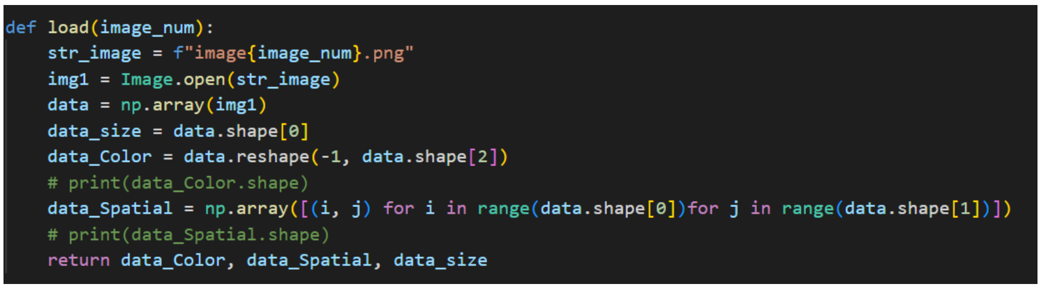
# HW6 311552055 蕭育泓

## Part a. code explanations

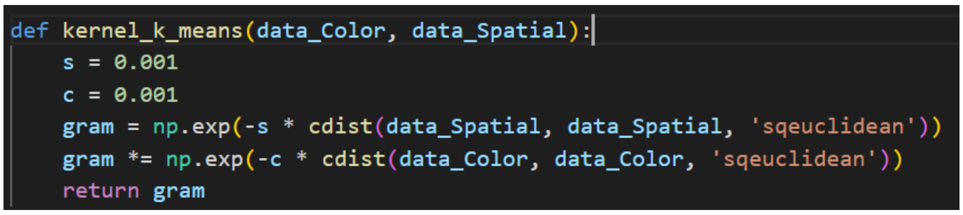
First, we need to load the data about the data_color(10000*3) ,the data_spatial(10000*2) and the data_size = 100



```python
def load(image_num):
    str_image = f"image{image_num}.png"
    img1 = Image.open(str_image)
    data = np.array(img1)
    data_size = data.shape[0]
    data_Color = data.reshape(-1, data.shape[2])
    # print(data_Color.shape)
    data_Spatial = np.array([(i, j) for i in range(data.shape[0])for j in range(data.shape[1])])
    # print(data_Spatial.shape)
    return data_Color, data_Spatial, data_size
```

## Part 1

kernel k-means

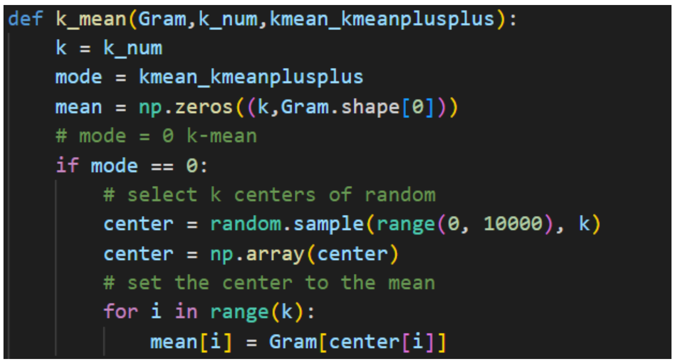


```python
def kernel_k_means(data_Color, data_Spatial):
    s = 0.001
    c = 0.001
    gram = np.exp(-s * cdist(data_Spatial, data_Spatial, 'sqeuclidean'))
    gram *= np.exp(-c * cdist(data_Color, data_Color, 'sqeuclidean'))
    return gram
```

Using the data_color and data_spatial to find out the gram matrix using the formula $k(x, x') = e^{-\gamma_s \|S(x)-S(x')\|^2} \times e^{-\gamma_c \|C(x)-C(x')\|^2}$

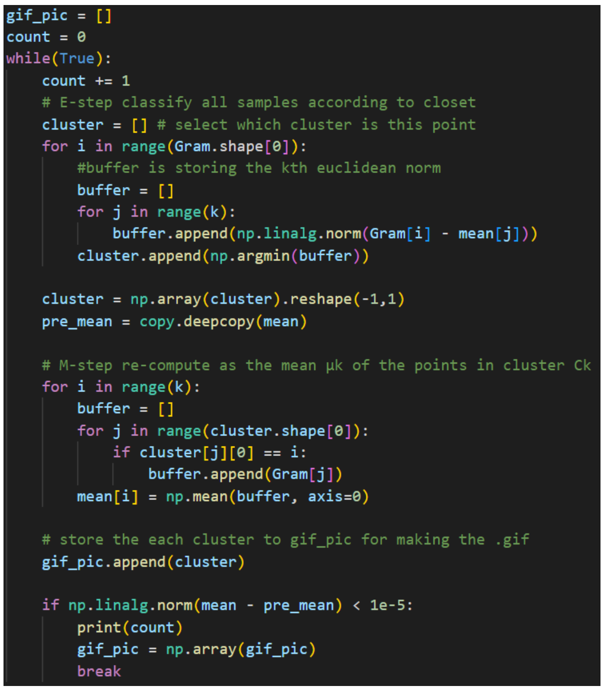
## In the k-mean.py

Function - k_mean



```python
def k_mean(Gram,k_num,kmean_kmeanplusplus):
    k = k_num
    mode = kmean_kmeanplusplus
    mean = np.zeros((k,Gram.shape[0]))
    # mode = 0 k-mean
    if mode == 0:
        # select k centers of random
        center = random.sample(range(0, 10000), k)
        center = np.array(center)
        # set the center to the mean
        for i in range(k):
            mean[i] = Gram[center[i]]
```

Picture1



```python
gif_pic = []
count = 0
while(True):
    count += 1
    # E-step classify all samples according to closet
    cluster = [] # select which cluster is this point
    for i in range(Gram.shape[0]):
        #buffer is storing the kth euclidean norm
        buffer = []
        for j in range(k):
            buffer.append(np.linalg.norm(Gram[i] - mean[j]))
        cluster.append(np.argmin(buffer))

    cluster = np.array(cluster).reshape(-1,1)
    pre_mean = copy.deepcopy(mean)

    # M-step re-compute as the mean μk of the points in cluster Ck
    for i in range(k):
        buffer = []
        for j in range(cluster.shape[0]):
            if cluster[j][0] == i:
                buffer.append(Gram[j])
        mean[i] = np.mean(buffer, axis=0)

    # store the each cluster to gif_pic for making the .gif
    gif_pic.append(cluster)

    if np.linalg.norm(mean - pre_mean) < 1e-5:
        print(count)
        gif_pic = np.array(gif_pic)
        break
```

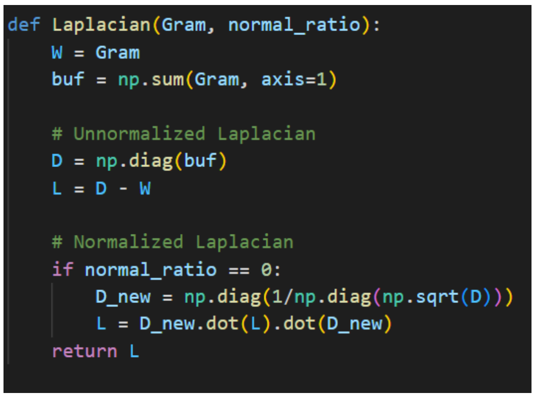
Picture2

Doing the k mean by using the random select mean in the beginning(Picture 1). We will repeat the step E (classify the point in which group) and step M (compte the new mean point) until the mean and pre_mean is converge(smaller than 1e-5). I also store the classify result for each loop.(Picture 2)

## In the spectral_clustering.py

Using some of the same functions in the k-mean.py like function - kernel_k_means, function – kmean to help me to generate the result. The most different things are we need to compute the Laplacian matrix with normal cut or ratio cut and find out the eigenspace.

## Function - Laplacian

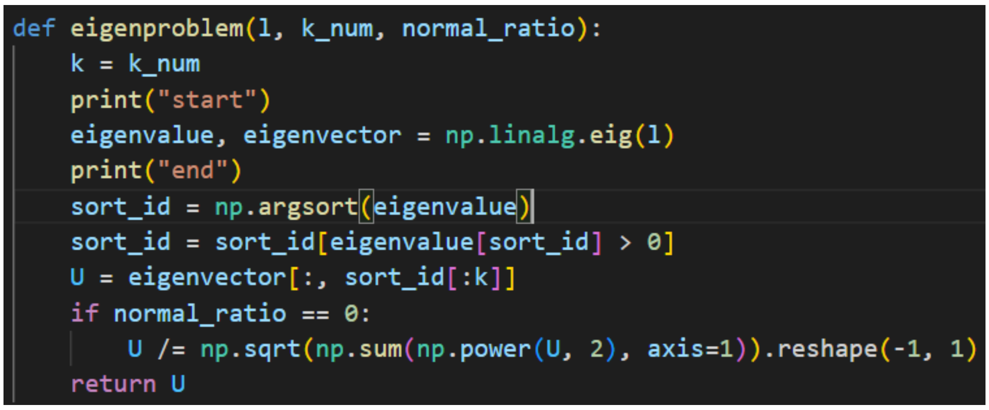


```python
def Laplacian(Gram, normal_ratio):
    W = Gram
    buf = np.sum(Gram, axis=1)

    # Unnormalized Laplacian
    D = np.diag(buf)
    L = D - W

    # Normalized Laplacian
    if normal_ratio == 0:
        D_new = np.diag(1/np.diag(np.sqrt(D)))
        L = D_new.dot(L).dot(D_new)
    return L
```

The Function – Laplacian, the Laplacian matrix(L) is be find from the matrix W and matrix D, the matrix W is the weight of the each edge which can find from the function-kernel_k_means and the matrix D is the diagonal matrix which which can find from the matrix D. If we select the normal cut, the Laplacian matrix(L) is represent as D-W. If we select the ratio cut, we will use formula below to find out the Laplacian matrix.

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

## Function - eigenproblem



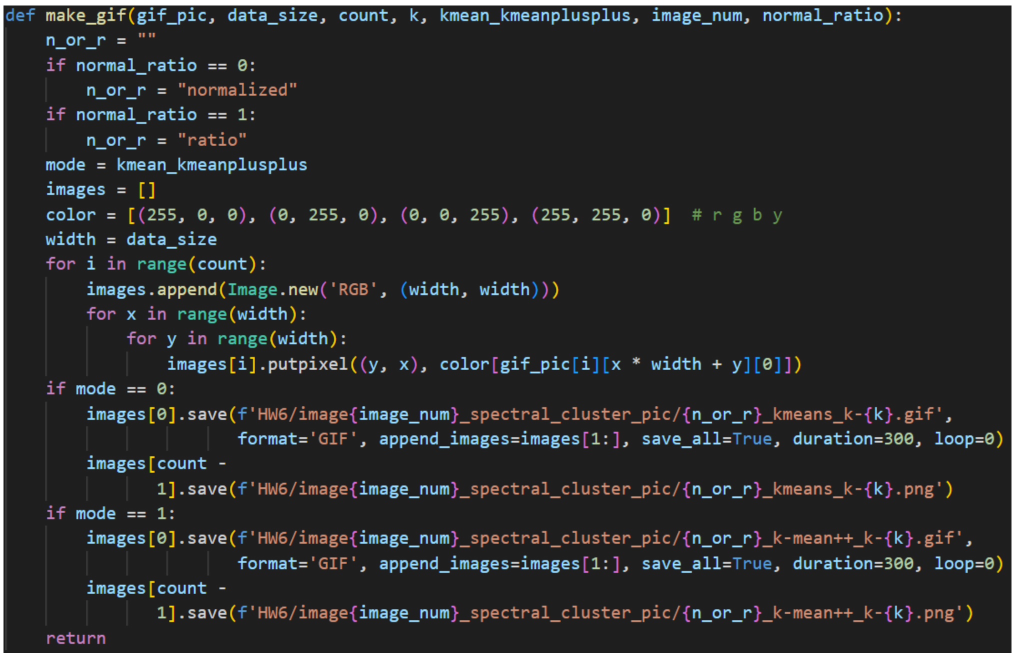
```python
def eigenproblem(l, k_num, normal_ratio):
    k = k_num
    print("start")
    eigenvalue, eigenvector = np.linalg.eig(l)
    print("end")
    sort_id = np.argsort(eigenvalue)
    sort_id = sort_id[eigenvalue[sort_id] > 0]
    U = eigenvector[:, sort_id[:k]]
    if normal_ratio == 0:
        U /= np.sqrt(np.sum(np.power(U, 2), axis=1)).reshape(-1, 1)
    return U
```

The Function – eigenproblem is used to find out the eigenspace with non-zero eigenvalue. If we select the normal cut , we need to normalizing the rows to norm 1 by the formula below.

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

We will find the clusters like kmean.py in spectral_clustering.py with the reuse functions. Both of the .py will make the .gif to show the result.

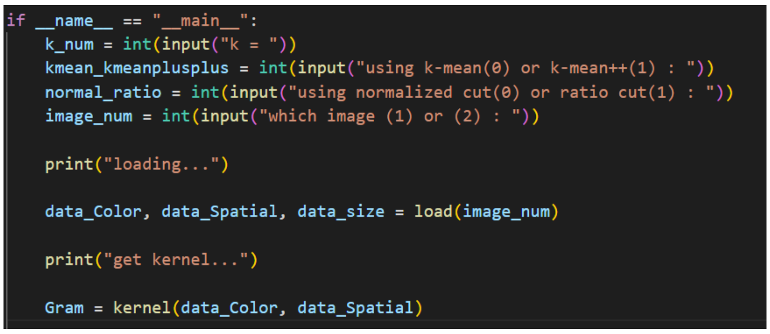
The function – make_gif



```python
def make_gif(gif_pic, data_size, count, k, kmean_kmeanplusplus, image_num, normal_ratio):
    n_or_r = ""
    if normal_ratio == 0:
        n_or_r = "normalized"
    if normal_ratio == 1:
        n_or_r = "ratio"
    mode = kmean_kmeanplusplus
    images = []
    color = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]  # r g b y
    width = data_size
    for i in range(count):
        images.append(Image.new('RGB', (width, width)))
        for x in range(width):
            for y in range(width):
                images[i].putpixel((y, x), color[gif_pic[i][x * width + y][0]])
    if mode == 0:
        images[0].save(f'HW6/image{image_num}_spectral_cluster_pic/{n_or_r}_kmeans_k-{k}.gif',
                       format='GIF', append_images=images[1:], save_all=True, duration=300, loop=0)
        images[count -
               1].save(f'HW6/image{image_num}_spectral_cluster_pic/{n_or_r}_kmeans_k-{k}.png')
    if mode == 1:
        images[0].save(f'HW6/image{image_num}_spectral_cluster_pic/{n_or_r}_k-mean++_k-{k}.gif',
                       format='GIF', append_images=images[1:], save_all=True, duration=300, loop=0)
        images[count -
               1].save(f'HW6/image{image_num}_spectral_cluster_pic/{n_or_r}_k-mean++_k-{k}.png')
    return
```

## Part 2

The different between part1 and part2 is the number of clusters(k), so in the main function we can select the value k what we want to be. In addition, we are same as part1.

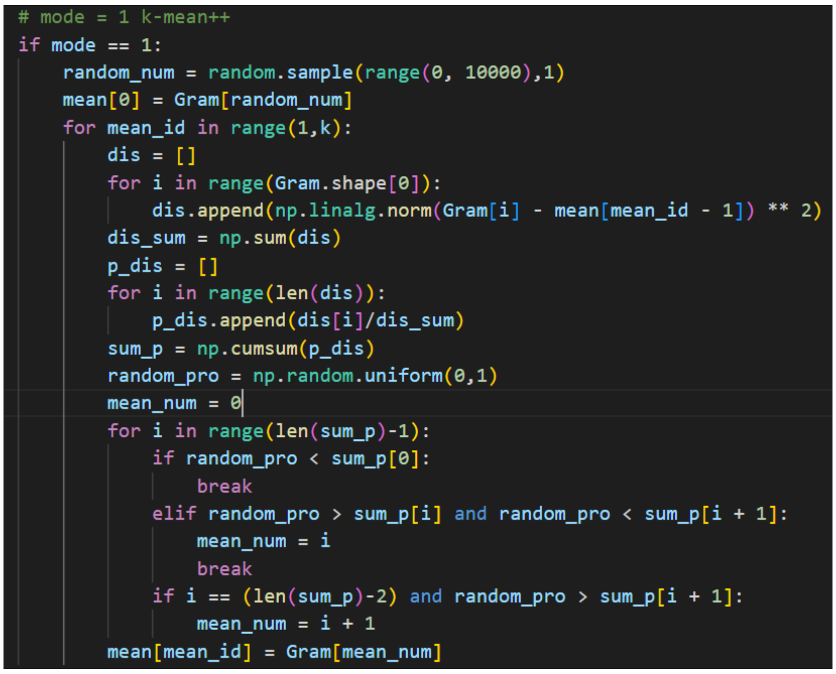


```python
if __name__ == "__main__":
    k_num = int(input("k = "))
    kmean_kmeanplusplus = int(input("using k-mean(0) or k-mean++(1) : "))
    normal_ratio = int(input("using normalized cut(0) or ratio cut(1) : "))
    image_num = int(input("which image (1) or (2) : "))

    print("loading...")

    data_Color, data_Spatial, data_size = load(image_num)

    print("get kernel...")

    Gram = kernel(data_Color, data_Spatial)
```

## Part 3

We try the different way k-means++ to show corresponding results for initialization of k-means clustering it can be select from the main function (the main function is in the part2).

The function – kmean is also express the way k-means++



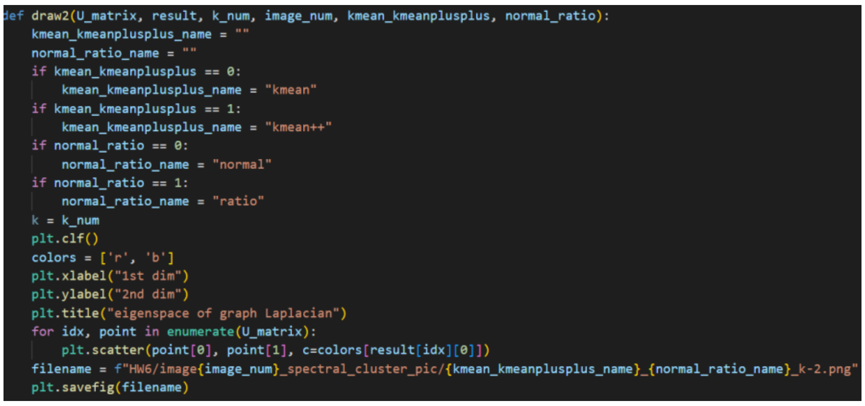
```python
# mode = 1 k-mean++
if mode == 1:
    random_num = random.sample(range(0, 10000),1)
    mean[0] = Gram[random_num]
    for mean_id in range(1,k):
        dis = []
        for i in range(Gram.shape[0]):
            dis.append(np.linalg.norm(Gram[i] - mean[mean_id - 1]) ** 2)
        dis_sum = np.sum(dis)
        p_dis = []
        for i in range(len(dis)):
            p_dis.append(dis[i]/dis_sum)
        sum_p = np.cumsum(p_dis)
        random_pro = np.random.uniform(0,1)
        mean_num = 0
        for i in range(len(sum_p)-1):
            if random_pro < sum_p[0]:
                break
            elif random_pro > sum_p[i] and random_pro < sum_p[i + 1]:
                mean_num = i
                break
            if i == (len(sum_p)-2) and random_pro > sum_p[i + 1]:
                mean_num = i + 1
        mean[mean_id] = Gram[mean_num]
```

First, we select a random mean in the beginning and then compute the distance between each point to the mean, accumulated value to find out the interval. We will random choose the number between 0 – 1, the more far from the mean has the more probability to be the next mean cluster.

## Part 4

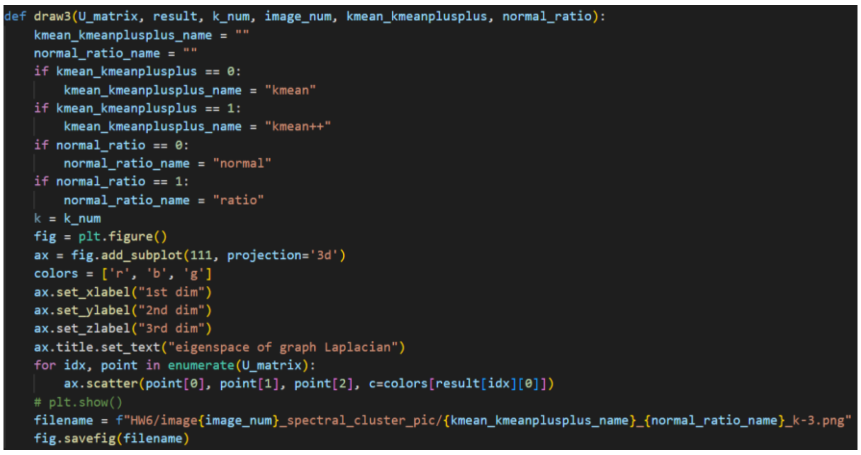
I will show the picture to example the same coordinates in the eigenspace of graph Laplacian for spectral clustering when k is 2 and 3.

The function-draw2



```python
def draw2(U_matrix, result, k_num, image_num, kmean_kmeanplusplus, normal_ratio):
    kmean_kmeanplusplus_name = ""
    normal_ratio_name = ""
    if kmean_kmeanplusplus == 0:
        kmean_kmeanplusplus_name = "kmean"
    if kmean_kmeanplusplus == 1:
        kmean_kmeanplusplus_name = "kmean++"
    if normal_ratio == 0:
        normal_ratio_name = "normal"
    if normal_ratio == 1:
        normal_ratio_name = "ratio"
    k = k_num
    plt.clf()
    colors = ['r', 'b']
    plt.xlabel("1st dim")
    plt.ylabel("2nd dim")
    plt.title("eigenspace of graph Laplacian")
    for idx, point in enumerate(U_matrix):
        plt.scatter(point[0], point[1], c=colors[result[idx][0]])
    filename = f"HW6/image{image_num}_spectral_cluster_pic/{kmean_kmeanplusplus_name}_{normal_ratio_name}_k-2.png"
    plt.savefig(filename)
```

The function-draw3



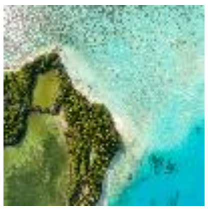
```python
def draw3(U_matrix, result, k_num, image_num, kmean_kmeanplusplus, normal_ratio):
    kmean_kmeanplusplus_name = ""
    normal_ratio_name = ""
    if kmean_kmeanplusplus == 0:
        kmean_kmeanplusplus_name = "kmean"
    if kmean_kmeanplusplus == 1:
        kmean_kmeanplusplus_name = "kmean++"
    if normal_ratio == 0:
        normal_ratio_name = "normal"
    if normal_ratio == 1:
        normal_ratio_name = "ratio"
    k = k_num
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    colors = ['r', 'b', 'g']
    ax.set_xlabel("1st dim")
    ax.set_ylabel("2nd dim")
    ax.set_zlabel("3rd dim")
    ax.title.set_text("eigenspace of graph Laplacian")
    for idx, point in enumerate(U_matrix):
        ax.scatter(point[0], point[1], point[2], c=colors[result[idx][0]])
    # plt.show()
    filename = f"HW6/image{image_num}_spectral_cluster_pic/{kmean_kmeanplusplus_name}_{normal_ratio_name}_k-3.png"
    fig.savefig(filename)
```

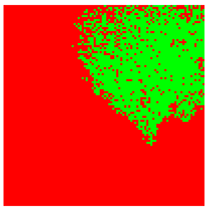
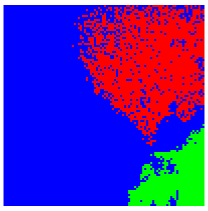
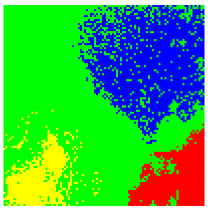
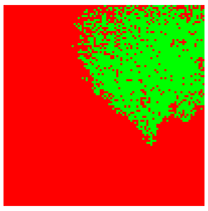
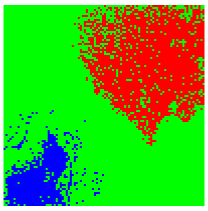
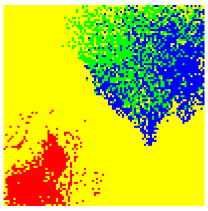
I will set the colors to show the different points and save the picture in the file.

## Part b. results & discussion

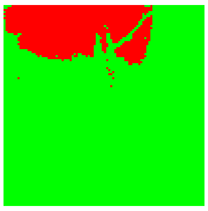
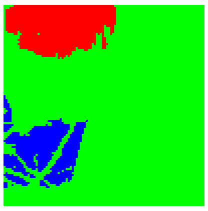
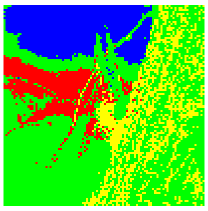
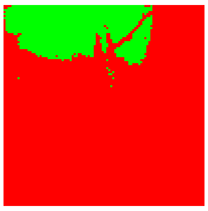
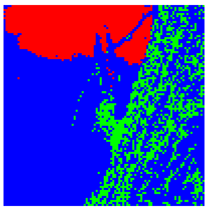
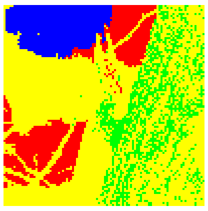
### In the k-mean.py results:

**Image1**



| | **K = 2** | **K = 3** | **K = 4** |
|---|---|---|---|
| **random** |  |  |  |
| **Kmean++** |  |  |  |

## Image2

| | K = 2 | K = 3 | K = 4 |
|---|---|---|---|
| **random** |  |  |  |
| **Kmean++** |  |  |  |

## In the k-mean.py discussion:

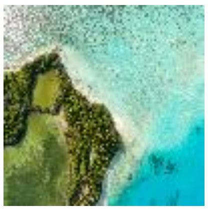
1. There were get the same result in random and kmean++.

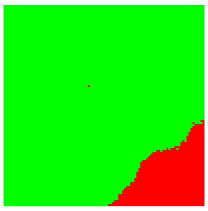
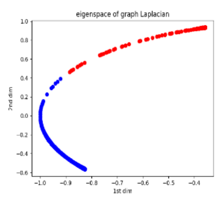
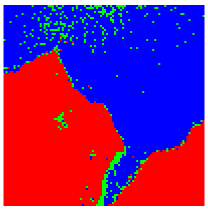
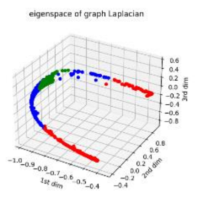
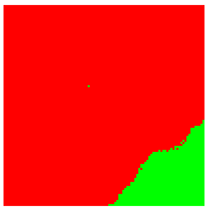
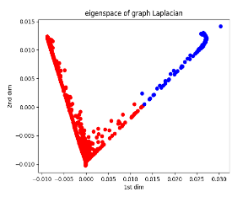
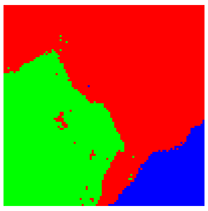
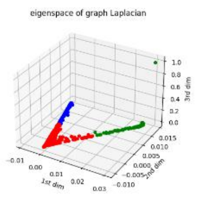
2. In the picture2, we can observe when k = 3, we get the more clear output which mean kmean++ has more stability. It is because the random case choose the cluster from the random.

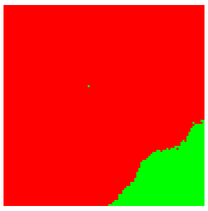
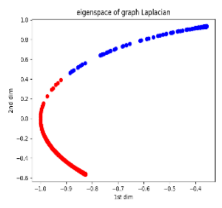
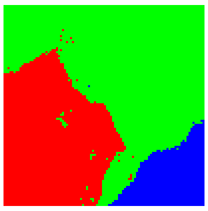
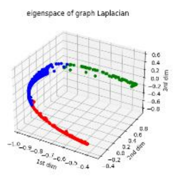
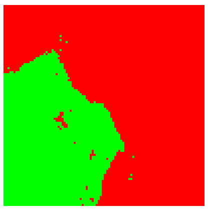
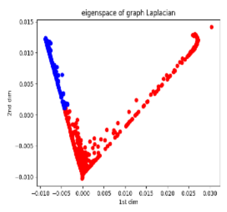
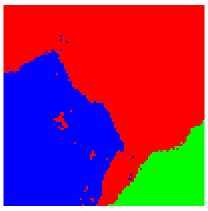
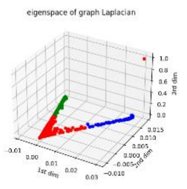
3. In the picture1, we can discover that the land and the sea is far different color in human eyes, but the computer think the cluster is between the two different color of the ocean. It is because of the kernel function and the color weight.

## In the spectral_clustering.py results:

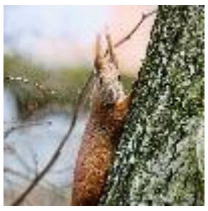
### Image1

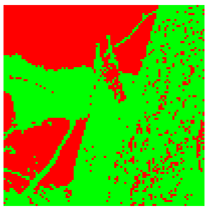
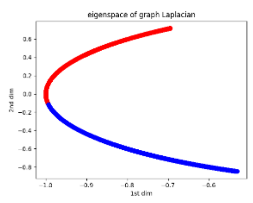
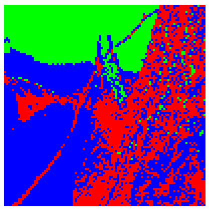
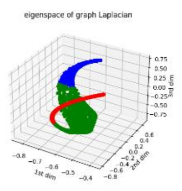
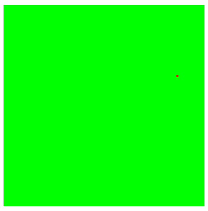
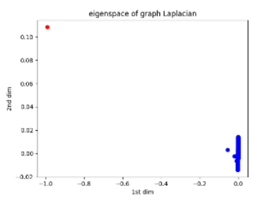
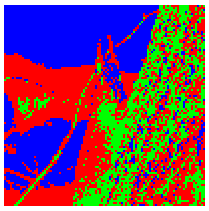
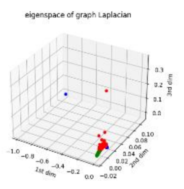


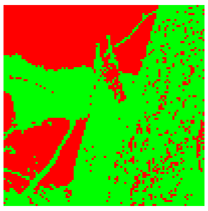
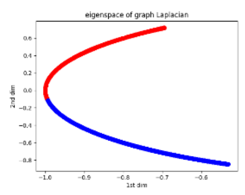
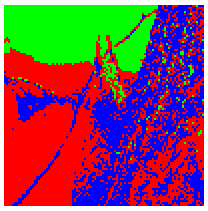
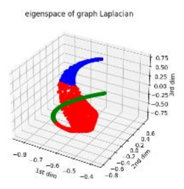
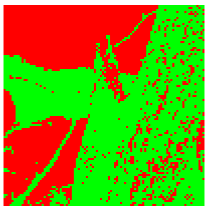
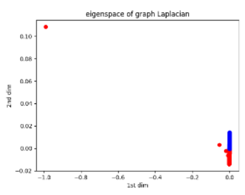
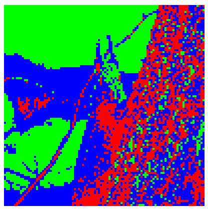
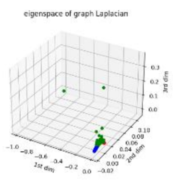
| Random | K = 2 | Examine point | K = 3 | Examine point |
|---|---|---|---|---|
| Normalized cut |  |  |  |  |
| Ratio cut |  |  |  |  |

| Kmean++ | K = 2 | Examine point | K = 3 | Examine point |
|---|---|---|---|---|
| Normalized cut |  |  |  |  |
| Ratio cut |  |  |  |  |

## Image2



| Random | K = 2 | Examine point | K = 3 | Examine point |
|---|---|---|---|---|
| Normalized cut |  |  |  |  |
| Ratio cut |  |  |  |  |

| Kmean++ | K = 2 | Examine point | K = 3 | Examine point |
|---|---|---|---|---|
| Normalized cut |  |  |  |  |
| Ratio cut |  |  |  |  |

## In the spectral_clustering.py discussion:

1. We can observe the worst result is the set of ratio cut and k=2 in image2.

2. There are no different between Normalized cut and ratio cut most of the time.

3. The k is higher and there is more different between Normalized cut and ratio cut.

4. There is same as k-mean.py, kmean++ has a little bester output than kmean.

5. In the examine point, there are useless in image2 ratio cut either random or kmean++.

6. The normalized cut is always differ to ratio cut.

7. In image1, the normalized cut in kmean++ and ratio cut in random output image are same in k=2.

## Part c. observations

1. The execution time in spectral_clustering.py is always about 10 more minutes that is far longer than k-mean.py.
2. When using the kernel kmean++, the execution time is a liitle more than using the random kernel.
3. When execute the spectral_clustering.py, I discover the converge time is less than k-mean.py.
4. The execution time in spectral_clustering.py takes the most of its time to compute the eigenvector and eigenvalue.