ML_HW6_report

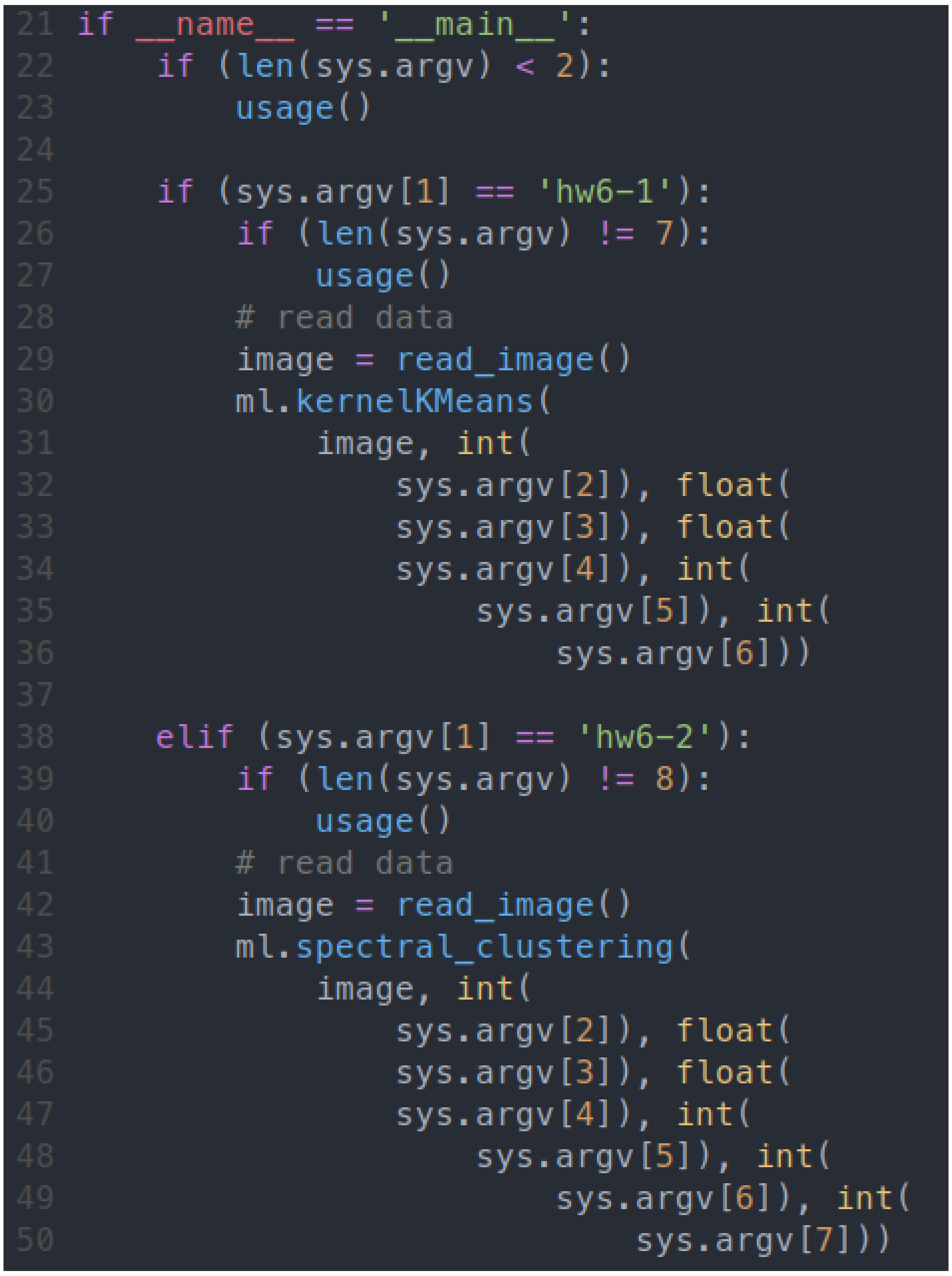
309554027 鍾弈言

A. code with detailed explanations (20%)

a. **Kernel k-means**

1. **main.py**

   I process all input parameters and read the image in main.py.

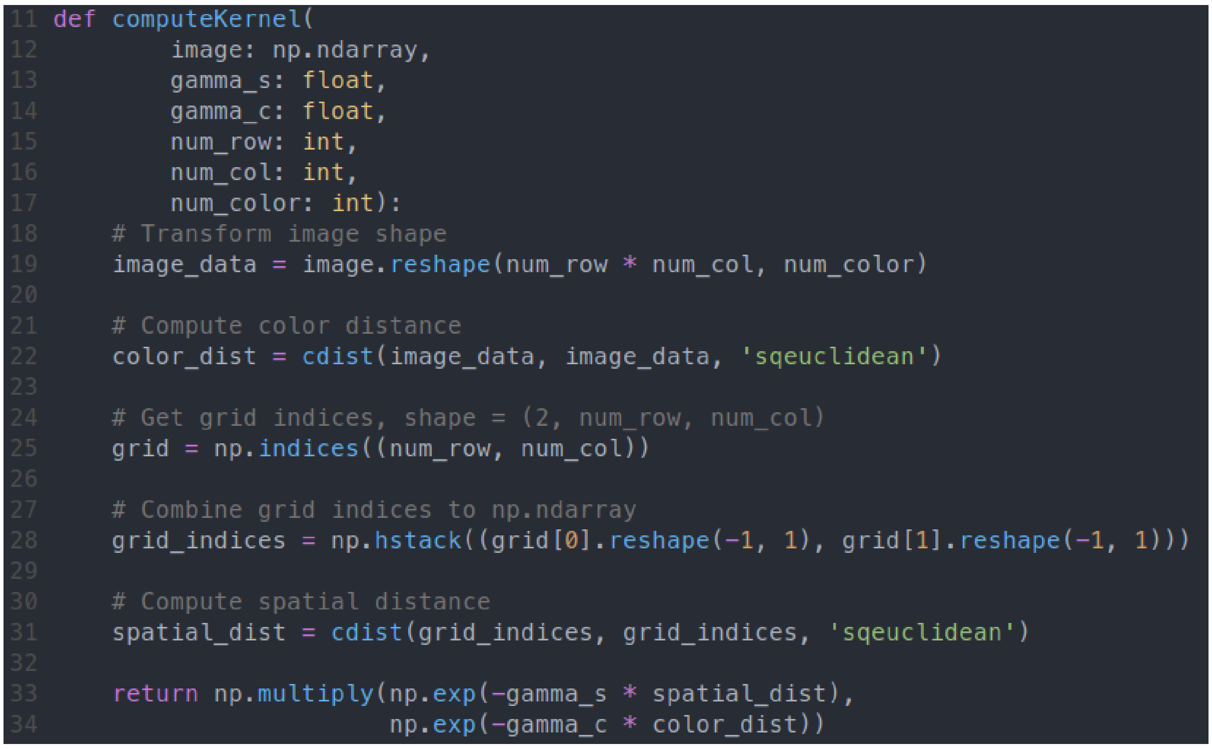


```python
if __name__ == '__main__':
    if (len(sys.argv) < 2):
        usage()

    if (sys.argv[1] == 'hw6-1'):
        if (len(sys.argv) != 7):
            usage()
        # read data
        image = read_image()
        ml.kernelKMeans(
            image, int(
                sys.argv[2]), float(
                sys.argv[3]), float(
                sys.argv[4]), int(
                    sys.argv[5]), int(
                        sys.argv[6]))

    elif (sys.argv[1] == 'hw6-2'):
        if (len(sys.argv) != 8):
            usage()
        # read data
        image = read_image()
        ml.spectral_clustering(
            image, int(
                sys.argv[2]), float(
                sys.argv[3]), float(
                sys.argv[4]), int(
                    sys.argv[5]), int(
                        sys.argv[6]), int(
                            sys.argv[7]))
```

2. **computeKernel() in kernel_kmeans.py**

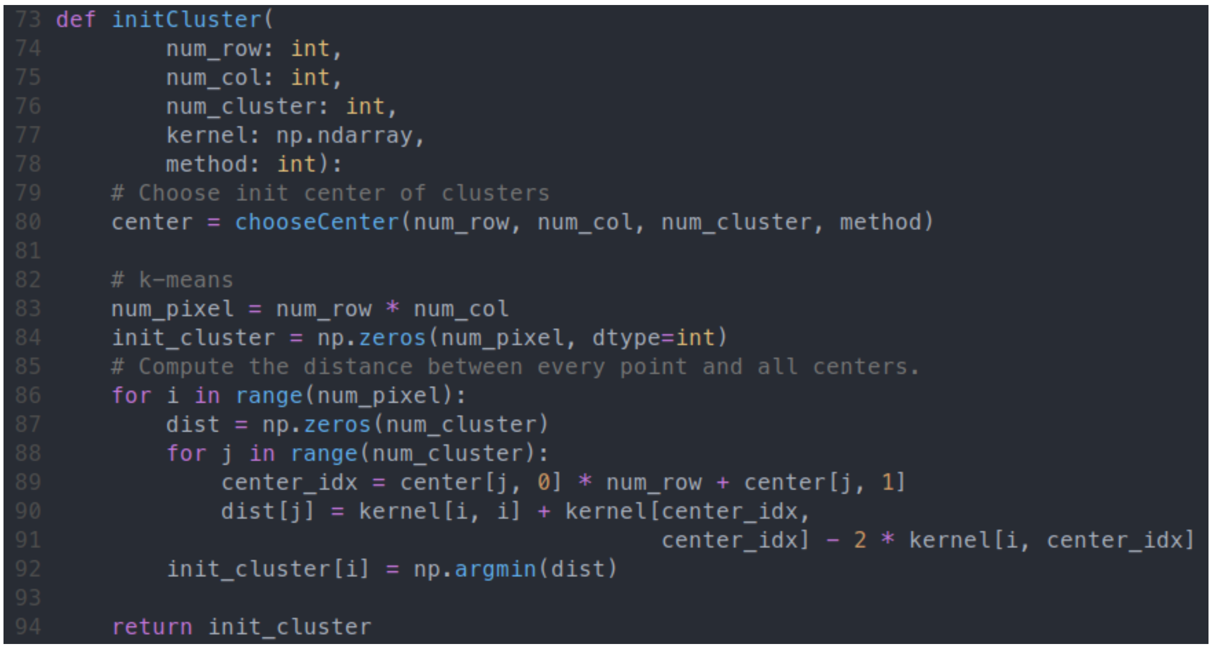
   Based on the new kernel definition provided by TA, I use the following code to calculate the Gram matrix.

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



```python
def computeKernel(
        image: np.ndarray,
        gamma_s: float,
        gamma_c: float,
        num_row: int,
        num_col: int,
        num_color: int):
    # Transform image shape
    image_data = image.reshape(num_row * num_col, num_color)

    # Compute color distance
    color_dist = cdist(image_data, image_data, 'sqeuclidean')

    # Get grid indices, shape = (2, num_row, num_col)
    grid = np.indices((num_row, num_col))

    # Combine grid indices to np.ndarray
    grid_indices = np.hstack((grid[0].reshape(-1, 1), grid[1].reshape(-1, 1)))

    # Compute spatial distance
    spatial_dist = cdist(grid_indices, grid_indices, 'sqeuclidean')

    return np.multiply(np.exp(-gamma_s * spatial_dist),
                       np.exp(-gamma_c * color_dist))
```

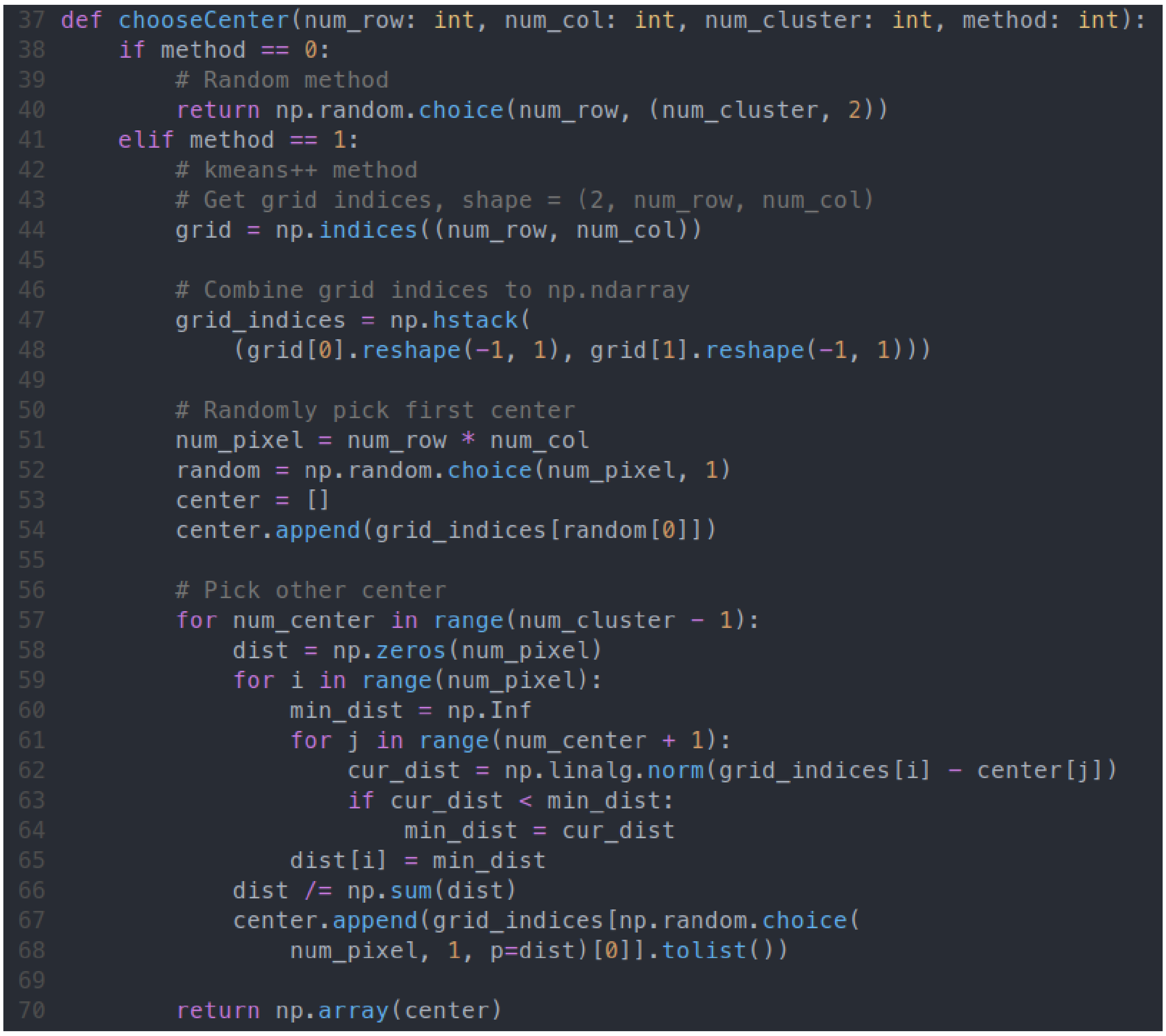
3. **initCluster()**
   Before doing k-means, I first initialize the cluster based on the input parameters num_cluster and method.



```python
def initCluster(
        num_row: int,
        num_col: int,
        num_cluster: int,
        kernel: np.ndarray,
        method: int):
    # Choose init center of clusters
    center = chooseCenter(num_row, num_col, num_cluster, method)

    # k-means
    num_pixel = num_row * num_col
    init_cluster = np.zeros(num_pixel, dtype=int)
    # Compute the distance between every point and all centers.
    for i in range(num_pixel):
        dist = np.zeros(num_cluster)
        for j in range(num_cluster):
            center_idx = center[j, 0] * num_row + center[j, 1]
            dist[j] = kernel[i, i] + kernel[center_idx,
                                            center_idx] - 2 * kernel[i, center_idx]
        init_cluster[i] = np.argmin(dist)

    return init_cluster
```

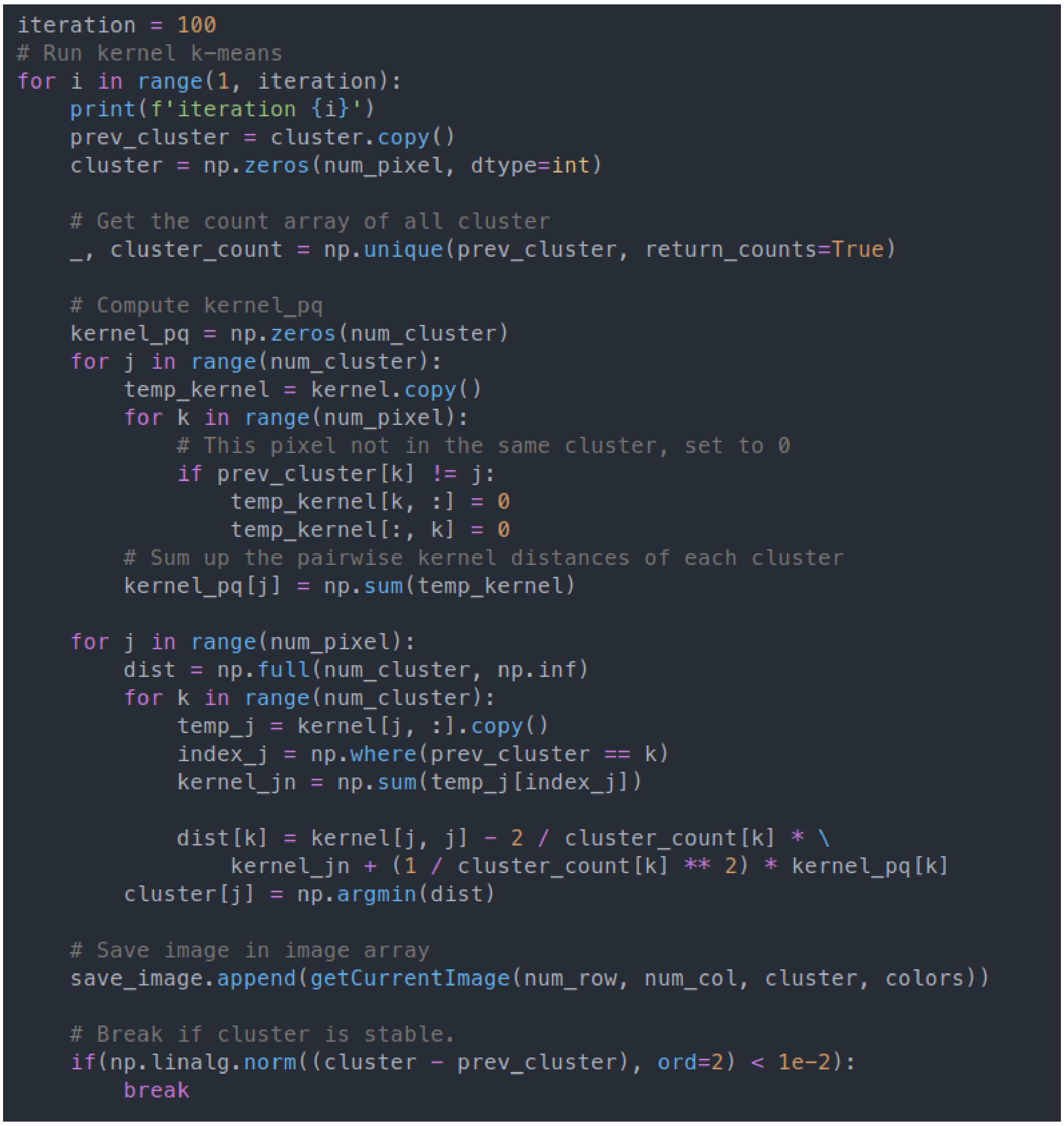
4. **initCluster() -> chooseCenter()**
   According to input parameters, "Method" determines the use of a random method or k-means++ method to select the

initialization center. Finally, the coordinates of the center are returned.



```python
def chooseCenter(num_row: int, num_col: int, num_cluster: int, method: int):
    if method == 0:
        # Random method
        return np.random.choice(num_row, (num_cluster, 2))
    elif method == 1:
        # kmeans++ method
        # Get grid indices, shape = (2, num_row, num_col)
        grid = np.indices((num_row, num_col))

        # Combine grid indices to np.ndarray
        grid_indices = np.hstack(
            (grid[0].reshape(-1, 1), grid[1].reshape(-1, 1)))

        # Randomly pick first center
        num_pixel = num_row * num_col
        random = np.random.choice(num_pixel, 1)
        center = []
        center.append(grid_indices[random[0]])

        # Pick other center
        for num_center in range(num_cluster - 1):
            dist = np.zeros(num_pixel)
            for i in range(num_pixel):
                min_dist = np.Inf
                for j in range(num_center + 1):
                    cur_dist = np.linalg.norm(grid_indices[i] - center[j])
                    if cur_dist < min_dist:
                        min_dist = cur_dist
                dist[i] = min_dist
            dist /= np.sum(dist)
            center.append(grid_indices[np.random.choice(
                num_pixel, 1, p=dist)[0]].tolist())

        return np.array(center)
```

5. **initCluster() -> bottom half**
   After selecting the initial center, I classify all pixels by the minimum distance in the feature space between the center and all pixels. Finally, I got the init cluster.
6. **computeKernelKMeans()**
   Now, I got the init cluster and gram matrix (variable kernel). Next, I repeatedly calculate the kernel k-means until the difference between the new cluster and the old cluster is small enough. Then it will break the loop and return the final result.



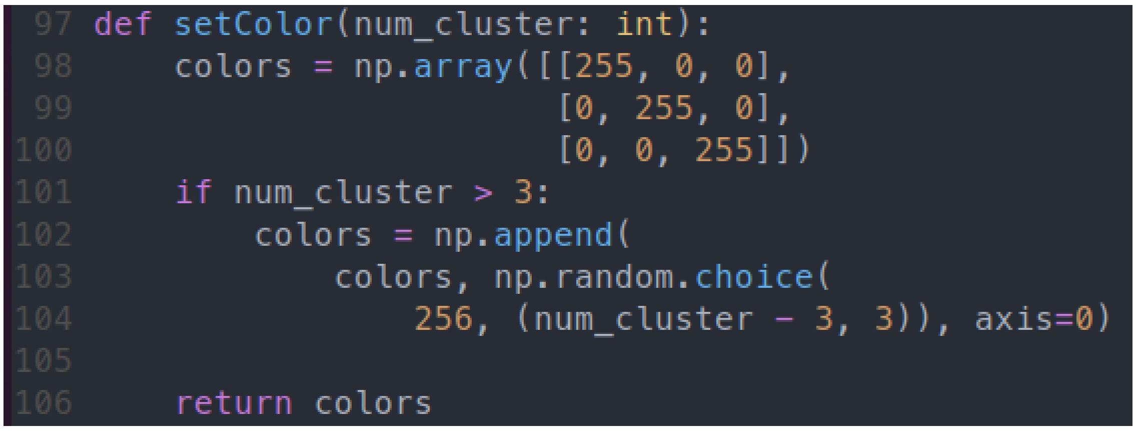
```python
iteration = 100
# Run kernel k-means
for i in range(1, iteration):
    print(f'iteration {i}')
    prev_cluster = cluster.copy()
    cluster = np.zeros(num_pixel, dtype=int)

    # Get the count array of all cluster
    _, cluster_count = np.unique(prev_cluster, return_counts=True)

    # Compute kernel_pq
    kernel_pq = np.zeros(num_cluster)
    for j in range(num_cluster):
        temp_kernel = kernel.copy()
        for k in range(num_pixel):
            # This pixel not in the same cluster, set to 0
            if prev_cluster[k] != j:
                temp_kernel[k, :] = 0
                temp_kernel[:, k] = 0
        # Sum up the pairwise kernel distances of each cluster
        kernel_pq[j] = np.sum(temp_kernel)

    for j in range(num_pixel):
        dist = np.full(num_cluster, np.inf)
        for k in range(num_cluster):
            temp_j = kernel[j, :].copy()
            index_j = np.where(prev_cluster == k)
            kernel_jn = np.sum(temp_j[index_j])

            dist[k] = kernel[j, j] - 2 / cluster_count[k] * \
                kernel_jn + (1 / cluster_count[k] ** 2) * kernel_pq[k]
        cluster[j] = np.argmin(dist)

    # Save image in image array
    save_image.append(getCurrentImage(num_row, num_col, cluster, colors))

    # Break if cluster is stable.
    if(np.linalg.norm((cluster - prev_cluster), ord=2) < 1e-2):
        break
```

Use the following formula to calculate the distance between all points and the center and update the new cluster.

$$\mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_p \sum_q \alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$

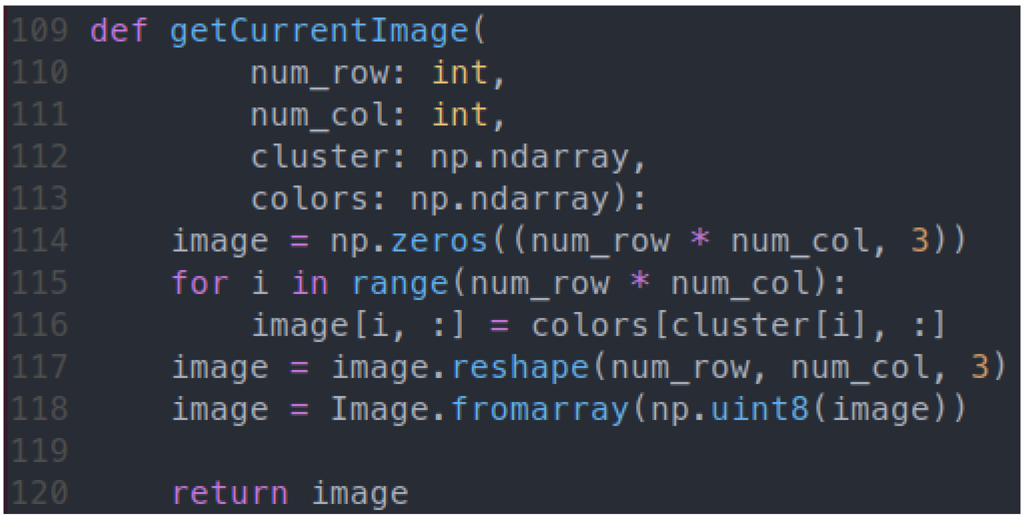
7. **setColor()**

   According to the input parameter <num_cluster> to produce the color array.



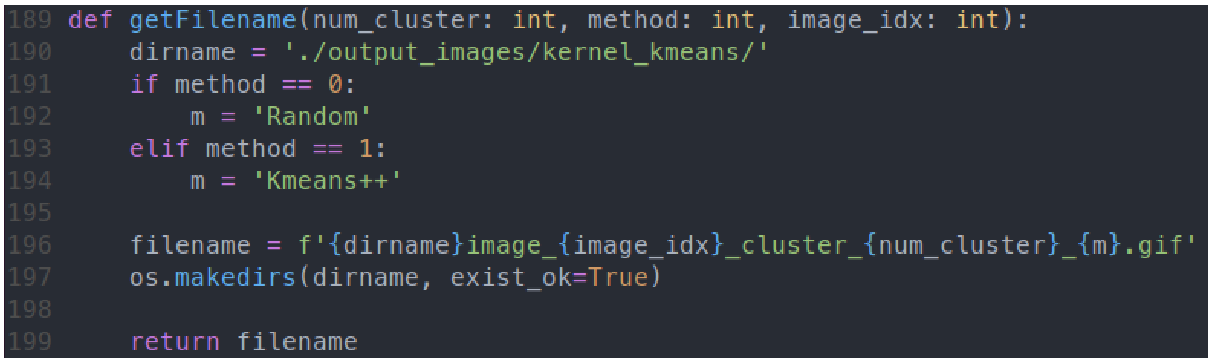
```python
def setColor(num_cluster: int):
    colors = np.array([[255, 0, 0],
                       [0, 255, 0],
                       [0, 0, 255]])
    if num_cluster > 3:
        colors = np.append(
            colors, np.random.choice(
                256, (num_cluster - 3, 3)), axis=0)

    return colors
```

8. **getCurrentImage()**
   According to the input cluster (updated each iteration), the corresponding image result is generated and returns the result.



```python
def getCurrentImage(
        num_row: int,
        num_col: int,
        cluster: np.ndarray,
        colors: np.ndarray):
    image = np.zeros((num_row * num_col, 3))
    for i in range(num_row * num_col):
        image[i, :] = colors[cluster[i], :]
    image = image.reshape(num_row, num_col, 3)
    image = Image.fromarray(np.uint8(image))

    return image
```

9. **getFilename()**
   According to the input parameter, producing the output filename.



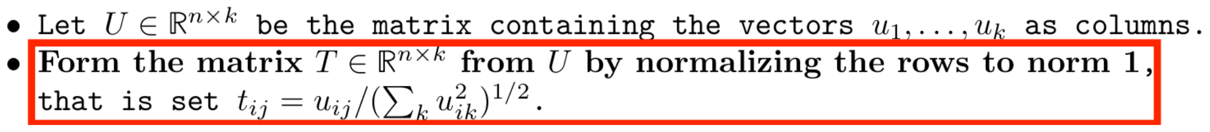
```python
def getFilename(num_cluster: int, method: int, image_idx: int):
    dirname = './output_images/kernel_kmeans/'
    if method == 0:
        m = 'Random'
    elif method == 1:
        m = 'Kmeans++'

    filename = f'{dirname}image_{image_idx}_cluster_{num_cluster}_{m}.gif'
    os.makedirs(dirname, exist_ok=True)

    return filename
```

b. Spectral clustering

1. computeKernel() in kernel_kmeans.py

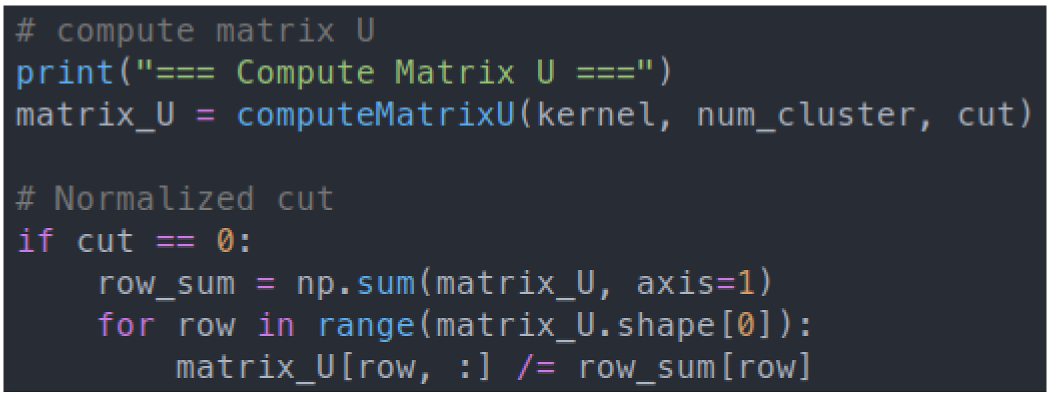
First, I compute the Gram matrix by computeKernel() which is defined in the kernel_kmeans.py. (The code paste in the previous page.)

2. computeMatrixU()



- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- $\boxed{\text{Form the matrix } T \in \mathbb{R}^{n \times k} \text{ from } U \text{ by normalizing the rows to norm 1, that is set } t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}.}$

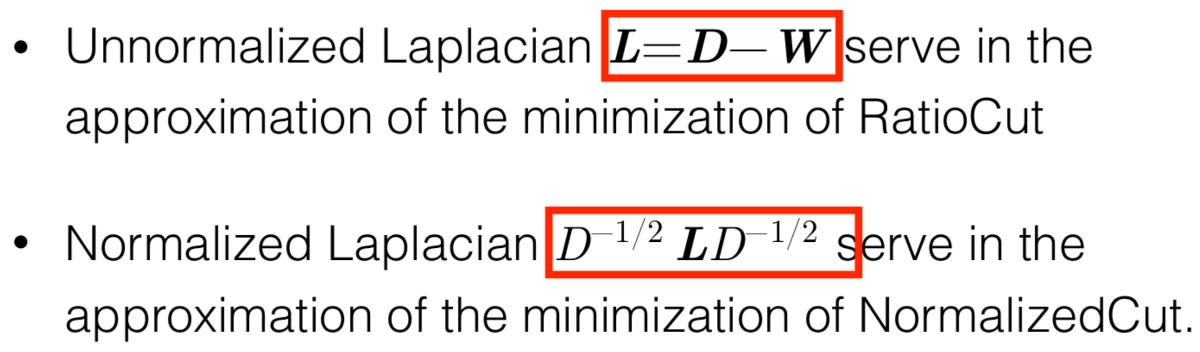
Based on the formula above, I construct the matrix U which contains all eigenvectors. If a normalized cut is used, I normalized the rows in the matrix U.



```python
# compute matrix U
print("=== Compute Matrix U ===")
matrix_U = computeMatrixU(kernel, num_cluster, cut)

# Normalized cut
if cut == 0:
    row_sum = np.sum(matrix_U, axis=1)
    for row in range(matrix_U.shape[0]):
        matrix_U[row, :] /= row_sum[row]
```

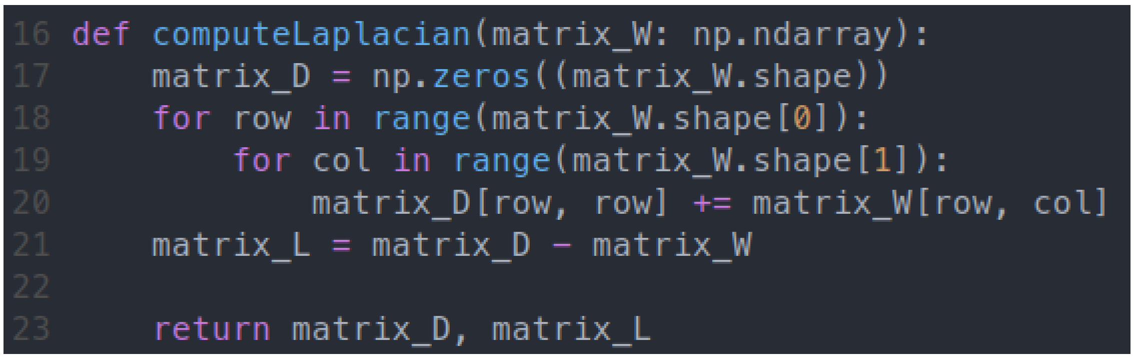
3. computeMatrixU() -> computeLaplacian()

According to the formula below,

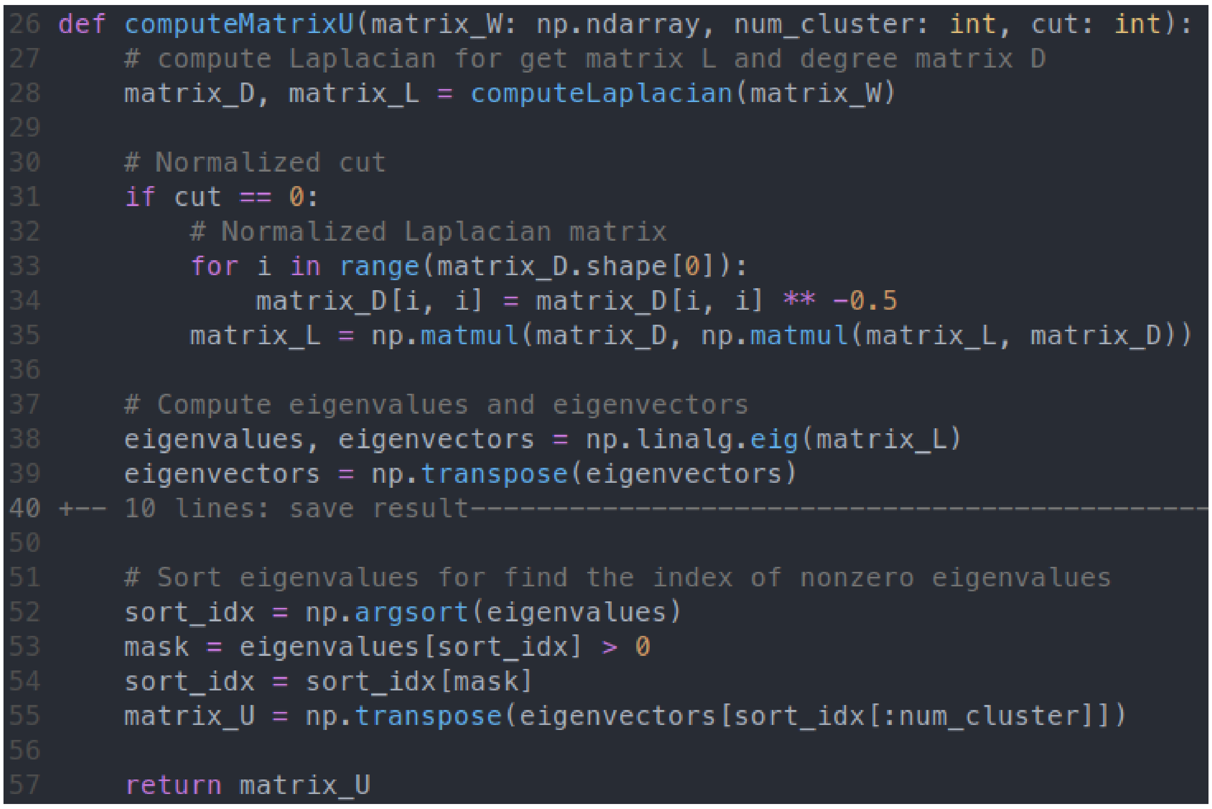


- Unnormalized Laplacian $\boxed{\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}}$ serve in the approximation of the minimization of RatioCut
- Normalized Laplacian $\boxed{D^{-1/2} \boldsymbol{L} D^{-1/2}}$ serve in the approximation of the minimization of NormalizedCut.

I compute the Laplacian matrix L and degree matrix D by computeLaplacian() function. If a normalized cut is used, the Laplacian matrix needs to be normalized after calling the function computeLaplacian(). Then, I call numpy to find the eigenvalues and eigenvectors of matrix L and return the matrix U which contains the first k eigenvectors with nonzero eigenvalues.

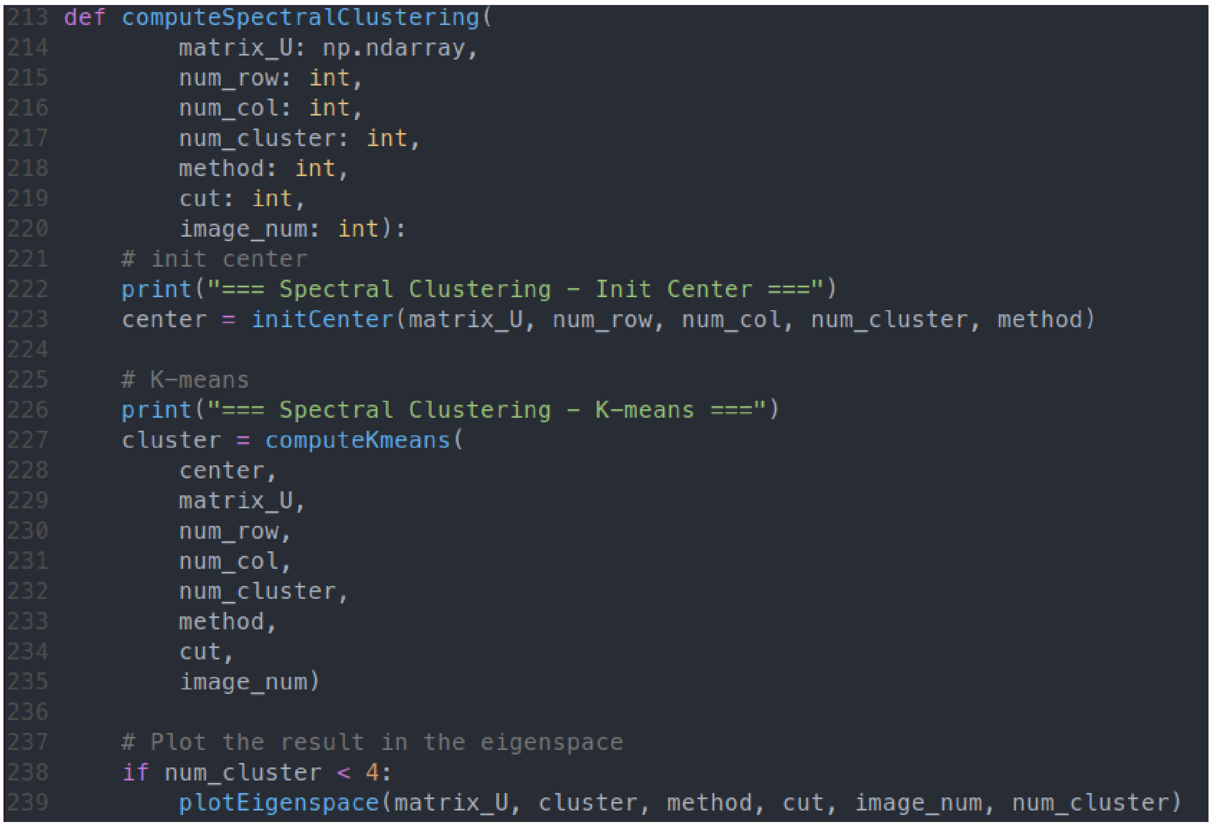


```python
def computeLaplacian(matrix_W: np.ndarray):
    matrix_D = np.zeros((matrix_W.shape))
    for row in range(matrix_W.shape[0]):
        for col in range(matrix_W.shape[1]):
            matrix_D[row, row] += matrix_W[row, col]
    matrix_L = matrix_D - matrix_W

    return matrix_D, matrix_L
```



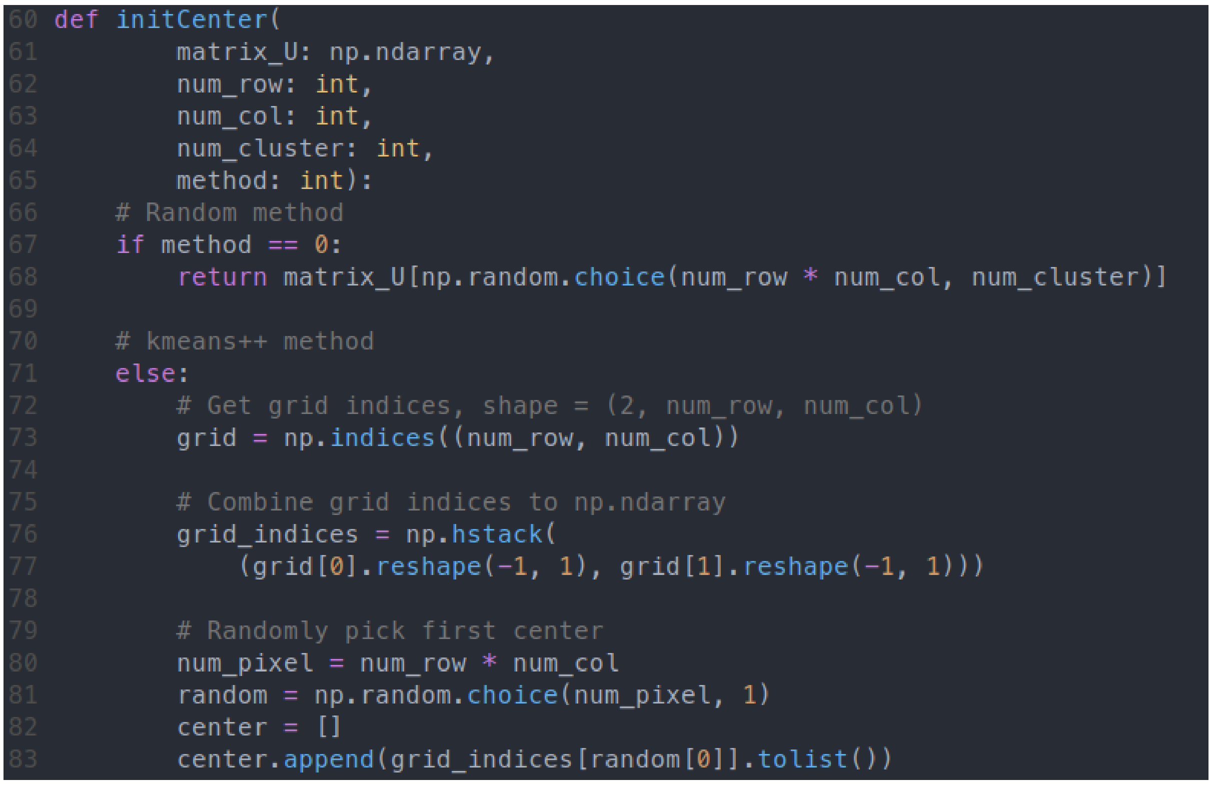
```python
def computeMatrixU(matrix_W: np.ndarray, num_cluster: int, cut: int):
    # compute Laplacian for get matrix L and degree matrix D
    matrix_D, matrix_L = computeLaplacian(matrix_W)

    # Normalized cut
    if cut == 0:
        # Normalized Laplacian matrix
        for i in range(matrix_D.shape[0]):
            matrix_D[i, i] = matrix_D[i, i] ** -0.5
        matrix_L = np.matmul(matrix_D, np.matmul(matrix_L, matrix_D))

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(matrix_L)
    eigenvectors = np.transpose(eigenvectors)
+-- 10 lines: save result--------------------------------------------

    # Sort eigenvalues for find the index of nonzero eigenvalues
    sort_idx = np.argsort(eigenvalues)
    mask = eigenvalues[sort_idx] > 0
    sort_idx = sort_idx[mask]
    matrix_U = np.transpose(eigenvectors[sort_idx[:num_cluster]])

    return matrix_U
```

4. **computeSpectralClustering()**
   It uses matrix U to compute init centers according to the given method and compute k-means to get the final convergence cluster. Finally, it plots all data points in the eigenspace.

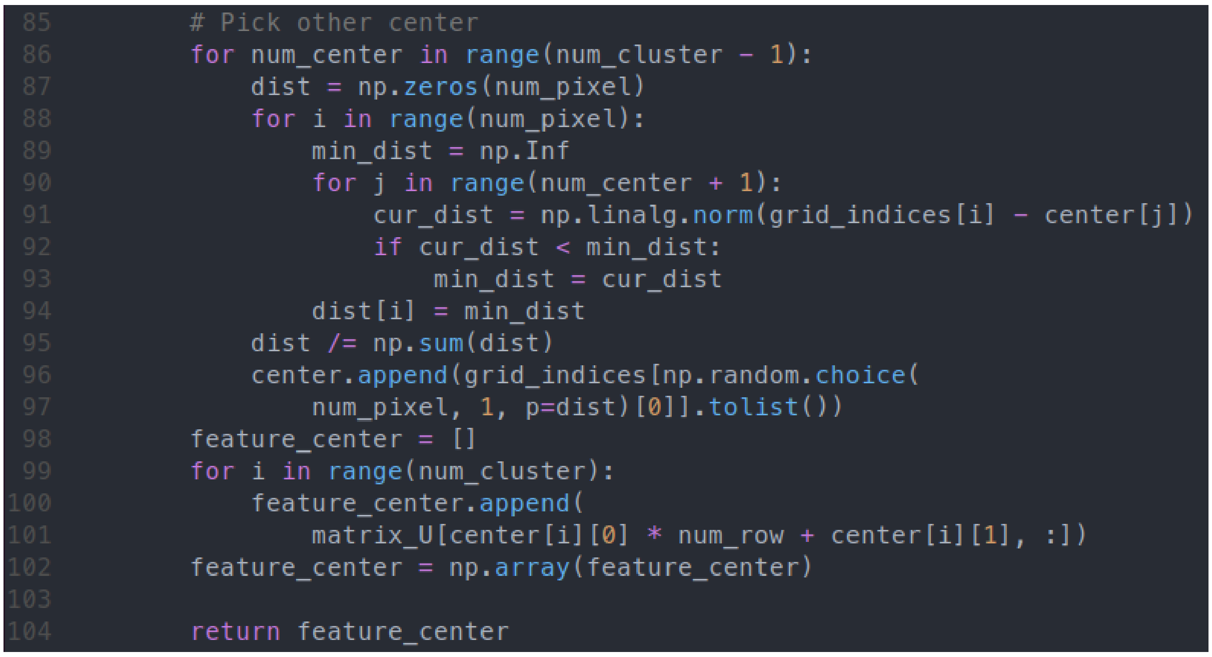


```python
def computeSpectralClustering(
        matrix_U: np.ndarray,
        num_row: int,
        num_col: int,
        num_cluster: int,
        method: int,
        cut: int,
        image_num: int):
    # init center
    print("=== Spectral Clustering - Init Center ===")
    center = initCenter(matrix_U, num_row, num_col, num_cluster, method)

    # K-means
    print("=== Spectral Clustering - K-means ===")
    cluster = computeKmeans(
        center,
        matrix_U,
        num_row,
        num_col,
        num_cluster,
        method,
        cut,
        image_num)

    # Plot the result in the eigenspace
    if num_cluster < 4:
        plotEigenspace(matrix_U, cluster, method, cut, image_num, num_cluster)
```

5. **computeSpectralClustering() -> initCenter()**
   Based on the input parameter "method", I will use a random method or k-means++ method to initialize centers. This function will return the coordinates of all centers in the eigenspace.

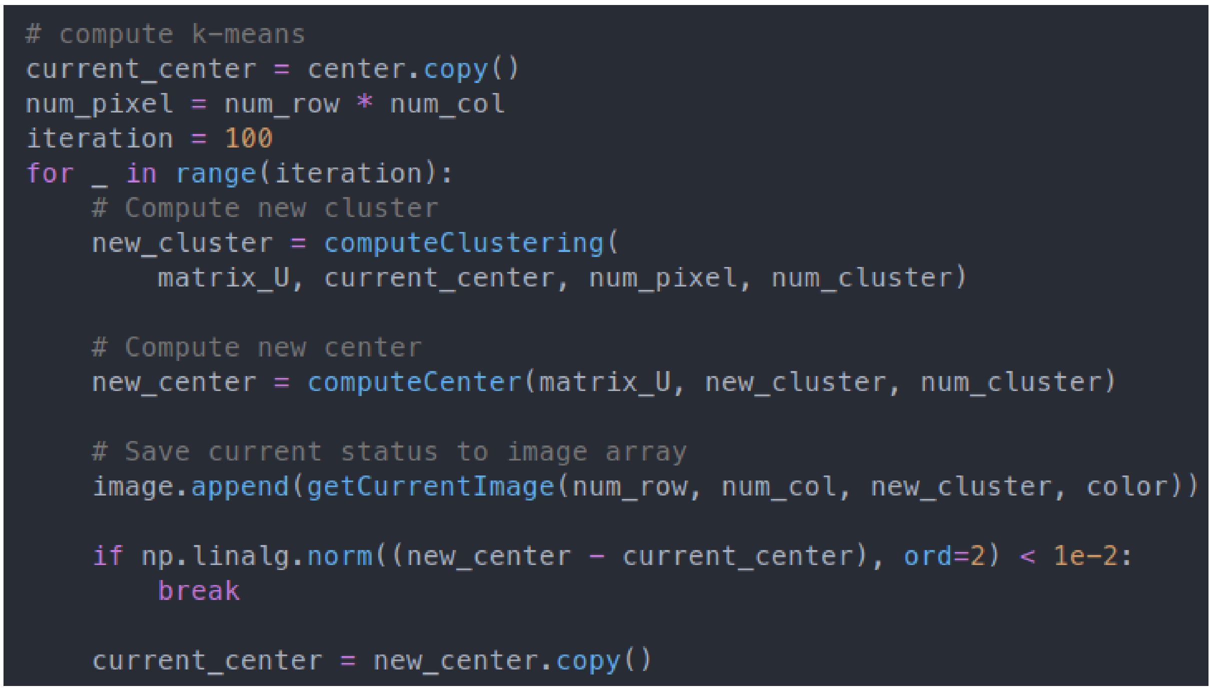


```python
def initCenter(
        matrix_U: np.ndarray,
        num_row: int,
        num_col: int,
        num_cluster: int,
        method: int):
    # Random method
    if method == 0:
        return matrix_U[np.random.choice(num_row * num_col, num_cluster)]

    # kmeans++ method
    else:
        # Get grid indices, shape = (2, num_row, num_col)
        grid = np.indices((num_row, num_col))

        # Combine grid indices to np.ndarray
        grid_indices = np.hstack(
            (grid[0].reshape(-1, 1), grid[1].reshape(-1, 1)))

        # Randomly pick first center
        num_pixel = num_row * num_col
        random = np.random.choice(num_pixel, 1)
        center = []
        center.append(grid_indices[random[0]].tolist())
```



```python
        # Pick other center
        for num_center in range(num_cluster - 1):
            dist = np.zeros(num_pixel)
            for i in range(num_pixel):
                min_dist = np.Inf
                for j in range(num_center + 1):
                    cur_dist = np.linalg.norm(grid_indices[i] - center[j])
                    if cur_dist < min_dist:
                        min_dist = cur_dist
                dist[i] = min_dist
            dist /= np.sum(dist)
            center.append(grid_indices[np.random.choice(
                num_pixel, 1, p=dist)[0]].tolist())
        feature_center = []
        for i in range(num_cluster):
            feature_center.append(
                matrix_U[center[i][0] * num_row + center[i][1], :])
        feature_center = np.array(feature_center)

        return feature_center
```

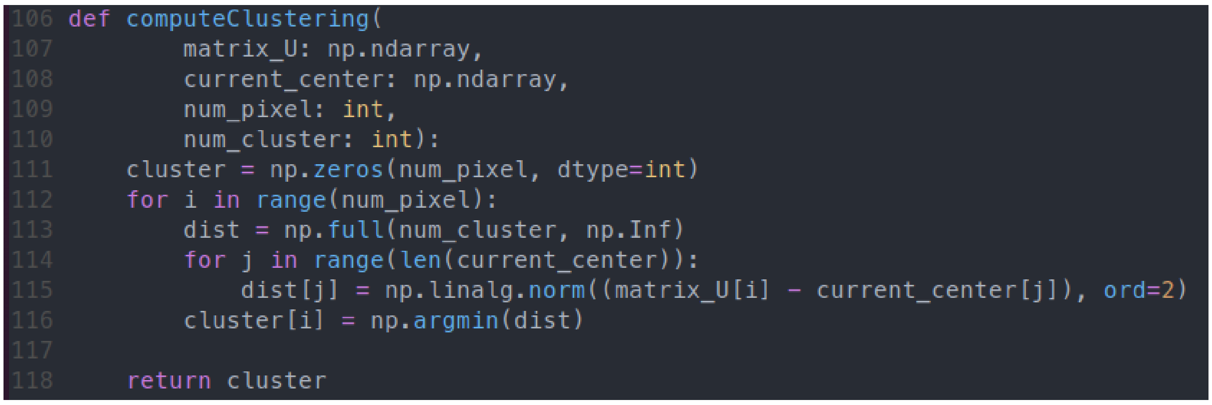
6. **computeSpectralClustering() -> computeKmeans()**
   After getting init centers, it can compute k-means to find the cluster of all points. In each iteration, it calls the function **computeClustering()** to update the clusters by input matrix U first. Then, it calls the function **computeCenter()** which uses new clustering to update the new centers. Next, it calls the function **getCurrentImage()** which uses given clustering to produce the image of current status and appends the image to the image array to produce the result gif graph. Finally, it checks the difference between the new centers and the old centers, if the difference is small enough(it means converges), then break the loop.



```python
# compute k-means
current_center = center.copy()
num_pixel = num_row * num_col
iteration = 100
for _ in range(iteration):
    # Compute new cluster
    new_cluster = computeClustering(
        matrix_U, current_center, num_pixel, num_cluster)

    # Compute new center
    new_center = computeCenter(matrix_U, new_cluster, num_cluster)

    # Save current status to image array
    image.append(getCurrentImage(num_row, num_col, new_cluster, color))

    if np.linalg.norm((new_center - current_center), ord=2) < 1e-2:
        break

    current_center = new_center.copy()
```

computeClustering() in computeKmeans()

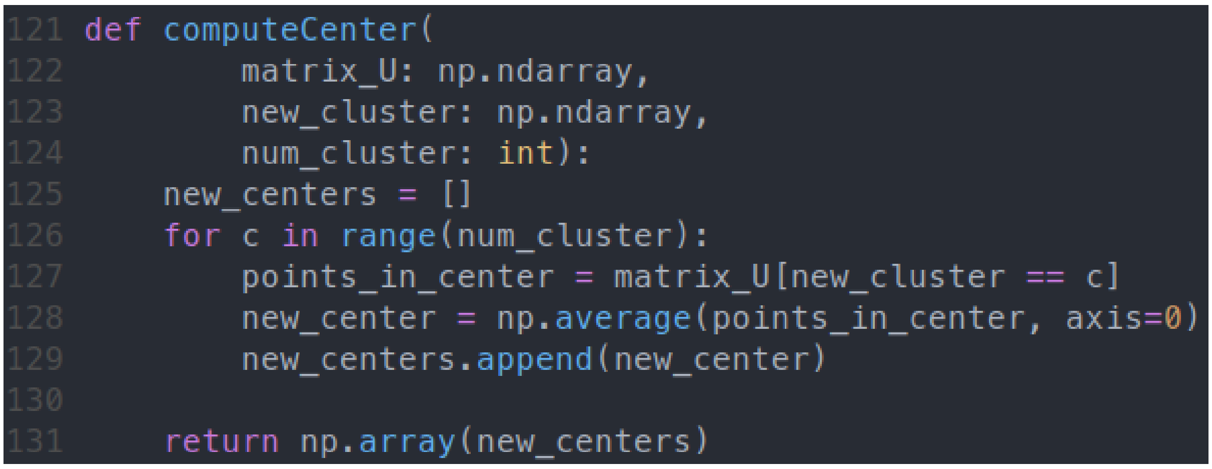
In the function computeClustering(), It computes the distance in eigenspace between all data points and all centers to classify all data points in new clustering.



```python
def computeClustering(
        matrix_U: np.ndarray,
        current_center: np.ndarray,
        num_pixel: int,
        num_cluster: int):
    cluster = np.zeros(num_pixel, dtype=int)
    for i in range(num_pixel):
        dist = np.full(num_cluster, np.Inf)
        for j in range(len(current_center)):
            dist[j] = np.linalg.norm((matrix_U[i] - current_center[j]), ord=2)
        cluster[i] = np.argmin(dist)

    return cluster
```

computeCenter() in computeKmeans()

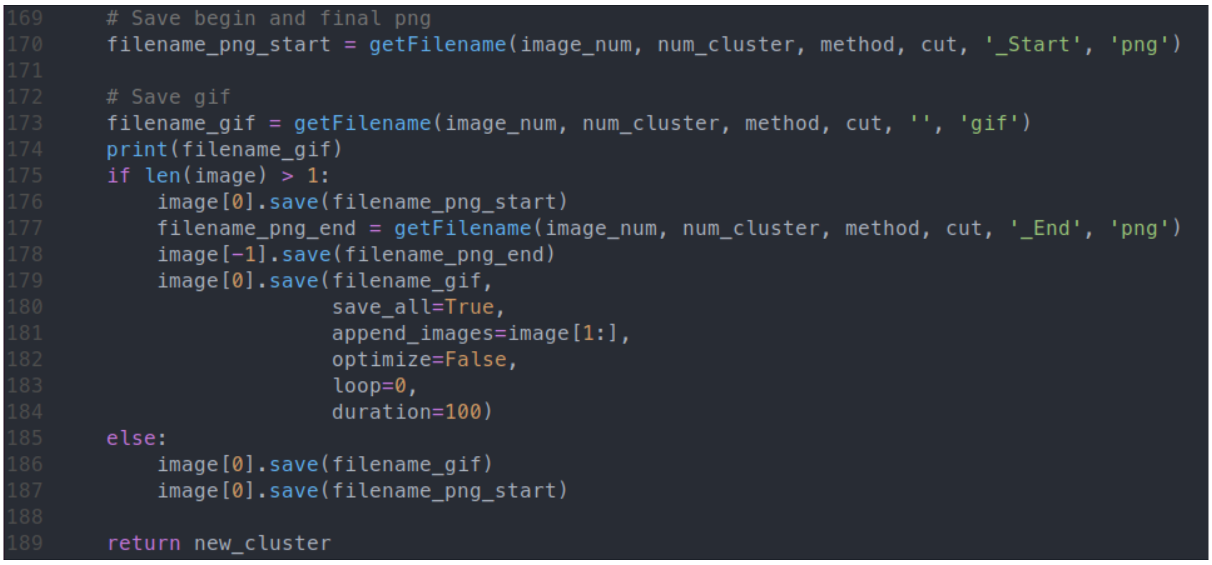
It computes the mean in each cluster and chooses the new center.



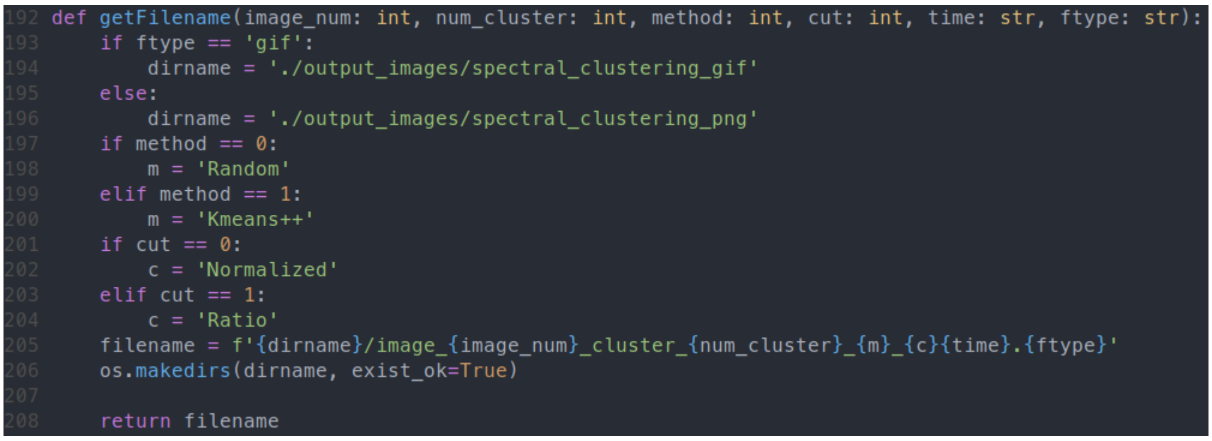
```python
def computeCenter(
        matrix_U: np.ndarray,
        new_cluster: np.ndarray,
        num_cluster: int):
    new_centers = []
    for c in range(num_cluster):
        points_in_center = matrix_U[new_cluster == c]
        new_center = np.average(points_in_center, axis=0)
        new_centers.append(new_center)

    return np.array(new_centers)
```

save result in computeKmeans()

After the clusters are stable(converge), all images resulting in each iteration are saved in the image array. Then I call the function getFilename() which, according to the input parameter, produces the appropriate filename, and produces the image of the final result.



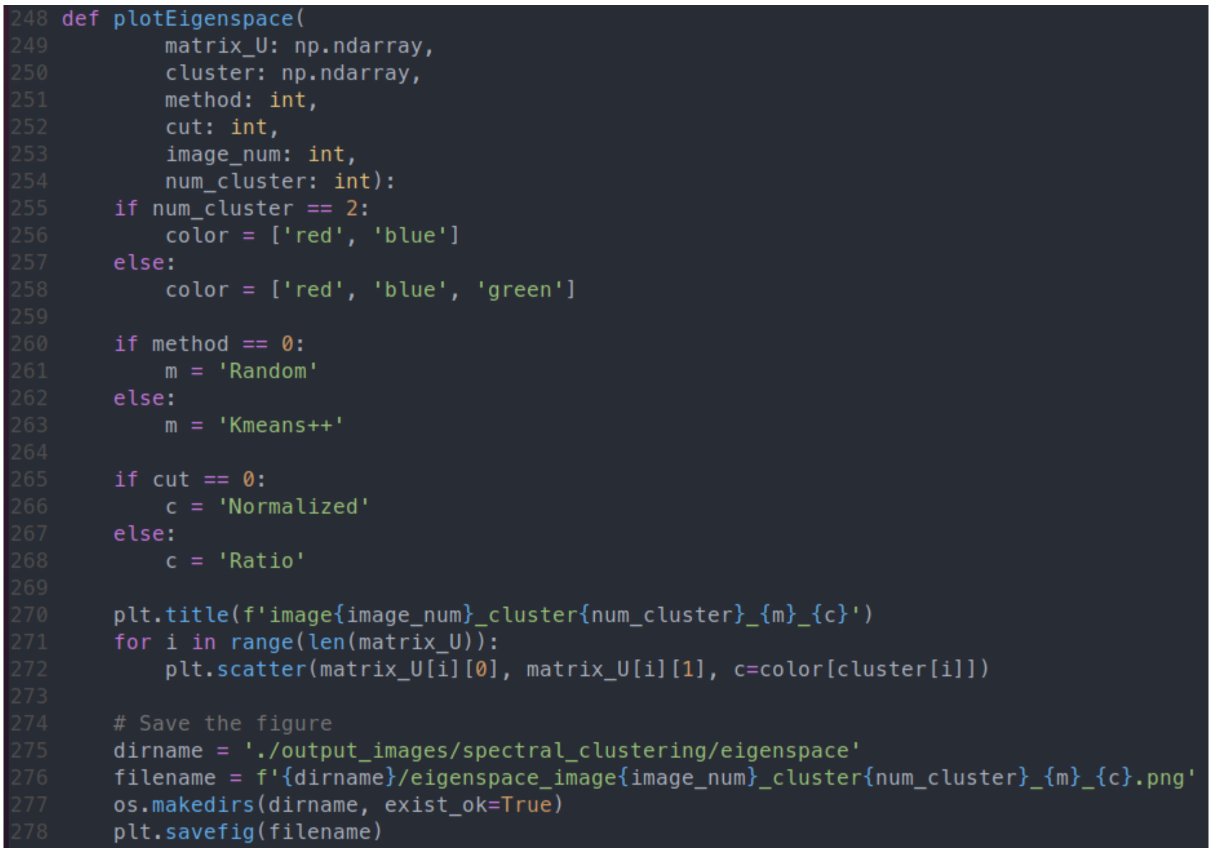
```python
    # Save begin and final png
    filename_png_start = getFilename(image_num, num_cluster, method, cut, '_Start', 'png')

    # Save gif
    filename_gif = getFilename(image_num, num_cluster, method, cut, '', 'gif')
    print(filename_gif)
    if len(image) > 1:
        image[0].save(filename_png_start)
        filename_png_end = getFilename(image_num, num_cluster, method, cut, '_End', 'png')
        image[-1].save(filename_png_end)
        image[0].save(filename_gif,
                      save_all=True,
                      append_images=image[1:],
                      optimize=False,
                      loop=0,
                      duration=100)
    else:
        image[0].save(filename_gif)
        image[0].save(filename_png_start)

    return new_cluster
```



```python
def getFilename(image_num: int, num_cluster: int, method: int, cut: int, time: str, ftype: str):
    if ftype == 'gif':
        dirname = './output_images/spectral_clustering_gif'
    else:
        dirname = './output_images/spectral_clustering_png'
    if method == 0:
        m = 'Random'
    elif method == 1:
        m = 'Kmeans++'
    if cut == 0:
        c = 'Normalized'
    elif cut == 1:
        c = 'Ratio'
    filename = f'{dirname}/image_{image_num}_cluster_{num_cluster}_{m}_{c}{time}.{ftype}'
    os.makedirs(dirname, exist_ok=True)

    return filename
```

7. plotEigenspace()

Plot all data points in the eigenspace.

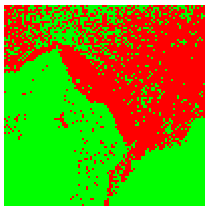
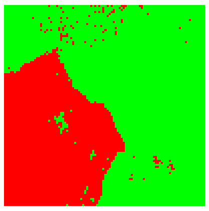
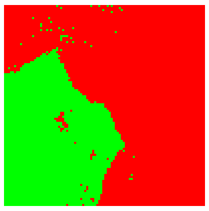
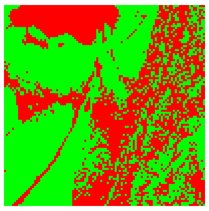
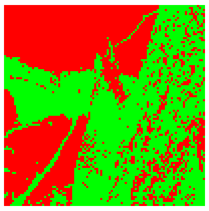
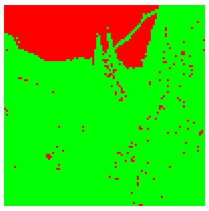


```python
def plotEigenspace(
        matrix_U: np.ndarray,
        cluster: np.ndarray,
        method: int,
        cut: int,
        image_num: int,
        num_cluster: int):
    if num_cluster == 2:
        color = ['red', 'blue']
    else:
        color = ['red', 'blue', 'green']

    if method == 0:
        m = 'Random'
    else:
        m = 'Kmeans++'

    if cut == 0:
        c = 'Normalized'
    else:
        c = 'Ratio'

    plt.title(f'image{image_num}_cluster{num_cluster}_{m}_{c}')
    for i in range(len(matrix_U)):
        plt.scatter(matrix_U[i][0], matrix_U[i][1], c=color[cluster[i]])

    # Save the figure
    dirname = './output_images/spectral_clustering/eigenspace'
    filename = f'{dirname}/eigenspace_image{image_num}_cluster{num_cluster}_{m}_{c}.png'
    os.makedirs(dirname, exist_ok=True)
    plt.savefig(filename)
```

B. experiments settings and results (20%) & discussion (30%)

Part1.

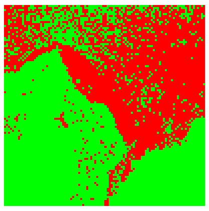
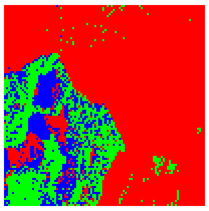
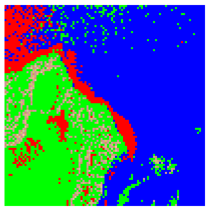
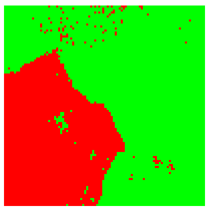
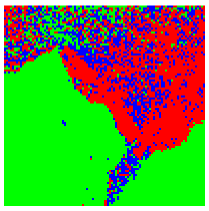
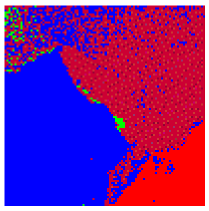
K = 2, choose center method: random

Image / method	Kernel K means	Spectral Clustering: Ratio cut	Spectral Clustering: Normalized Cut
Image 1			
Image 2			

Part2.

K = 2, 3, 4, choose center method: random

Image 1

method / K	2	3	4
Kernel K means			
Spectral Clustering: Ratio cut			

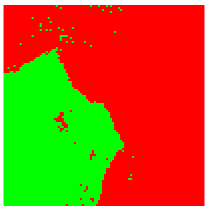
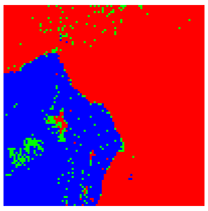
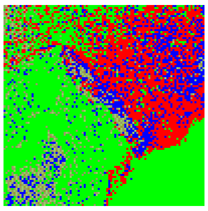
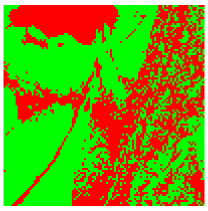
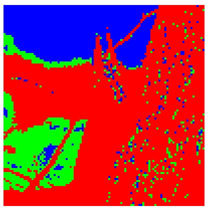
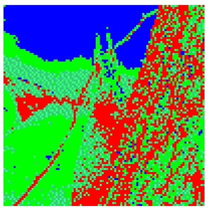
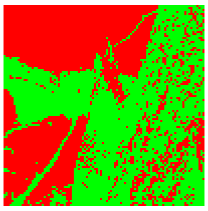
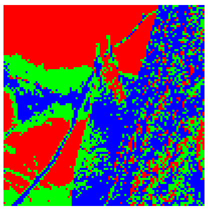
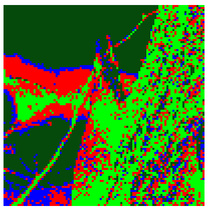
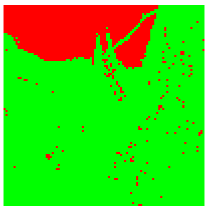
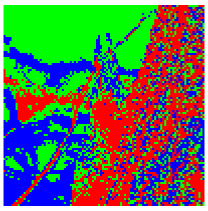
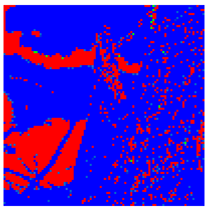
Spectral Clustering: Normalized Cut			

Image2

method / K	2	3	4
Kernel K means			
Spectral Clustering: Ratio cut			
Spectral Clustering: Normalized Cut			

Part3.
K = 2

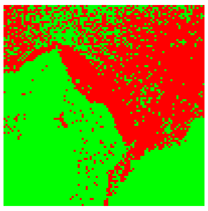
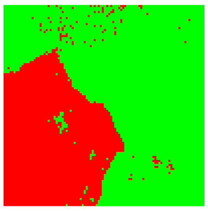
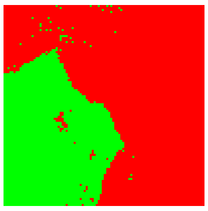
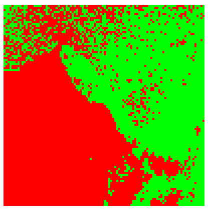
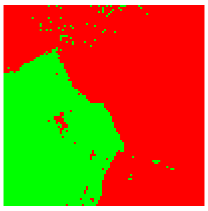
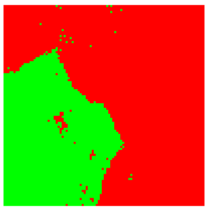
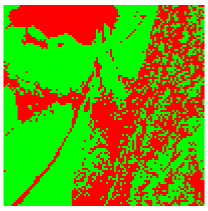
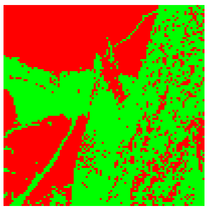
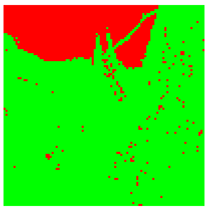
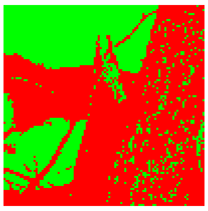
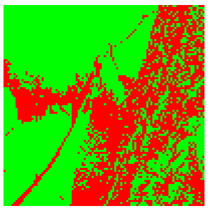
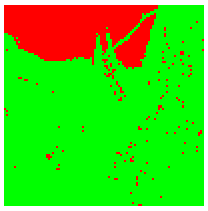
	Kernel K means	Spectral Clustering: Ratio cut	Spectral Clustering: Normalized Cut
image1: random			

image1: k-means++			
image2: random			
image2: k-means++			

K = 3

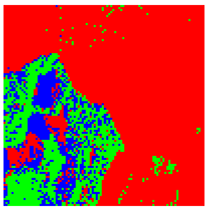
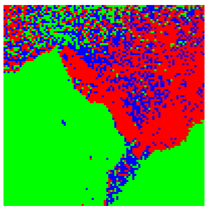
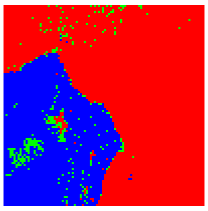
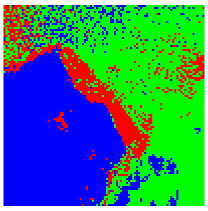
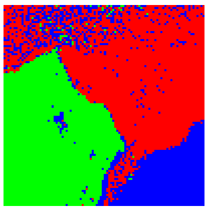
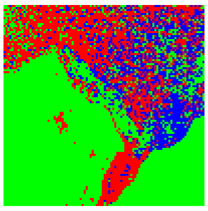
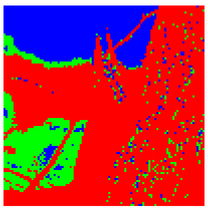
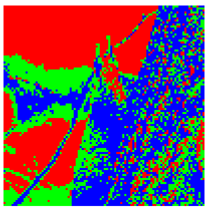
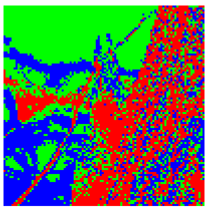
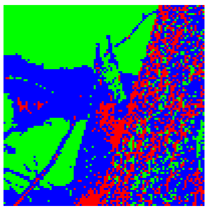
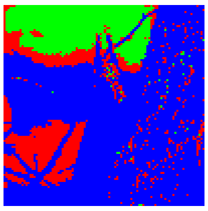
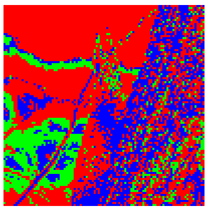
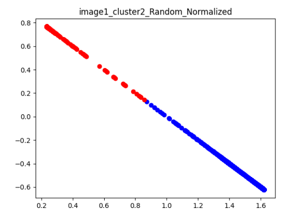
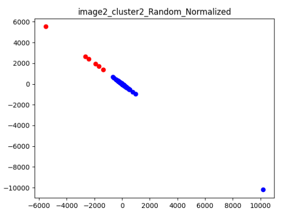
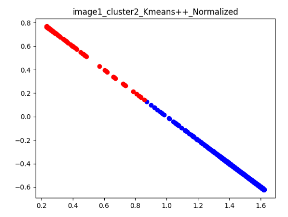
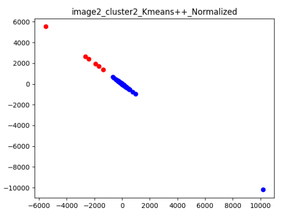
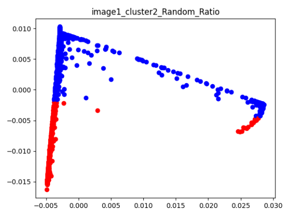
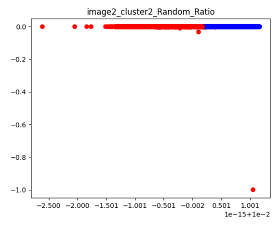
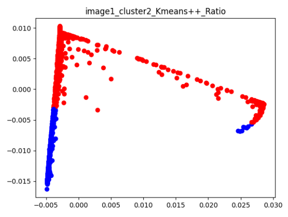
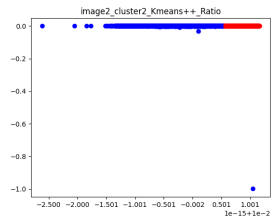
	Kernel K means	Spectral Clustering: Ratio cut	Spectral Clustering: Normalized Cut
image1: random			
image1: k-means++			
image2: random			

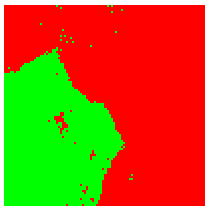
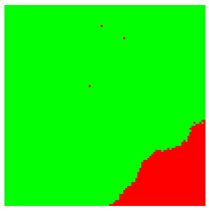
image2: k-means++			

Part4.
K = 2

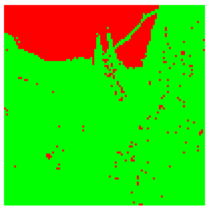
	image1	image2
Normalized Cut + random		
Normalized Cut + k-means++		
Ratio Cut + random		
Ratio Cut + k-means++		

C. observations and discussion (10%)

a. First of all, the clustering results are not bad. We can see the approximate outline of the original image from the classification result(eg. rabbit shape and coastline).
b. The Random method needs more iterations to converge than the k-means++ method.
c. Although the k-means method will converge faster than the random method, the result may be worse than before. As we can see in the figure below, when the k-means method converges and breaks the loop, almost all data points have been classified into the same clustering.

   Start: , End: 

d. The k-means++ method can get better initial clustering than the random method.
e. If k > 5, some clustering have only a few points.
f. The normalized cut method seems to not classify the data points well. The result of normalized cut will classify a lot of data points into the same clustering.

   Strat: , End: