0880803 – 馮清海

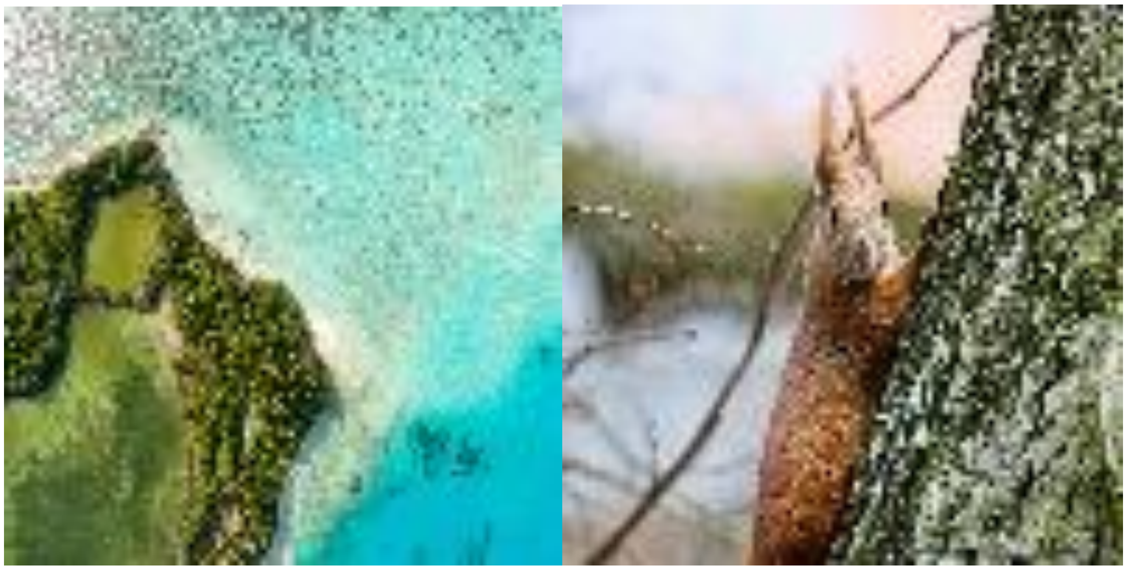
Thanh – Hai, Phung

HOMEWORK 6

MACHINE LEARNING

K Mean Clustering, Spectral Clustering

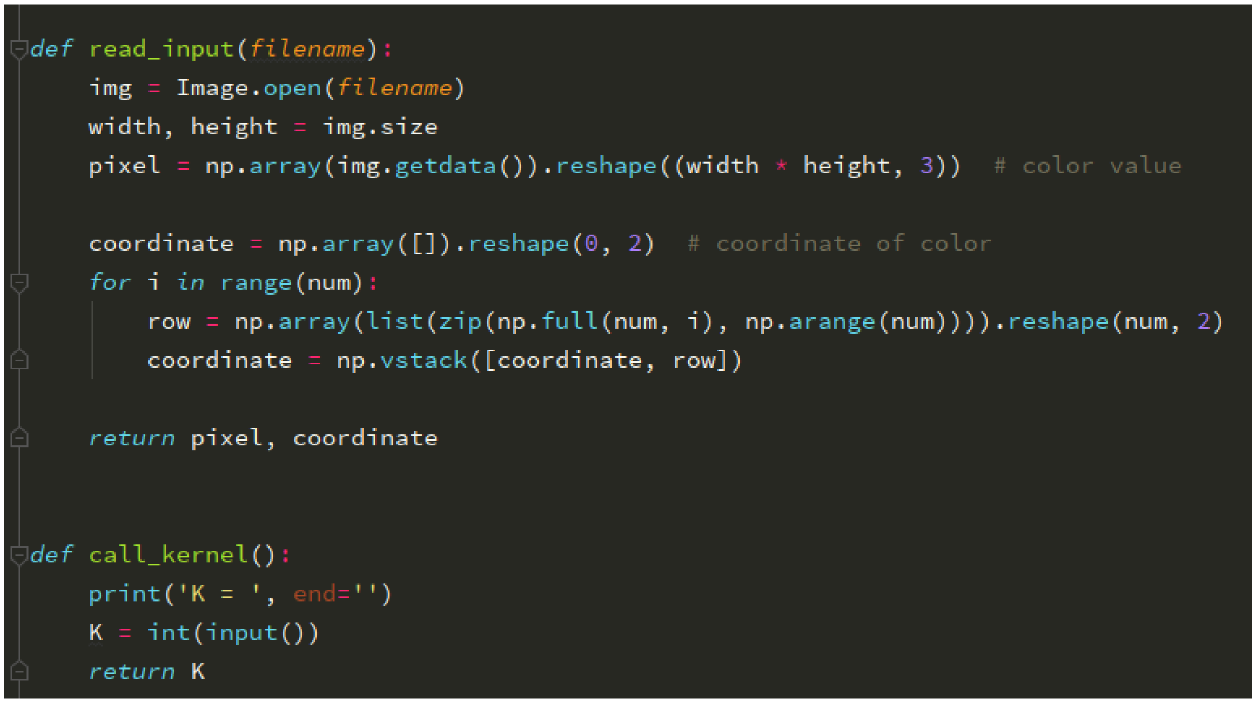
Input image



I. Kernel K mean

Reading input

<code>



```python
def read_input(filename):
    img = Image.open(filename)
    width, height = img.size
    pixel = np.array(img.getdata()).reshape((width * height, 3))  # color value

    coordinate = np.array([]).reshape(0, 2)  # coordinate of color
    for i in range(num):
        row = np.array(list(zip(np.full(num, i), np.arange(num)))).reshape(num, 2)
        coordinate = np.vstack([coordinate, row])

    return pixel, coordinate


def call_kernel():
    print('K = ', end='')
    K = int(input())
    return K
```

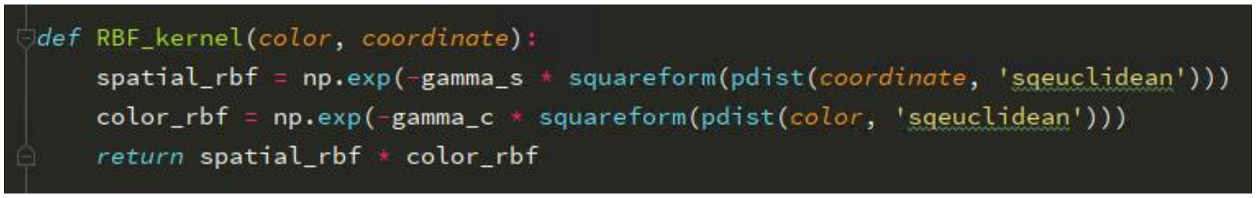
Firstly, we read the input picture using pillow library, separating the coordinate and pixel of the RGB picture, calling the kernel (e.g. 2, 3, 4)

Method

In this algorithm, we implement two RBF kernel to be one to calculate the Gram matrix by the following formula

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

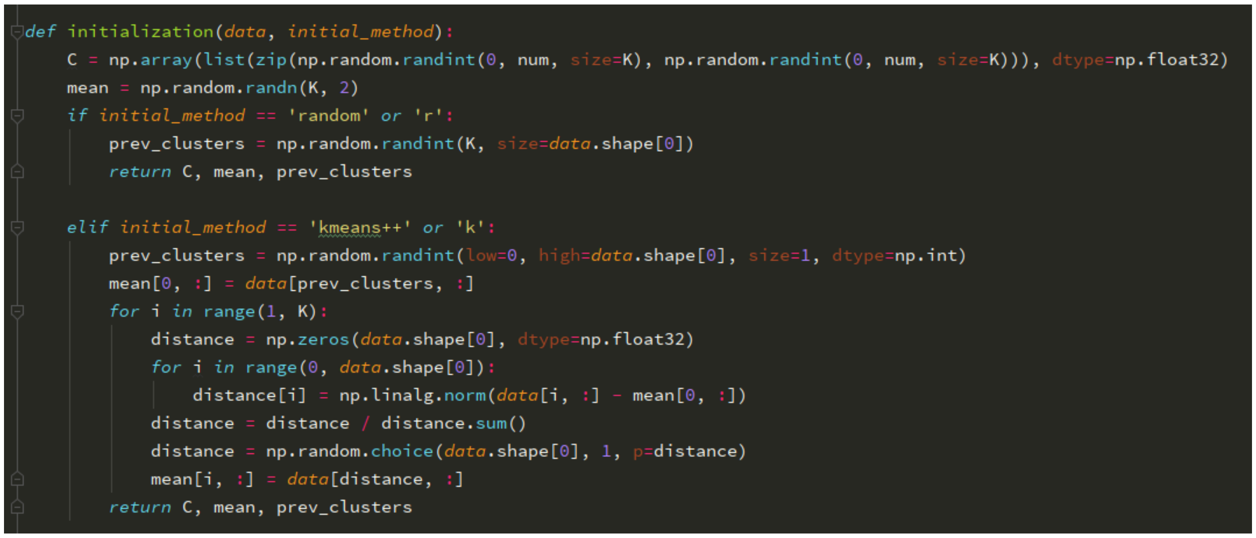
<code>



```python
def RBF_kernel(color, coordinate):
    spatial_rbf = np.exp(-gamma_s * squareform(pdist(coordinate, 'sqeuclidean')))
    color_rbf = np.exp(-gamma_c * squareform(pdist(color, 'sqeuclidean')))
    return spatial_rbf * color_rbf
```

Then following we define the initialization K mean cluster, in this project, we did 2 types of methods which are **RANDOM** and **KMEAN++**.

<code>

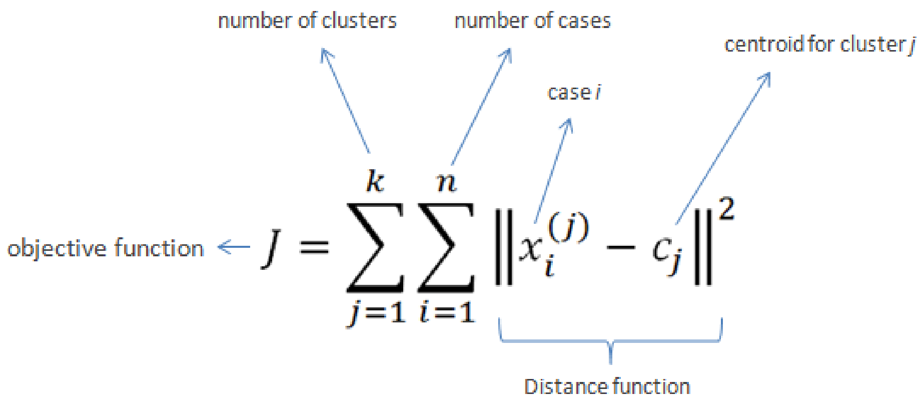


```python
def initialization(data, initial_method):
    C = np.array(list(zip(np.random.randint(0, num, size=K), np.random.randint(0, num, size=K))), dtype=np.float32)
    mean = np.random.randn(K, 2)
    if initial_method == 'random' or 'r':
        prev_clusters = np.random.randint(K, size=data.shape[0])
        return C, mean, prev_clusters

    elif initial_method == 'kmeans++' or 'k':
        prev_clusters = np.random.randint(low=0, high=data.shape[0], size=1, dtype=np.int)
        mean[0, :] = data[prev_clusters, :]
        for i in range(1, K):
            distance = np.zeros(data.shape[0], dtype=np.float32)
            for i in range(0, data.shape[0]):
                distance[i] = np.linalg.norm(data[i, :] - mean[0, :])
            distance = distance / distance.sum()
            distance = np.random.choice(data.shape[0], 1, p=distance)
            mean[i, :] = data[distance, :]
        return C, mean, prev_clusters
```

In the fundamental of Kernel K – Mean Clustering, it said that the K mean ++ has converges faster than other methods by defining the initial cluster instead of **None** as other methods. Let's see if it is True or Not in the following result.

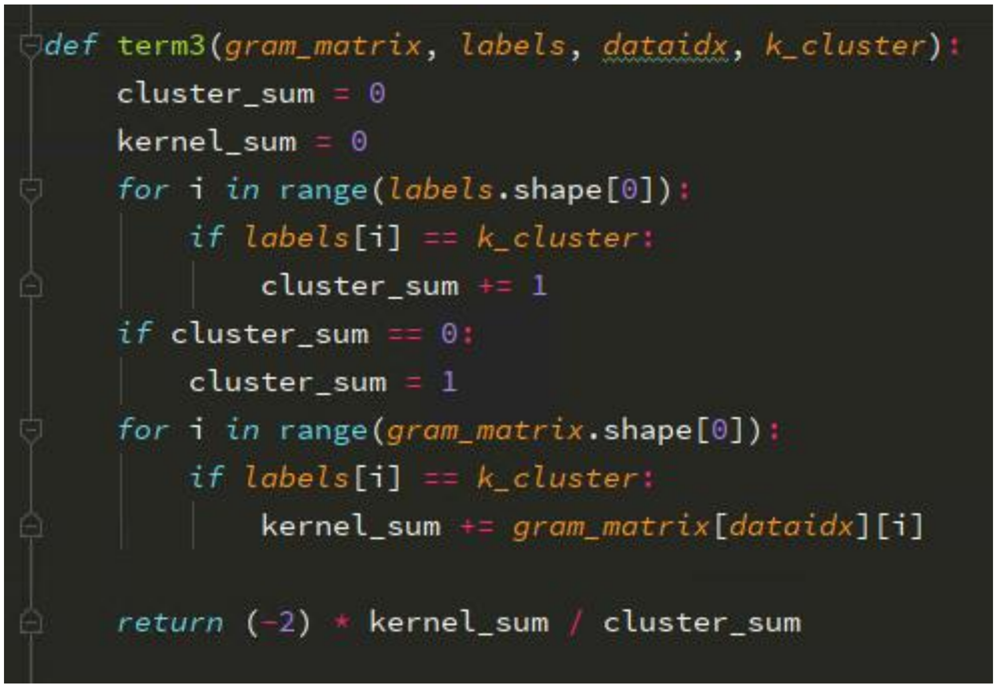
Following we define the two terms of calculating the cluster centers by two step:



$$J = \sum_{j=1}^{k}\sum_{i=1}^{n}\left\|x_i^{(j)} - c_j\right\|^2$$

Step 1: We cluster the data in *K* group where *K* is predefined

<code>



```python
def term3(gram_matrix, labels, dataidx, k_cluster):
    cluster_sum = 0
    kernel_sum = 0
    for i in range(labels.shape[0]):
        if labels[i] == k_cluster:
            cluster_sum += 1
    if cluster_sum == 0:
        cluster_sum = 1
    for i in range(gram_matrix.shape[0]):
        if labels[i] == k_cluster:
            kernel_sum += gram_matrix[dataidx][i]

    return (-2) * kernel_sum / cluster_sum
```

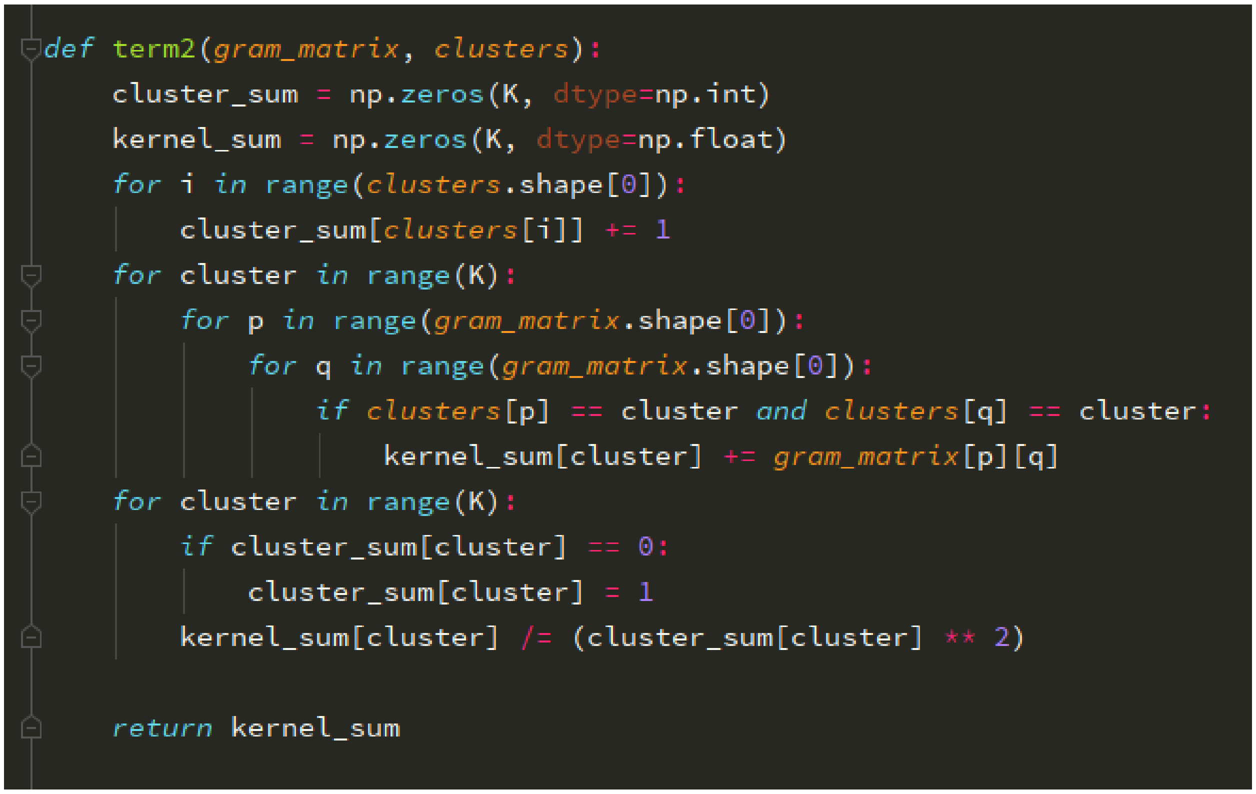
Then we count the number of elements of each cluster by ***labels[i] == k_cluster***

Step 2: Applying the initialization method by choosing random data point in same group. It can be “random” or “kmeans++” as above defined.

Step 3: Assign the data point with their nearest cluster center according to Euclidean distance function.

In this step, we calculate the second term of applied kernel distance

<code>

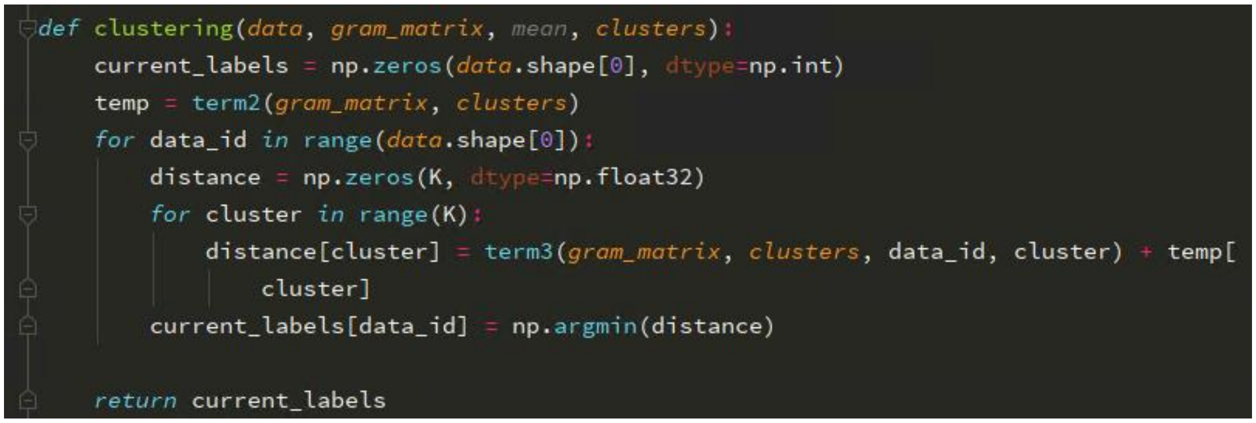


```python
def term2(gram_matrix, clusters):
    cluster_sum = np.zeros(K, dtype=np.int)
    kernel_sum = np.zeros(K, dtype=np.float)
    for i in range(clusters.shape[0]):
        cluster_sum[clusters[i]] += 1
    for cluster in range(K):
        for p in range(gram_matrix.shape[0]):
            for q in range(gram_matrix.shape[0]):
                if clusters[p] == cluster and clusters[q] == cluster:
                    kernel_sum[cluster] += gram_matrix[p][q]
    for cluster in range(K):
        if cluster_sum[cluster] == 0:
            cluster_sum[cluster] = 1
        kernel_sum[cluster] /= (cluster_sum[cluster] ** 2)

    return kernel_sum
```

Step 4: We calculate the centroid (or mean) of the same data point which was assigned in the previous step)

<code>



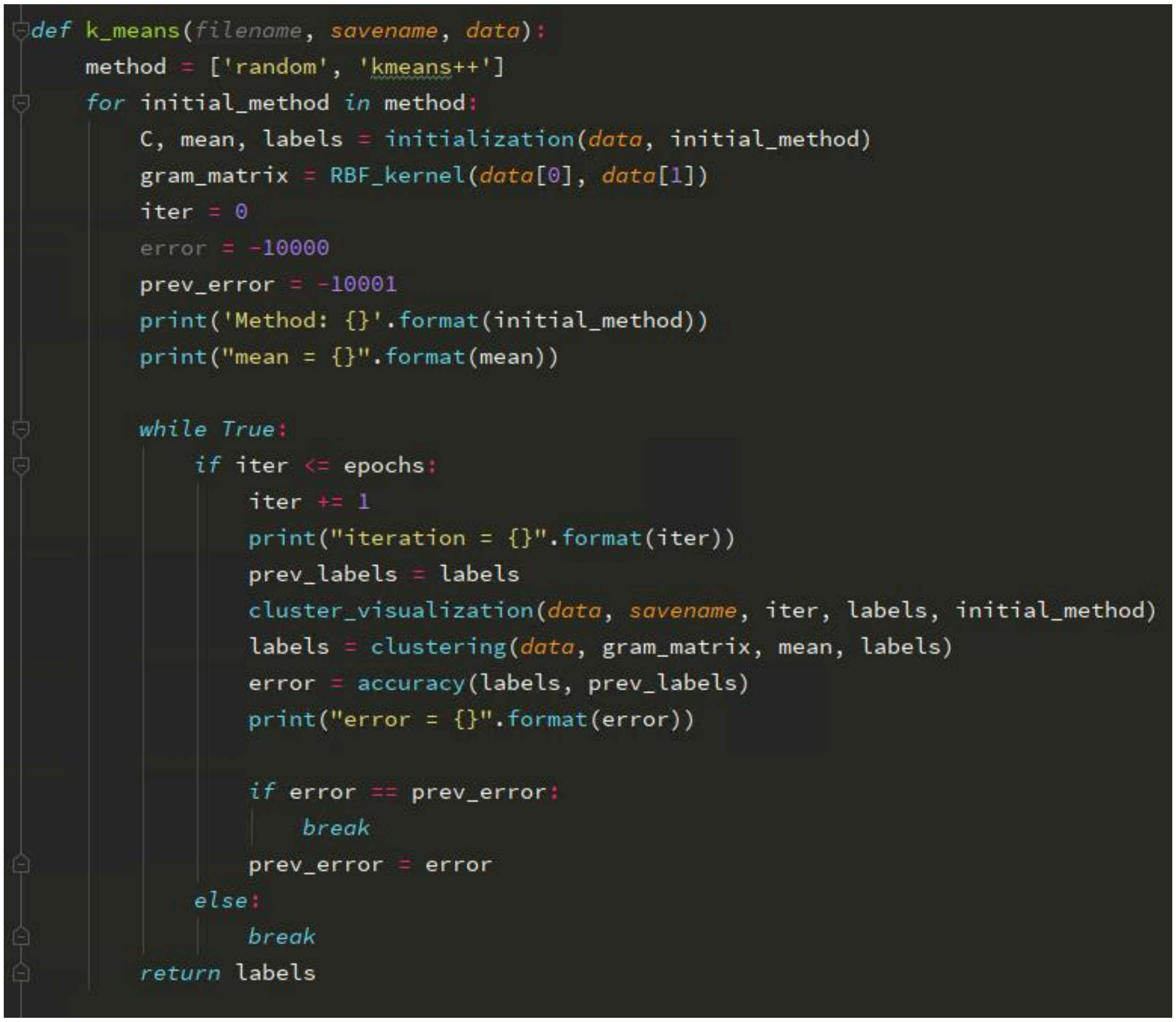
```python
def clustering(data, gram_matrix, mean, clusters):
    current_labels = np.zeros(data.shape[0], dtype=np.int)
    temp = term2(gram_matrix, clusters)
    for data_id in range(data.shape[0]):
        distance = np.zeros(K, dtype=np.float32)
        for cluster in range(K):
            distance[cluster] = term3(gram_matrix, clusters, data_id, cluster) + temp[
                cluster]
        current_labels[data_id] = np.argmin(distance)

    return current_labels
```

Assign the data point to the nearest mean.

Step 5: We repeat the step 2 and 3 until it converges.

<code>



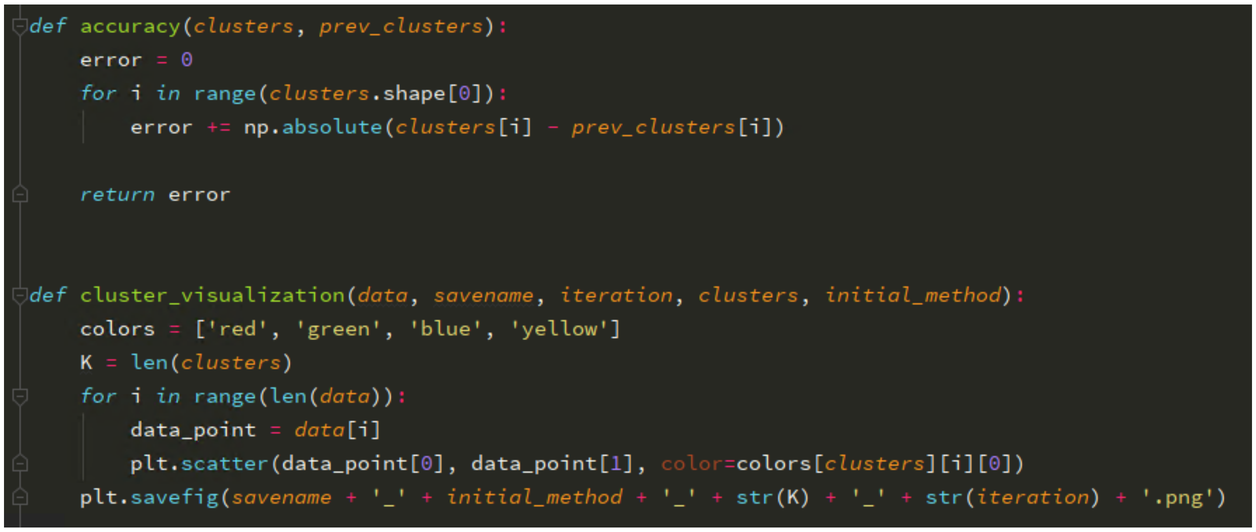
```
def k_means(filename, savename, data):
    method = ['random', 'kmeans++']
    for initial_method in method:
        C, mean, labels = initialization(data, initial_method)
        gram_matrix = RBF_kernel(data[0], data[1])
        iter = 0
        error = -10000
        prev_error = -10001
        print('Method: {}'.format(initial_method))
        print("mean = {}".format(mean))

        while True:
            if iter <= epochs:
                iter += 1
                print("iteration = {}".format(iter))
                prev_labels = labels
                cluster_visualization(data, savename, iter, labels, initial_method)
                labels = clustering(data, gram_matrix, mean, labels)
                error = accuracy(labels, prev_labels)
                print("error = {}".format(error))

                if error == prev_error:
                    break
                prev_error = error
            else:
                break
        return labels
```

Following that we calculate the accuracy of the step assign labels and cluster of the datapoint for observing how and how long it take to converge.

<code>



```
def accuracy(clusters, prev_clusters):
    error = 0
    for i in range(clusters.shape[0]):
        error += np.absolute(clusters[i] - prev_clusters[i])

    return error


def cluster_visualization(data, savename, iteration, clusters, initial_method):
    colors = ['red', 'green', 'blue', 'yellow']
    K = len(clusters)
    for i in range(len(data)):
        data_point = data[i]
        plt.scatter(data_point[0], data_point[1], color=colors[clusters][i][0])
    plt.savefig(savename + '_' + initial_method + '_' + str(K) + '_' + str(iteration) + '.png')
```

Table 1: Result of K – mean Clustering with different method, different kernels.

**Error: Image 1, Image 2

Mode	Random						K mean ++					
Kernel / Iteration	2		3		4		2		3		4	
1	4968	4902	9437	9339	13951	12781	4928	4922	8536	8778	11687	12292
2	446	974	930	1598	1741	1794	912	628	1689	1683	1964	3061
3	120	218	95	734	879	902	176	218	531	842	340	1225
4	63	86	19	331	545	618	28	89	222	582	94	687
5	20	54	7	194	353	402	4	33	174	368	26	443
6	7	46	4	140	284	297	0	9	100	208	9	314
7	2	29	2	109	378	211	0	6	60	101	9	210
8	1	20	1	82	906	135		3	41	69	1	205

9	0	14	0	36	1272	81		3	35	43	0	205
10	0	6		21	539	46			17	35	0	
11		2		13	218	28			10	20		
12		0		15	114	21			11	18		
13				12	59	11			8	8		
14				5	34	8			6	8		
15				3	17	4			0			
16				2	16	3						
17				2	7	0						
18					4	0						
19					2							
20					0							

COMMENT:

- Based on the table above, we may see the changing of distance between the clusters of different data point, the **Random** is said that it converges slower than **Kmean++** method, but in this implementation I didn't see it has much sense. In some case, the Kmean++ have slower converges than Random. May be it because of the input structure don't have a good spherical-like shape. We can easily see it in the table I, where the Random method have better performance and time consuming in Image 1 compare to Image 2, which have more complicate spherical shape.
- One more comment is that it looks likes the Kernel K mean couldn't figure out the clusters correctly, we may have mentioned it in the gif file showing time series of an individual method, from kernel 2 to kernel 4 with different method, different type of images. In other words, data points in smaller clusters may be left away from the centroid itself in order to focus more to the larger clusters.

- Last but not the least, Kmean assumes spherical shapes of clusters which have radius equal to the distance between the centroid and the furthest data point and it didn't work well when clusters are in different shape such as elliptical clusters.

II. Spectral Clustering

In this implementation, the Spectral Clustering is implement same as the K mean method. There are few essential points here make them different:

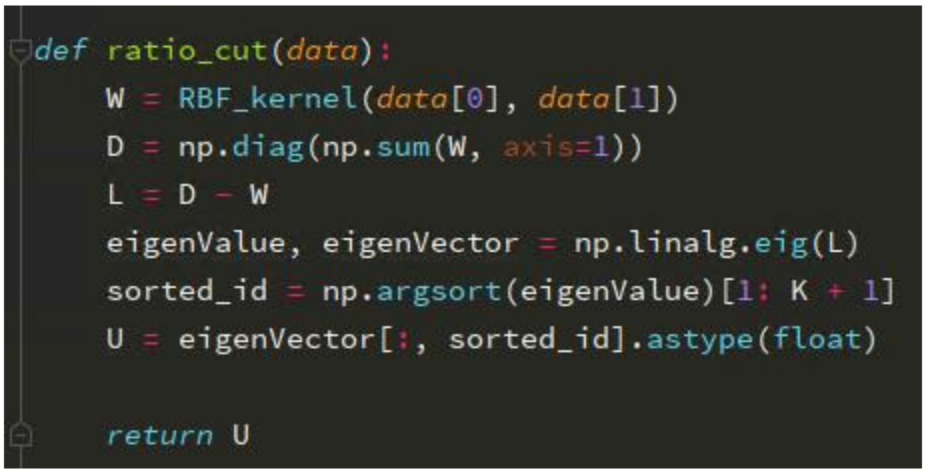
- The K mean clustering divide the objects into k clusters such that some metric relative to the centroids of the clusters is minimized.
- The Spectral clustering make the data points as nodes of a connected graph and clusters are found by partitioning this graph, based on its spectral decomposition into the subgraphs.

To define the Spectral Clustering method, we need to use the Ratio Cut (graph cut) and Normalized Cut to sort the eigenvalue and eigenvector of L (define by the weight and degree)

We can see the different between the Ratio and Normalize here by defining the weight eigenvalue of every data point later.

- Unnormalized Laplacian $\boxed{L = D - W}$ serve in the approximation of the minimization of RatioCut
- Normalized Laplacian $\boxed{D^{-1/2} L D^{-1/2}}$ serve in the approximation of the minimization of NormalizedCut.

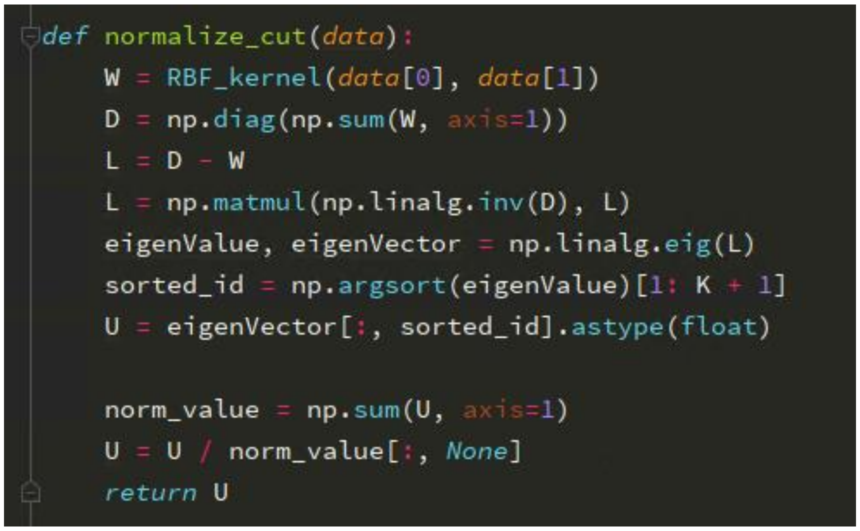
<code>



```
def ratio_cut(data):
    W = RBF_kernel(data[0], data[1])
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    eigenValue, eigenVector = np.linalg.eig(L)
    sorted_id = np.argsort(eigenValue)[1: K + 1]
    U = eigenVector[:, sorted_id].astype(float)

    return U
```

<code>



```python
def normalize_cut(data):
    W = RBF_kernel(data[0], data[1])
    D = np.diag(np.sum(W, axis=1))
    L = D - W
    L = np.matmul(np.linalg.inv(D), L)
    eigenValue, eigenVector = np.linalg.eig(L)
    sorted_id = np.argsort(eigenValue)[1: K + 1]
    U = eigenVector[:, sorted_id].astype(float)

    norm_value = np.sum(U, axis=1)
    U = U / norm_value[:, None]
    return U
```

$$s(A,B)=\sum_{i\in A}\sum_{j\in B} w_{ij}$$

- Ratio Cut

$$J_{Rcut}(A,B) = \frac{s(A,B)}{|A|}+\frac{s(A,B)}{|B|}$$

- Normalized Cut

$$d_A=\sum_{i\in A} d_i$$

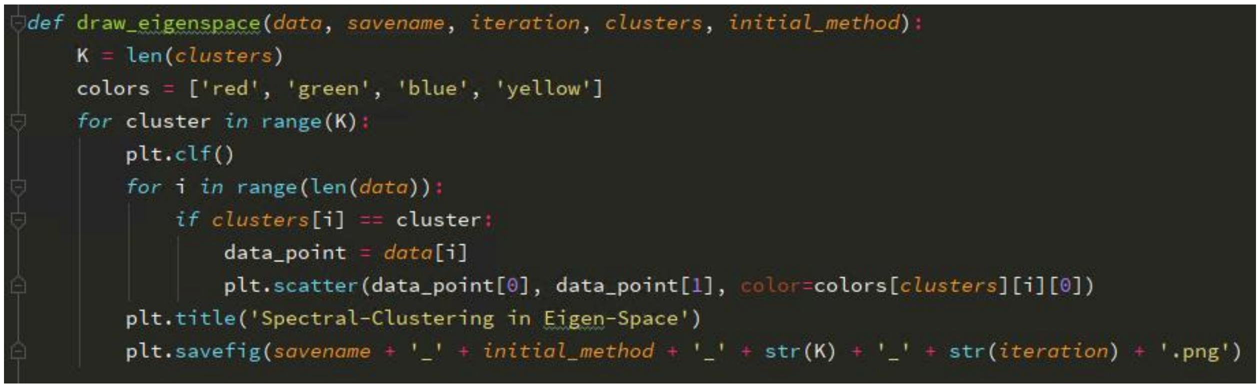
$$J_{Ncut}(A,B) = \frac{s(A,B)}{d_A}+\frac{s(A,B)}{d_B}$$

$$= \frac{s(A,B)}{s(A,A)+s(A,B)}+\frac{s(A,B)}{s(B,B)+s(A,B)}$$

We need to find the Min similar between A and B for both Normalize and Ratio Cut, then we need to calculate the L by the Diagonal Matrix D and L.

And use the matrix U sorted by the eigenvector and eigenid itself to calculate the clustering of the data point input.

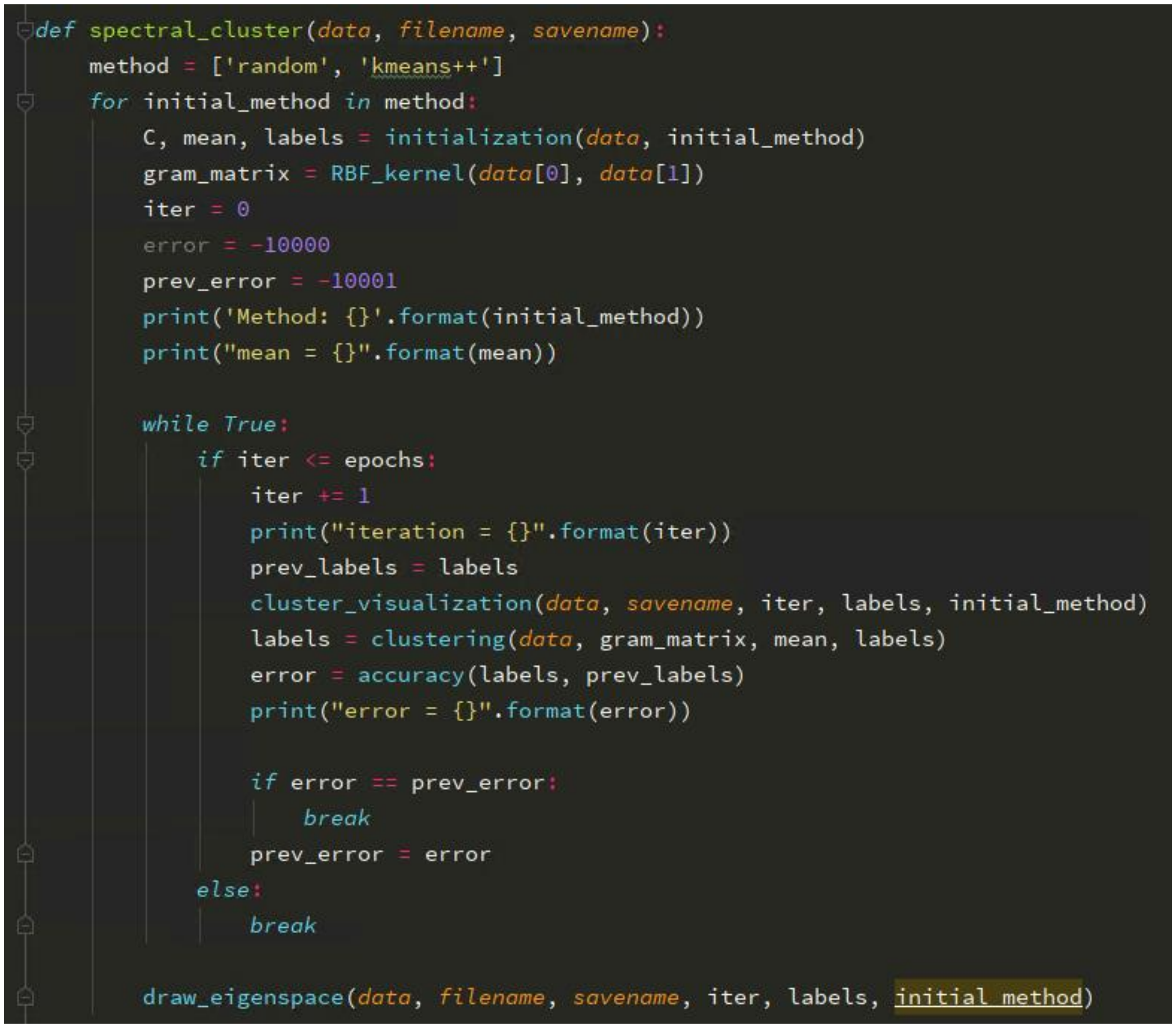
Visualize the eigen space map.



```python
def draw_eigenspace(data, savename, iteration, clusters, initial_method):
    K = len(clusters)
    colors = ['red', 'green', 'blue', 'yellow']
    for cluster in range(K):
        plt.clf()
        for i in range(len(data)):
            if clusters[i] == cluster:
                data_point = data[i]
                plt.scatter(data_point[0], data_point[1], color=colors[clusters][i][0])
        plt.title('Spectral-Clustering in Eigen-Space')
        plt.savefig(savename + '_' + initial_method + '_' + str(K) + '_' + str(iteration) + '.png')
```

Calculate the clustering by repeat the step 2 and 3 by Normalize Cut and Ratio Cut

<code>



```
def spectral_cluster(data, filename, savename):
    method = ['random', 'kmeans++']
    for initial_method in method:
        C, mean, labels = initialization(data, initial_method)
        gram_matrix = RBF_kernel(data[0], data[1])
        iter = 0
        error = -10000
        prev_error = -10001
        print('Method: {}'.format(initial_method))
        print("mean = {}".format(mean))

        while True:
            if iter <= epochs:
                iter += 1
                print("iteration = {}".format(iter))
                prev_labels = labels
                cluster_visualization(data, savename, iter, labels, initial_method)
                labels = clustering(data, gram_matrix, mean, labels)
                error = accuracy(labels, prev_labels)
                print("error = {}".format(error))

                if error == prev_error:
                    break
                prev_error = error
            else:
                break

        draw_eigenspace(data, filename, savename, iter, labels, initial_method)
```

Table 2: Result of Spectra Clustering with different method, different kernels.

**Error: Image 1, Image 2

Mode		Random						K mean ++					
Iteration	Kernel	2		3		4		2		3		4	
1	Normalize Cut	5019	4983	10096	9810	14811	9998	4960	5041	8852	9302	12943	12315
	Ratio Cut	4987	5026	6697	9819	15093	10003	4977	5045	8123	8678	11883	13533
2	Normalize Cut	5026	4126	9958	4260	14832	11056	258	107	738	1719	4169	1851
	Ratio Cut	6396	5098	6396	10196	19188	10186	264	161	867	823	3075	1154
3	Normalize Cut	551	217	1207	3866	2527	2650	172	57	520	1242	1116	784
	Ratio Cut	388	65	2822	3373	3078	3362	109	122	1150	631	1601	958
4	Normalize Cut	198	71	1771	3273	2228	3368	108	34	452	618	187	520
	Ratio Cut	158	54	597	1014	2250	4995	49	114	1179	556	986	763
5	Normalize Cut	46	42	397	1015	1603	2989	44	30	254	256	127	449
	Ratio Cut	71	44	821	358	2311	2654	27	90	964	531	694	754
6	Normalize Cut	10	48	12	294	584	971	17	20	137	148	111	461
	Ratio Cut	35	34	678	180	1372	1578	14	60	764	520	645	709
7	Normalize Cut	2	48	4	127	615	1191	9	18	63	63	101	414

	Ratio Cut	17	33	619	153	773	622	9	32	652	506	616	653
8	Normalize Cut	0	None	0	65	519	1029	6	19	34	33	83	349
	Ratio Cut	10	20	447	127	359	334	3	25	518	401	554	554
9	Normalize Cut	0	None	0	41	388	846	3	8	15	21	46	247
	Ratio Cut	4	17	328	113	367	226	1	24	452	337	524	486
10	Normalize Cut	None	None	None	26	319	805	3	4	6	18	36	184
	Ratio Cut	1	11	195	66	335	172	0	13	349	304	500	440
11	Normalize Cut	None	None	None	10	252	864	None	1	6	5	17	107
	Ratio Cut	0	9	147	42	343	169	0	13	248	234	478	411
12	Normalize Cut	None	None	None	10	189	960		1	None	5	17	77
	Ratio Cut	0	7	93	34	348	163		None	155	210	436	340
13	Normalize Cut		None	None	None	138	988			None	None	None	66
	Ratio Cut		4	53	23	343	156			98	153	387	294
14	Normalize Cut		None	None	None	88	631			None	None	None	22
	Ratio Cut		1	40	9	348	110			64	92	344	249
15	Normalize Cut		None	None	None	65	368			None	None	None	9
	Ratio Cut		0	39	4	368	87			41	53	312	205

16	Normalize Cut			None	None	31	178			None	None	None	9
	Ratio Cut			27	3	386	67			29	41	319	208
17	Normalize Cut			None	None	27	50			None	None	None	None
	Ratio Cut			16	4	370	41			24	30	325	175
18	Normalize Cut			None	None	14	26			None	None	None	None
	Ratio Cut			18	2	348	43			22	14	317	131
19	Normalize Cut			None	None	7	5			None	None	None	None
	Ratio Cut			6	2	313	51			11	6	309	109
20	Normalize Cut			None		2	0			None	None		None
	Ratio Cut			3		290	37			13	3		88
21	Normalize Cut			None		0	0			None	None		None
	Ratio Cut			1		263	36			4	6		68

COMMENT:

- There are two methods for calculate the L and L_{sym} as kernel matrix. The success of spectral clustering compare to K mean is mainly based on the fact that it does not make strong assumptions on the form of the clusters. The method of convex sets to be precise.
- It is not trivial to have good clusters of data point, but it can be separate the two data point of kernel instead of mixing it as Kmean. We can easily to see that the Spectral separate so well the neighbors of different clusters.

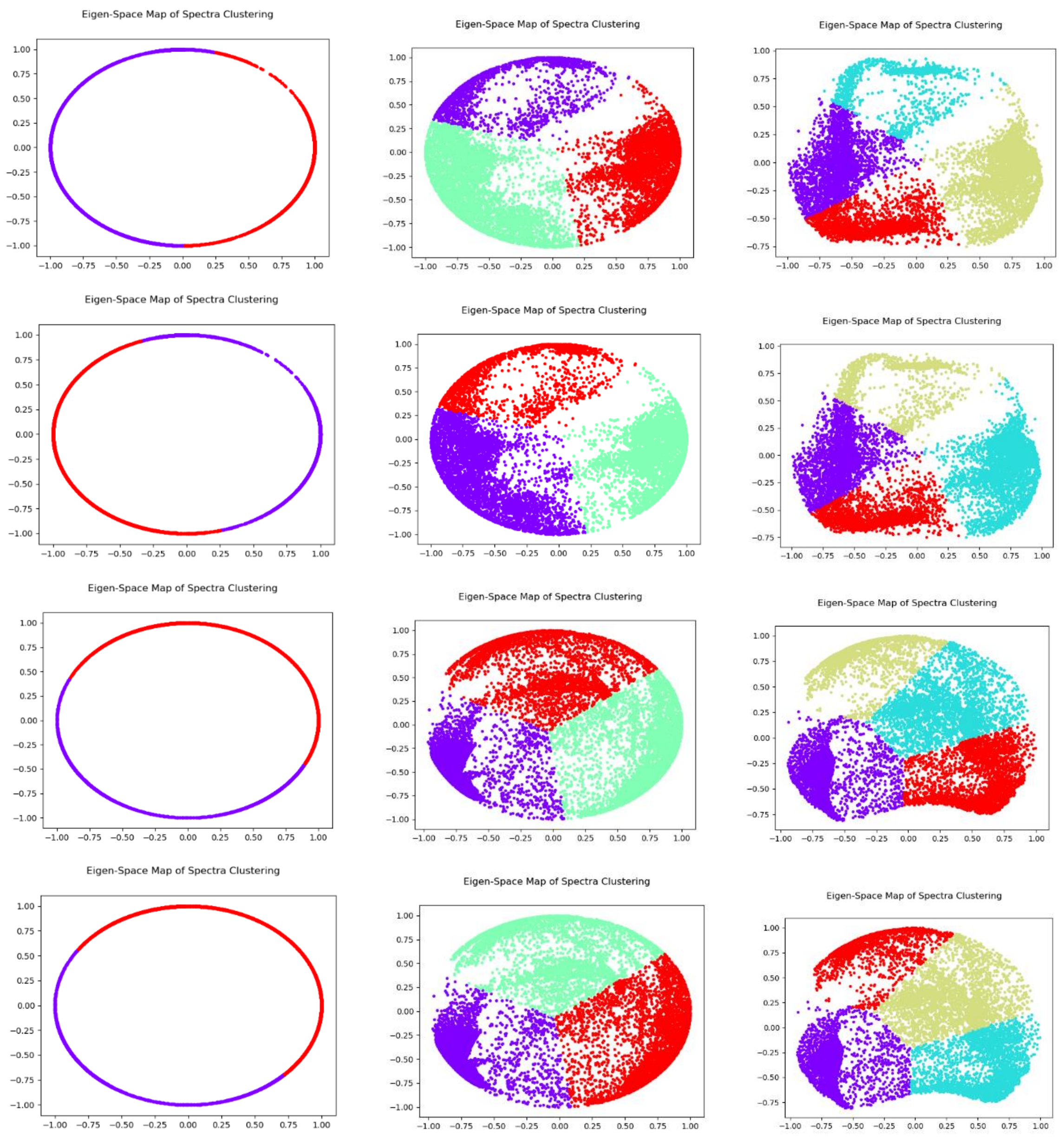


Figure 1: Eigen-space of Cluster point by Normalize Cut. From left to right, from top to bottom.

Image 1 Normalize Cut with 2 – 3 – 4 cluster point in Kmean++ method,

Image 1 Normalize Cut with 2 – 3 – 4 cluster point in Random method,

Image 2 Normalize Cut with 2 – 3 – 4 cluster point in Kmean++ method,

Image 2 Normalize Cut with 2 – 3 – 4 cluster point in Random method,

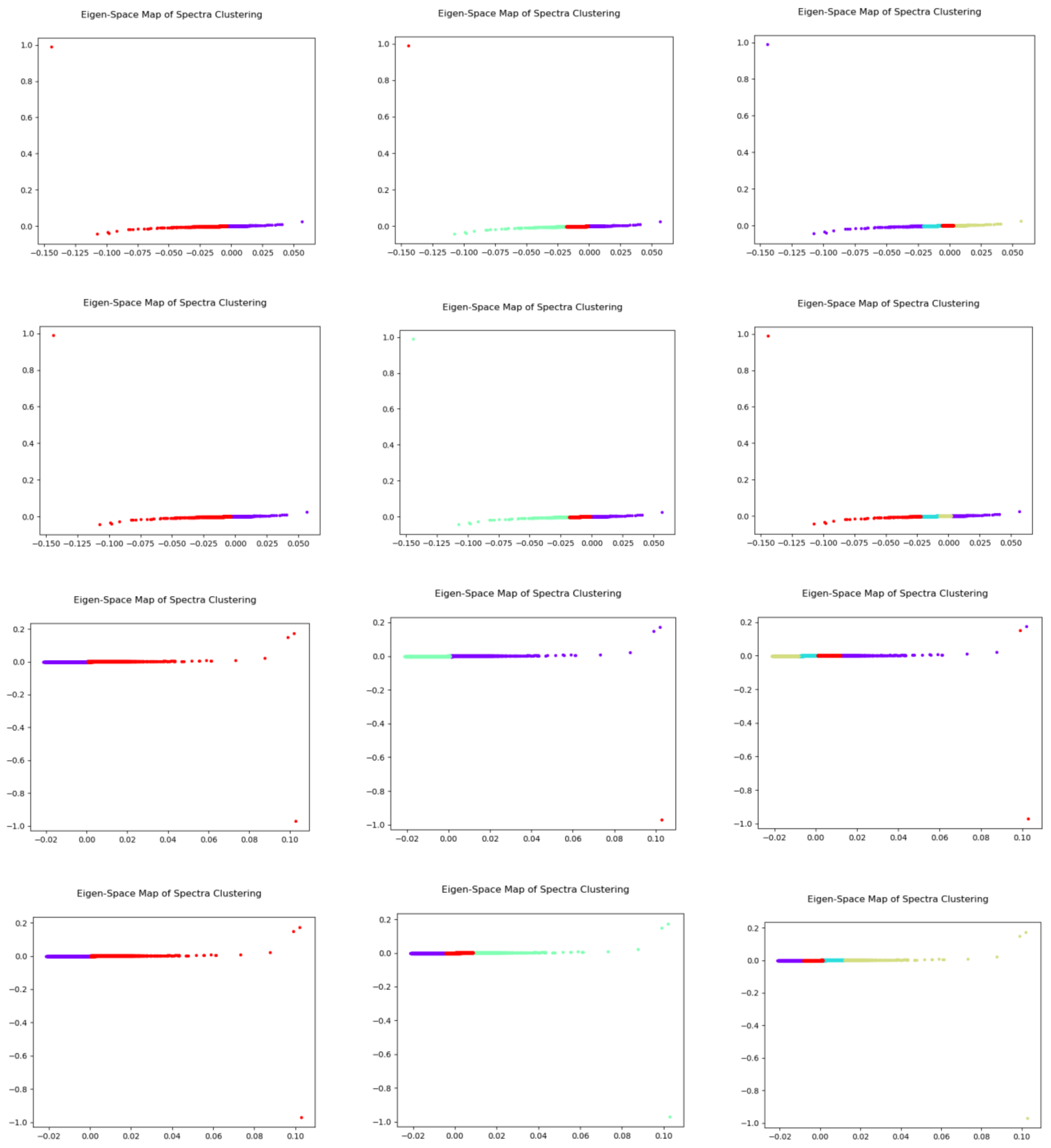


Figure 1: Eigen-space of Cluster point by Normalize Cut. From left to right, from top to bottom.

Image 1 Normalize Cut with 2 – 3 – 4 cluster point in Kmean++ method,

Image 1 Normalize Cut with 2 – 3 – 4 cluster point in Random method,

Image 2 Normalize Cut with 2 – 3 – 4 cluster point in Kmean++ method,

Image 2 Normalize Cut with 2 – 3 – 4 cluster point in Random method, If you have any concern about my code or other question don't hesitate to contact me!

My email is: *haiphung106@gmail.com*