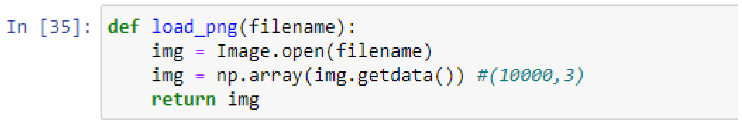
ML HW06 Report

a. Code

Part 1. Kernel k-means and Spectral clustering

I. Kernel k-means

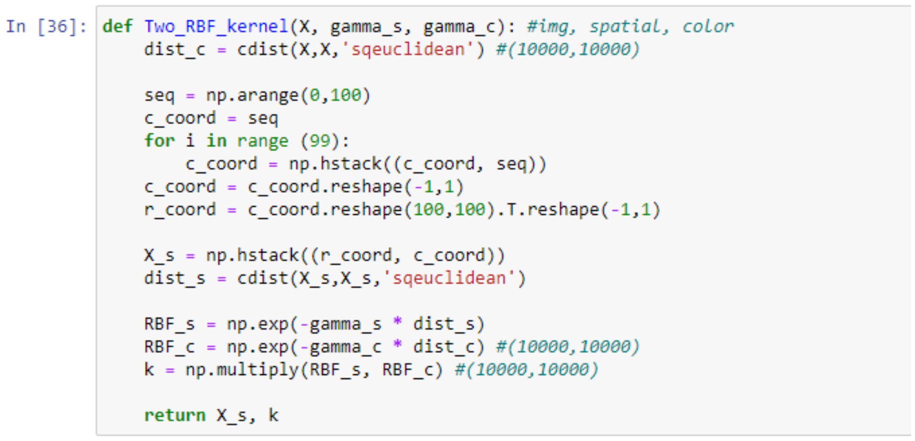
1. load_png() is a function that load the input image with PIL.



```
In [35]: def load_png(filename):
             img = Image.open(filename)
             img = np.array(img.getdata()) #(10000,3)
             return img
```

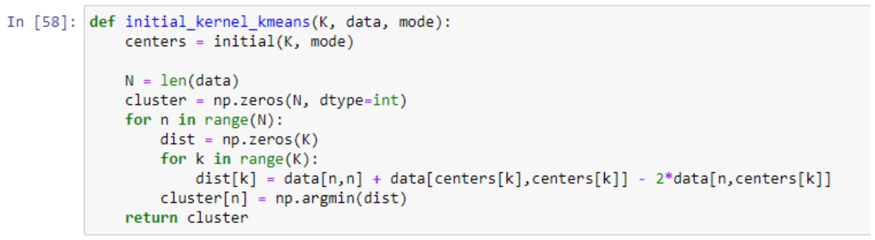
2. Two_RBF_kernel() is a function that use two RBF kernel which we can see in Spec
   - It considers both spatial and color info. Into this kernel.
   - X is input image which have color info.
   - X_s contains coordinate for each pixels for each image.
   - RBF_s is RBF kernel generate from spatial info.
   - RBF_c is RBF kernel generate from color info.
   - Finally return k equals to multiply RBF_s and RBF_c

$$k(x,x') = e^{-\gamma_s \|S(x)-S(x')\|^2} \times e^{-\gamma_c \|C(x)-C(x')\|^2}$$



```
In [36]: def Two_RBF_kernel(X, gamma_s, gamma_c): #img, spatial, color
             dist_c = cdist(X,X,'sqeuclidean') #(10000,10000)

             seq = np.arange(0,100)
             c_coord = seq
             for i in range (99):
                 c_coord = np.hstack((c_coord, seq))
             c_coord = c_coord.reshape(-1,1)
             r_coord = c_coord.reshape(100,100).T.reshape(-1,1)

             X_s = np.hstack((r_coord, c_coord))
             dist_s = cdist(X_s,X_s,'sqeuclidean')

             RBF_s = np.exp(-gamma_s * dist_s)
             RBF_c = np.exp(-gamma_c * dist_c) #(10000,10000)
             k = np.multiply(RBF_s, RBF_c) #(10000,10000)

             return X_s, k
```

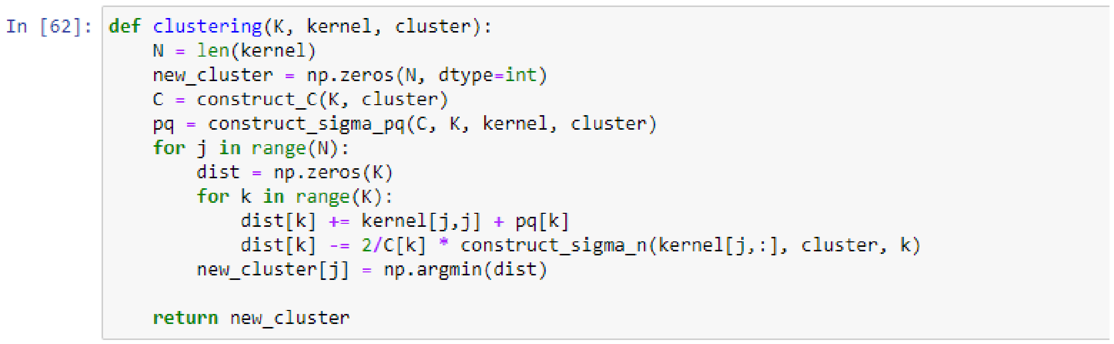
3. Initial_kernel_kmean()
   - Call initial function(I will explain it in Part 3.) to find initial center determine by which mode you choose
     It can be random or k means ++.
   - Use found centers to initial clustering. We can calculate each pixels' distance to each center.
     $||\phi(X_n) - \phi(\mu_n)|| = k(x_n, x_n) + k(\mu_k, \mu_k) - 2k(x_n, \mu_k).$
     Pixel will be clustered to the center that is closest to.



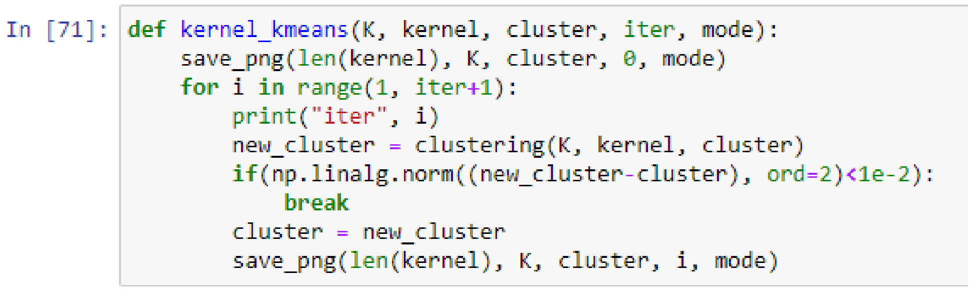
```
In [58]: def initial_kernel_kmeans(K, data, mode):
             centers = initial(K, mode)

             N = len(data)
             cluster = np.zeros(N, dtype=int)
             for n in range(N):
                 dist = np.zeros(K)
                 for k in range(K):
                     dist[k] = data[n,n] + data[centers[k],centers[k]] - 2*data[n,centers[k]]
                 cluster[n] = np.argmin(dist)
             return cluster
```

4. kernel_kmeans()
   - kernel_means simply repeat clustering function. It will stop when cluster do not change anymore.
   - Clustering follows the equation which in the lecture ppt

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \sum_{n=1}^{N} \alpha_{kn}\phi(x_n)\right\|$$
$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_p\sum_q \underline{\alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)}_{\text{Gram matrix!}}$$

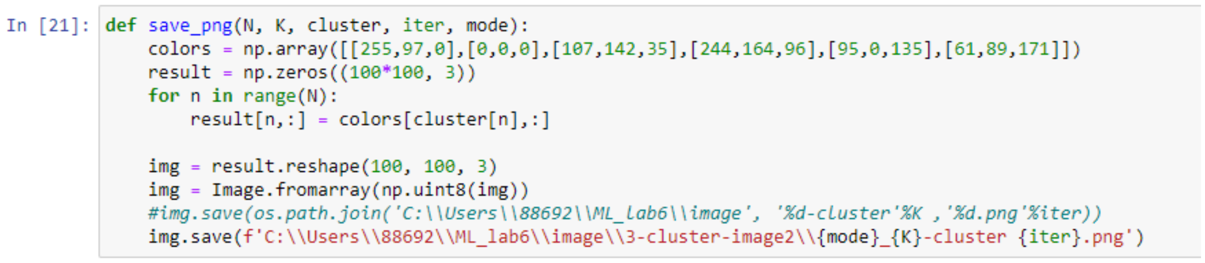


```
In [62]: def clustering(K, kernel, cluster):
             N = len(kernel)
             new_cluster = np.zeros(N, dtype=int)
             C = construct_C(K, cluster)
             pq = construct_sigma_pq(C, K, kernel, cluster)
             for j in range(N):
                 dist = np.zeros(K)
                 for k in range(K):
                     dist[k] += kernel[j,j] + pq[k]
                     dist[k] -= 2/C[k] * construct_sigma_n(kernel[j,:], cluster, k)
                 new_cluster[j] = np.argmin(dist)

             return new_cluster
```



```
In [71]: def kernel_kmeans(K, kernel, cluster, iter, mode):
             save_png(len(kernel), K, cluster, 0, mode)
             for i in range(1, iter+1):
                 print("iter", i)
                 new_cluster = clustering(K, kernel, cluster)
                 if(np.linalg.norm((new_cluster-cluster), ord=2)<1e-2):
                     break
                 cluster = new_cluster
                 save_png(len(kernel), K, cluster, i, mode)
```

5. Save_png(): we can visualize cluster image with the following code.



```
In [21]: def save_png(N, K, cluster, iter, mode):
             colors = np.array([[255,97,0],[0,0,0],[107,142,35],[244,164,96],[95,0,135],[61,89,171]])
             result = np.zeros((100*100, 3))
             for n in range(N):
                 result[n,:] = colors[cluster[n],:]

             img = result.reshape(100, 100, 3)
             img = Image.fromarray(np.uint8(img))
             #img.save(os.path.join('C:\\Users\\88692\\ML_lab6\\image', '%d-cluster'%K ,'%d.png'%iter))
             img.save(f'C:\\Users\\88692\\ML_lab6\\image\\3-cluster-image2\\{mode}_{K}-cluster {iter}.png')
```

II. Spectral clustering

# For spectral clustering, we need to compute eigenvectors of Laplacian. There are two types of Laplacian, unnormalized and normalized which serve in ratio cut and normalized cut respectively.

1. Global variable: `CLUSTER_NUM` is the number of the cluster, `POINT_NUM` is the number of data

points (pixels), and LENGTH is the number of pixels of the image.

```
POINT_NUM = 10000
LENGTH = 100
K = 2
```

2. Unnormalized Laplacian (ratio cut)

   subtract similarity matrix from degree matrix. And we use kernel matrix to represent similarity matrix. Below are the formula and its implementation

$$L = D - W$$

```
kernel = computeKernel(image.reshape(-1, 3))
D = np.sum(kernel, axis=1)
```

```
L = D - kernel
```

3. Normalized Laplacian (normalized cut)

$$L = I - D^{-1}WD^{-1}$$

```
kernel = computeKernel(image.reshape(-1, 3))
D = np.sum(kernel, axis=1)
```

```
D_sqr = np.diag(np.power(D, -0.5))
L = np.identity(POINT_NUM) - D_sqr @ kernel @ D_sqr
```

4. Compute eigenvectors (both ratio cut and normalized cut)

   Next step is to compute the first k eigenvectors of L where k is the number of clusters. Then form the matrix U with those k eigenvectors. Here we use `np.linalg.eig` to help us compute the result.

```
eigen_val, eigen_vec = np.linalg.eig(L)
idx = np.argsort(eigen_val)
eigen_vec = eigen_vec[:, idx]
U = eigen_vec[:, 1:CLUSTER_NUM+1].real
```

   That is done for ratio cut, but for normalized cut, we have to form the matrix T from U by normalizing the rows to norm 1, as teacher said in video

$$T \in \mathbb{R}^{n*k} \quad where \quad t_{ij} = \frac{u_{ij}}{(\sum_k u_{ik}^2)^{(1/2)}}$$

```
T = np.zeros(U.shape, dtype=np.float64)
for i in range(POINT_NUM):
    T[i] = U[i] / np.sqrt(np.sum(U[i] ** 2))
```

we can use matrix U / T as our new data points and do the k-means clustering. The code below is the implementation of k-means.

5. Kmeans(): a function implement k-means

```python
def kMeans(image, imagename, eigen, cut, method):
    iteration = 0
    centroids, cluster = init(eigen, method)
    draw(image, imagename, cluster, iteration, cut)
    while 1:
        iteration += 1
        pre_cluster = cluster
        cluster = clustering(eigen, centroids)
        centroids = updateCentroids(eigen, centroids, cluster)
        draw(image, imagename, cluster, iteration, cut)
        error = computeError(cluster, pre_cluster)
        print(f'Iter: {iteration}: {error}')
        if error == 0:
            break
    if CLUSTER_NUM == 2:
        drawEigenSpace(eigen, cluster, cut)
```

6. Clustering(): Assign each data point to one of k clusters. The data points belong to which clusters depends on the distance between data points and the centroids of clusters. Below is the implementation.

```python
def clustering(eigen, centroids):
    cluster = np.zeros(POINT_NUM, dtype=int)
    for i in range(POINT_NUM):
        distance = np.zeros(K, dtype=np.float32)
        for j in range(K):
            distance[j] = np.sum(np.absolute(eigen[i] - centroids[j]))
        cluster[i] = np.argmin(distance)
    return cluster
```

7. updateCentroids(): update the centroids. We compute the average value of each cluster, then assign it as a new centroid of that cluster.

```python
def updateCentroids(eigen, centroids, cluster):
    centroids = np.zeros(centroids.shape, dtype=np.float64)
    cnt = np.zeros(K, dtype=np.int32)
    for i in range(POINT_NUM):
        centroids[cluster[i]] += eigen[i]
        cnt[cluster[i]] += 1
    for i in range(K):
        if cnt[i] == 0:
            cnt[i] = 1
        centroids[i] /= cnt[i]
    return centroids
```

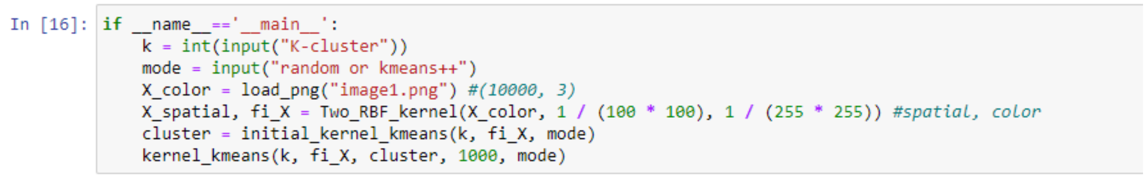
8. draw(): we can visualize cluster image with the following code.

```python
def draw(image, imagename, cluster, iteration, cut):
    title = f'Spectral Clustering Iteration-{iteration}'
    plt.clf()
    plt.subplot(111, aspect='equal')
    cluster_x = []
    cluster_y = []
    for j in range(K):
        cluster_x.append([])
        cluster_y.append([])
    for i in range(POINT_NUM):
        cluster_x[cluster[i]].append((i // LENGTH) / (LENGTH - 1))
        cluster_y[cluster[i]].append((i % LENGTH) / (LENGTH - 1))
    for j in range(K):
        plt.scatter(cluster_y[j], cluster_x[j], s=2, c=[colormap[j]])
    plt.xlim(0, 1)
    plt.ylim(1, 0)
    plt.suptitle(title)
    plt.tight_layout()
    plt.savefig(f'D:\Download_vivaldi\ML_HW06\ML_HW06\spectral-clustering/{imagename}_{K}-cluster_{cut}_{iteration}.png')
```

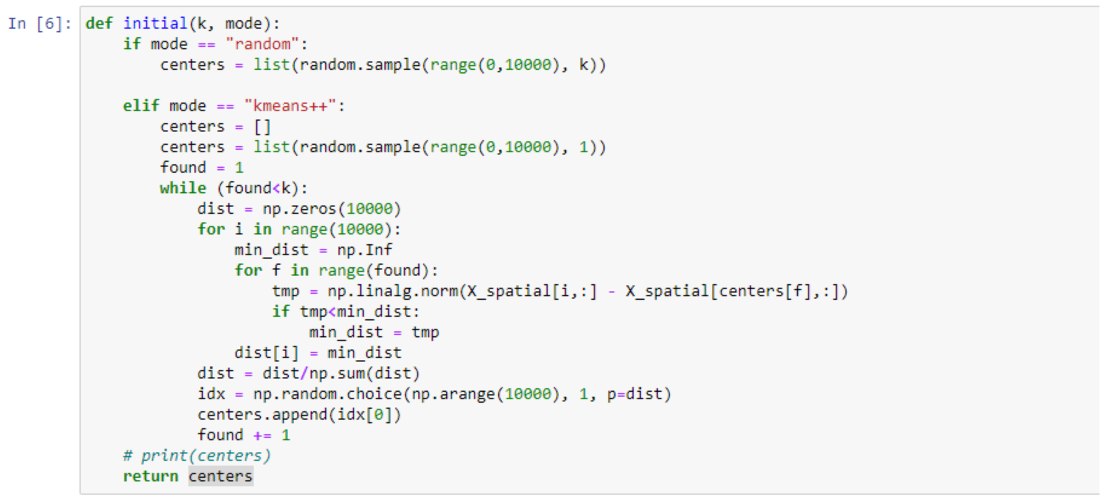
Part 2. Cluster data into 2 clusters, try more clusters

I. Kernel k-means (initial method use k means ++)

1. Main_function() is will get the info. about how many cluster you want to sperate, and which mode you want to select, then in initial function will get k center for clustering
   - k contain the cluster number (e.g. 2-cluster k = 2 etc.)



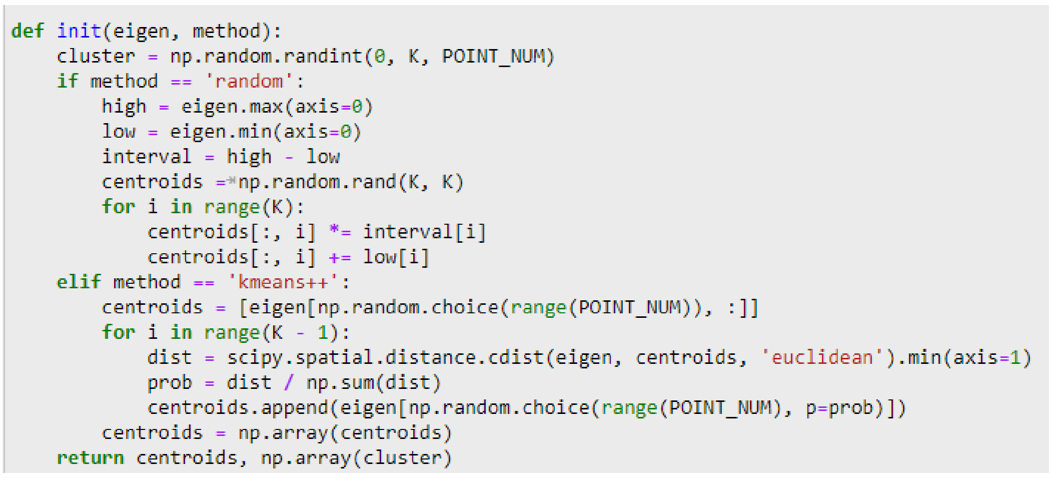
```
In [16]: if __name__=='__main__':
             k = int(input("K-cluster"))
             mode = input("random or kmeans++")
             X_color = load_png("image1.png") #(10000, 3)
             X_spatial, fi_X = Two_RBF_kernel(X_color, 1 / (100 * 100), 1 / (255 * 255)) #spatial, color
             cluster = initial_kernel_kmeans(k, fi_X, mode)
             kernel_kmeans(k, fi_X, cluster, 1000, mode)
```



```
In [6]: def initial(k, mode):
            if mode == "random":
                centers = list(random.sample(range(0,10000), k))

            elif mode == "kmeans++":
                centers = []
                centers = list(random.sample(range(0,10000), 1))
                found = 1
                while (found<k):
                    dist = np.zeros(10000)
                    for i in range(10000):
                        min_dist = np.Inf
                        for f in range(found):
                            tmp = np.linalg.norm(X_spatial[i,:] - X_spatial[centers[f],:])
                            if tmp<min_dist:
                                min_dist = tmp
                        dist[i] = min_dist
                    dist = dist/np.sum(dist)
                    idx = np.random.choice(np.arange(10000), 1, p=dist)
                    centers.append(idx[0])
                    found += 1
        #    print(centers)
            return centers
```

II. Spectral clustering (initial method use k means ++)

1. Main_function() is will get the info. about how many cluster you want to sperate, and which mode you want to select, then in initial function will get k center for clustering
   - k contain the cluster number (e.g. 2-cluster k = 2 etc.)

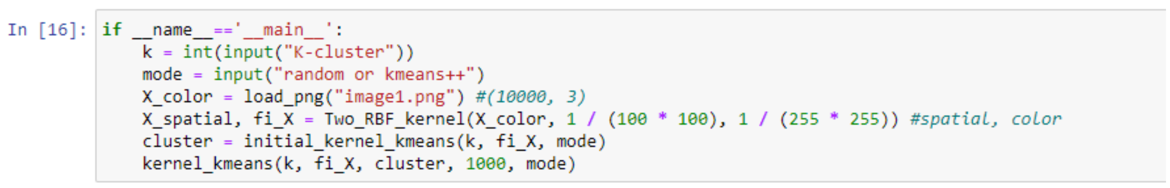


```
def init(eigen, method):
    cluster = np.random.randint(0, K, POINT_NUM)
    if method == 'random':
        high = eigen.max(axis=0)
        low = eigen.min(axis=0)
        interval = high - low
        centroids = np.random.rand(K, K)
        for i in range(K):
            centroids[:, i] *= interval[i]
            centroids[:, i] += low[i]
    elif method == 'kmeans++':
        centroids = [eigen[np.random.choice(range(POINT_NUM)), :]]
        for i in range(K - 1):
            dist = scipy.spatial.distance.cdist(eigen, centroids, 'euclidean').min(axis=1)
            prob = dist / np.sum(dist)
            centroids.append(eigen[np.random.choice(range(POINT_NUM), p=prob)])
        centroids = np.array(centroids)
    return centroids, np.array(cluster)
```

Part 3. Initialization of k mean and spectral with different methods (eg. k mean++, random)

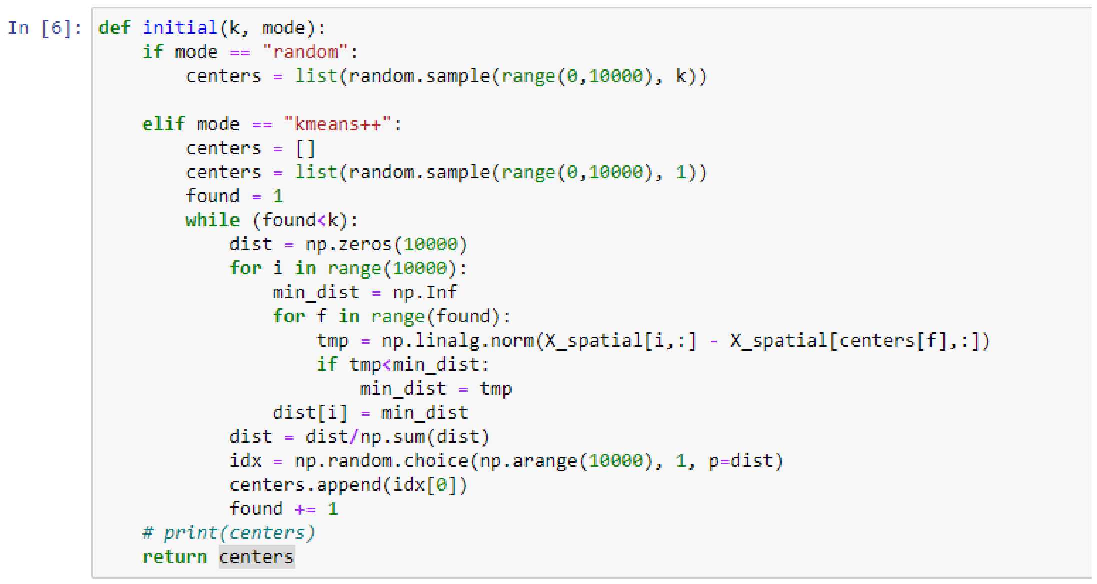
I. K means clustering (k mean++, random)

1. Main_function() is will get the info. about how many cluster you want to sperate, and which mode you want to select.
   - Mode will effect which initial center will be choose



```
In [16]: if __name__=='__main__':
             k = int(input("K-cluster"))
             mode = input("random or kmeans++")
             X_color = load_png("image1.png") #(10000, 3)
             X_spatial, fi_X = Two_RBF_kernel(X_color, 1 / (100 * 100), 1 / (255 * 255)) #spatial, color
             cluster = initial_kernel_kmeans(k, fi_X, mode)
             kernel_kmeans(k, fi_X, cluster, 1000, mode)
```

2. Initial() is a function to find initial center determine by which mode you choose
   - Mode == "random": random from exist data points
   - Mode == "kmeans++": random find one data point and find farthest data point
     And repeat finding farthest data point until found k initial data points

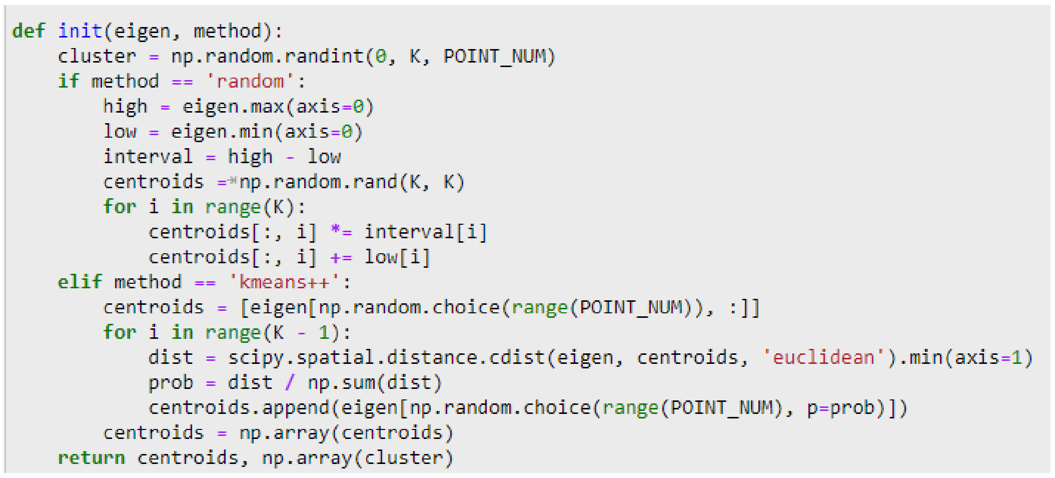


```
In [6]: def initial(k, mode):
            if mode == "random":
                centers = list(random.sample(range(0,10000), k))

            elif mode == "kmeans++":
                centers = []
                centers = list(random.sample(range(0,10000), 1))
                found = 1
                while (found<k):
                    dist = np.zeros(10000)
                    for i in range(10000):
                        min_dist = np.Inf
                        for f in range(found):
                            tmp = np.linalg.norm(X_spatial[i,:] - X_spatial[centers[f],:])
                            if tmp<min_dist:
                                min_dist = tmp
                        dist[i] = min_dist
                    dist = dist/np.sum(dist)
                    idx = np.random.choice(np.arange(10000), 1, p=dist)
                    centers.append(idx[0])
                    found += 1
            # print(centers)
            return centers
```

II. Spectral (normalize and ratio cut)

1. Initial() is a function to find initial center determine by which mode you choose
- Mode == "random": random from exist data points
- Mode == "kmeans++": random find one data point and find farthest data point

And repeat finding farthest data point until found k initial data points

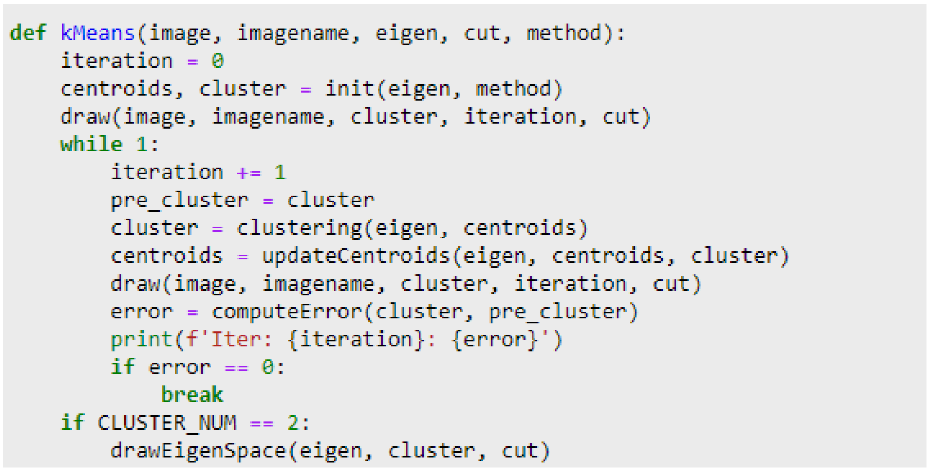


```
def init(eigen, method):
    cluster = np.random.randint(0, K, POINT_NUM)
    if method == 'random':
        high = eigen.max(axis=0)
        low = eigen.min(axis=0)
        interval = high - low
        centroids = np.random.rand(K, K)
        for i in range(K):
            centroids[:, i] *= interval[i]
            centroids[:, i] += low[i]
    elif method == 'kmeans++':
        centroids = [eigen[np.random.choice(range(POINT_NUM)), :]]
        for i in range(K - 1):
            dist = scipy.spatial.distance.cdist(eigen, centroids, 'euclidean').min(axis=1)
            prob = dist / np.sum(dist)
            centroids.append(eigen[np.random.choice(range(POINT_NUM), p=prob)])
        centroids = np.array(centroids)
    return centroids, np.array(cluster)
```

Part 4. Spectral clustering does have the same coordinates in the eigenspace

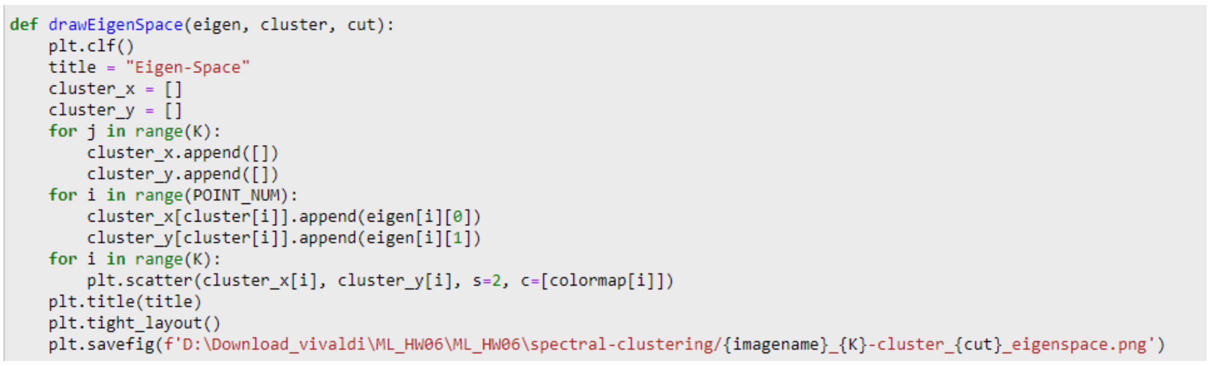
In kMeans, we sent the eigenvector get from compute Normalized Laplacian eigenvector in step 4

Normalized sent T as eigen, ratio sent U as eigen



```python
def kMeans(image, imagename, eigen, cut, method):
    iteration = 0
    centroids, cluster = init(eigen, method)
    draw(image, imagename, cluster, iteration, cut)
    while 1:
        iteration += 1
        pre_cluster = cluster
        cluster = clustering(eigen, centroids)
        centroids = updateCentroids(eigen, centroids, cluster)
        draw(image, imagename, cluster, iteration, cut)
        error = computeError(cluster, pre_cluster)
        print(f'Iter: {iteration}: {error}')
        if error == 0:
            break
    if CLUSTER_NUM == 2:
        drawEigenSpace(eigen, cluster, cut)
```

Then plot the eigenspace to observe the coordinate of Laplacian



```python
def drawEigenSpace(eigen, cluster, cut):
    plt.clf()
    title = "Eigen-Space"
    cluster_x = []
    cluster_y = []
    for j in range(K):
        cluster_x.append([])
        cluster_y.append([])
    for i in range(POINT_NUM):
        cluster_x[cluster[i]].append(eigen[i][0])
        cluster_y[cluster[i]].append(eigen[i][1])
    for i in range(K):
        plt.scatter(cluster_x[i], cluster_y[i], s=2, c=[colormap[i]])
    plt.title(title)
    plt.tight_layout()
    plt.savefig(f'D:\Download_vivaldi\ML_HW06\ML_HW06\spectral-clustering/{imagename}_{K}-cluster_{cut}_eigenspace.png')
```

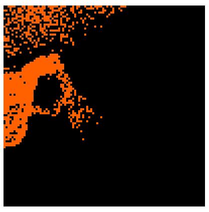
b. Experiments setting and results & Disscusion

Part 1. Kernel k-means and Spectral clustering

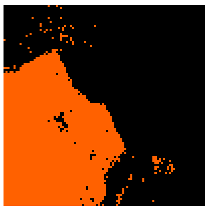
I. Kernel k-means

1. Image1.png use random (only show the iteration = 1 and result cluster)

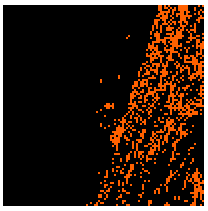
- Iteration 1



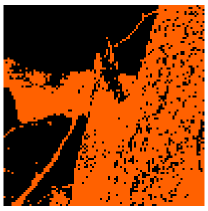
- Iteration 5



2. Image2.png use random (only show the result cluster)
   - Iteration 1

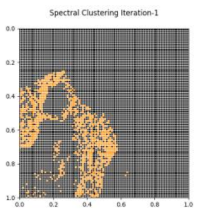


   - Iteration 16

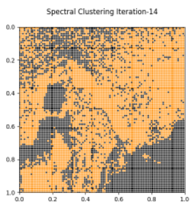


II. Spectral clustering with ratio

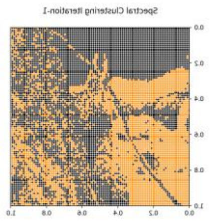
1. Image1.png use random (only show the iteration = 1 and result cluster)
   - Iteration 1



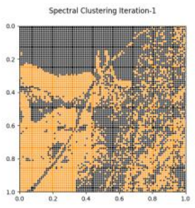
   - Iteration 14



2. Image2.png use random (only show the result cluster)
   - Iteration 1

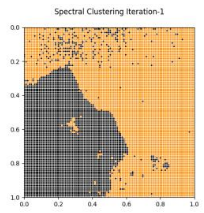


   - Iteration 16

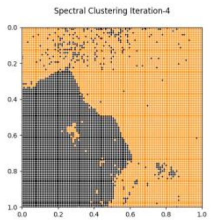


III. Spectral clustering with normalize

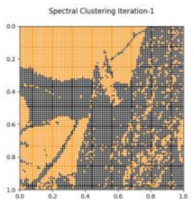
1. Image1.png use random (only show the iteration = 1 and result cluster)
    - Iteration 1



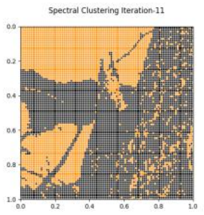
    - Iteration 4



2. Image2.png use random (only show the result cluster)
    - Iteration 1



    - Iteration 11



IV. Discussion:

1. In image1 we can see some black dot within orange cluster. This is due to the reason that the dots have similar color with the island and also close to the island, so algorithm give them in the same cluster.
2. Some part of image2 is labeled as background. This is not what we expected to have. Nevertheless, it is reasonable. Some parts of trunk has similar color, light blue, with sky in the background. Since I consider color in my kernel, these blue dots would be considered similar to blue sky. Thus, they may be classify as same cluster

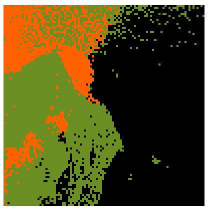
Part 2. Cluster data into 2 clusters, try more clusters (e.g. 3 or 4)

I. Kernel k-means (initial method use k means ++)

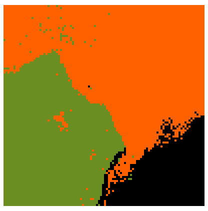
1. 3-cluster

    (1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

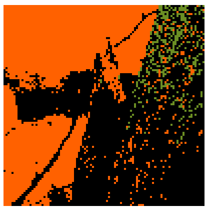


- Iteration 24

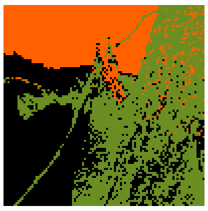


(2) Image2.png use random (only show the result cluster)

- Iteration 1



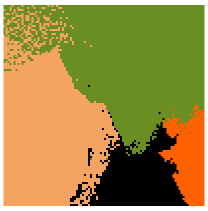
- Iteration 41



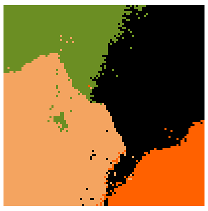
2. 4-cluster

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

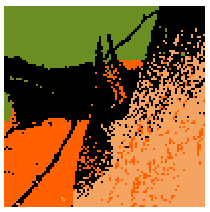


- Iteration 22

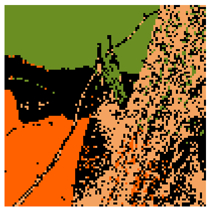


(2) Image2.png use random (only show the result cluster)

- Iteration 1



- Iteration 24

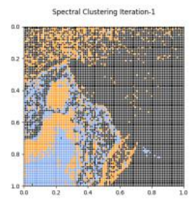


II. Spectral clustering with normalized

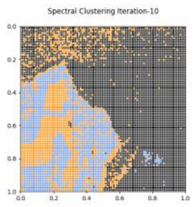
1. 3-cluster

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

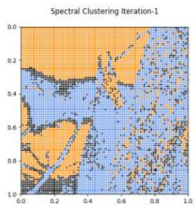


- Iteration 10

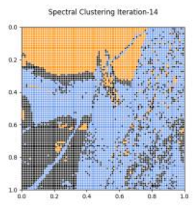


(2) Image2.png use random (only show the result cluster)

- Iteration 1



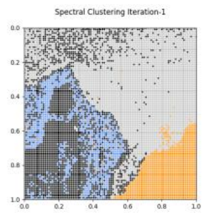
- Iteration 14



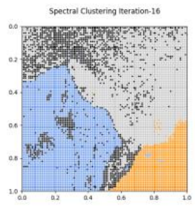
2. 4-cluster

(1) Image1.png use random (only show the iteration = 0 and result cluster)

- Iteration 1

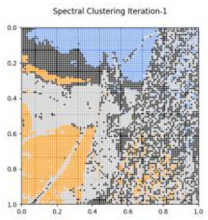


- Iteration 16

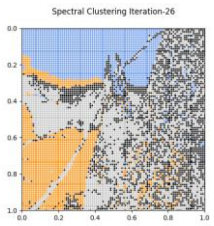


(2) Image2.png use random (only show the result cluster)

- Iteration 1



- Iteration 26

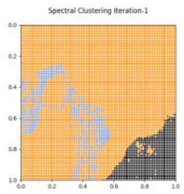


III. Spectral clustering with ratio

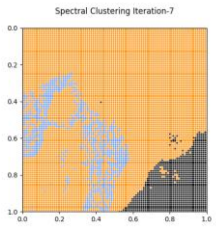
1. 3-cluster

(3) Image1.png use random (only show the iteration = 0 and result cluster)

- Iteration 1

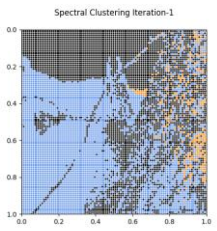


- Iteration 7

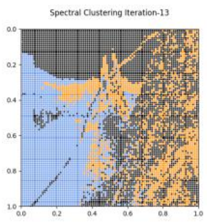


(4) Image2.png use random (only show the result cluster)

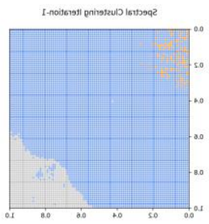
- Iteration 1



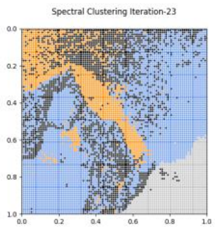
- Iteration 13



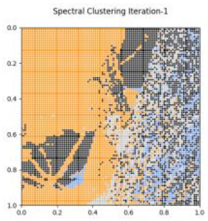
2. 4-cluster
   (3) Image1.png use random (only show the iteration = 1 and result cluster)
   - Iteration 1



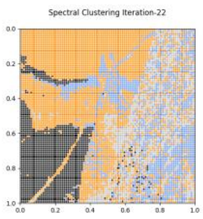
- Iteration 23



(4) Image2.png use random (only show the result cluster)
- Iteration 1



- Iteration 37



IV. Discussion:

1. Within this different cluster experiment, I think that decide which dots belong to which cluster is following the color. It is obvious in 4-cluster. The result shows that dark green is one cluster, light green is another cluster. The result doesn't like what we except.

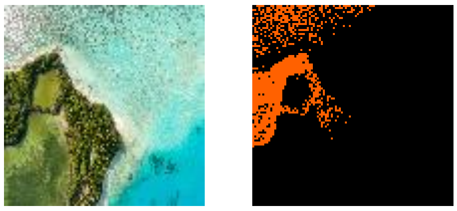
Part 3. Initialization of k mean and spectral with different methods (eg. k mean++, random)

I. K means clustering (random, k mean++)

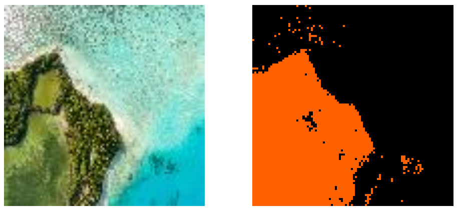
1. 2-cluster with random

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

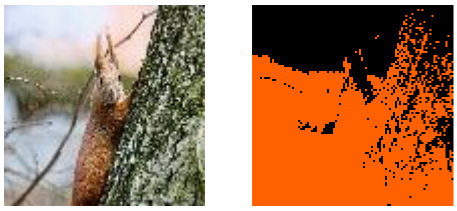


- Iteration 5

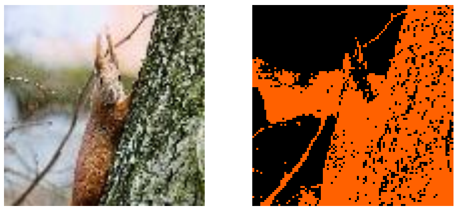


(2) Image2.png use random (only show the result cluster)

- Iteration 1



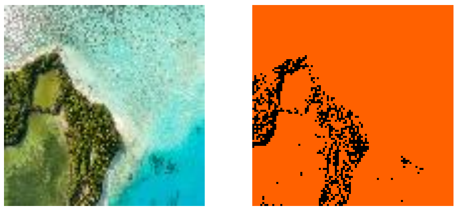
- Iteration 27



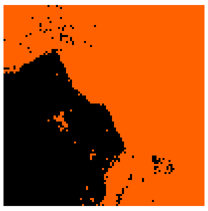
2. 2-cluster with k mean++

(1) Image1.png use random (only show the iteration = 0 and result cluster)

- Iteration 1

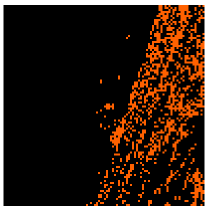


- Iteration 6

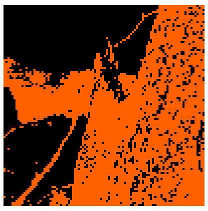


(2) Image2.png use random (only show the result cluster)

- Iteration 1



- Iteration 16

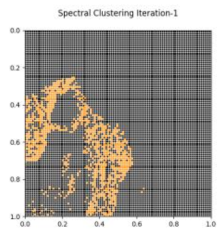


II. Spectral with ratio cut

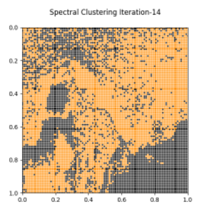
1. 2-cluster with random

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

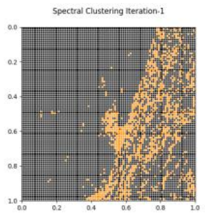


- Iteration 14

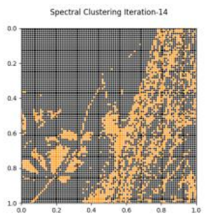


(2) Image2.png use random (only show the result cluster)

- Iteration 1



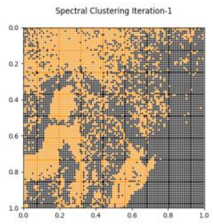
- Iteration 27



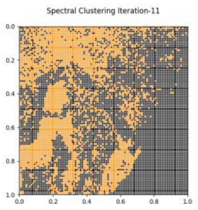
2. 2-cluster with k mean++

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

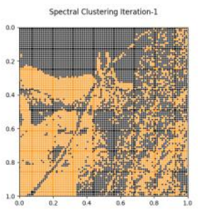


- Iteration 11

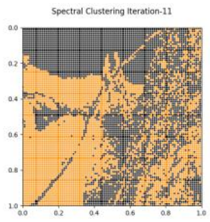


(2) Image2.png use random (only show the result cluster)

- Iteration 1



- Iteration 16

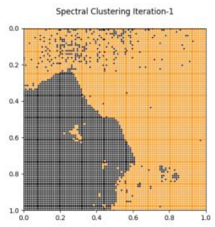


III. Spectral with normalized cut

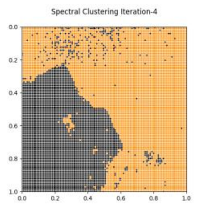
1. 2-cluster with random

(1) Image1.png use random (only show the iteration = 0 and result cluster)

- Iteration 0

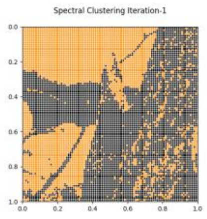


- Iteration 4

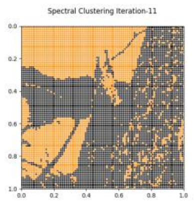


(2) Image2.png use random (only show the result cluster)

- Iteration 1



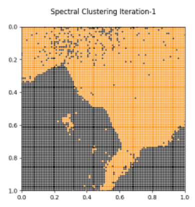
- Iteration 11



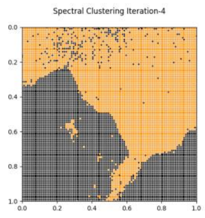
2. 2-cluster with k mean++

(1) Image1.png use random (only show the iteration = 1 and result cluster)

- Iteration 1

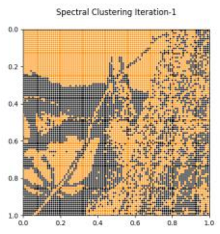


- Iteration 4

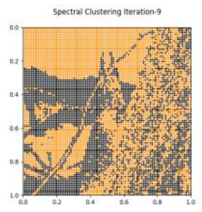


(2) Image2.png use random (only show the result cluster)

- Iteration 1



- Iteration 9



IV. Discussion:

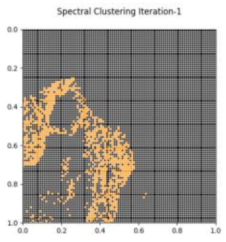
1. Random initial v.s. k-means++:
   - These two initial may lead to same result. However, there is higher chances for random initial to result in bad clustering.
   - Random initial may get really close centers at the beginning of the program. Thus, it requires more time to converge to final clustering than k-means++.
   - k-mean++ guarantee to find far initial centers. It is really important to have centers far from each others. It may have no huge difference in this assignment since the input data is not huge. The cost is still ok to have a worse initial. However, it would show great difference to have good initial in larger computational or data cases. Thus, it is good to have k-mean++ than simply random.

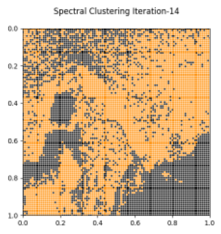
Part 4. Spectral clustering does have the same coordinates in the eigenspace

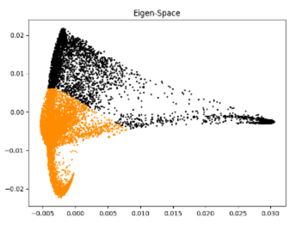
I. Spectral with ratio cut

I only visualize the eigenspace for 2-cluster and we can see that the data points with the same cluster gather in eigenspace.

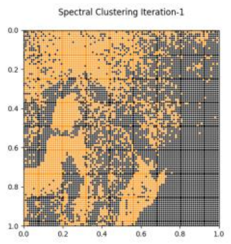
1. 2-cluster random

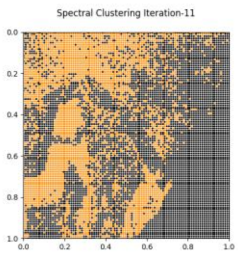


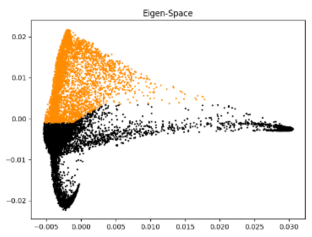




2. 2-cluster kmeans++



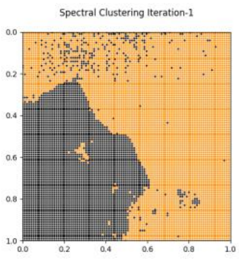


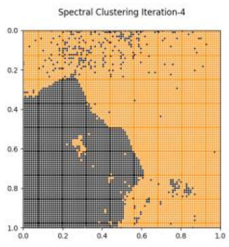


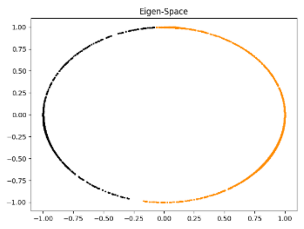
II. Spectral with normalized cut

I only visualize the eigenspace for 2-cluster and we can see that the data points with the same cluster gather in eigenspace.

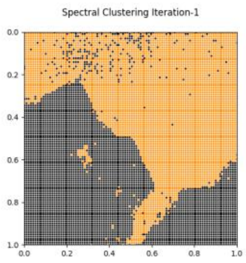
1. 2-cluster random

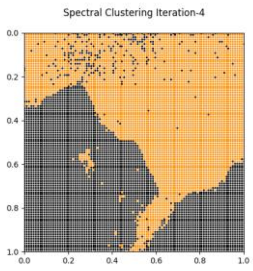


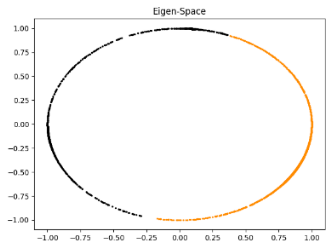




2. 2-cluster kmeans++







III. Dicussion

1. No matter random or kmeans++ the eigenspace seems the same
2. ratio cut its eigenspace spread more irregular.
3. Normalized cut its eigenspace spread more regular like a circule.

c. Discussion & observations

1. It is a unsupervised learning. There is no correct k to be chosen. However, we can see from the result that there are better choices. Take image1.png as an example. It is a more reasonable result use clustering to separate the island and ocean. Thus, it is good to choose k as 2. Choosing k as 3 may still be reasonable, since there are significant 2 difference colors in ocean. We can label them with different class. However, it would be weird to choose k as 5 or 6 or even larger
2. k is a data-dependent variable. There is no general solution to find one
3. the experiment ratio cut and normalized cut spend a lot of time, I think it take a long time to solve the eigenvector problem.