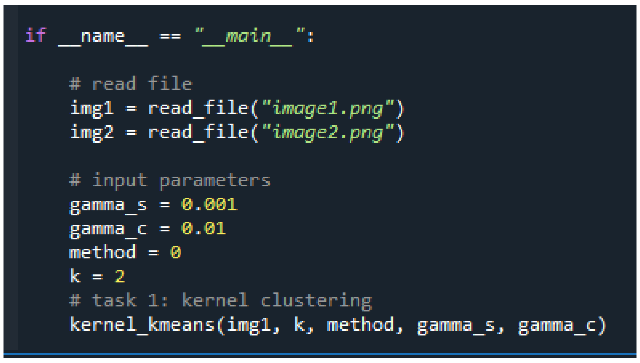
ML HW6

312554006 Ming-Chih, Lo

1. Code with detailed explanations

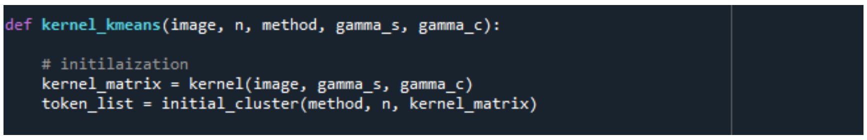
i. Part 1, Kmeans



```
if __name__ == "__main__":

    # read file
    img1 = read_file("image1.png")
    img2 = read_file("image2.png")

    # input parameters
    gamma_s = 0.001
    gamma_c = 0.01
    method = 0
    k = 2
    # task 1: kernel clustering
    kernel_kmeans(img1, k, method, gamma_s, gamma_c)
```

(1) Main function: run the kernel_kmeans function

(2) There are several parts in the kernel_kmeans function. Firstly, I calculate the kernel similarity based on the hybrid RBF kernel specified in the spec. Moreover, to initialize the cluster based on the "method" parameters

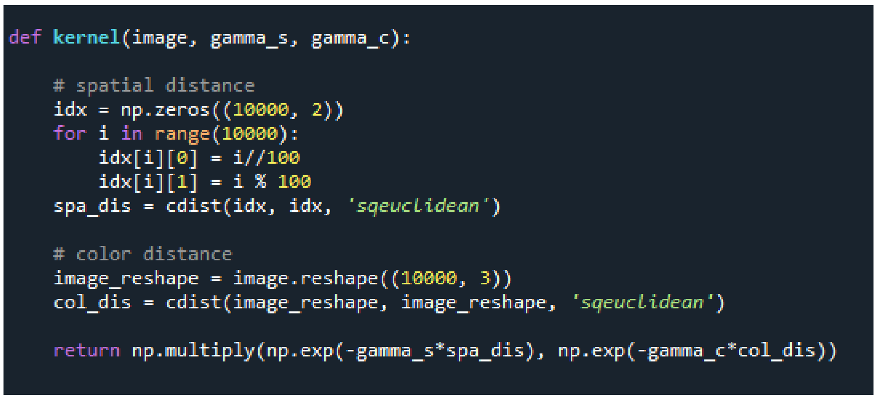


```
def kernel_kmeans(image, n, method, gamma_s, gamma_c):

    # initilaization
    kernel_matrix = kernel(image, gamma_s, gamma_c)
    token_list = initial_cluster(method, n, kernel_matrix)
```

To be more specific, the kernel function is as follow:

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

We use the following kernel function to calculate spatial distance and color distance. After it, we get a 10000*10000 matrix, which is our pairwise kernel.



```
def kernel(image, gamma_s, gamma_c):

    # spatial distance
    idx = np.zeros((10000, 2))
    for i in range(10000):
        idx[i][0] = i//100
        idx[i][1] = i % 100
    spa_dis = cdist(idx, idx, 'sqeuclidean')

    # color distance
    image_reshape = image.reshape((10000, 3))
    col_dis = cdist(image_reshape, image_reshape, 'sqeuclidean')

    return np.multiply(np.exp(-gamma_s*spa_dis), np.exp(-gamma_c*col_dis))
```

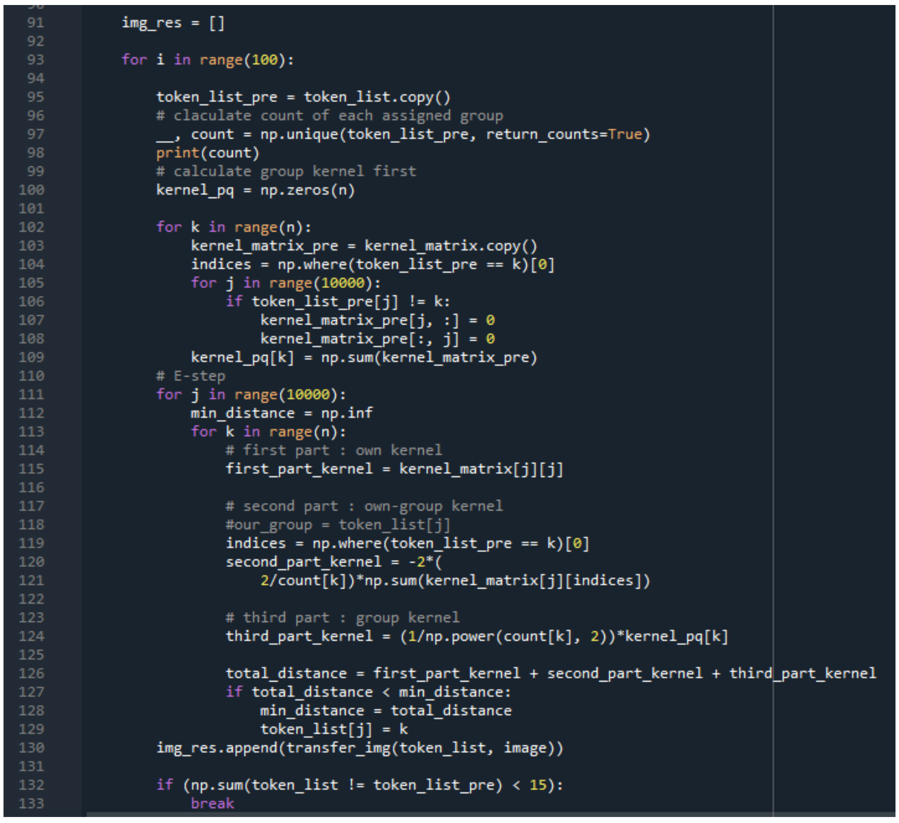
Also, the default initialize method here is to random assign each data points into k cluster. (Hard to converge in this way)

```
def initial_cluster(method, n, kernel_matrix):

    # original method: to divide different parts depend on positions
    if method == 0:
        cluster_label = np.random.randint(0, n, size=10000)
        return cluster_label
    # k-means++
    else:
```

(3) Based on the slides provided by pf. Chiu (pg.22, unsupervised learning), we can calculate the distance b/w each point and each cluster with the following formulas:

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)\right\|$$
$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_{n}\alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q}\alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$

Here, I divide the whole formulas into three parts, and calculate them separately, as specified in line 95-125, and add them together in line 126



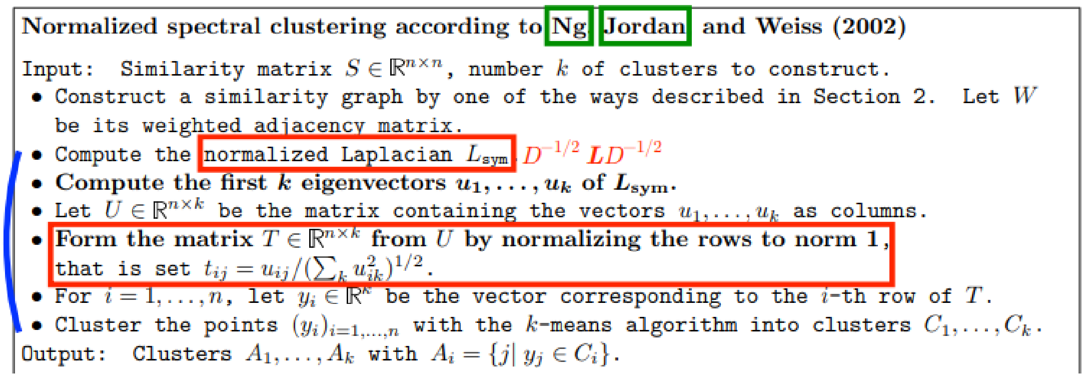
```
 91     img_res = []
 92
 93     for i in range(100):
 94
 95         token_list_pre = token_list.copy()
 96         # claculate count of each assigned group
 97         __, count = np.unique(token_list_pre, return_counts=True)
 98         print(count)
 99         # calculate group kernel first
100         kernel_pq = np.zeros(n)
101
102         for k in range(n):
103             kernel_matrix_pre = kernel_matrix.copy()
104             indices = np.where(token_list_pre == k)[0]
105             for j in range(10000):
106                 if token_list_pre[j] != k:
107                     kernel_matrix_pre[j, :] = 0
108                     kernel_matrix_pre[:, j] = 0
109             kernel_pq[k] = np.sum(kernel_matrix_pre)
110         # E-step
111         for j in range(10000):
112             min_distance = np.inf
113             for k in range(n):
114                 # first part : own kernel
115                 first_part_kernel = kernel_matrix[j][j]
116
117                 # second part : own-group kernel
118                 #our_group = token_list[j]
119                 indices = np.where(token_list_pre == k)[0]
120                 second_part_kernel = -2*(
121                     2/count[k])*np.sum(kernel_matrix[j][indices])
122
123                 # third part : group kernel
124                 third_part_kernel = (1/np.power(count[k], 2))*kernel_pq[k]
125
126                 total_distance = first_part_kernel + second_part_kernel + third_part_kernel
127                 if total_distance < min_distance:
128                     min_distance = total_distance
129                     token_list[j] = k
130         img_res.append(transfer_img(token_list, image))
131
132         if (np.sum(token_list != token_list_pre) < 15):
133             break
```

After getting the pairwise distance (M step), we reassign each point to the nearest cluster (E step), as specified in line 127-130

// The num of iteration is set to 100, and the early stop condition is when the changes in new cluster and the old one are less than 5.

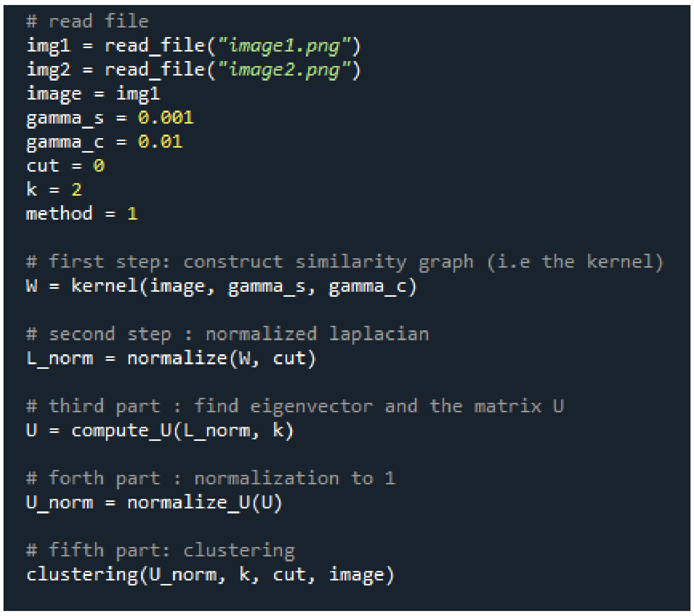
i. Part 1, Spectral clustering

(1) Following the algorithm (slides pg. 74), the overall code structure is as follows:



Normalized spectral clustering according to $\boxed{\text{Ng}}$ $\boxed{\text{Jordan}}$ and Weiss (2002)

```
Input:  Similarity matrix $S \in \mathbb{R}^{n\times n}$, number $k$ of clusters to construct.
 • Construct a similarity graph by one of the ways described in Section 2.  Let $W$
   be its weighted adjacency matrix.
 • Compute the $\boxed{\text{normalized Laplacian } L_{\text{sym}}}$ $D^{-1/2} L D^{-1/2}$
 • Compute the first $k$ eigenvectors $u_1, \dots, u_k$ of $L_{\text{sym}}$.
 • Let $U \in \mathbb{R}^{n\times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ as columns.
 • Form the matrix $T \in \mathbb{R}^{n\times k}$ from $U$ by normalizing the rows to norm 1,
   that is set $t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}$.
 • For $i = 1, \dots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the $i$-th row of $T$.
 • Cluster the points $(y_i)_{i=1,\dots,n}$ with the $k$-means algorithm into clusters $C_1, \dots, C_k$.
Output:  Clusters $A_1, \dots, A_k$ with $A_i = \{j \mid y_j \in C_i\}$.
```



```
# read file
img1 = read_file("image1.png")
img2 = read_file("image2.png")
image = img1
gamma_s = 0.001
gamma_c = 0.01
cut = 0
k = 2
method = 1

# first step: construct similarity graph (i.e the kernel)
W = kernel(image, gamma_s, gamma_c)

# second step : normalized laplacian
L_norm = normalize(W, cut)

# third part : find eigenvector and the matrix U
U = compute_U(L_norm, k)

# forth part : normalization to 1
U_norm = normalize_U(U)

# fifth part: clustering
clustering(U_norm, k, cut, image)
```

(2) the kernel function here is the same one in the Kmeans method. Using the kernel, we can get the weight graph.

(3) the normalization part is based on different cut method, to normalize the Laplacian matrix.

```python
def normalize(W, cut):
    # based on the definition: L = D - W, and D is diagnoal degree matrix
    D = np.zeros_like(W)
    for i in range(len(D)):
        D[i][i] = np.sum(W[i])
    L = D - W

    # based on different cut method, we have different nomralized way
    # normalize cut

    if cut == 0:
        for i in range(10000):
            D[i][i] = 1 / np.sqrt(D[i][i])
        L_norm = D@L@D
        return L_norm

    # ratio cut
    else:
        L_norm = L
        return L_norm
```

We can get L matrix from D – W, and based on the definition, the D matrix is the diagonal matrix that contains all connected data points.

Moreover, based on different cut method, we have different normalization methods: For normalize cut, we use sqrt(D)@L@sqrt(D) and for ratio cut, we use the same Laplacian matrix.

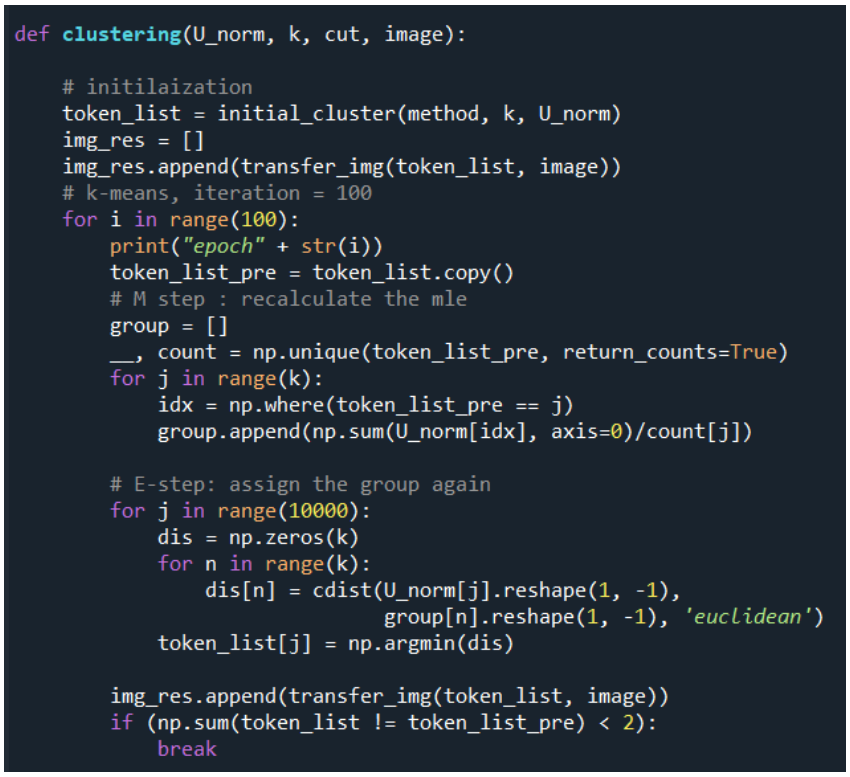
(4) get eigenvalues and eigenvectors by using the argsort function, and reply the first k eigenvectors with nonzero eigenvalues.

```python
def compute_U(L_norm, k):
    e_val, e_vec = np.linalg.eig(L_norm)

    non_zero_indices = np.where(np.abs(e_val) > 1e-10)[0]
    e_val = e_val[non_zero_indices]
    e_vec = e_vec[:, non_zero_indices]

    sorted_indices = np.argsort(e_val)
    e_val = e_val[sorted_indices]
    e_vec = e_vec[:, sorted_indices]

    return e_vec[:, :k]
```

(5) the following part is to normalize the U matrix the rows to norm 1 by using the normalized cut.

```python
def normalize_U(U):
    row_norms = np.linalg.norm(U, axis=1, keepdims=True)
    return U / row_norms
```

(6) Final part, clustering:

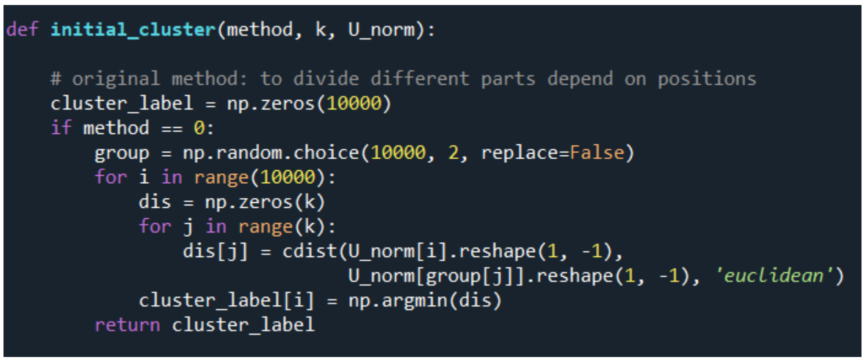


```python
def clustering(U_norm, k, cut, image):

    # initilaization
    token_list = initial_cluster(method, k, U_norm)
    img_res = []
    img_res.append(transfer_img(token_list, image))
    # k-means, iteration = 100
    for i in range(100):
        print("epoch" + str(i))
        token_list_pre = token_list.copy()
        # M step : recalculate the mle
        group = []
        __, count = np.unique(token_list_pre, return_counts=True)
        for j in range(k):
            idx = np.where(token_list_pre == j)
            group.append(np.sum(U_norm[idx], axis=0)/count[j])

        # E-step: assign the group again
        for j in range(10000):
            dis = np.zeros(k)
            for n in range(k):
                dis[n] = cdist(U_norm[j].reshape(1, -1),
                               group[n].reshape(1, -1), 'euclidean')
            token_list[j] = np.argmin(dis)

        img_res.append(transfer_img(token_list, image))
        if (np.sum(token_list != token_list_pre) < 2):
            break
```

Similar to Kmeans, we first initialize the assigned group for each point. (explained later)

Next, based on the initial cluster, I try to recalculate the center (M step) and using the new cluster center to recalculate the distance. Finally, assign each data’s group again (E step).

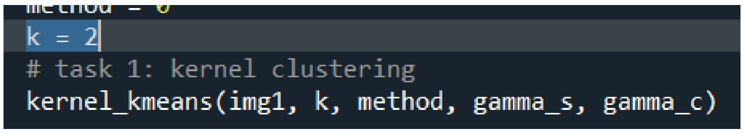
About the initial method, here I randomize k rows from matrix U ,and find out the initial pairwise assigned group.



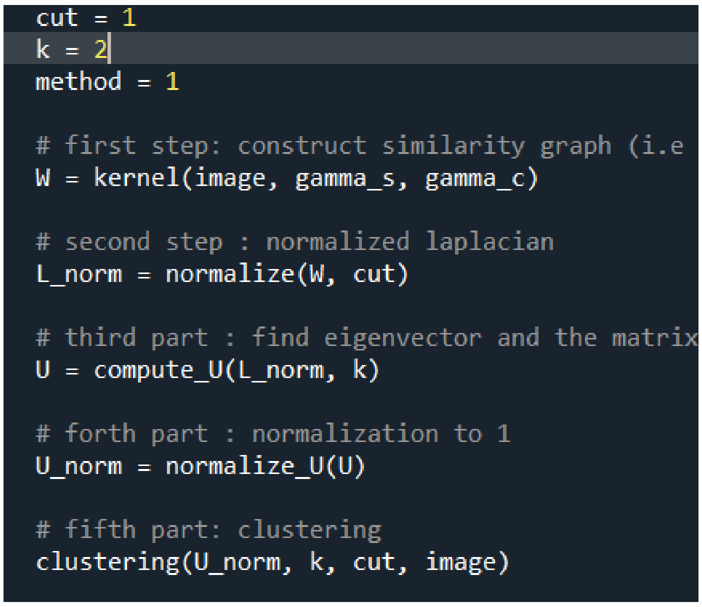
```python
def initial_cluster(method, k, U_norm):

    # original method: to divide different parts depend on positions
    cluster_label = np.zeros(10000)
    if method == 0:
        group = np.random.choice(10000, 2, replace=False)
        for i in range(10000):
            dis = np.zeros(k)
            for j in range(k):
                dis[j] = cdist(U_norm[i].reshape(1, -1),
                               U_norm[group[j]].reshape(1, -1), 'euclidean')
            cluster_label[i] = np.argmin(dis)
    return cluster_label
```

ii. more clusters

To try out more clusters, here just input the k with higher values. Same in Kmeans and Spectral clustering.



```
method = 0
k = 2
# task 1: kernel clustering
kernel_kmeans(img1, k, method, gamma_s, gamma_c)
```

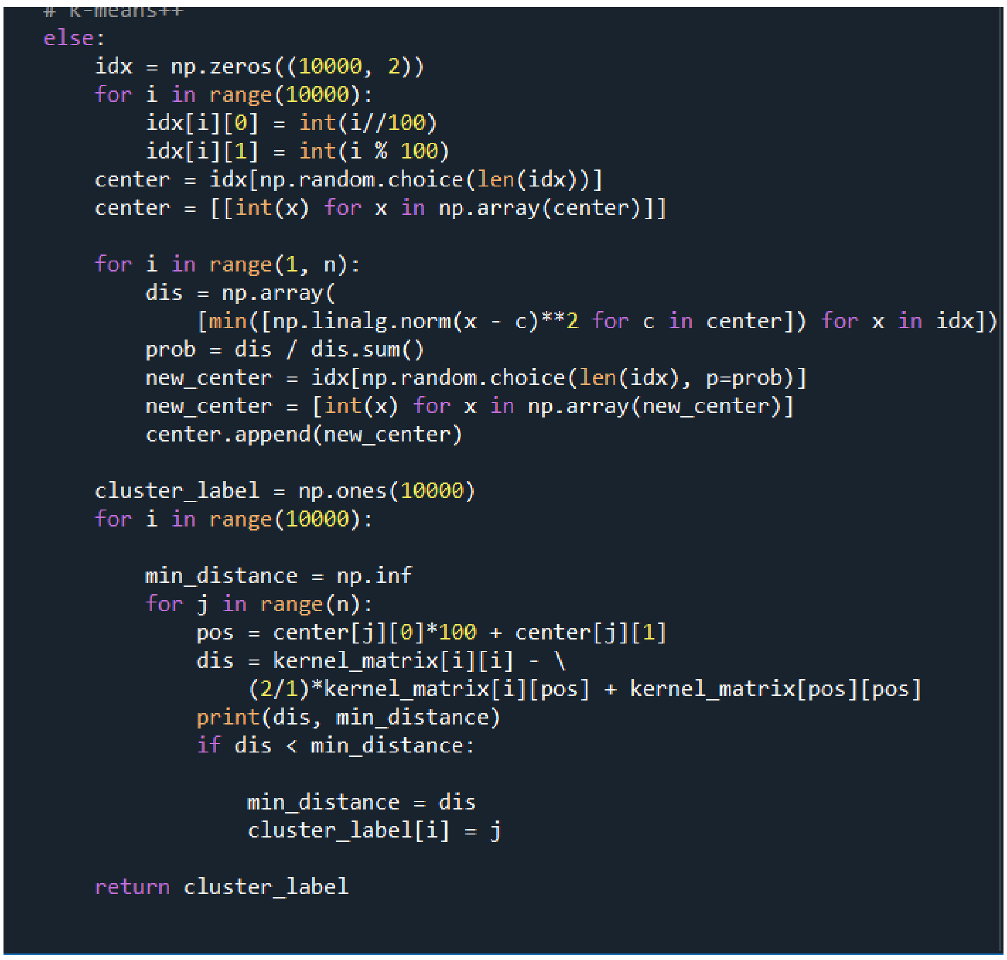


```
cut = 1
k = 2
method = 1

# first step: construct similarity graph (i.e
W = kernel(image, gamma_s, gamma_c)

# second step : normalized laplacian
L_norm = normalize(W, cut)

# third part : find eigenvector and the matrix
U = compute_U(L_norm, k)

# forth part : normalization to 1
U_norm = normalize_U(U)

# fifth part: clustering
clustering(U_norm, k, cut, image)
```

iii. different initializations: the kmeans++

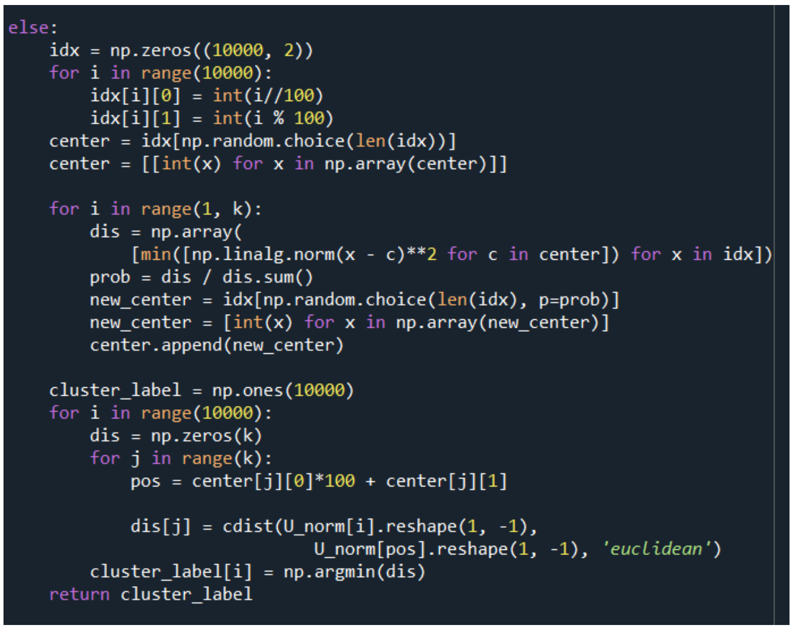
Besides the setting of gamma_s and gamma_c in the kernel calculation, I've tried kmeasns++ to choose the initial cluster center more appropriately. The core idea of kmeans++ is to find a distant center depends on their related space.

In Kernel-Kmeans and spectral clustering, I use the same way:
a. finds the initial cluster
b. finds another central point based on the distance probability distribution
c. until the center numbers meet the number of our cluster numbers.
d. Finally, reassign each data point to the nearest distanced center.



```python
# K-means++
else:
    idx = np.zeros((10000, 2))
    for i in range(10000):
        idx[i][0] = int(i//100)
        idx[i][1] = int(i % 100)
    center = idx[np.random.choice(len(idx))]
    center = [[int(x) for x in np.array(center)]]

    for i in range(1, n):
        dis = np.array(
            [min([np.linalg.norm(x - c)**2 for c in center]) for x in idx])
        prob = dis / dis.sum()
        new_center = idx[np.random.choice(len(idx), p=prob)]
        new_center = [int(x) for x in np.array(new_center)]
        center.append(new_center)

    cluster_label = np.ones(10000)
    for i in range(10000):

        min_distance = np.inf
        for j in range(n):
            pos = center[j][0]*100 + center[j][1]
            dis = kernel_matrix[i][i] - \
                (2/1)*kernel_matrix[i][pos] + kernel_matrix[pos][pos]
            print(dis, min_distance)
            if dis < min_distance:

                min_distance = dis
                cluster_label[i] = j

    return cluster_label
```

// The kmeans++ in Kernel K-means

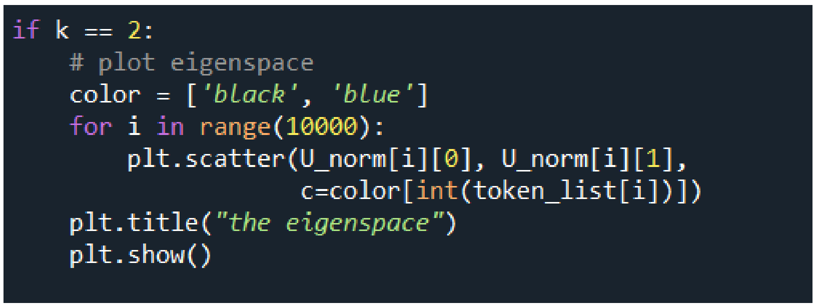


```python
else:
    idx = np.zeros((10000, 2))
    for i in range(10000):
        idx[i][0] = int(i//100)
        idx[i][1] = int(i % 100)
    center = idx[np.random.choice(len(idx))]
    center = [[int(x) for x in np.array(center)]]

    for i in range(1, k):
        dis = np.array(
            [min([np.linalg.norm(x - c)**2 for c in center]) for x in idx])
        prob = dis / dis.sum()
        new_center = idx[np.random.choice(len(idx), p=prob)]
        new_center = [int(x) for x in np.array(new_center)]
        center.append(new_center)

    cluster_label = np.ones(10000)
    for i in range(10000):
        dis = np.zeros(k)
        for j in range(k):
            pos = center[j][0]*100 + center[j][1]

            dis[j] = cdist(U_norm[i].reshape(1, -1),
                           U_norm[pos].reshape(1, -1), 'euclidean')
        cluster_label[i] = np.argmin(dis)
    return cluster_label
```

// kmeans++ in Spectral clustering

iv. Experiments on the coordinates in the eigenspace

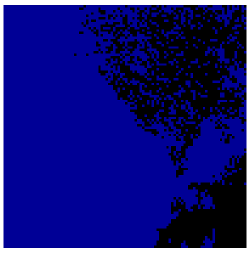
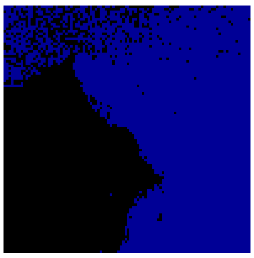
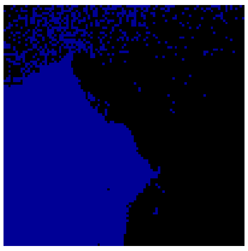
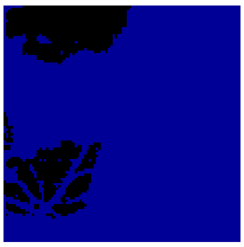
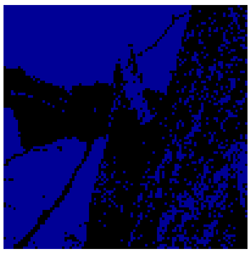
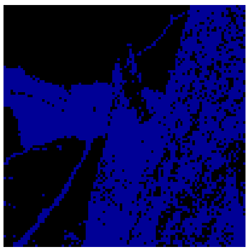
For visualization, when the number of cluster = 2, I plot the eigenspace with different color based on their assigned group.



```python
if k == 2:
    # plot eigenspace
    color = ['black', 'blue']
    for i in range(10000):
        plt.scatter(U_norm[i][0], U_norm[i][1],
                    c=color[int(token_list[i])])
    plt.title("the eigenspace")
    plt.show()
```

2. Experiment setting and results & discussion

i. Part 1 result

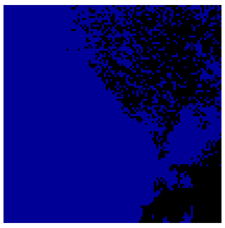
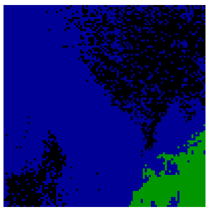
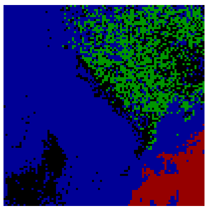
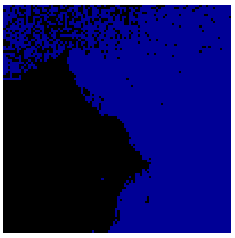
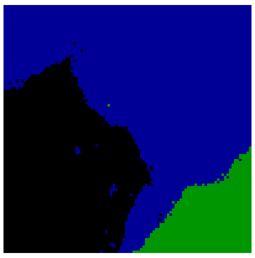
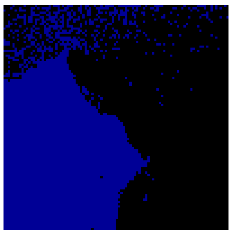
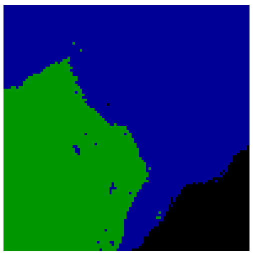
	Kernel K-means	Spectral clustering, normalized cut	Spectral clustering, ratio cut
Image1			
Image2			

For the hyperparameters, here I use gamma_s = 0.001 and gamma_c = 0.01

Also, the initial clusters is assigned by random.

ii. part 2 result

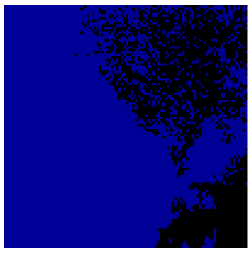
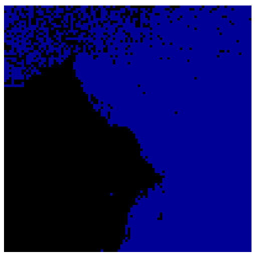
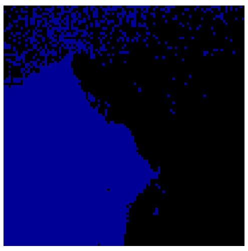
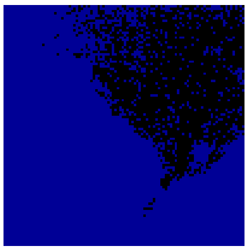
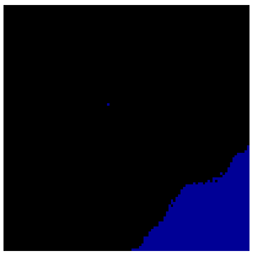
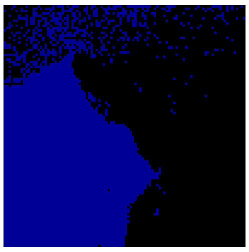
(1) image 1, for different cluster size and the hyperparameter setting is the same as part 1:

	K = 2	K = 3	K = 4
Kernel K-means			
Spectral clustering, normalized cut			Can't show due to Memory space limit (my PC issue)
Spectral clustering, ratio cut			Can't show due to Memory space limit (my PC issue)

iii. part 3 result

Due to memory space limit, here I juust show the difference by using different initialization with some cases.

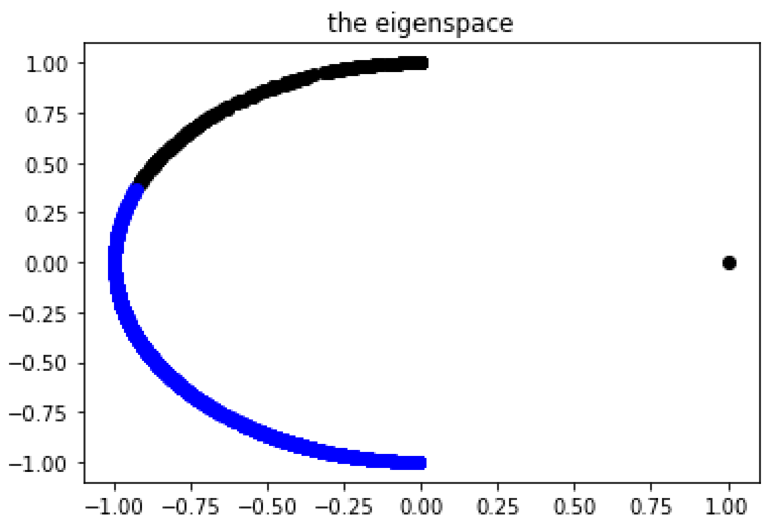
For k = 2, image = image1, and all other parameters remain the same.

	Kernel K-means	Spectral clustering, normalized cut	Spectral clustering, ratio cut
Random			
Kmeans++			

iv. part 4 result

under the condition of k = 2, image = image1, ratio cut, and all other conditions are the same:

the eigenspace is as follows:



where the x-axis is the first column in matrix U and y axis is the second column in matrix U.

3. Observation

i. the difference:

(1) Using the spectral clustering can have a better cluster result (points are separated more apparent)

(2) Also, with the number of k increases, the spectral clustering is better

(3) the normalized cut method seems better than the ratio cut based on the result.

ii. the execution time

(1) using the kmeans++ in the initialization stage, it's faster to execute.(but not necessarily better) Especially on the spectral clustering method, it's more efficient.

(2) For spectral clustering, the computation of matrix U is extremely time-consuming, seems there are more efficient calculations that I didn't take advantage of.

iii. others

(1) I think there are some calculation errors with my eigenvalue / eigenvector calculation because my eigenspace with colors is wrong, they are supposed to be diagonal with separated colors.

(2) It’s quite hard to tune the hyperparameters. Choosing the wrong gamma of RBF kernels may lead to odd results.

(3) This homework really needs a lot of time to execute the final result and the gif. I consider that there must have some more efficient algorithms that I don’t have use it here.