# Kernel Kmeans

0516312 陳皓婷

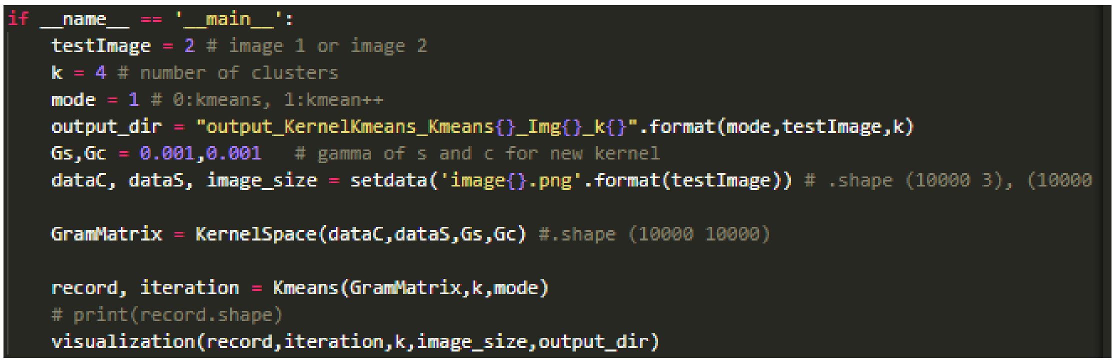
***<In main function>***

First, set the parameters (testImage, k, mode, gamma S and gamma C for kernel).

Then, use *setdata* function to get the point data from the image.

Use *KernelSpace* function to get GramMatrix and *Kmeans* function to updata the centers.

Finally, use *visualization* to show the result.



```python
if __name__ == '__main__':
    testImage = 2 # image 1 or image 2
    k = 4 # number of clusters
    mode = 1 # 0:kmeans, 1:kmean++
    output_dir = "output_KernelKmeans_Kmeans{}_Img{}_k{}".format(mode,testImage,k)
    Gs,Gc = 0.001,0.001   # gamma of s and c for new kernel
    dataC, dataS, image_size = setdata('image{}.png'.format(testImage)) # .shape (10000 3), (10000

    GramMatrix = KernelSpace(dataC,dataS,Gs,Gc) #.shape (10000 10000)

    record, iteration = Kmeans(GramMatrix,k,mode)
    # print(record.shape)
    visualization(record,iteration,k,image_size,output_dir)
```

***<setdata>***

Use Image library to read to color of pixels and named as dataC.

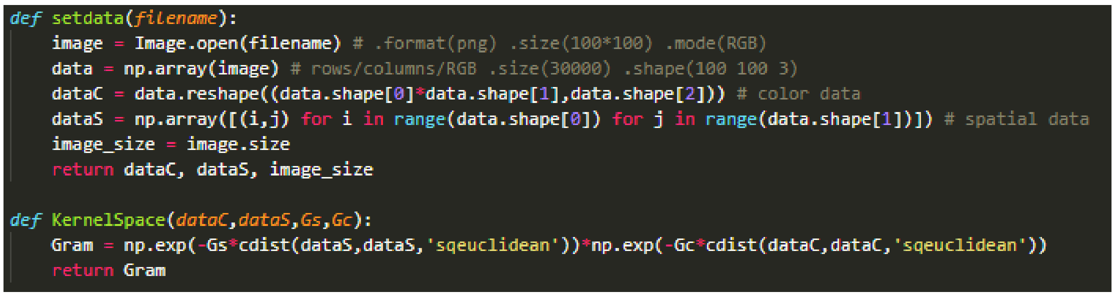
Then, set dataS as the spatial information of pixels.

Finally, return dataC, data and the size of the image.

***<KernelSpace>***

Use the data from *setdata* to compute GramMatrix

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



```python
def setdata(filename):
    image = Image.open(filename) # .format(png) .size(100*100) .mode(RGB)
    data = np.array(image) # rows/columns/RGB .size(30000) .shape(100 100 3)
    dataC = data.reshape((data.shape[0]*data.shape[1],data.shape[2])) # color data
    dataS = np.array([(i,j) for i in range(data.shape[0]) for j in range(data.shape[1])]) # spatial data
    image_size = image.size
    return dataC, dataS, image_size

def KernelSpace(dataC,dataS,Gs,Gc):
    Gram = np.exp(-Gs*cdist(dataS,dataS,'sqeuclidean'))*np.exp(-Gc*cdist(dataC,dataC,'sqeuclidean'))
    return Gram
```

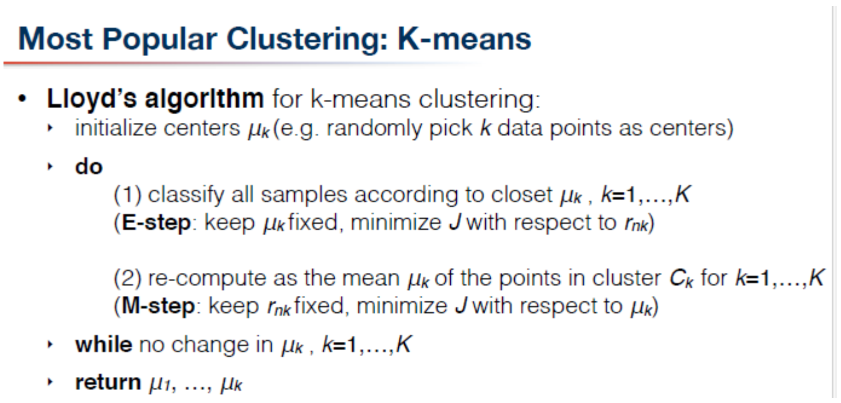
***<Kmeans>***

Call *firstMean* function to get the first mean

Then do E-step and M-step sequentially in a loop until result convergence.

E-step: distribute the points to corresponding cluster by choosing the min- distance one.

M-step: update the centers by computing the new means



## Most Popular Clustering: K-means

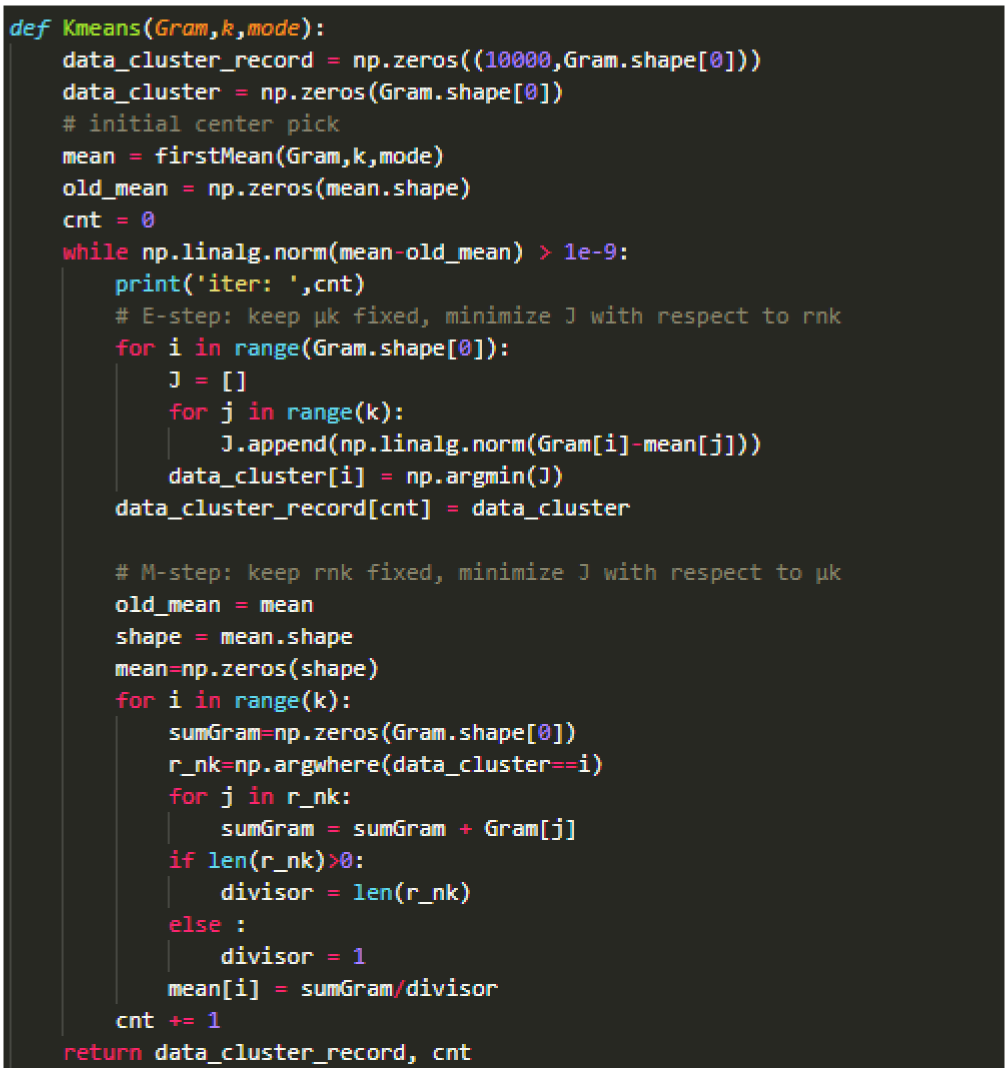
- **Lloyd's algorithm** for k-means clustering:
  - initialize centers $\mu_k$ (e.g. randomly pick $k$ data points as centers)
  - **do**

    (1) classify all samples according to closet $\mu_k$ , $k$=1,…,$K$
    (**E-step**: keep $\mu_k$ fixed, minimize $J$ with respect to $r_{nk}$)

    (2) re-compute as the mean $\mu_k$ of the points in cluster $C_k$ for $k$=1,…,$K$
    (**M-step**: keep $r_{nk}$ fixed, minimize $J$ with respect to $\mu_k$)
  - **while** no change in $\mu_k$ , $k$=1,…,$K$
  - **return** $\mu_1$, …, $\mu_k$

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|x_n - \mu_k\|^2 \qquad r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_k \|x_n - \mu_k\| \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

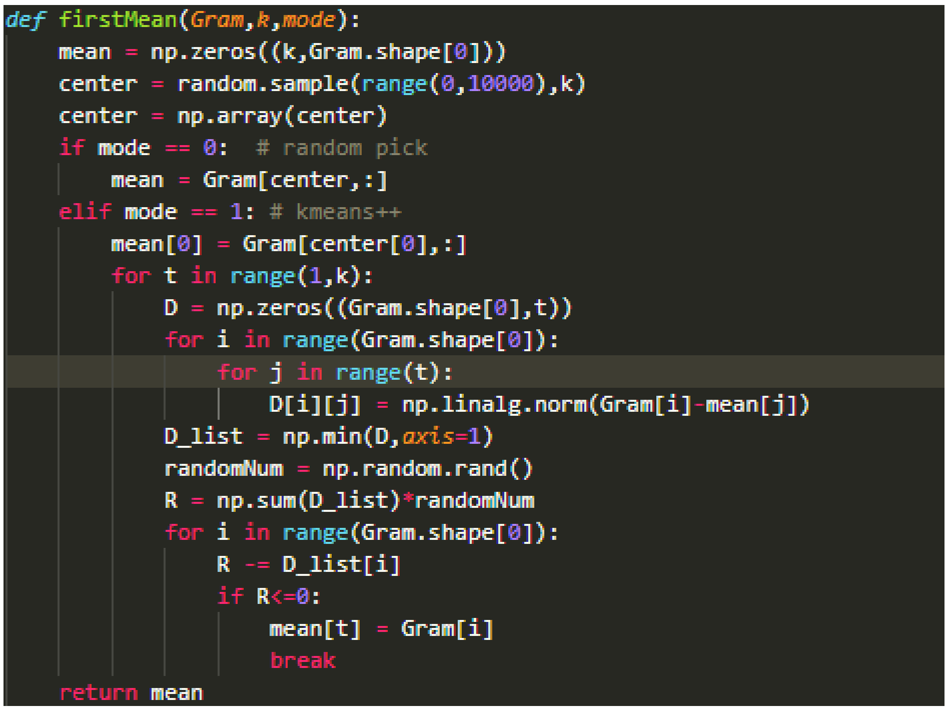


```python
def Kmeans(Gram,k,mode):
    data_cluster_record = np.zeros((10000,Gram.shape[0]))
    data_cluster = np.zeros(Gram.shape[0])
    # initial center pick
    mean = firstMean(Gram,k,mode)
    old_mean = np.zeros(mean.shape)
    cnt = 0
    while np.linalg.norm(mean-old_mean) > 1e-9:
        print('iter: ',cnt)
        # E-step: keep μk fixed, minimize J with respect to rnk
        for i in range(Gram.shape[0]):
            J = []
            for j in range(k):
                J.append(np.linalg.norm(Gram[i]-mean[j]))
            data_cluster[i] = np.argmin(J)
        data_cluster_record[cnt] = data_cluster

        # M-step: keep rnk fixed, minimize J with respect to μk
        old_mean = mean
        shape = mean.shape
        mean=np.zeros(shape)
        for i in range(k):
            sumGram=np.zeros(Gram.shape[0])
            r_nk=np.argwhere(data_cluster==i)
            for j in r_nk:
                sumGram = sumGram + Gram[j]
            if len(r_nk)>0:
                divisor = len(r_nk)
            else :
                divisor = 1
            mean[i] = sumGram/divisor
        cnt += 1
    return data_cluster_record, cnt
```

***<firstMean>***

The ways to choose the first set of means, one is random pick the point, and another is use the Kmeans++ way to get the means.

How the Kmeans++ way perform (ref): https://kknews.cc/zh-tw/code/b4axoe6.html

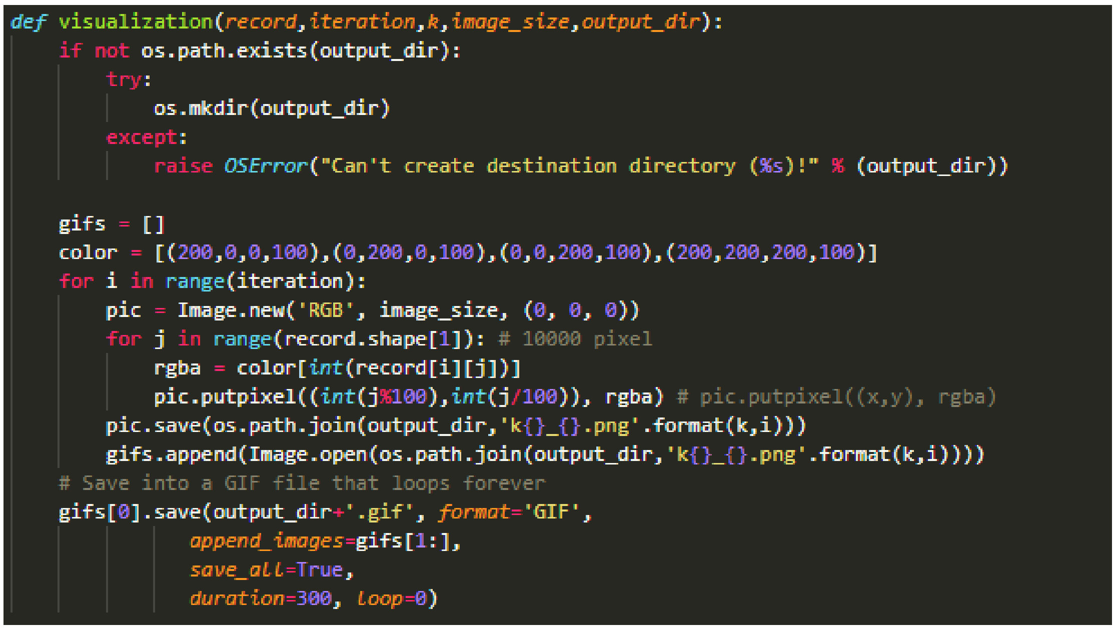


```python
def firstMean(Gram,k,mode):
    mean = np.zeros((k,Gram.shape[0]))
    center = random.sample(range(0,10000),k)
    center = np.array(center)
    if mode == 0:  # random pick
        mean = Gram[center,:]
    elif mode == 1: # kmeans++
        mean[0] = Gram[center[0],:]
        for t in range(1,k):
            D = np.zeros((Gram.shape[0],t))
            for i in range(Gram.shape[0]):
                for j in range(t):
                    D[i][j] = np.linalg.norm(Gram[i]-mean[j])
            D_list = np.min(D,axis=1)
            randomNum = np.random.rand()
            R = np.sum(D_list)*randomNum
            for i in range(Gram.shape[0]):
                R -= D_list[i]
                if R<=0:
                    mean[t] = Gram[i]
                    break
    return mean
```

***<visualization>***

Visualize the results by draw the points belongs to different cluster in different color and make the processing into GIF file.

The GIF file name is according to use kmean++or not, the image number and how many clusters to divid.

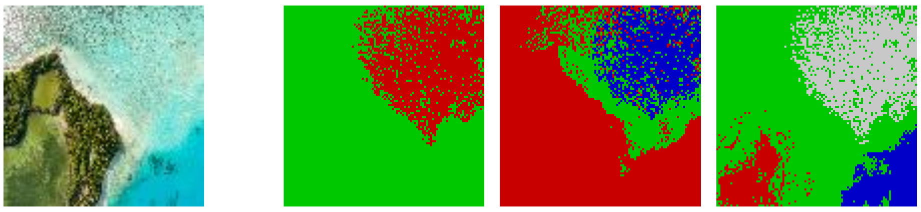


```python
def visualization(record,iteration,k,image_size,output_dir):
    if not os.path.exists(output_dir):
        try:
            os.mkdir(output_dir)
        except:
            raise OSError("Can't create destination directory (%s)!" % (output_dir))

    gifs = []
    color = [(200,0,0,100),(0,200,0,100),(0,0,200,100),(200,200,200,100)]
    for i in range(iteration):
        pic = Image.new('RGB', image_size, (0, 0, 0))
        for j in range(record.shape[1]): # 10000 pixel
            rgba = color[int(record[i][j])]
            pic.putpixel((int(j%100),int(j/100)), rgba) # pic.putpixel((x,y), rgba)
        pic.save(os.path.join(output_dir,'k{}_{}.png'.format(k,i)))
        gifs.append(Image.open(os.path.join(output_dir,'k{}_{}.png'.format(k,i))))
    # Save into a GIF file that loops forever
    gifs[0].save(output_dir+'.gif', format='GIF',
                append_images=gifs[1:],
                save_all=True,
                duration=300, loop=0)
```

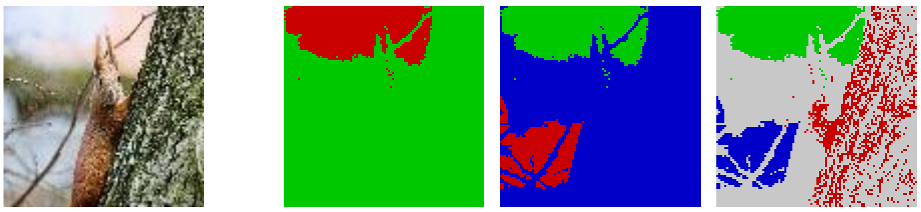
Result:

Kernel Kmeans with random pick Kmeans

Origin => k=2 k=3 k=4

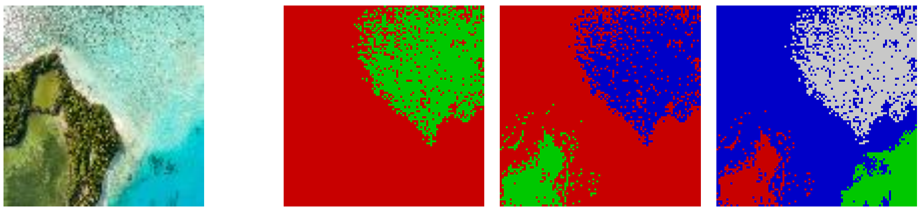


Origin => k=2 k=3 k=4

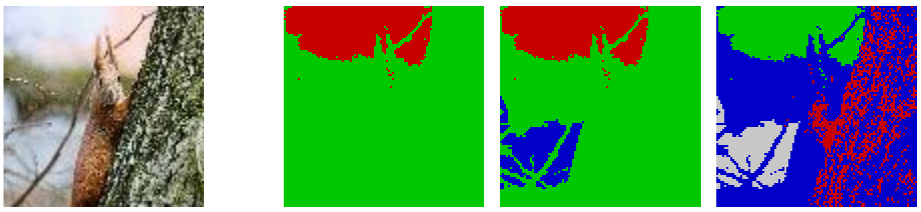


Kernel Kmeans with Kmeans++

Origin => k=2 k=3 k=4



Origin => k=2 k=3 k=4



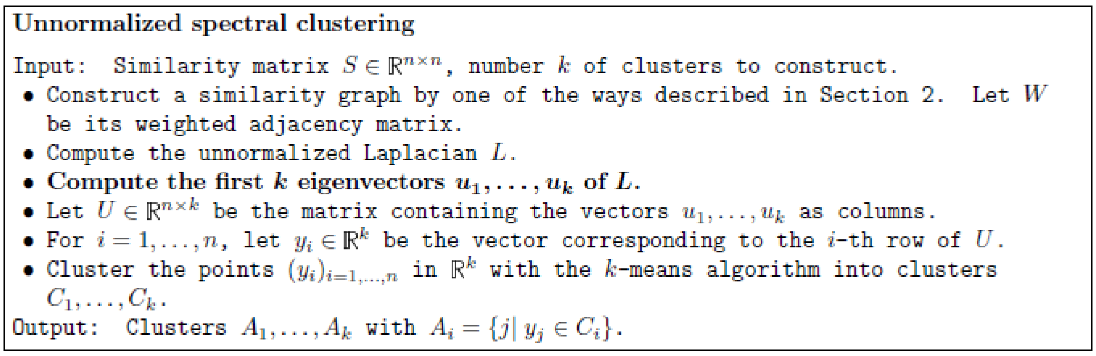
Observation:

The outputs of kmeans++ and random pick kmeans are almost look like same, but Kmean++ is better because it can make the first means not too close, which makes the different between the output of Kmeans++_Image1_k3 and the output of random_pick_kmeans_Image1_k3.

Using new kernel of mix the RBF with color data and the RBF with space data, which enables us to consider both the color feature and space feature

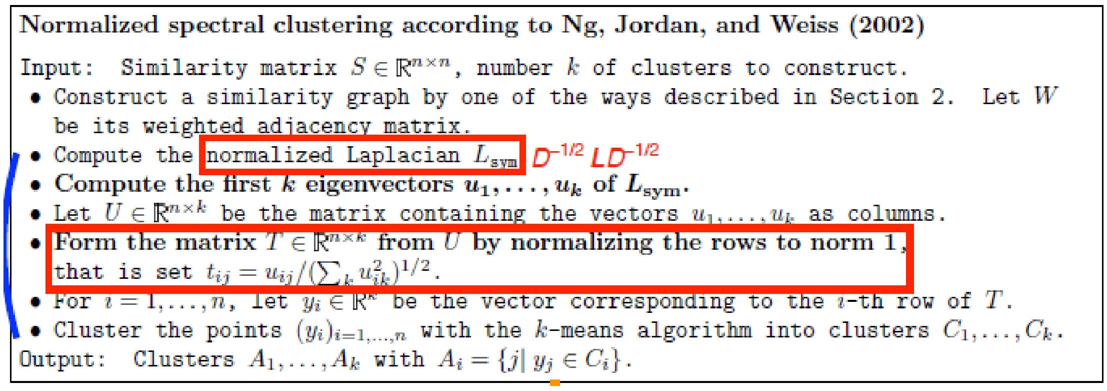
## Spectral Clustering

Use the algorithm of Unnormalized spectral clustering and the algorithm of Normalized spectral clustering to compute the clustering to get the result



**Unnormalized spectral clustering**

```
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 unnormalized Laplacian $L$.
  • Compute the first $k$ eigenvectors $u_1, \dots, u_k$ of $L$.
  • Let $U \in \mathbb{R}^{n\times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ as columns.
  • For $i = 1, \dots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the $i$-th row of $U$.
  • Cluster the points $(y_i)_{i=1,\dots,n}$ in $\mathbb{R}^k$ with the $k$-means algorithm into clusters
    $C_1, \dots, C_k$.
Output: Clusters $A_1, \dots, A_k$ with $A_i = \{j \mid y_j \in C_i\}$.
```



**Normalized spectral clustering according to Ng, 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 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 $A_i = \{j \mid y_j \in C_i\}$.
```

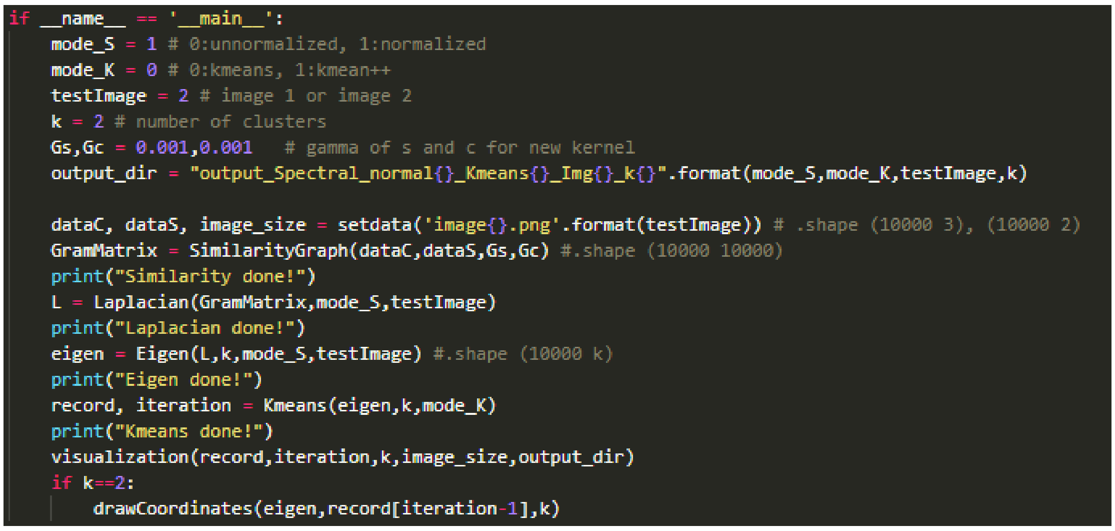
***<In main function>***

First, set the parameters (mode_S, mode_K, testImage, k, Gs and Gc for kernel).

Then, use *setdata* function to get the point data from the image.

Use *SimilarityGraph* function to get Similarity matrix, *Laplacian* function to comput the Laplacian L, *Eigen* function to get eigenvectors information and *Kmeans* function to updata the centers.

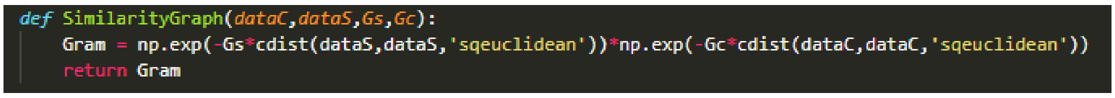
Finally, use *visualization* and *drawCoordinates* to show the result.



```python
if __name__ == '__main__':
    mode_S = 1 # 0:unnormalized, 1:normalized
    mode_K = 0 # 0:kmeans, 1:kmean++
    testImage = 2 # image 1 or image 2
    k = 2 # number of clusters
    Gs,Gc = 0.001,0.001    # gamma of s and c for new kernel
    output_dir = "output_Spectral_normal{}_Kmeans{}_Img{}_k{}".format(mode_S,mode_K,testImage,k)

    dataC, dataS, image_size = setdata('image{}.png'.format(testImage)) # .shape (10000 3), (10000 2)
    GramMatrix = SimilarityGraph(dataC,dataS,Gs,Gc) #.shape (10000 10000)
    print("Similarity done!")
    L = Laplacian(GramMatrix,mode_S,testImage)
    print("Laplacian done!")
    eigen = Eigen(L,k,mode_S,testImage) #.shape (10000 k)
    print("Eigen done!")
    record, iteration = Kmeans(eigen,k,mode_K)
    print("Kmeans done!")
    visualization(record,iteration,k,image_size,output_dir)
    if k==2:
        drawCoordinates(eigen,record[iteration-1],k)
```

***<SimilarityGraph>***

Same as the *KernelSpace* function in Kernel Kmeans.

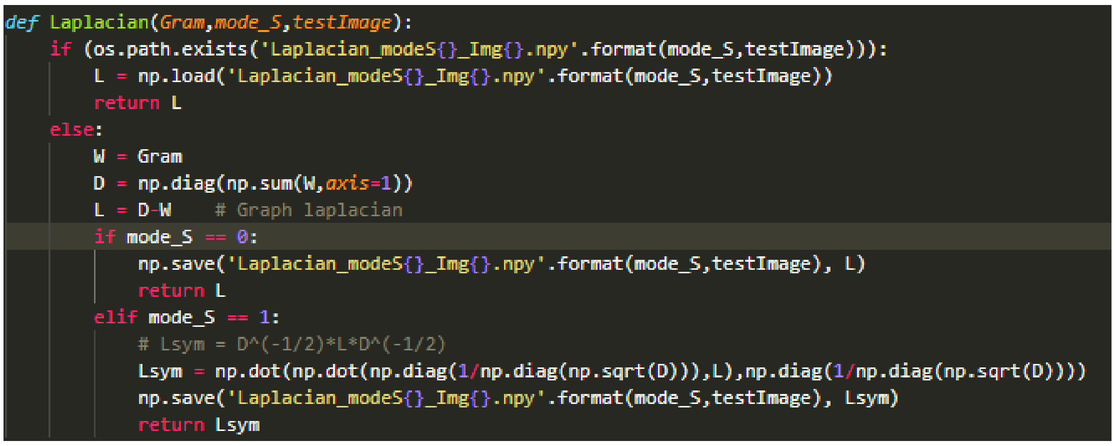


```python
def SimilarityGraph(dataC,dataS,Gs,Gc):
    Gram = np.exp(-Gs*cdist(dataS,dataS,'sqeuclidean'))*np.exp(-Gc*cdist(dataC,dataC,'sqeuclidean'))
    return Gram
```

***<Laplacian>***

If the Spectral is unnormalized, use L = D-W as the Laplacian L.

If the Spectral is normalized, compute Lsym = D^(-1/2)*L*D(-1/2)



```python
def Laplacian(Gram,mode_S,testImage):
    if (os.path.exists('Laplacian_modeS{}_Img{}.npy'.format(mode_S,testImage))):
        L = np.load('Laplacian_modeS{}_Img{}.npy'.format(mode_S,testImage))
        return L
    else:
        W = Gram
        D = np.diag(np.sum(W,axis=1))
        L = D-W    # Graph laplacian
        if mode_S == 0:
            np.save('Laplacian_modeS{}_Img{}.npy'.format(mode_S,testImage), L)
            return L
        elif mode_S == 1:
            # Lsym = D^(-1/2)*L*D^(-1/2)
            Lsym = np.dot(np.dot(np.diag(1/np.diag(np.sqrt(D))),L),np.diag(1/np.diag(np.sqrt(D))))
            np.save('Laplacian_modeS{}_Img{}.npy'.format(mode_S,testImage), Lsym)
            return Lsym
```

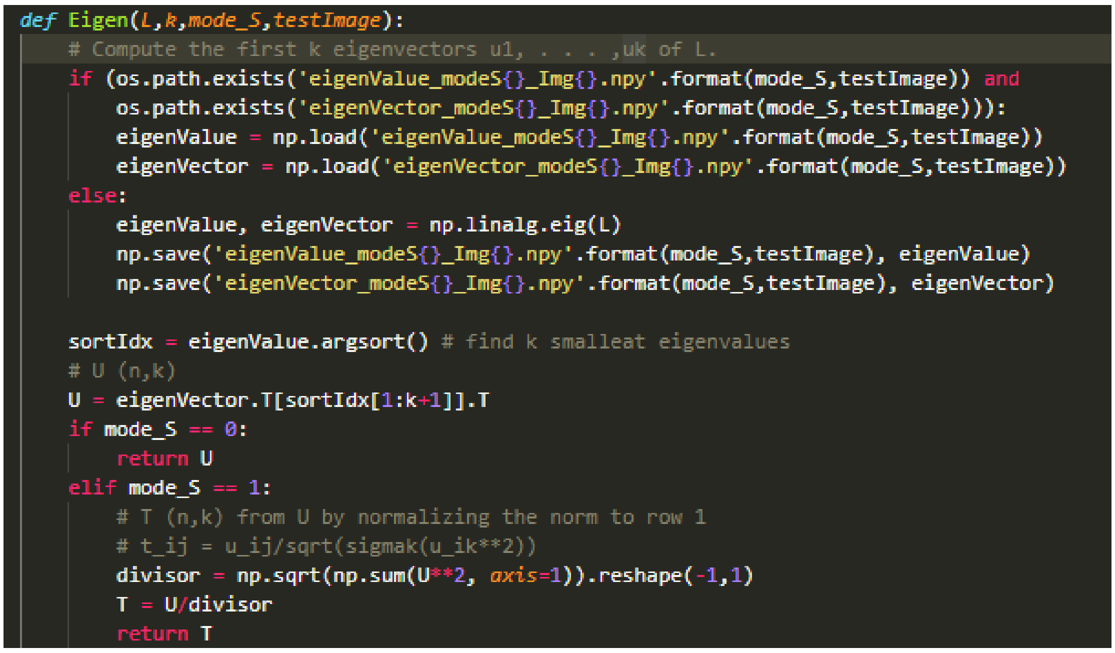
***<Eigen>***

Find the first k eigenvectors of L and compute U and T(if normalized)

Use *np.linalg.eig()* to get the eigenvalue and eigenvector of L, then sort eigenvalue to get the first k eigen vector as columns of U.

If Spectral is normalized, compute T by following the formula

$$t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}.$$

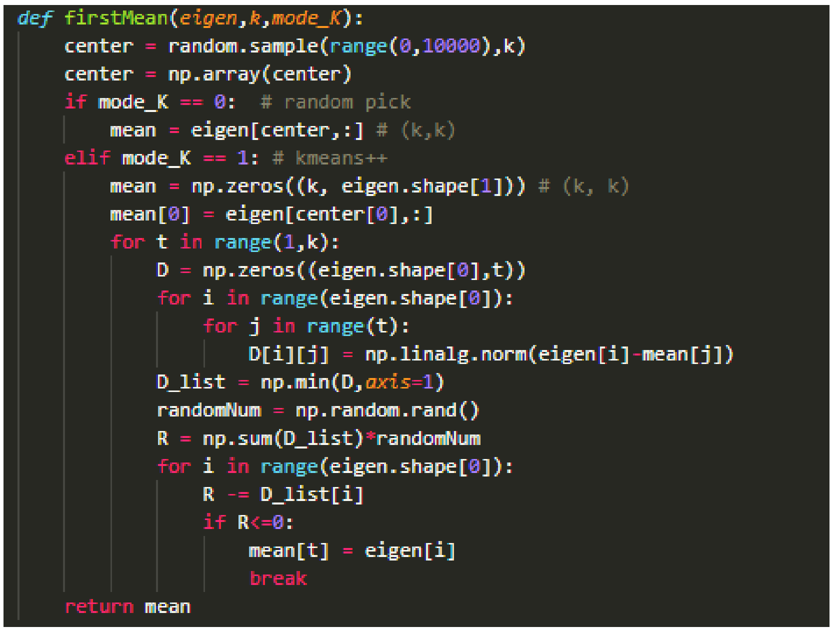


```python
def Eigen(L,k,mode_S,testImage):
    # Compute the first k eigenvectors u1, . . . ,uk of L.
    if (os.path.exists('eigenValue_modeS{}_Img{}.npy'.format(mode_S,testImage)) and
        os.path.exists('eigenVector_modeS{}_Img{}.npy'.format(mode_S,testImage))):
        eigenValue = np.load('eigenValue_modeS{}_Img{}.npy'.format(mode_S,testImage))
        eigenVector = np.load('eigenVector_modeS{}_Img{}.npy'.format(mode_S,testImage))
    else:
        eigenValue, eigenVector = np.linalg.eig(L)
        np.save('eigenValue_modeS{}_Img{}.npy'.format(mode_S,testImage), eigenValue)
        np.save('eigenVector_modeS{}_Img{}.npy'.format(mode_S,testImage), eigenVector)

    sortIdx = eigenValue.argsort() # find k smalleat eigenvalues
    # U (n,k)
    U = eigenVector.T[sortIdx[1:k+1]].T
    if mode_S == 0:
        return U
    elif mode_S == 1:
        # T (n,k) from U by normalizing the norm to row 1
        # t_ij = u_ij/sqrt(sigmak(u_ik**2))
        divisor = np.sqrt(np.sum(U**2, axis=1)).reshape(-1,1)
        T = U/divisor
        return T
```

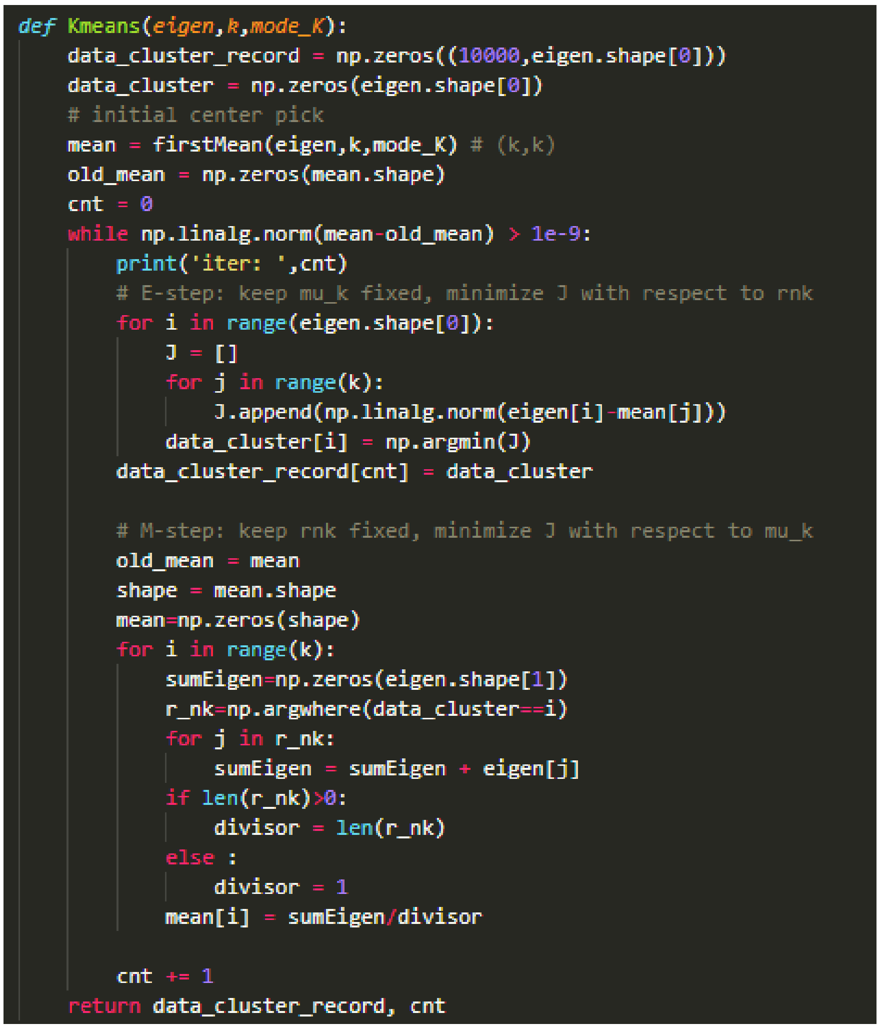
***<Kernel Kmeans>***

Use the result of *Eigen* function to update the centers.

The *firstMean* function and *Kmeans* function is similar to the ones in Kernel Kmeans



```python
def firstMean(eigen,k,mode_K):
    center = random.sample(range(0,10000),k)
    center = np.array(center)
    if mode_K == 0:  # random pick
        mean = eigen[center,:] # (k,k)
    elif mode_K == 1: # kmeans++
        mean = np.zeros((k, eigen.shape[1])) # (k, k)
        mean[0] = eigen[center[0],:]
        for t in range(1,k):
            D = np.zeros((eigen.shape[0],t))
            for i in range(eigen.shape[0]):
                for j in range(t):
                    D[i][j] = np.linalg.norm(eigen[i]-mean[j])
            D_list = np.min(D,axis=1)
            randomNum = np.random.rand()
            R = np.sum(D_list)*randomNum
            for i in range(eigen.shape[0]):
                R -= D_list[i]
                if R<=0:
                    mean[t] = eigen[i]
                    break
    return mean
```

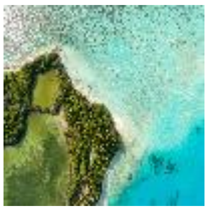
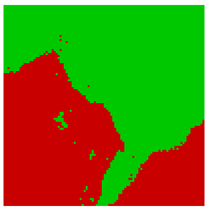
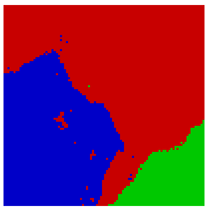
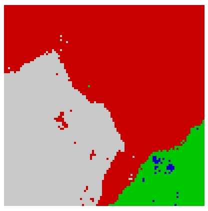


```python
def Kmeans(eigen,k,mode_K):
    data_cluster_record = np.zeros((10000,eigen.shape[0]))
    data_cluster = np.zeros(eigen.shape[0])
    # initial center pick
    mean = firstMean(eigen,k,mode_K) # (k,k)
    old_mean = np.zeros(mean.shape)
    cnt = 0
    while np.linalg.norm(mean-old_mean) > 1e-9:
        print('iter: ',cnt)
        # E-step: keep mu_k fixed, minimize J with respect to rnk
        for i in range(eigen.shape[0]):
            J = []
            for j in range(k):
                J.append(np.linalg.norm(eigen[i]-mean[j]))
            data_cluster[i] = np.argmin(J)
        data_cluster_record[cnt] = data_cluster

        # M-step: keep rnk fixed, minimize J with respect to mu_k
        old_mean = mean
        shape = mean.shape
        mean=np.zeros(shape)
        for i in range(k):
            sumEigen=np.zeros(eigen.shape[1])
            r_nk=np.argwhere(data_cluster==i)
            for j in r_nk:
                sumEigen = sumEigen + eigen[j]
            if len(r_nk)>0:
                divisor = len(r_nk)
            else :
                divisor = 1
            mean[i] = sumEigen/divisor

        cnt += 1
    return data_cluster_record, cnt
```

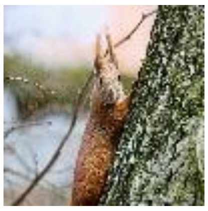
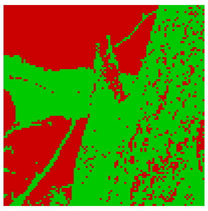
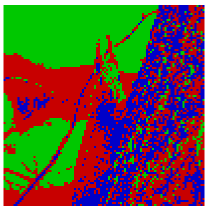
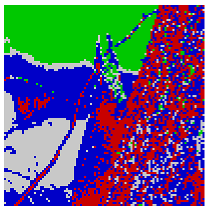
Result:

Unnormalized Spectral Clustering with Kernel Kmeans with random pick Kmeans

Origin => k=2 k=3 k=4

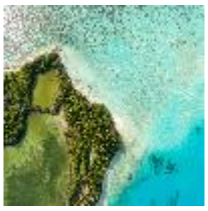
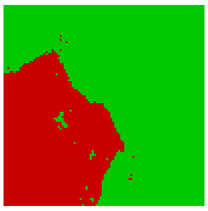
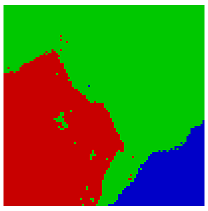
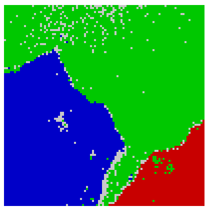





Origin => k=2 k=3 k=4

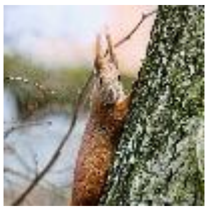
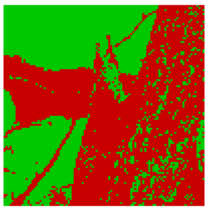
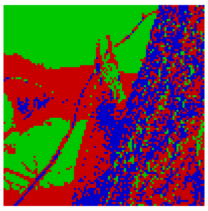
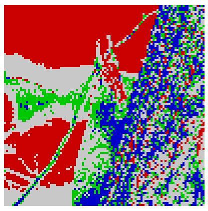





Unnormalized Spectral Clustering with Kernel Kmeans with Kmeans++

Origin => k=2 k=3 k=4

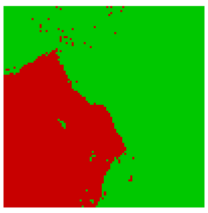
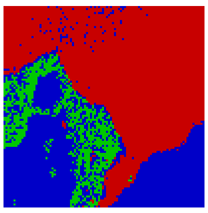
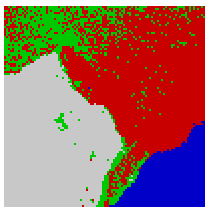





Origin => k=2 k=3 k=4

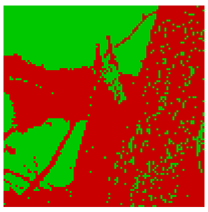
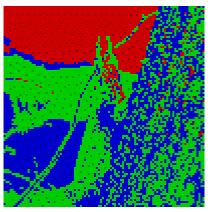
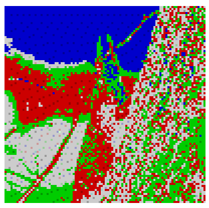





Normalized Spectral Clustering with Kernel Kmeans with random pick Kmeans

Origin => k=2 k=3 k=4

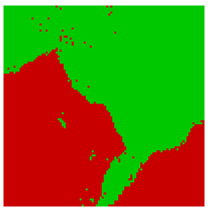
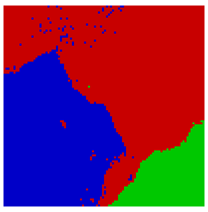
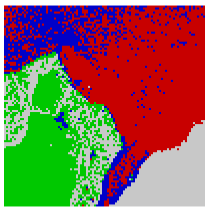




Origin => k=2 k=3 k=4

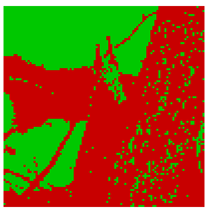
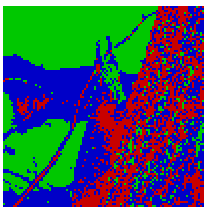
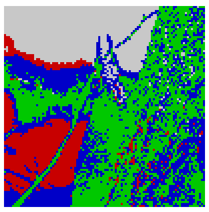




Normalized Spectral Clustering with Kernel Kmeans with Kmeans++

Origin => k=2 k=3 k=4

Origin => k=2 k=3 k=4

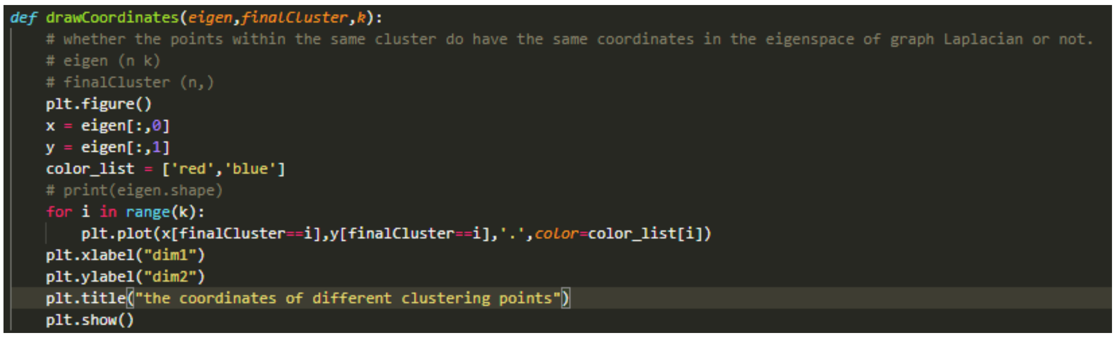




Observation:

- If there are many boundaries in the origin picture (ex. Image2), the outputs of the pixels to clusters are more likely to be different.
- The output of Unnormalized Spectral with Kernel Kmeans with random pick Kmeans is similar to Unnormalized Spectral with Kernel Kmeans with Kmeans++. And the output of Normalized Spectral with Kernel Kmeans with random pick Kmeans is similar to Normalized Spectral with Kernel Kmeans with Kmeans++.
- In smaller k (k=2), outputs of Normalized Spectral are more likely to look as same as the ones of Unmalized Spectral. And with larger k (k=4), the outputs are more likely to look different.

***<drawCoordinates>***

To test if the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian.

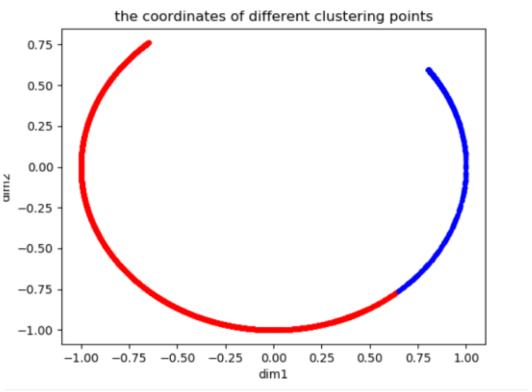


```python
def drawCoordinates(eigen,finalCluster,k):
    # whether the points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not.
    # eigen (n k)
    # finalCluster (n,)
    plt.figure()
    x = eigen[:,0]
    y = eigen[:,1]
    color_list = ['red','blue']
    # print(eigen.shape)
    for i in range(k):
        plt.plot(x[finalCluster==i],y[finalCluster==i],'.',color=color_list[i])
    plt.xlabel("dim1")
    plt.ylabel("dim2")
    plt.title("the coordinates of different clustering points")
    plt.show()
```

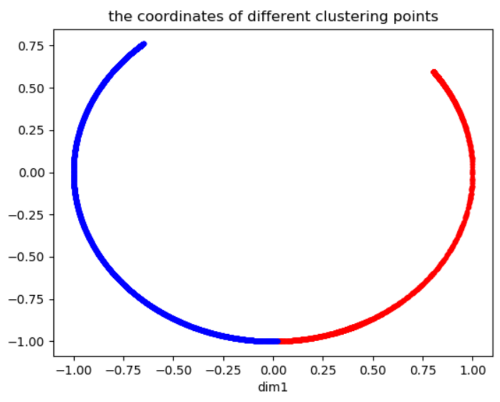
Result:

When Image2, k = 2 (clusters) :

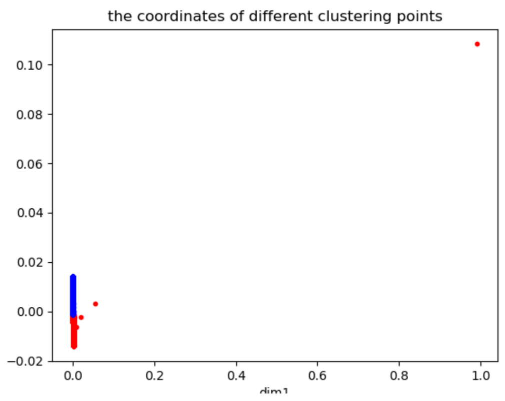
Normalized & initial kmeans



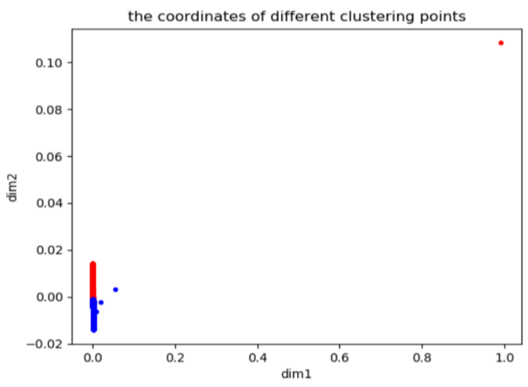
Normalized & kmeans++



Unnormalized & initial kmeans



Unnormalized & kmeans++



According to the result, I observed that the data points within the same cluster do not have the same coordinates in the eigenspace of graph Laplacian, but their coordinates are close and relative to others in the same cluster.

Besides, the coordinates of origin kmeans and the coordinates of kmeans++ is similar, and the coordinates of Normalized Spectral and the coordinates of Unnormalized Spectral is quite different.