw(G, node_size = 5)

def cluster(W, k):
    D = np.diag(np.sum(W, axis = 1))
    L = D - W
    _, eigvectors = eig(L)
    U = eigvectors[:,:k]
    kmeans = KMeans(n_clusters = k).fit(U)    # Using sklearn's K-means algorithm
    return kmeans.labels_

clusters = cluster(W, 2)

accuracy = np.mean(clusters == correct_adj)
if accuracy < 0.5:
    accuracy = 1 - accuracy

print(f'Accuracy: {accuracy * 100}%')
```

```
Accuracy: 99.9%
```



### E)

In [14]:

```
def transform_zalando(z):  # Re-define Zalando_plot to return "plottable" matrix
    n = 28  # Image size
    A = np.reshape(z, (n, n))

    # Normalize it
    I = np.argmax(np.abs(z))
    za = z[I]
    A = A / za

    B = 1 - A # It looks nicer with a white background.
    return B
```

In [15]:

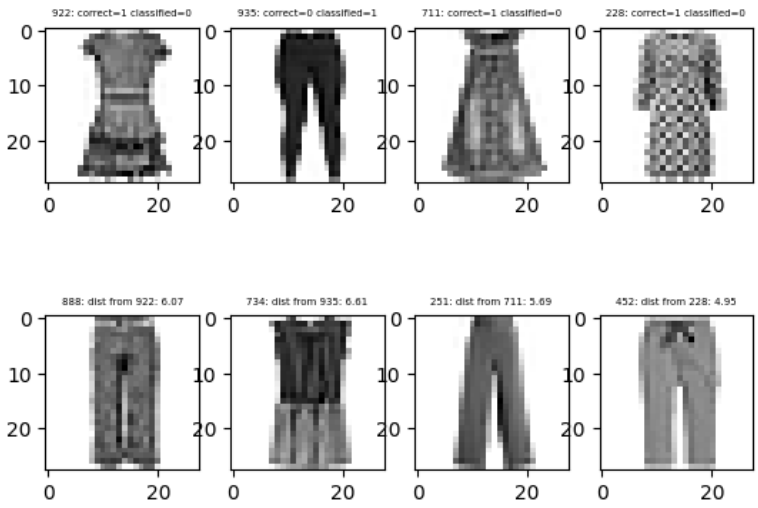
```
vec = 1 * (predictions != correct_adj)
wrong_guesses = vec.nonzero()[0]
```

In [16]:

```
idxs = [random.randrange(len(wrong_guesses)) for _ in range(4)]
II = wrong_guesses[idxs]
imagesd = {
    idx: {
        "prediction": predictions[idx],
        "actual": correct_adj[idx]
    }
    for idx in II
}
```

In [17]:

```
fig, axs = plt.subplots(2, 4)

for i, image_idx in enumerate(imagesd):
    # Plot items
    d = imagesd[image_idx]
    axs[0, i].imshow(transform_zalando(items[:,image_idx]),cmap='gray', origin='upper')
    axs[0, i].set_title(f'{image_idx}: correct={d["actual"]} classified={d["prediction"]}', fontsize=5)

for i, image_idx in enumerate(imagesd):
    # Subset item classOc
    item = items[:, [image_idx]]
    item_class = correct_adj[image_idx]

    # Subset relevant items to compare + do comparison
    relevant_comparison_items = items[:, (correct_adj != item_class)]
    a = np.tile(item, (1, relevant_comparison_items.shape[1]))
    norms = np.linalg.norm(a - relevant_comparison_items, axis=0)
    norms = norms[norms != 0]
    closest_idx, dist = norms.argmin(), norms.min()
    closest_item = relevant_comparison_items[:, [closest_idx]]

    # Find original index
    closest_item_mat = np.tile(closest_item, (1, 1000))
    closest_item_norm = np.linalg.norm(closest_item_mat - items, axis=0)
    closest_item_orig_idx = np.argmax(closest_item_norm == 0)

    # Plot the closest item from the other class
    axs[1, i].imshow(transform_zalando(closest_item),cmap='gray', origin='upper')
    axs[1, i].set_title(f'{closest_item_orig_idx}: dist from {image_idx}: {round(dist, 2)}', fontsize=5)
```



## F)

In [18]:

```
x = 2
W_weight = np.ones((28, 28))
W_weight[11:16,:] = x
w_weight = W_weight.ravel()
w_weight = w_weight / np.linalg.norm(w_weight)
zalando_plot(w_weight)
```

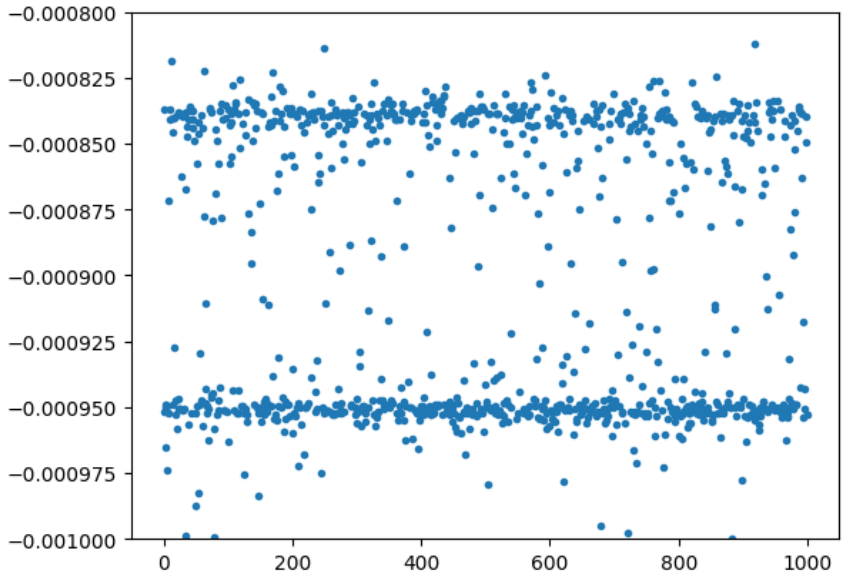
The plot above shows that the weight matrix put an increased in the middle of the zalando_items, which probably is the area where they differ the most.

Therefore, we hope that using this weighting will contribute to a better accuracy and a clearer separation in plots like the one in part d

In [19]:

```
S = build_distance(items, w_weight)
W = build_weights(S)
D = np.diag(np.sum(W, axis = 1))
L = D - W

eigvals, eigvectors = eig(L)


xvals = np.zeros((eigvectors.shape[0],1))
plt.plot(eigvectors[:,1], '.')
plt.ylim(bottom = -0.001, top = -0.0008)


tau = np.median(eigvectors[:,1])
predictions = 1*(eigvectors[:,1] > tau)
correct_adj = (correct-1).T[0]
faccuracy = np.mean(predictions == correct_adj)
print(f'Accuracy: {faccuracy*100}%')
```

```
Accuracy: 91.2%
```




The amount that it is better is measured by the difference in accuracy. That is,

In [20]:

```
print(f'Difference in accuracy: {round((faccuracy - daccuracy) * 100, 3)} percentage points')
```

```
Difference in accuracy: 9.2 percentage points
```

## F - bonus

Another alternative which could generate an even better accuracy is if we weigh the lower part of the image more, since the difference between dresses and pants is probably most obvious when looking at the lowest part.

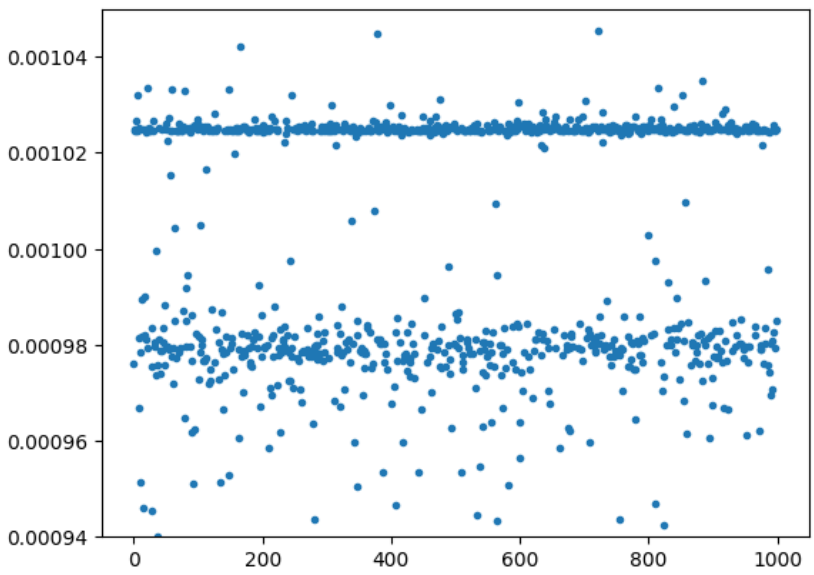
In [21]:

```
x = 2
W_weight = np.ones((28, 28))
W_weight[20:,:] = x
w_weight = W_weight.ravel()
w_weight = w_weight / np.linalg.norm(w_weight)
zalando_plot(w_weight)
```

In [24]:

```
S = build_distance(items, w_weight)
W = build_weights(S)
D = np.diag(np.sum(W, axis = 1))
L = D - W

eigvals, eigvectors = eig(L)


xvals = np.zeros((eigvectors.shape[0],1))
plt.plot(eigvectors[:,1], '.')
plt.ylim(bottom = 0.00094, top = 0.00105)


tau = np.median(eigvectors[:,1])
predictions = 1*(eigvectors[:,1] < tau)
correct_adj = (correct-1).T[0]
accuracy = np.mean(predictions == correct_adj)
print(f'Accuracy: {accuracy*100}%')
```

```
Accuracy: 98.4%
```




When weighting the loser part of the image we get an even better accuracy and, as seen in the plot, the separation is clearer

# SF2526 - HW2 - Group 7

David Ahnlund, Ludvig Wärnberg Gerdin

February 17, 2023

## Exercise 5

### 12c

Explores algebraic multiplicity with regard to spectral clustering. In particular, it defines the algebraic multiplicity. On top of that, the video shows that linear combinations of eigenvectors form new eigenvectors. The formed eigenvectors are not unique, but the eigenspace that they form - the space spanned by the eigenvectors - is unique.

### 14a

Explores the Rayleigh-Ritz theorem and how it can be used to uncover the eigenvalues given the Rayleigh quotient. In words, obtaining - for instance - the second eigenvalue using the theorem, we receive the eigenvalue by minimising the Rayleigh-quotient with respect to all vectors such that the first eigenvalue is orthogonal to those vectors. It explores the theorem through two examples.

### 14c

Explores the mincut algorithm for clustering and shows that the method might return unituitive results clustering. Given that, shows that a modified method - RatioCut - returns sometimes returns more intuitive results and therefore often serves as a more appropriate method for clustering. Does so using two examples of graphs and their resulting partitions with the mincut algorithm.

### 16a

Explores the derivatives for both eigenvalues and eigenvectors. Illustrates the proof for the derivation of those derivatives. Formulas can be derived for the eigenvalues and eigenvectors as a function of epsilon, giving us a quantification of the slope at the origin of the plots of those formulas as a function of epsilon

## 18a

Illustrates the proof for the Rayleigh-Ritz theorem. Important point to take away is the notion that the formula can be restricted to diagonal matrices without loss of generality. Using the normalisation condition for the z-values and the notion that eigenvalues are ordered by magnitude, allows us to fill the proof.