Homework 6 – Machine Learning

REPORT

Max LEMASQUERIER
0845034

I – CODE

```
import numpy as np
import matplotlib.pyplot as plt
import imageio
from scipy.spatial.distance import *
from tqdm import tqdm
```

In these first line, I’m just importing the usefull libraries.

```
def read_input(image):
    im = imageio.imread(image)
    im=im.reshape((-1,3))
    return im
```

Then here is the function that I used to load the image data into a variable. During my work, I’ve focused on the *image1.png* and not on the image 2, because the computation were already quite long to have good results with the image 1, I just had no time to do the same computations on the image 2 as well. But it should work all the same for both images.

```
def compute_rbf_kernel(image,gamma1,gamma2):
    kernel = np.zeros((image.shape[0],image.shape[0]))
    temp=[]
    for i in range(100):
        for j in range(100):
            temp.append([i,j])
    temp = np.asarray(temp)
    temp2 = np.exp(-gamma1*cdist(temp,temp))
    color = cdist(image,image)
    kernel = np.multiply(temp2,np.exp(-gamma2*color))
    return kernel
```

```python
def initialization(data, k):
    """
        Function to initialize the means of the different clusters
        and the classification first step.
    """
    n = data.shape[0]
    initialize_method = "2"
    means = np.random.rand(k,3)*255
    if initialize_method == "1":    # RANDOM INIT
        classif_prec = np.random.randint(k, size=n)
    elif initialize_method == "2":      # Structured init : every 2 columns
        classif_prec = []
        for i in range(n):
            if i % 2 == 1:
                classif_prec.append(0)
            else:
                classif_prec.append(1)
        classif_prec = np.asarray(classif_prec)
    elif initialize_method == "3":  # More accurate method, based on the random mean
        classif_prec = np.zeros(n, dtype=np.int)
        temp = np.zeros(n)
        null_vector = np.zeros([1, 2])
        for i in range(0, n):
            temp[i] = np.linalg.norm(data[i,:] - null_vector[0,:])
        mean_temp = np.mean(temp)
        for i in range(0, n):
            if temp[i] >= mean_temp:
                classif_prec[i] = 0
            else:
                classif_prec[i] = 1
    return means, np.asarray(classif_prec)
```

This is the function I used to compute the kernel matrix of the image, including as asked in the statement : a part concerning the spatial distance of the pixels, and another part concerning the color smiliarities, that I described as just a spatial distance between 2 different rgb vectors.

This part is about the initialization of the clustering. It turned out that it is quite important, and changes the results I could obtain by changing the initalization method. I thought it would end up being the same but in fact not really. So in this function I initialize the means of the different clusters, and a first classification of the pixels, which is quite important for how the algorithm will classify the pixels next. There are 3 different methods here, my favorite is the random one because it really starts from nothing to get to some quite impressive clustering. But I think the third method is much closer from an accurate description of the image.

```python
def deuxieme_terme(data, kernel_data, classification, data_number, cluster_number, k):
    result = 0
    number_in_cluster = 0
    for i in range(0, data.shape[0]):
        if classification[i] == cluster_number:
            number_in_cluster += 1
    if number_in_cluster == 0:
        number_in_cluster = 1
    for i in range(0, data.shape[0]):
        if classification[i] == cluster_number:
            result += kernel_data[data_number][i]
    return -2 * (result / number_in_cluster)
```

This function is computing the second term of the euclidean distance from a point to the center of the different clusters. This is important to determine afterwards, which cluster we will attribute to every pixel. And below is a similar function to compute the third term of this distance :

```python
def troisieme_terme(kernel_data, classification, k):
    """
        Function to compute the third term of the euclidean distance
        from a point to the center of the different clusters.
    """
    temp = np.zeros(k)
    temp1 = np.zeros(k)
    for i in range(0, classification.shape[0]):
        temp[classification[i]] += 1
    for i in range(0, k):
        for p in range(0, kernel_data.shape[0]):
            for q in range(p + 1, kernel_data.shape[1]):
                if classification[p] == i and classification[q] == i:
                    temp1[i] += kernel_data[p,q]
    for i in range(0, k):
        if temp[i] == 0:
            temp[i] = 1
        temp1[i] /= (temp[i] ** 2)
    return temp1
```

Next is the classifier :

```python
def classifier(data, kernel_data, means, classification):
    """
    Attribute a cluster to every pixel of an image, based on the kernel results
    """
    temp_classification = np.zeros([data.shape[0]], dtype=np.int)
    third_term = troisieme_terme(kernel_data, classification, means.shape[0])
    for i in tqdm(range(0, data.shape[0])):
        temp = np.zeros([means.shape[0]], dtype=np.float32) # temp size: k
        for j in range(0, means.shape[0]):
            temp[j] = deuxieme_terme(data, kernel_data, classification, i, j, means.shape[0]) + third_term[j]
        temp_classification[i] = np.argmin(temp)
    return temp_classification
```

As explained above, it uses the other functions in order to compute the full euclidean distance, and takes the minimum value of a corresponding cluster to attribute it to a data point, which is here a pixel.

```python
def nb_erreur(classification, classif_prec):
    """
        Count the number of pixels that have change the cluster between an iteration
    """
    error = 0
    for i in range(0, classification.shape[0]):
        error += np.absolute(classification[i] - classif_prec[i])
    return error
```

This function is counting the number of pixels that left a cluster to join a new one, on every iteration so that we can stop when every pixel is fixed, and not changing it’s belonging to a cluster anymore.

```python
def display_clusters(k, data, means, classification, iteration, filename):
    title = "Kernel-K-Means Iteration-" + str(iteration)
    plt.clf()
    plt.suptitle(title)
    plt.imshow(classification.reshape((100,100)),cmap='gray')
    plt.show()
```

This function is just a displayer of figure. Once I classified every pixel to different clusters, I can plot a map of the assignment to see wether it looks like the original picture or not. I will show some results after.

```python
def kkmeans(data, kernel_data, filename):
    k = 2                                            # cluster number
    means, classif_prec = initialization(data, k)    # INITIALIZE everything
    iteration, erreur_prec = 1, 0
    display_clusters(k, data, means, classif_prec, iteration, filename) # display the inital assignment of clusters
    classification = classifier(data, kernel_data, means, classif_prec)
    error = nb_erreur(classification, classif_prec)
    for i in range(50): # Limit to 50 iteration, after that if it did not converge already, it will stop.
        display_clusters(k, data, means, classification, iteration, filename)
        iteration += 1
        classif_prec = classification
        classification = classifier(data, kernel_data, means, classification)
        error = nb_erreur(classification, classif_prec)
        print(error)
        if error == erreur_prec:
            break
        erreur_prec = error
    means = update(data, means, classification) # update the clusters mean to have the final centroids
    display_clusters(k, data, means, classification, iteration, filename)   # display the final assignment.
```

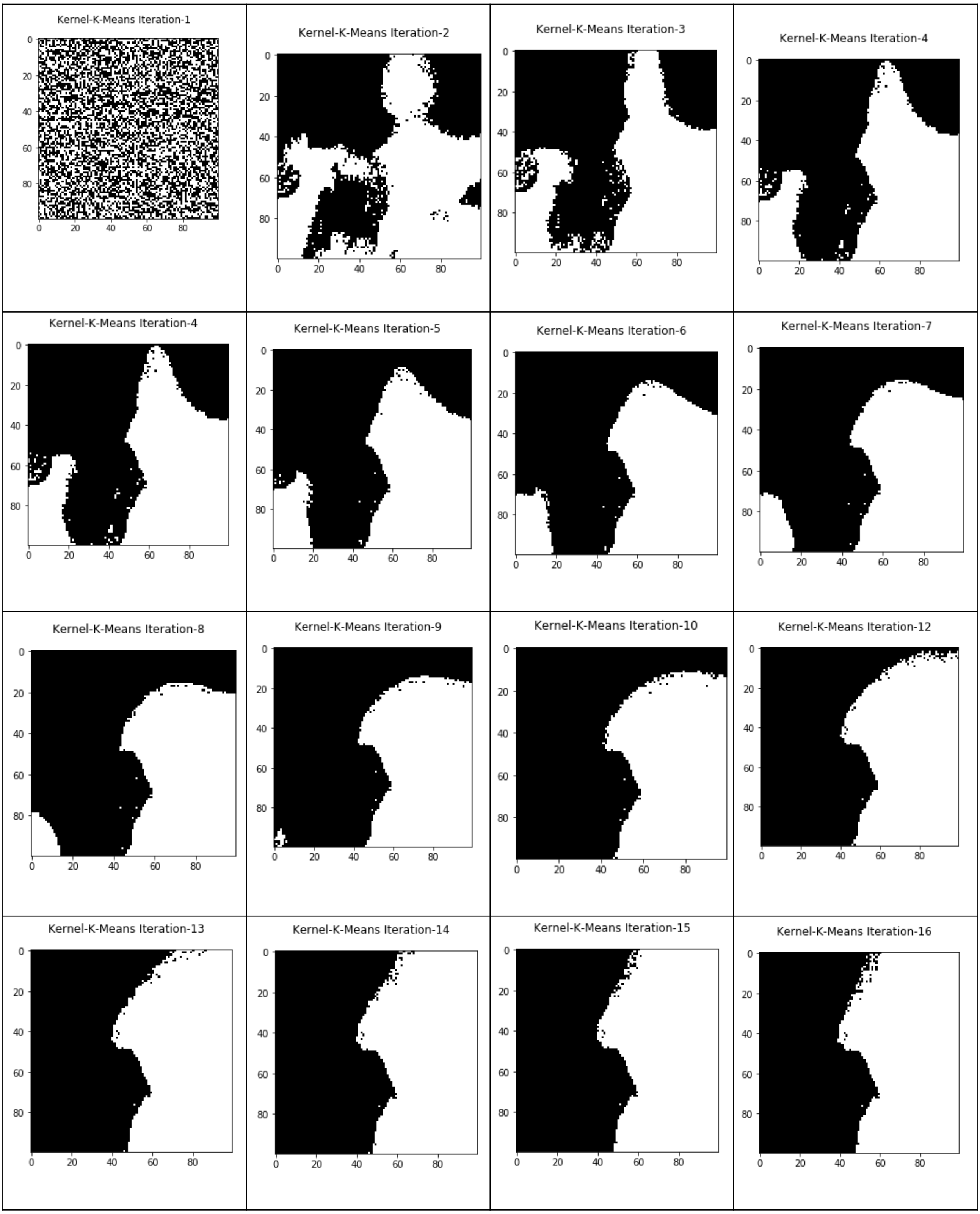
This is the main function, not the hardest one, because it just runs the other functions. So we just do the usual process, initalization, classification and iterate until it converges or reaches a max iteration number.

II – RESULTS

From what I understood, the spectral is the same as k mean, but using the output cut and ratio cut as input for the spectral clustering.
So I will only introduce the results I had with the kernel k-means algorithm. Every iteration was about 5 minute to compute because the third term of the euclidean distance was quite long to compute, and the classification step was therefore quite long. Plus I had to loop over 10 000 two times which also takes a while.

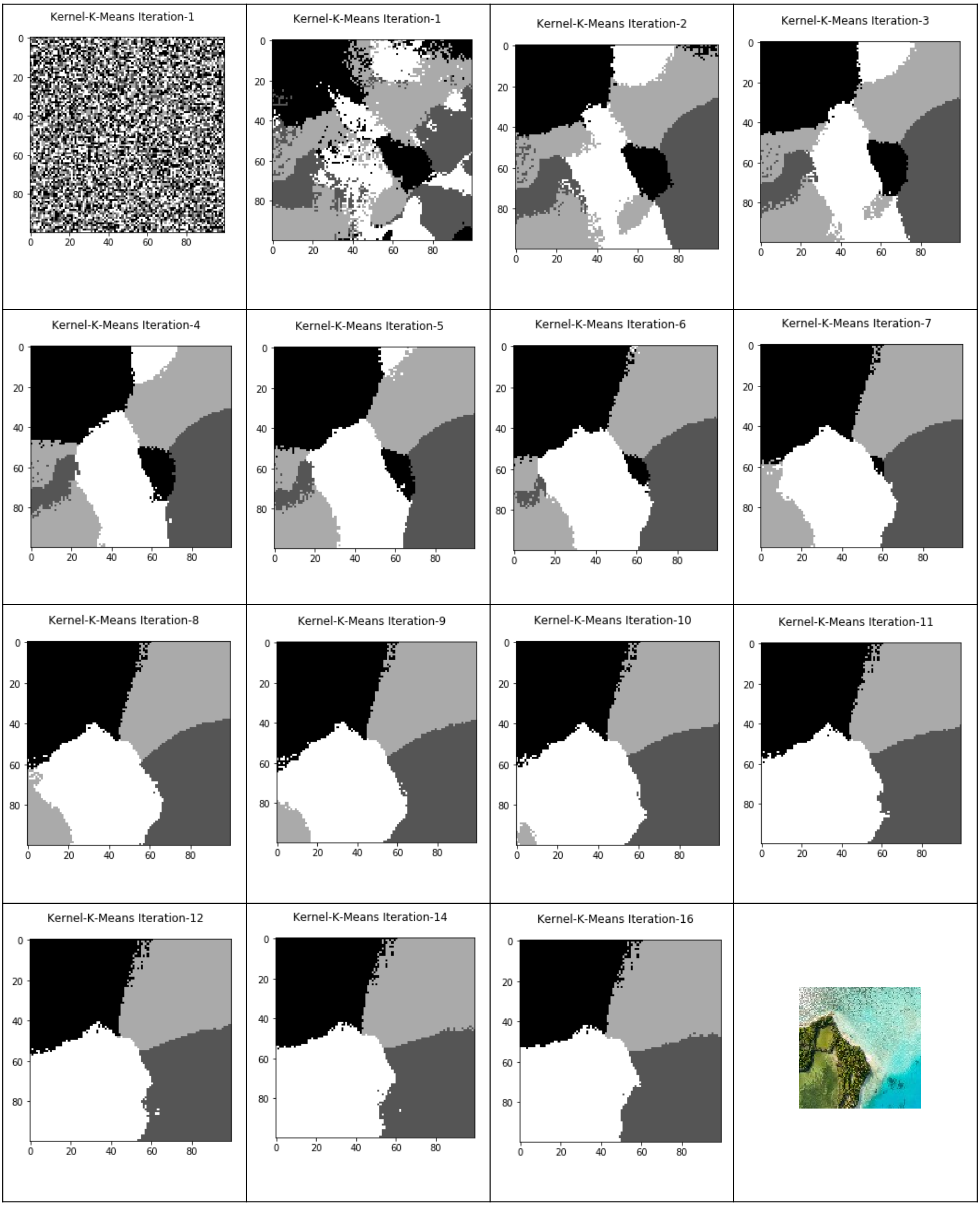
Here is our base image :

First, with a random initialization and 2 clusters :



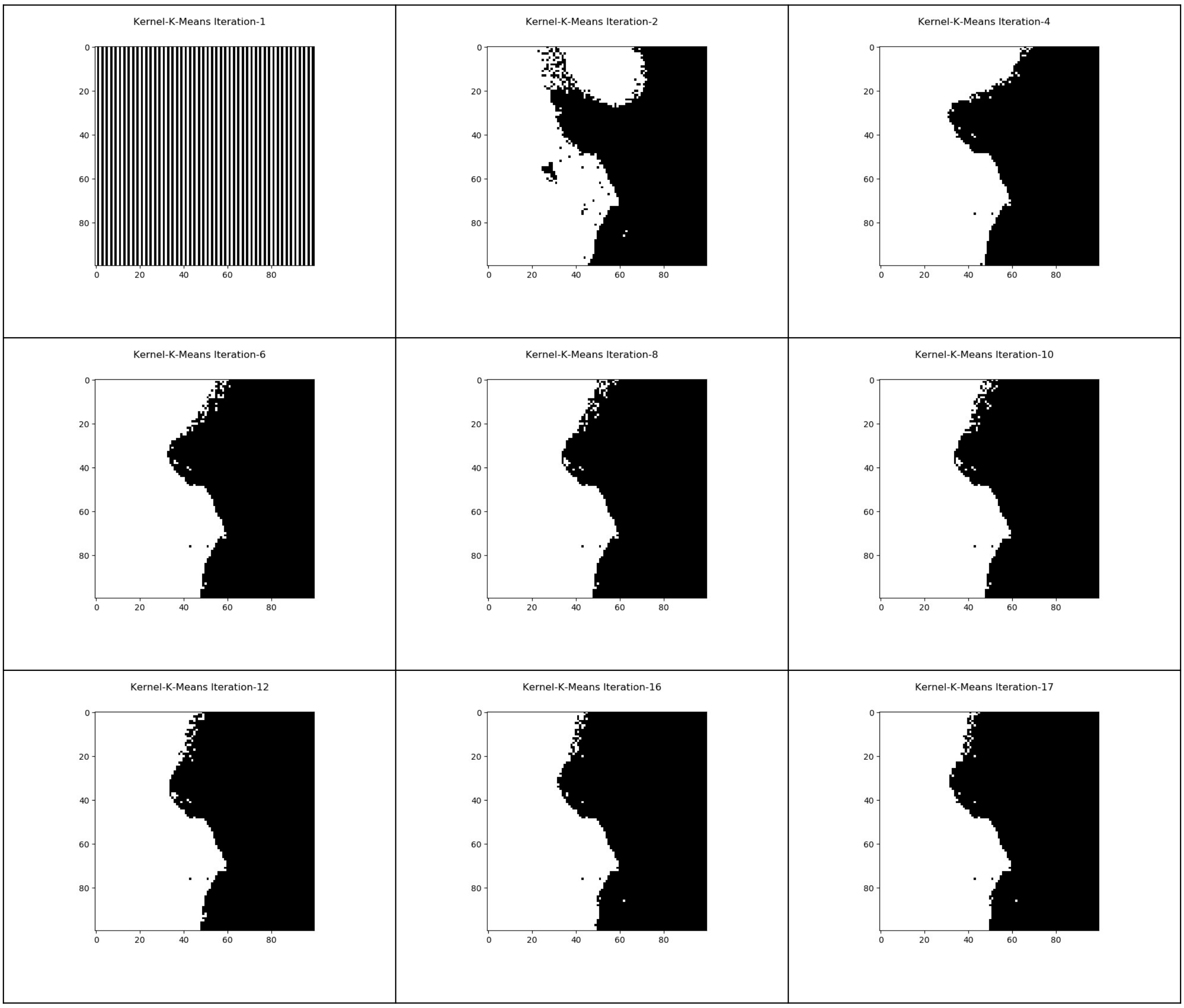
You can see the gif as well which should be in the compressed file. We see a nice evolution after iterating.

With the same initialization with 4 clusters :

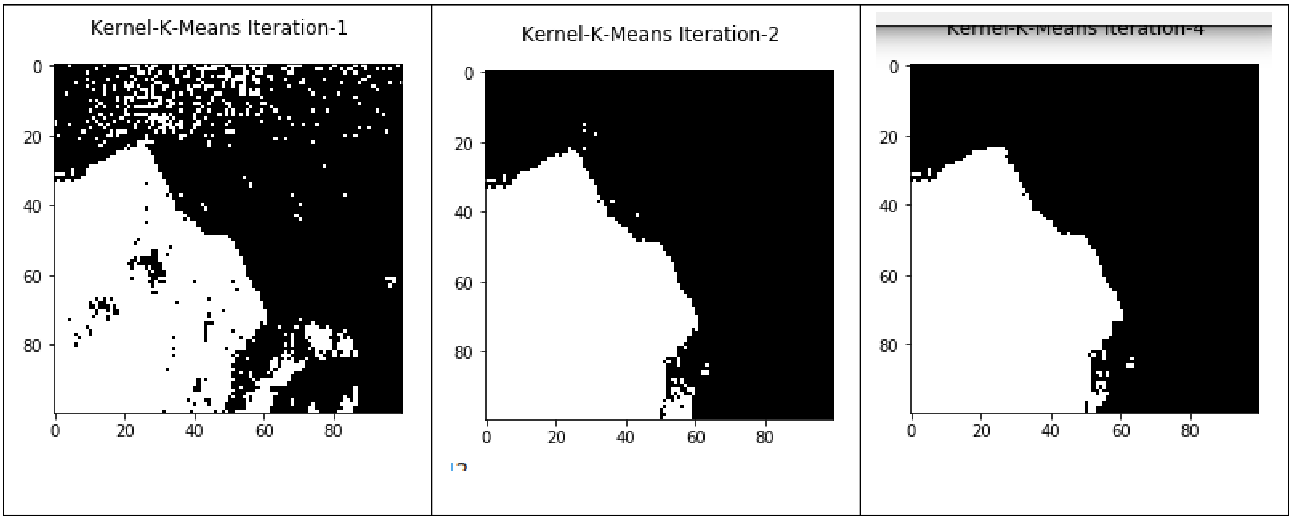


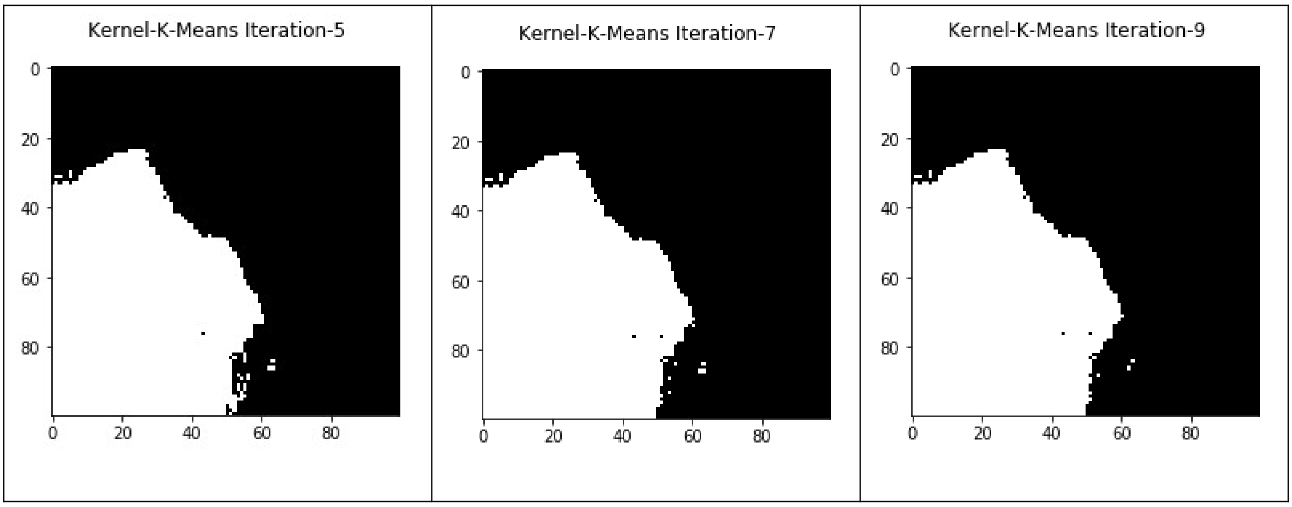
We recognize in a better way in my opinion the different regions of the initial picture. Very happy about this result.

Then with different init method :



And :





So for clustering with 2 regions only, this method is way better than the previous ones. We clearly identify the land and the see here.

The spectral clustering aims to take the eigenvalues of the kernel matrix in order to reduce the dimensionnality of the problem. I didn’t have time to fully run the spectral clustering because I still have to debug a few stuff. But I missed some time, so I can’t discuss about the eigenspace of graph Laplacian.

The idea was to compute the Laplacian, then to compute its eigenvalues and eigenvectors before using a very similar to k-means algorithm.