 **The gif animations and some markdown notations just won't work on pdf file! So please refer to my hackmd or gif files in the zip file handed in to check the animations!**

my hackmd: <https://hackmd.io/ssPhbafhQL-PYuvMMB49wQ> (<https://hackmd.io/ssPhbafhQL-PYuvMMB49wQ>)

## Code

### Part 1

#### Kernel K-means

##### *The main structure*

```
1  if __name__ == "__main__":
2      parser = argparse.ArgumentParser()
3      parser.add_argument("--i", help = "path of the input image",
4                          type = str, default = "data/image1.jpg")
5      parser.add_argument("--o", help = "path of the output directory",
6                          type = str, default = "output_kernel_kmeans/output1")
7      parser.add_argument("--k", help = "number of clusters",
8                          type = int, default = 4)
9      parser.add_argument("--s", help = "parameter gamma_s of the mix RBF kernel",
10                         type = float, default = 0.001)
11     parser.add_argument("--c", help = "parameter gamma_c of the mix RBF kernel",
12                         type = float, default = 0.001)
13     parser.add_argument("--m", help = "mode to use in initial clustering, 3 available
14                             random, normal, kmeans+", type = str, default = "kmeans+")
15     parser.add_argument("--epoch", help = "how many epochs at most",
16                         type = int, default = 20)
17     args = parser.parse_args()
18
19     gamma_s = args.s
20     gamma_c = args.c
21     K = args.k
22     mode = args.m
23     epoch = args.epoch
24     output_dir = (args.o).split(sep = "/")[0]
25     output_dir_img = (args.o).split(sep = "/")[1]
26     if not os.path.exists(output_dir):
27         try:
28             os.mkdir(output_dir)
29         except:
30             raise OSError("Failed to construct the output directory {} !"
31                             .format(output_dir))
32     if not os.path.exists(os.path.join(output_dir, output_dir_img)):
33         try:
34             os.mkdir(os.path.join(output_dir, output_dir_img))
35         except:
36             raise OSError("Failed to construct the output directory {} !"
37                             .format(output_dir_img))
38     image, coords, w, h = load_image(args.i)
39     Kernel_Kmeans(args.o, image, coords, w, h, gamma_s, gamma_c, K, mode, epoch)
```

Firstly, by module ***argparse***, we can pass some arguments to the python file, including

1. **-i**: the path of input image, one at a time.
2. **-o**: the path of output directory. The result gif file will be stored in here.
3. **-k**: the number of clusters to be used. Default to be **4**.
4. **-s**: the hyper-parameter of the kernel used, which is related to *spatial information*. See the section **function: RBF\_kernel\_mix()** later for more details. Default to be **0.001**.
5. **-c**: the hyper-parameter of the kernel used, which is related to *color information*. See the section **function: RBF\_kernel\_mix()** later for more details. Default to be **0.001**.
6. **-m**: the method to initialize kernel k-means. There are 3 ways available: **random**, **normal** and **kmeans++**. See the section **function: get\_initial\_center()** later for more details. Default to be **kmeans++**.
7. **-epoch**: the maximum iterations when doing the clustering processes. Default to be **20**.

In our code, the hyper-parameters from **-s** and **-c** will be passed to variables `gamma_s` and `gamma_c`, respectively. The cluster number will be stored in `κ`. The initialize method for kernel k-means will be stored in `mode`. Finally, the iteration epochs will be stored in `epoch`.

We will automatically create new folders for the output path if it does not exist.

Basically, we use function `load_image()` to load the image from the input path, and return the pixels array, the coordinates array for each pixels, the width and height of the image.

Then, we pour all the paths, arrays and other parameters into the function `Kernel_Kmeans()` to execute the kernel k-means clustering.

All the details for clustering will be given in the introductions of these functions.

**function: load\_image()**

```

1  def load_image(path: str):
2      try:
3          img = Image.open(path)
4          img = img.convert('RGB')
5      except:
6          raise IOError("Failed to open the image {} !".format(path))
7
8      w, h = img.size
9
10     image = np.array(img.getdata()).reshape((w*h, 3))
11     coords = np.empty((0, 2))
12     for i in range(h):
13         for j in range(w):
14             coords = np.append(coords, [[i,j]], axis = 0)
15
16     return image, coords, w, h

```

In this function, we will open the input image from the path given. We will make sure that color channels are arranged in **R-G-B** order.

We will also obtain and return the weight and height of the image. These 2 values will be used when conducting visualization later on.

We return the image array `image`, which has the shape of  $(W \times H, 3)$ . In this case, both images are  $100 \times 100$ , thus `image` has the shape of  $(10000, 3)$ . `image` contains the **color information**.

As for the **spatial information**, we will generate and return a coordinate array `coords` containing the positions of each pixels. `coords` has the shape of  $(W \times H, 2)$ , in this case,  $(10000, 2)$ . It looks like this:

```

1  array([[0, 0],[0, 1],...,[0, 99],[1, 0],...,[99, 99]])

```

### ***function: Kernel\_Kmeans()***

This functions is the whole processes of kernel k-means.

```

1 def Kernel_Kmeans(output_path, image, coords, w, h, gamma_s, gamma_c, K, mode, epc
2     image_list = []
3     n = image.shape[0]
4
5     kernel = RBF_kernel_mix(image, coords, gamma_s, gamma_c)
6
7     initial_centers = get_initial_center(K, n, kernel, mode)
8
9     threshold = 1e-5
10    for e in range(1, epoch+1):
11        final_clusters = np.empty(n, dtype = np.uint8)
12
13        # E-step
14        for i in range(n):
15            dist = []
16            for k in range(K):
17                dist.append(np.linalg.norm(kernel[i, :] - initial_centers[k, :]))
18            dist = np.array(dist)
19            final_clusters[i] = np.argmin(dist)
20
21        # M-step
22        final_centers = np.zeros_like(initial_centers)
23        for k in range(K):
24            masked = np.where(final_clusters == k, True, False)
25            final_centers[k] = np.sum(kernel[mask, :], axis = 0)
26            if np.sum(masked) > 0:
27                final_centers[k] /= np.sum(masked)
28
29        if (np.linalg.norm(final_centers - initial_centers) < threshold):
30            break
31
32        img = visualize(n, w, h, final_clusters, e, output_path)
33        image_list.append(img)
34        initial_centers = final_centers
35
36    image_list[0].save(os.path.join(output_path, "GIF.gif"), save_all = True,
37                      append_images = image_list[1:], duration = 200, loop = 0)

```

**Some preparation:** `image_list` is a list storing all images generated in each iteration of clustering processes. This will be used to generate a gif animation file.

`n` is the total number of pixels, 10000 in this case.

First, we pass the color and spatial information of the image, `image` and `coords`, as well as the hyper-parameters `gamma_s` and `gamma_c` into the function

`RBF_kernel_mix()`, which will return the Gram matrix of kernel functions. We will store that in the variable `kernel`. More detailed explanations regarding the kernel will be given in the section: **function: RBF\_kernel\_mix()** later on.

And then, we can use the kernel `kernel` with the initialization method specified, which is default to

**kmeans++** in `mode`, to calculate the initial  $K$  cluster centers, `initial_centers`, by the help of function **get\_initial\_center()**. Again,  $K$  is default to be 4.

`initial_centers` has the shape of  $(K, 10000)$ ,  $K = 4$  by default.

More information regarding the function and all the candidate initialization methods will be described later on in the section **function: get\_initial\_center()**.

As long as we have the initial cluster centers, we can do the kernel k-means clustering processes, which are the codes in the *for loop*.

In **expectation step**, for each pixel in kernel space, we calculate its distances between the centers of all  $K$  current clusters. We then pick the cluster whose center in kernel space has the minimum distance among all  $K$  clusters from the pixel. The predicted cluster belonging for all 10000 pixels will be stored in the (10000, ) array `final_clusters` .

In **maximization step**, we calculate the mean of all data points in each cluster, and obtain the new centers of all  $K$  clusters, which are stored in `final_centers` . The `final_centers` has the same shape as `initial_centers` ,  $(K, 10000)$ ,  $K = 4$  by default.

In each iteration, after both E-step and M-step are done, we will check if the clustering processes converged, by calculating the **Frobenius norm** of the difference between `initial_centers` and `final_centers` , since they both are matrixes. The threshold is set to be  $10^{-9}$  in `threshold` . If the norm is less than the threshold, we consider it converged.

If the process is not converged, we then do the visualization based on the clustering results in `final_clusters` by the function **visualize()**. The function will return an image array constructed by the clustering results. We will store it in `image_list` for generating a gif later on.

The details concerning visualization will be listed in the later section: **function: visualize()**.

Finally, set the `initial_centers` as the current `final_centers` , and go into the next iteration.

Outside the for loop, we use all the clustering result images in `image_list` to make a gif animation.

### ***function: RBF\_kernel\_mix()***

This function is to calculate the Gram matrixes of kernel functions based on the color information and the spatial information.

```
1 def RBF_kernel_mix(image, coords, gamma_s, gamma_c):
2     dist_coors = cdist(coords, coords, "sqeuclidean") # 10000*10000
3     dist_colors = cdist(image, image, "sqeuclidean")
4
5     RBF_coors = np.exp(-gamma_s * dist_coors)
6     RBF_colors = np.exp(-gamma_c * dist_colors)
7     mix_kernel = np.multiply(RBF_coors, RBF_colors)
8
9     return mix_kernel
```

The kernel used here is a mix kernel of two RBF kernels, one considering the spatial information and the other considering the color information.

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

with  $S(x)$  being the **spatial** information, i.e. the **coordinate** of the pixel  $x$  (a 2d vector in `coords`) and  $C(x)$  being the **color** information, i.e. the **RGB** values of the pixel  $x$  (a 3d vector in `image`).

As for the hyper-parameters regarding the spatial kernel and the color kernel,  $\gamma_s$  and  $\gamma_c$ , they are denoted as `gamma_s` and `gamma_c` in my code. Both the parameters are default to **0.001**.

### ***function: get\_initial\_center()***

This function is used to generate  $K$  initial cluster centers by different initialization methods.

```

1 def get_initial_center(K, n, kernel, mode):
2     if mode == "random":
3         initial_cluster = kernel[list(random.sample(range(0, n), K)), :]
4     elif mode == "normal":
5         initial_cluster = np.empty((K, n))
6         mean_vector = np.mean(kernel, axis = 0)
7         std_vector = np.std(kernel, axis = 0)
8         for feature in range(n):
9             initial_cluster[:, feature] = np.random.normal(mean_vector[feature],
10                 std_vector[feature], K)
11     elif mode == "kmeans++":
12         clusters = []
13         clusters.append(random.randint(0, n-1))
14         cluster_num = 1
15         while (cluster_num < K):
16             dist = np.empty((n, cluster_num))
17             for i in range(n):
18                 for c in range(cluster_num):
19                     dist[i, c] = np.linalg.norm(kernel[i, :]-kernel[clusters[c], :
20                 dist_min = np.min(dist, axis = 1)
21                 sum_rand = np.sum(dist_min) * np.random.rand()
22                 for i in range(n):
23                     sum_rand -= dist_min[i]
24                     if sum_rand <= 0:
25                         clusters.append(i)
26                         break
27                 cluster_num += 1
28             initial_cluster = kernel[clusters, :]
29     else:
30         raise ValueError("Unknown clustering mode: {}".format(mode))
31
32     return initial_cluster

```

We provide 3 different initialization methods: **random**, **normal** and **kmeans++**.

#### <**random** mode>

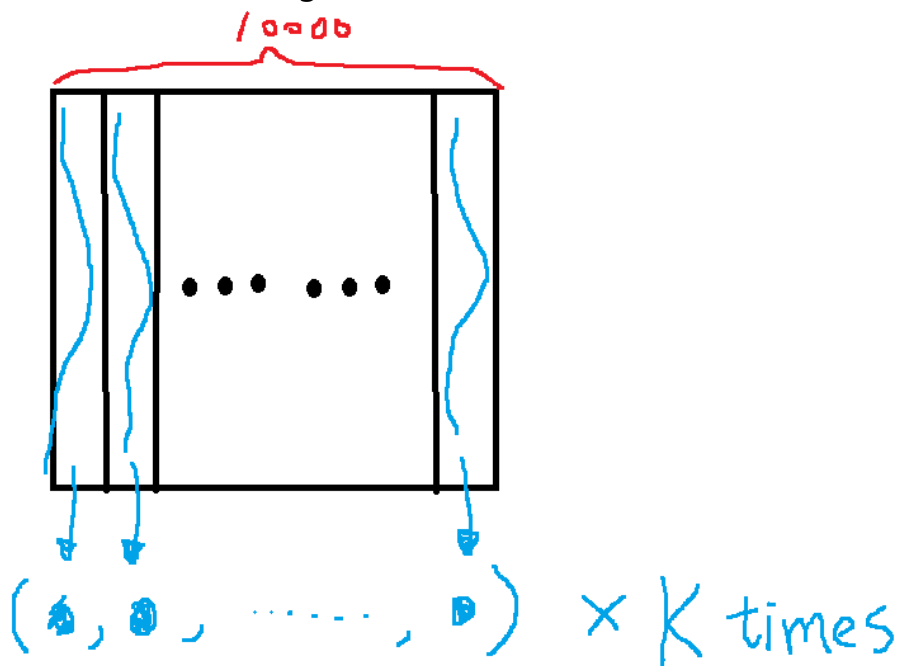
In random mode, the centers of all  $K$  clusters are **randomly chosen(sampled)** from  $K$  pixels in kernel spaces `kernel` .

<**normal** mode>

In normal mode, we generate all centers based on the **Gaussian distributions** of each 10000 *features*(dimensions) in kernel spaces `kernel` .

This is done by first calculate the **mean** and **standard deviation** of each feature(dimension). Now, for each dimension, we can fit a normal distribution with the mean and standard deviation calculated.

To generate a center of a cluster, we just need to sample a value from each of the 10000 normal distributions of each feature(dimension). Do this  $K$  times, we can then generate centers of  $K$  clusters.





<**kmeans++ mode**>

In k-means++ mode, we use a modified initialization method from K-means++ clustering algorithm. The concept is to randomly choose these cluster centers as well, but *the pixel which is far from all previously chosen cluster centers should be more probable to be picked.*

The first cluster center is randomly picked from all the pixels.

As for the second to the  $K$ -th cluster centers, suppose that we're currently generating the  $i$ -th center, which means that we already generated  $i - 1$  cluster centers.

The algorithm processes are as follows:

1. For each pixel in kernel space, calculate its distances from the  $i - 1$  chosen cluster centers, and then record the **minimum** distance.
2. **sum up** the minimum distances of all 10000 pixels, and then **multiply** it by a **random floating point number within [0,1) range**.
3. Use the **roulette method** to decide which pixel to choose as the  $i$ -th cluster center.

If the value from step 2 is equal to or less than the minimum distance of the current pixel, **choose current pixel as the new center**. Otherwise, subtract the minimum distance of the current pixel from the value in step 2 and **move on to check the next pixel**.

**function: visualize()**

This function is to visualize the clustering results and return as an image array.

```

1  def visualize(n, w, h, clusters, epoch, output_path):
2      colors = []
3      for r in [0, 255]:
4          for g in [0, 255]:
5              for b in [0, 255]:
6                  colors.append([r, g, b])
7      colors = np.array(colors)
8
9      img = np.empty((h, w, 3))
10     for i in range(n):
11         img[i // w, i % w, :] = colors[clusters[i], :]
12
13     img = Image.fromarray(np.uint8(img))
14     # img.save(os.path.join(output_path, "%03d.png" % epoch))
15
16     return img

```

Each clusters should be colored in different colors. The RGB values for each colors are store in a list of tuples `colors` . We only generate 8 colors (R, G, B can either be 0 or 255), so technically the most cluster number available in our program is **8**.

Plot each pixel in the image array `img` by its cluster. `img` is a  $(H, W, 3)$  3d array, which in this case  $H = W = 100$ . Before returning the image, turn `img` from an array to a *Image* object.

We will return the result image by calling this function in each iteration of K-means process.

After all that, we will use this image to generate a animated gif file as described previously in section

**function: Kernel\_Kmeans().**

## ***Spectral Clustering (Ratio Cut)***

All the preparations and arguments to be passed for this code is almost the same as those used in ***kernel k-means***. So please refer to **The main structure** section in kernel k-means before.

There's only one additional argument in spectral clustering: **-n**, which indicates whether we are going to use **ratio cut (unnormalized)** or **normal cut (normalized)** in spectral clustering.

Set **0** for ratio cut and **1** for normal cut. The argument is default to **0**. We also set the variable `normalized` to receive the value of this argument.

***The main structure***

```

1  if normalized == 0: # unnormalized spectral clustering, ratio cut used
2      image, coords, w, h = load_image(args.i)
3      image_list = []
4      n = image.shape[0]
5
6      W = RBF_kernel_mix(image, coords, gamma_s, gamma_c)
7      D, L = Laplacian_and_Degree(W, n)
8
9      save_parameters(os.path.join(output_path, "ratio_cut.pkl"), L, D, W,
10                     None, None, False)
11      # L, D, W = load_parameters(os.path.join(output_path, "ratio_cut.pkl"), False)
12
13      eigenvalues, eigenvectors = np.linalg.eig(L)
14      np.save(os.path.join(output_path, "ratio_eigenvalues.npy"), eigenvalues)
15      np.save(os.path.join(output_path, "ratio_eigenvectors.npy"), eigenvectors)
16      # eigenvalues = np.load(os.path.join(output_path, "ratio_eigenvalues.npy"))
17      # eigenvectors = np.load(os.path.join(output_path, "ratio_eigenvectors.npy"))
18
19      sorted_idx = np.argsort(eigenvalues)
20      masked = np.where(eigenvalues[sorted_idx] > 0, True, False)
21      sorted_idx = sorted_idx[masked]
22
23      U = eigenvectors[:, sorted_idx[0:K]]
24      initial_centers = get_initial_center(K, n, U, mode)
25
26      threshold = 1e-5
27      for e in range(1, epoch+1):
28          final_clusters = np.empty(n, dtype = np.uint8)
29
30          # E-step
31          for i in range(n):
32              dist = []
33              for k in range(K):
34                  dist.append(np.linalg.norm(U[i, :] - initial_centers[k, :]))
35              dist = np.array(dist)
36              final_clusters[i] = np.argmin(dist)
37
38          # M-step
39          final_centers = np.zeros_like(initial_centers)
40          for k in range(K):
41              masked = np.where(final_clusters == k, True, False)
42              final_centers[k] = np.sum(U[masked, :], axis = 0)
43              if np.sum(masked) > 0:
44                  final_centers[k] /= np.sum(masked)
45
46          if (np.linalg.norm(final_centers - initial_centers) < threshold):
47              break
48
49          img = visualize(n, w, h, final_clusters, e, output_path, "ratio")
50          image_list.append(img)
51          initial_centers = final_centers
52      image_list[0].save(os.path.join(output_path, "ratio_GIF.gif"), save_all = True)
53
54      if K == 3:
55          visualize_eigenspaces(U, final_clusters, output_path, "ratio")

```

First, we load in the image by function **load\_image()**, which returns the image RGB array `image`, the coordinates array for pixels `coords`, and the width as well as the height of the image.

This function is the same as that used in kernel k-means, so please refer to the **function: load\_image()** section there if you want to know the details.

And then, the following process are very similar to those described in the **function: Kernel\_Kmeans()** section of kernel k-means. So, we would like to bring up those

differences only.

**difference 1:** matrix  $W$

We use `w` to store the Gram matrix of kernel functions from the function **RBF\_kernel\_mix()** instead of `kernel`. They are the same thing, just the name difference. The reason we do so is to align it with the  $W$  matrix in the handout slides.

$W$  is the **weighting matrix**, which has its elements  $W_{ij}$  being the similarity scores between the node (pixel)  $x_i$  and  $x_j$ , calculated from the kernel functions:

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

As for the details about this kernel, please refer to the **function: RBF\_kernel\_mix()** section in kernel k-means.

**difference 2:** matrix  $D$  and  $L$

As for the Degree matrix  $D$  and the graph Laplacian matrix  $L$ , we will use the function

**Laplacian\_and\_Degree()** to construct them, and store in `D` and `L`, respectively. The detailed explanations about these 2 matrixes will be given in the section **function: Laplacian\_and\_Degree()** later on.

**difference 3:** save and load parameters

After calculating matrixes  $W$ ,  $D$  and  $L$ , we will use the function **save\_parameters()**. Next time when we want to execute the clustering again on the same image, we can directly use the function **load\_parameters()** to load the these 3 matrixes in, for saving calculation time. Details about these 2 functions will be described later.

**difference 4:** calculate the eigenpairs

Based on the handout slides, we know that the best indicator-like matrix  $U$  is a  $n \times K$  matrix ( $n = 10000$  and  $K = 3$  by default), composed of the  $K$  eigenvectors of the graph Laplacian matrix  $L$ , corresponding to the  $K$  smallest **non-null** eigenvalues. So, We need to solve the eigenpairs of matrix  $L$ .

After the calculations, we use `numpy.save()` to save the eigenpairs. Next time when we are doing clustering on the same image again, we can directly use

`numpy.load()` to load in the eigenpairs. Again, this saves a lots of time since calculating the eigen decomposition of a  $10000 \times 10000$  matrix  $L$  takes quite a long time.

When the eigenpairs are solved, we can pick out the  $K$  eigenvectors corresponding to the  $K$  smallest **non-null** eigenvalues to construct matrix  $U$ . And this is what the following codes do.

```
1 sorted_idx = np.argsort(eigenvalues)
2 masked = np.where(eigenvalues[sorted_idx] > 0, True, False)
3 sorted_idx = sorted_idx[mask]
4
5 U = eigenvectors[:, sorted_idx[0:K]]
```

After these, the processes for executing K-means are basically the same as those in kernel k-means, just that

**difference 5:** matrix  $U$  instead of  $W$  is used in `get_initial_center()`

The kernel matrix we used while finding the initial cluster centers in **get\_initial\_center()** are the matrix  $U$  instead of the original gram matrix  $W$ , which is literally the same thing as `kernel` used in kernel k-means.

**difference 6:** visualization of coordinates in eigen space

We also do the visualization of coordinates of all pixels in the eigen space constructed by  $U$ .

Details about the visualization here will be elaborated in the section **function: visualize\_eigenspaces()** later on.

For visualization reason, only when  $K = 3$  clusters we will visualize the eigen space, which is a **3D** space.

### ***function: Laplacian\_and\_Degree()***

This function takes the weighting matrix  $W$ , and return the degree matrix  $D$  as well as the graph Laplacian matrix  $L$ . All 3 matrixes have the shape of (10000, 10000) in this case.

```
1 def Laplacian_and_Degree(W, n):
2     D = np.zeros_like(W)
3     L = np.zeros_like(W)
4
5     D = np.diag(np.sum(W, axis = 1))
6
7     L = D - W
8
9     return D, L
```

The degree matrix  $D$  is a diagonal matrix, with the element  $d_{ii} = \sum_{j=1}^n W_{ij}$ . That is to say, the degree of pixel  $i$  is exactly the summation over the weights of all its neighbors, since  $W_{ij} = 0$  if pixel  $j$  is not  $i$ 's neighbor. And then, we can define the graph Laplacian matrix  $L$  as:

$$L = D - W$$

### ***function: save\_parameters()***

This function is to save the parameters, typically those matrixes:

1. In **unnormalized** mode (*ratio cut*), the function save  $L$ ,  $D$ ,  $W$ .

For the explanations about these 3 matrixes, you can

refer to the section **function: Laplacian\_and\_Degree()** above.

2. In **normalized** mode (*normal cut*), the function save  $L$ ,  $D$ ,  $W$ ,  $L_{sym}$  and  $D_{sym}$ .

For the explanations about the matrixes  $L_{sym}$  and  $D_{sym}$ , you can refer to the section **function:**

**Normalized\_Laplacian\_and\_Degree()** later while we are introducing ***normalized mode (normal cut)***.

```
1 def save_parameters(output_path: str, L, D, W, Lsym, Dsym, normalized: bool):
2     if normalized:
3         parameters = {
4             'L': L,
5             'D': D,
6             'W': W,
7             'Lsym': Lsym,
8             'Dsym': Dsym
9         }
10    else:
11        parameters = {
12            'L': L,
13            'D': D,
14            'W': W
15        }
16    file = open(output_path, "wb")
17    pickle.dump(parameters, file)
18    file.close()
```

Here, we use `dump()` function from the `pickle` module to save these parameters to the file.

### ***function: load\_parameters()***

This function is to load in the parameters, typically these matrixes:

1. In **unnormalized** mode (*ratio cut*), the function load  $L$ ,  $D$ ,  $W$ .

For the explanations about these 3 matrixes, you can refer to the section **function: Laplacian\_and\_Degree()** above.

2. In **normalized** mode (*normal cut*), the function load  $L$ ,  $D$ ,  $W$ ,  $L_{sym}$  and  $D_{sym}$ .

For the explanations about the matrixes  $L_{sym}$  and  $D_{sym}$ , you can refer to the section **function:**

**Normalized\_Laplacian\_and\_Degree()** later while we are introducing ***normalized mode (normal cut)***.

```

1 def load_parameters(output_path: str, normalized: bool):
2     with open(output_path, "rb") as file:
3         parameters = pickle.load(file)
4
5     L = parameters['L']
6     D = parameters['D']
7     W = parameters['W']
8     if normalized:
9         Lsym = parameters["Lsym"]
10        Dsym = parameters["Dsym"]
11        return L, D, W, Lsym, Dsym
12    return L, D, W

```

### ***function: visualize\_eigenspaces()***

This function is to draw out each pixel in the  $K$  dimensional eigen space.  $K$  is the cluster number. For visualization reason, only when  $K = 3$  we will call this function to visualize these pixel in **3D** eigen space.

```

1 def visualize_eigenspaces(U, clusters, output_path, mode: str):
2     fig = plt.figure()
3     ax = fig.add_subplot(111, projection = '3d')
4     for i in np.arange(3):
5         ax.scatter(U[:, 0][clusters == i], U[:, 1][clusters == i], U[:, 2][clusters == i])
6
7     ax.set_xlabel('eigenvector dim 1')
8     ax.set_ylabel('eigenvector dim 2')
9     ax.set_zlabel('eigenvector dim 3')
10
11    plt.savefig(os.path.join(output_path, mode + '_eigenspaces.png'))

```

We first use `add_subplot()` from package `matplotlib.pyplot` with argument `projection = "3d"` to build a 3d subplot. And then draw a *3d scatter plot* for the pixels based on their coordinates in the eigen space of the graph Laplacian matrix  $L$  in **unnormalized mode**. If in **normalized mode**, then the scatter plot will be drawn on the eigen space of the **normalized** graph Laplacian matrix  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ . The details about the normalized graph Laplacian matrix will be shown in the next part: the **normalized mode (normal cut)**.

## ***Spectral Clustering (Normal Cut)***

All the preparations and arguments to be passed for this code is almost the same as those used in **kernel k-means**. So please refer to **The main structure** section in kernel k-means before.



There's only one additional argument in spectral clustering: **-n**, which indicates whether we are going to use **ratio cut (unnormalized)** or **normal cut (normalized)** in spectral clustering.

Set **0** for ratio cut and **1** for normal cut. The argument is default to **0**. We also set the variable `normalized` to receive the value of this argument.

***The main structure***

```

1 elif normalized == 1: # normalized spectral clustering, normal cut used
2     image, coords, w, h = load_image(args.i)
3     image_list = []
4     n = image.shape[0]
5
6     W = RBF_kernel_mix(image, coords, gamma_s, gamma_c)
7     D, L = Laplacian_and_Degree(W, n)
8     Dsym, Lsym = Normalized_Laplacian_and_Degree(D, L)
9
10    save_parameters(os.path.join(output_path, "normal_cut.pkl"), L, D, W,
11    Lsym, Dsym, True)
12    # L, D, W, Lsym, Dsym = load_parameters(os.path.join(output_path,
13    # "normal_cut.pkl"), True)
14
15    eigenvalues, eigenvectors = np.linalg.eig(Lsym)
16    np.save(os.path.join(output_path, "normal_eigenvalues.npy"), eigenvalues)
17    np.save(os.path.join(output_path, "normal_eigenvectors.npy"), eigenvectors)
18    # eigenvalues = np.load(os.path.join(output_path, "normal_eigenvalues.npy"))
19    # eigenvectors = np.load(os.path.join(output_path, "normal_eigenvectors.npy"))
20
21    sorted_idx = np.argsort(eigenvalues)
22    masked = np.where(eigenvalues[sorted_idx] > 0, True, False)
23    sorted_idx = sorted_idx[mask]
24
25    U = eigenvectors[:, sorted_idx[0:K]]
26    norm = np.linalg.norm(U, axis = 1).reshape(-1, 1)
27    U_norm = U / norm
28
29    initial_centers = get_initial_center(K, n, U_norm, mode)
30
31    threshold = 1e-5
32    for e in range(1, epoch+1):
33        final_clusters = np.empty(n, dtype = np.uint8)
34
35        # E-step
36        for i in range(n):
37            dist = []
38            for k in range(K):
39                dist.append(np.linalg.norm(U_norm[i, :] - initial_centers[k, :]))
40            dist = np.array(dist)
41            final_clusters[i] = np.argmin(dist)
42
43        # M-step
44        final_centers = np.zeros_like(initial_centers)
45        for k in range(K):
46            masked = np.where(final_clusters == k, True, False)
47            final_centers[k] = np.sum(U_norm[mask], :, axis = 0)
48            if np.sum(mask) > 0:
49                final_centers[k] /= np.sum(mask)
50
51        if (np.linalg.norm(final_centers - initial_centers) < threshold):
52            break
53
54        img = visualize(n, w, h, final_clusters, e, output_path, "normal")
55        image_list.append(img)
56        initial_centers = final_centers
57    image_list[0].save(os.path.join(output_path, "normal_GIF.gif"), save_all = True,
58    append_images = image_list[1:], duration = 200, loop = 0)
59
60    if K == 3:
61        visualize_eigenspaces(U_norm, final_clusters, output_path, "normal")

```

All the processes are basically the same as those in previous part: ***unnormalized mode (normal cut)***. Just some differences listed as below:

**difference 1:** normalized degree matrix and normalized graph Laplacian matrix

Aside from the ordinary weighting matrix, degree matrix and graph Laplacian matrix, namely  $W$ ,  $D$ ,  $L$ , as described in the previous **normalized mode** part, we also have to calculate additional 2 matrixes: the normalized degree matrix  $D_{sym} = D^{-\frac{1}{2}}$  and the normalized graph Laplacian matrix  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ . More information will be revealed in the section

**function: Normalized\_Laplacian\_and\_Degree()** later on.

**difference 2:** Solve the eigen problem of  $L_{sym}$  instead of  $L$

Based on the handout slides, we know that the best indicator-like matrix  $U$  is a matrix composed of the  $K$  eigenvectors of the **normalized** graph Laplacian matrix  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ , corresponding to the  $K$  smallest **non-null** eigenvalues.

So, We need to solve the eigenpairs of matrix  $L_{sym}$ .

After the calculations, we use `numpy.save()` to save the eigenpairs. Next time when we are doing clustering on the same image again, we can directly use

`numpy.load()` to load in the eigenpairs. Again, this saves a lots of time since calculating the eigen decomposition of a big matrix takes quite a long time.

**difference 3:** Use the normalized indicator matrix

$U_{norm}$  instead of  $U$

When the eigenpairs are solved, we can pick out the  $K$  eigenvectors corresponding to the  $K$  smallest **non-null** eigenvalues to construct matrix  $U$ .

However,  $U$  is not the normalized indicator matrix yet!

To get the **normalized** indicator matrix  $U_{norm}$ , we must normalized each **row vectors** of  $U$  to **unit length**.

And this is what the following codes do.

```
1 sorted_idx = np.argsort(eigenvalues)
2 masked = np.where(eigenvalues[sorted_idx] > 0, True, False)
3 sorted_idx = sorted_idx[mask]
4
5 U = eigenvectors[:, sorted_idx[0:K]]
6 norm = np.linalg.norm(U, axis = 1).reshape(-1, 1) # L2 norm for each rows.
7 U_norm = U / norm
```

**difference 4:** matrix  $U_{norm}$  instead of  $U$  is used in `get_initial_center()`

The kernel matrix we used while finding the initial cluster centers in **get\_initial\_center()** are the **normalized** matrix  $U_{norm}$  instead of the original indicator matrix  $U$  used in **unnormalized mode** (ratio cut).

**difference 5:** Use  $U_{norm}$  instead of  $U$  to visualize coordinates in eigen space

We also do the visualization of coordinates of all pixels in the eigen space constructed by the **normalized**  $U_{norm}$  instead of the original indicator matrix  $U$ .

Apart from these differences, all processes and functions used are almost the same as those used in previous **unnormalized mode (ratio cut)** part.

**function: Normalized\_Laplacian\_and\_Degree()**

This function is used to calculate and return the **normalized** degree matrix  $D_{sym}$  and the **normalized** graph Laplacian matrix  $L_{sym}$ , from the original degree matrix and graph Laplacian matrix, namely  $D$  and  $L$ .

```

1 def Normalized_Laplacian_and_Degree(D, L):
2     Dsym = np.diag(np.reciprocal(np.diag(np.sqrt(D))))
3     Lsym = np.linalg.multi_dot([Dsym, L, Dsym])
4
5     return Dsym, Lsym

```

According to the handout slides, the **normalized** degree matrix  $D_{sym}$  is:

$$D_{sym} = D^{-\frac{1}{2}}$$

The **normalized** graph Laplacian matrix  $L_{sym}$  is:

$$L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

## Part 2

---

No matter in kernel k-means part or spectral clustering part; no matter ratio cut is used or normal cut is used, to try different cluster numbers, you just need to adjust the argument passing in.

What I am talking about is this argument: **-k**, which indicates the cluster number  $K$  you would like to use in clustering processes.

```

1 parser.add_argument("--k", help = "number of clusters",
2                       type = int, default = 4)
3 .....
4
5 K = args.k

```

The default cluster number used in **kernel k-means** is **4**, set for no particular reason.

As for the default cluster number used in **spectral clustering** is **3**, since we can visualize each pixels in the **3D** eigen space.

We would like to try out different cluster numbers in this "Part 2". Please refer to the **experiment and discussion** sections of Part 2.

## Part 3

---

No matter in kernel k-means part or in spectral clustering part, there are 3 different initialization methods we've tried. They are **random mode**, **normal mode** and **kmeans++ mode**. The following descriptions about these 3 methods are the same as those in the section **function: get\_initial\_center()** in kernel k-means part. The experiment results and discussion will be given later in the **experiment and discussion** sections of Part 3.

```

1  def get_initial_center(K, n, kernel, mode):
2      if mode == "random":
3          initial_cluster = kernel[list(random.sample(range(0, n), K)), :]
4      elif mode == "normal":
5          initial_cluster = np.empty((K, n))
6          mean_vector = np.mean(kernel, axis = 0)
7          std_vector = np.std(kernel, axis = 0)
8          for feature in range(n):
9              initial_cluster[:, feature] = np.random.normal(mean_vector[feature],
10                  std_vector[feature], K)
11      elif mode == "kmeans++":
12          clusters = []
13          clusters.append(random.randint(0, n-1))
14          cluster_num = 1
15          while (cluster_num < K):
16              dist = np.empty((n, cluster_num))
17              for i in range(n):
18                  for c in range(cluster_num):
19                      dist[i, c] = np.linalg.norm(kernel[i, :]-kernel[clusters[c], :])
20              dist_min = np.min(dist, axis = 1)
21              sum_rand = np.sum(dist_min) * np.random.rand()
22              for i in range(n):
23                  sum_rand -= dist_min[i]
24                  if sum_rand <= 0:
25                      clusters.append(i)
26                      break
27              cluster_num += 1
28          initial_cluster = kernel[clusters, :]
29      else:
30          raise ValueError("Unknown clustering mode: {}".format(mode))
31
32      return initial_cluster

```

### <random mode>

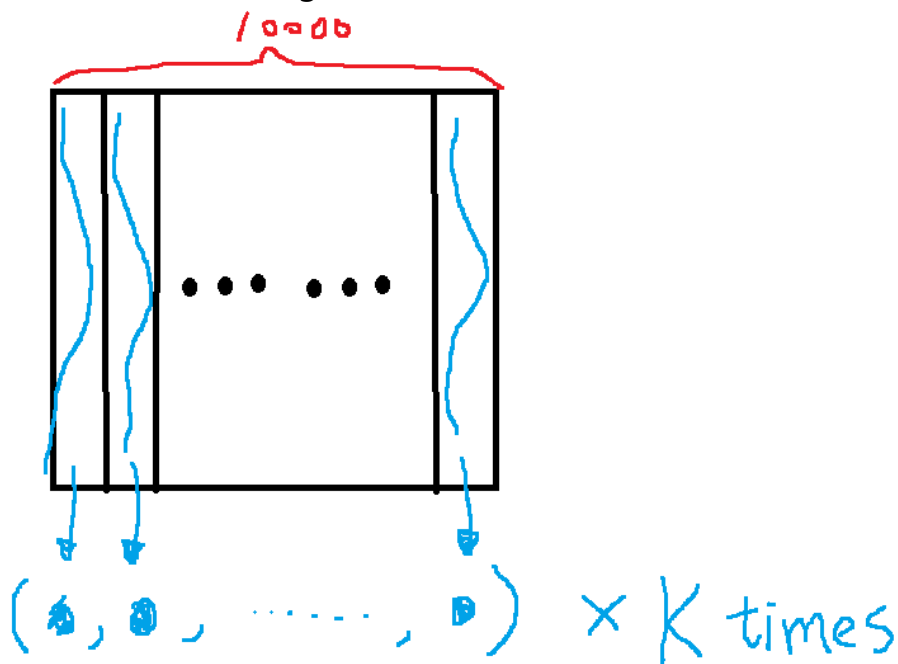
In random mode, the centers of all  $K$  clusters are **randomly chosen(sampled)** from  $K$  pixels in kernel spaces `kernel` .

<**normal** mode>

In normal mode, we generate all centers based on the **Gaussian distributions** of each 10000 *features*(dimensions) in kernel spaces `kernel` .

This is done by first calculate the **mean** and **standard deviation** of each feature(dimension). Now, for each dimension, we can fit a normal distribution with the mean and standard deviation calculated.

To generate a center of a cluster, we just need to sample a value from each of the 10000 normal distributions of each feature(dimension). Do this  $K$  times, we can then generate centers of  $K$  clusters.



<**kmeans++ mode**>

In k-means++ mode, we use a modified initialization method from K-means++ clustering algorithm. The concept is to randomly choose these cluster centers as well, but *the pixel which is far from all previously chosen cluster centers should be more probable to be picked.*

The first cluster center is randomly picked from all the pixels.

As for the second to the  $K$ -th cluster centers, suppose that we're currently generating the  $i$ -th center, which means that we already generated  $i - 1$  cluster centers.

The algorithm processes are as follows:

1. For each pixel in kernel space, calculate its distances from the  $i - 1$  chosen cluster centers, and then record the **minimum** distance.
2. **sum up** the minimum distances of all 10000 pixels, and then **multiply** it by a **random floating point number within [0,1) range**.
3. Use the **roulette method** to decide which pixel to choose as the  $i$ -th cluster center.

If the value from step 2 is equal to or less than the minimum distance of the current pixel, **choose current pixel as the new center**. Otherwise, subtract the minimum distance of the current pixel from the value in step 2 and **move on to check the next pixel**.

## Part 4

This part is specially for **spectral clustering** (both normalized cut and ratio cut). We would like to examine whether the data points (pixels) within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not.

This is done by the *visualizing these points in the eigen space*, which is done by the function



**visualize\_eigenspaces()**. The following explanation is basically a copy from the section **function**:

**visualize\_eigenspaces()** in Part 1, unnormalized mode (ratio cut).

```
1 def visualize_eigenspaces(U, clusters, output_path, mode: str):
2     fig = plt.figure()
3     ax = fig.add_subplot(111, projection = '3d')
4     for i in np.arange(3):
5         ax.scatter(U[:, 0][clusters == i], U[:, 1][clusters == i], U[:, 2][clusters == i])
6
7     ax.set_xlabel('eigenvector dim 1')
8     ax.set_ylabel('eigenvector dim 2')
9     ax.set_zlabel('eigenvector dim 3')
10
11     plt.savefig(os.path.join(output_path, mode + '_eigenspaces.png'))
```

We first use `add_subplot()` from package

`matplotlib.pyplot` with argument `projection = "3d"` to build a 3d subplot. And then draw a *3d scatter plot* for the pixels based on their coordinates in the eigen space of the graph Laplacian matrix  $L$  in **unnormalized mode**.

If in **normalized mode**, then the scatter plot will be drawn on the eigen space of the **normalized** graph Laplacian matrix  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$ .

Finally, for visualization reason, only when  $K = 3$  we will call this function to visualize these pixel in **3D** eigen space.

## Experiment & Discussion



**The gif animations won't work on pdf file! So please**

***refer to my hackmd or gif files in the zip file handed in!***

my hackmd: <https://hackmd.io/ssPhbafhQL-PYuvMMB49wQ> (<https://hackmd.io/ssPhbafhQL-PYuvMMB49wQ>)

### Part 1

In this part, we would like to show the result gif animations of the clustering procedure of kernel k-means and spectral clustering (both ratio cut and normal cut)

### Kernel K-means

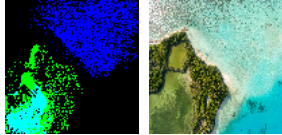
### ***Experimental settings (the arguments)***

All left with default values.

1. cluster number:  $K = 4$
2. initialization methods: **kmeans++**
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

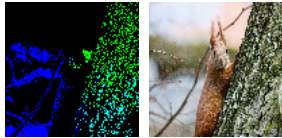
for the meaning of these 2 parameters, please refer to the **function: RBF\_kernel\_mix()** section in the kernel k-means part 1.

*image1*



file name: **kernel\_kmeans\_GIF1.gif**

*image2*



file name: **kernel\_kmeans\_GIF2.gif**

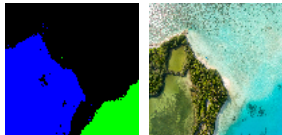
### **Spectral Clustering (Ratio Cut)**

#### ***Experimental settings (the arguments)***

All left with default values.

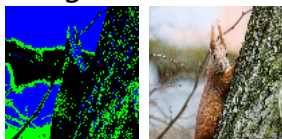
1. cluster number:  $K = 3$ , for visualizing eigenspace.
2. initialization methods: **kmeans++**
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$
4. normalized mode: **0** (deactivated, **unnormalized**)

*image1*



file name: **ratio\_GIF1.gif**

*image2*



file name: **ratio\_GIF2.gif**

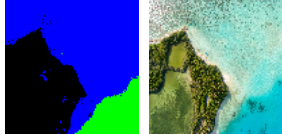
### **Spectral Clustering (Normal Cut)**

## **Experimental settings (the arguments)**

All left with default values.

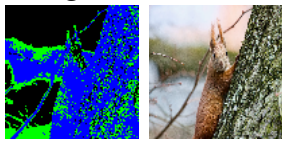
1. cluster number:  $K = 3$ , for visualizing eigenspace.
2. initialization methods: **kmeans++**
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$
4. normalized mode: **1** (activated, **normalized**)

*image1*



file name: **normal\_GIF1.gif**

*image2*



file name: **normal\_GIF2.gif**

## **Discussions**

One can see that in **kernel K-means**, the first image with  $K = 4$  clusters has quite good clustering result. However, one can also see that the **initialization centers** do affect the clustering a lot. Because the deep green area at the left-bottom area of the image is deemed as a cluster, and  $K = 4$ , so the other deep green area which are the **real trees** are deemed as the same cluster as the *light blue sea*. As for the image 2 in **kernel K-means**, since the cluster number is only  $K = 4$ , the **animal creature** which has closer color as the **tree trunk** are clustered together, resulting a bad result.

As for the **spectral clustering**, we can observe a phenomenon that **the results of image 1 converge very quick**, no matter which type of cut is used. This may be because that the first image has more homogenous color areas. What I means is that no matter green island or blue sea, they are typically in the same color in big areas. On the contrary, borders of some areas in image 2 are **gradient(漸層的)** colored, thus the predicted clusters of border pixels change in the clustering procedure.

## **Part 2**

---

In this part, we want to experiment whether different cluster numbers improve our performance of clustering or not.

## Kernel K-means

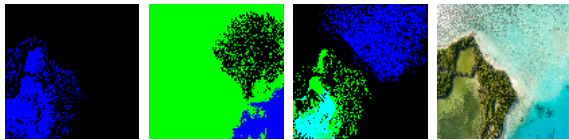
### ***Experimental settings (the arguments)***

1. cluster number:  $K = 2, K = 3, K = 4$  (default in part 1)

2. initialization methods: **kmeans++**

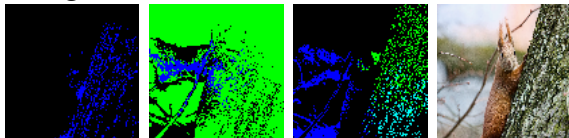
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

*image1*, from  $K = 2$  to  $K = 4$



file name: **GIF\_2\_1.gif, GIF\_3\_1.gif, kernel\_kmeans\_GIF1.gif**

*image2*, from  $K = 2$  to  $K = 4$



file name: **GIF\_2\_2.gif, GIF\_3\_2.gif, kernel\_kmeans\_GIF2.gif**

### ***Discussion***

In kernel k-means, one can see that in image 1, the most suitable cluster number will be the default value **4**.  $K = 2$  and  $K = 3$  are both too few. With  $K = 4$ , We can already classify 4 main areas: *deep green area of the island, light green area of the island, light blue area of the sea and finally, the other area.*

As for the second image, there is an interesting phenomenon: in ground truth image, we can see that there's two **light blue** areas. However, the pixels in these 2 areas are classified in **different clusters**!

## Spectral Clustering (Ratio Cut)

### ***Experimental settings (the arguments)***

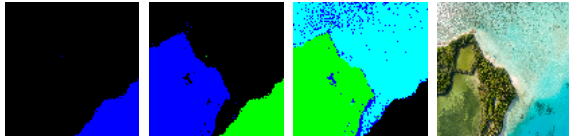
1. cluster number:  $K = 2, K = 3, K = 4$  (default value is  $K = 3$ )

2. initialization methods: **kmeans+ +**

3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

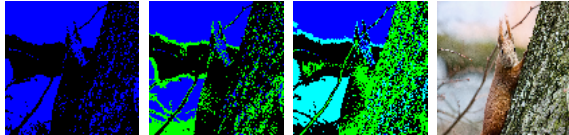
4. normalized mode: **0** (deactivated, **unnormalized**)

*image1*, from  $K = 2$  to  $K = 4$



file name: **ratio\_GIF\_2\_1.gif**, **ratio\_GIF1.gif**,  
**ratio\_GIF\_4\_1.gif**

*image2*, from  $K = 2$  to  $K = 4$



file name: **ratio\_GIF\_2\_2.gif**, **ratio\_GIF2.gif**,  
**ratio\_GIF\_4\_2.gif**

### **Discussion**

In spectral clustering, one can see that in image 1, when  $K = 2$  and  $K = 3$ , the results converge really quick. The possible reason is also described in part 1

**(homogenously colored areas).**

An interesting situation in  $K = 4$  shows that ***the clustering procedures are highly affected by outliers!***

One can see that initially there are 4 clusters.

Nevertheless, one of the cluster shrinks to only 1 point (colored in deep blue). This point might be an outlier.

As for the second image, there are nothing special. Only one thing that's been mentioned in previous part: in  $K = 4$ , two blue areas are classified in different clusters.

## **Spectral Clustering (Normal Cut)**

### **Experimental settings (the arguments)**

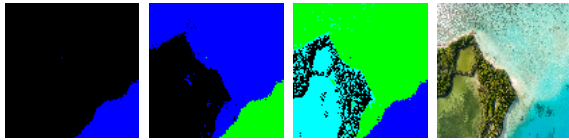
1. cluster number:  $K = 2, K = 3, K = 4$  (default value is  $K = 3$ )

2. initialization methods: **kmeans+ +**

3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

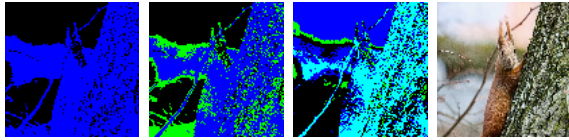
4. normalized mode: **1** (activated, **normalized**)

*image1*, from  $K = 2$  to  $K = 4$



file name: **normal\_GIF\_2\_1.gif**, **normal\_GIF1.gif**,  
**normal\_GIF\_4\_1.gif**

*image2*, from  $K = 2$  to  $K = 4$



file name: **normal\_GIF\_2\_2.gif**, **normal\_GIF2.gif**,  
**normal\_GIF\_4\_2.gif**

### ***Discussions***

As for normal cut (normalized mode), one can see that the procedures on image 1 still converge really quick due to homogenously colored areas.

As for the second image, both blue areas are classified into 2 different clusters from  $K = 4$ .

## **Part 3**

---

In this part, we would like to try different ways to initialize kernel k-means and spectral clustering (both normalized cut and ratio cut).

We provide 3 different methods: **random**, **normal** and **kmeans++**. About the descriptions of these methods, please refer to the previous section: **function: `get_initial_center()`** in the code explanation part kernel k-means.

### **Kernel K-means**

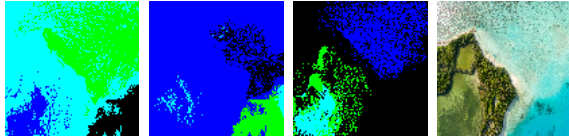
#### ***Experimental settings (the arguments)***

1. cluster number:  $K = 4$

2. initialization methods: **random**, **normal**, **kmeans++** (default)

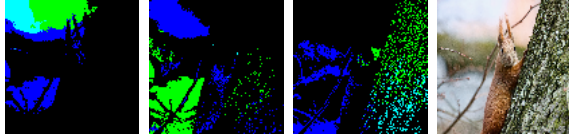
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

*image1*, in **random**, **normal**, **kmeans++** order



file name: **GIF\_random\_1.gif**, **GIF\_normal\_1.gif**,  
**kernel\_kmeans\_GIF1.gif**

*image2*, in **random**, **normal**, **kmeans++** order



file name: **GIF\_random\_2.gif**, **GIF\_normal\_2.gif**,  
**kernel\_kmeans\_GIF2.gif**

### ***Discussion***

One can see that all these 3 methods perform equally good on image 1 in **kernel k-means**. A little difference is that one of the cluster in **random mode** and **normal mode** refers to the deep blue sea area in ground truth, while one of the cluster in **kmeans++ mode** refers to the deep green areas on the island in ground truth.

The difference might be because of different concepts regarding choosing initial centers behind these methods. In **kmeans++**, one important concept is that the current center chosen **has to be far from other previously chosen ones**.

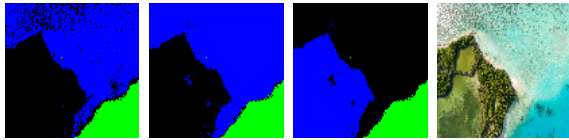
In image 2, we can observe that **kmeans++** mode converge relatively slow, compared to the other 2 methods.

## **Spectral Clustering (Ratio Cut)**

### ***Experimental settings (the arguments)***

1. cluster number:  $K = 3$ , for visualizing the eigenspace
2. initialization methods: **random**, **normal**, **kmeans++** (default)
3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$
4. normalized mode: **0** (deactivated, **unnormalized**)

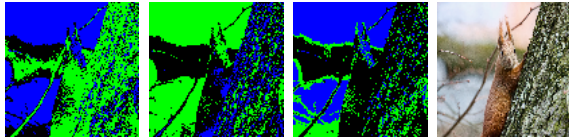
*image1*, in **random, normal, kmeans++** order



file name: **ratio\_GIF\_random\_1.gif**,

**ratio\_GIF\_normal\_1.gif, ratio\_GIF1.gif**

*image2*, in **random, normal, kmeans++** order



file name: **ratio\_GIF\_random\_2.gif**,

**ratio\_GIF\_normal\_2.gif, ratio\_GIF2.gif**

### ***Discussion***

In spectral clustering with ratio cut, one can see that in image 1, all 3 methods converge quickly, and the clustering results are basically the same.

However, in the second image, we can observe the same phenomenon as that in kernel k-means: **kmeans++** mode converge relatively slow, compared to the other 2 methods.

## **Spectral Clustering (Normal Cut)**

### ***Experimental settings (the arguments)***

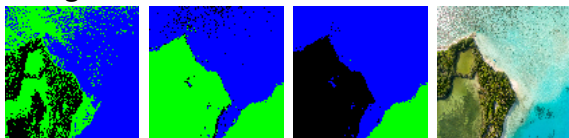
1. cluster number:  $K = 3$ , for visualizing the eigenspace

2. initialization methods: **random, normal, kmeans++** (default)

3. hyper-parameters for the mix kernel:  $\gamma_s = \gamma_c = 0.001$

4. normalized mode: **1** (activated, **normalized**)

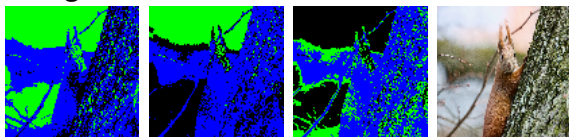
*image1*, in **random, normal, kmeans++** order



file name: **normal\_GIF\_random\_1.gif**,

**normal\_GIF\_normal\_1.gif, normal\_GIF1.gif**

*image2*, in **random, normal, kmeans++** order



file name: **normal\_GIF\_random\_2.gif**,

**normal\_GIF\_normal\_2.gif, normal\_GIF2.gif**



## ***Discussions***

In spectral clustering with normal cut, one can see that in image 1, the **kmeans++ mode** converges very quick. The **random mode** converges the slowest, though it actually reaches the same clustering result as that of **kmeans++ mode**.

As for the **normal** mode, we can observe that it converges into almost 2 clusters only (we use  $K = 3$ , however.). This might be due to a bad sampled centers, which is a pretty outlier value.

As for the results of the second image, we can see that all 3 methods have agreement in the clustering results. The only difference is the speed to converge.

## **Part 4**

---

In this part, we would like to examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not, for spectral clustering (both normalized cut and ratio cut). We will do this by **visualizing the eigenspace** and plot the data points in **scatter plot**, by the help of the function **visualize\_eigenspaces()**. For more implementation details, you can refer to the section **function: visualize\_eigenspaces()** in part 1, spectral clustering (normal cut).

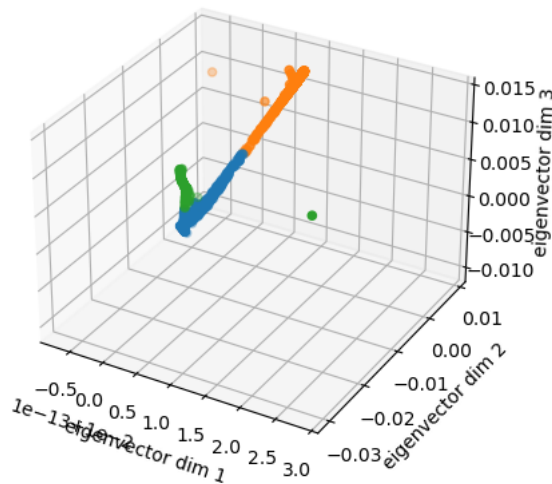
Since we are plotting a **3D** scatter plot, we will do this visualization **only when  $K = 3$  clusters**.

### **Spectral Clustering (Ratio Cut)**

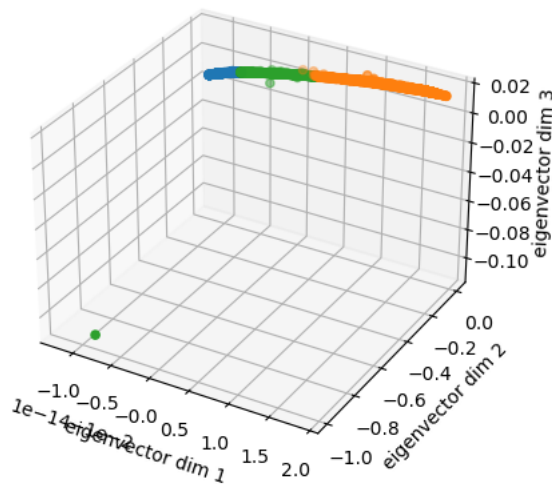
#### ***Experimental settings and results***

All arguments are left as default values.

**image 1:** filename = **ratio\_eigenspaces\_1.png**



**image 2:** filename = **ratio\_eigenspaces\_2.png**



## Discussion

In spectral clustering with **ratio cut**, one can see that in image 1, the orange cluster and blue cluster are basically classified depending on **eigenvector 2**, while the green cluster is classified by the third and first eigenvectors.

As for the second image, we can see that the procedure yields a neat classifying result in the eigenspace! *We can even use **eigenvector 1** only, to conduct the clustering!!!*

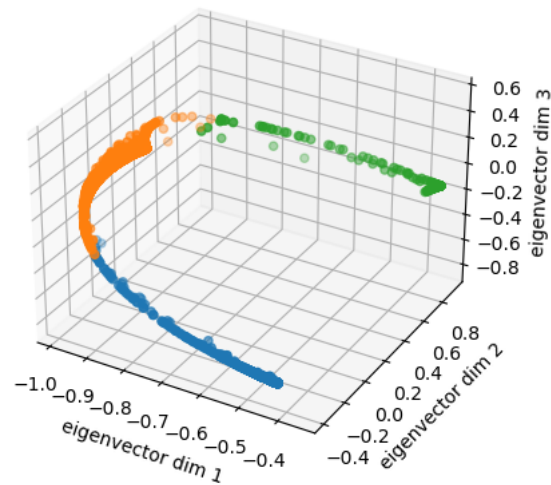
Also, it seems that there's an outlier of the green cluster in the eigenspace.

## Spectral Clustering (Normal Cut)

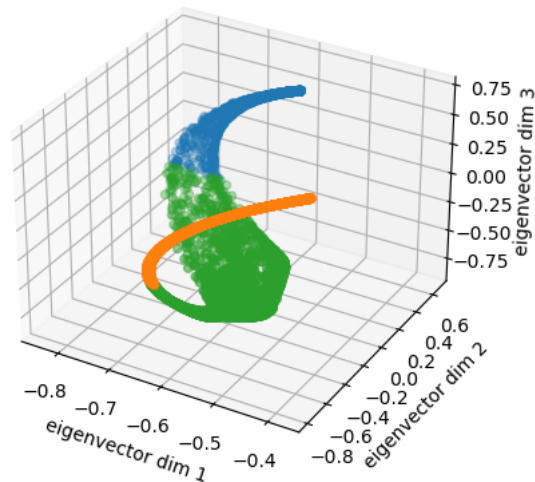
## Experimental settings and results

All arguments are left as default values.

**image 1:** filename = **normal\_eigenspaces\_1.png**



**image 2:** filename = **normal\_eigenspaces\_2.png**



## Discussion

In spectral clustering with **normal cut**, the results are pretty different from those with **ratio cut**.

In the result of image 1, we can see that if we are try to classify these 3 clusters, we can first use **eigenvector 1** to isolate the orange cluster out. And then, we use **eigenvector 2** to separate the blue cluster form the green cluster.

As for the second image, its scatter plot presents an interesting horn-like shape in the eigenspace.

We can first use **eigenvector 3** to isolate the green cluster

out. And then, we use both **eigenvector 1** and **eigenvector 2** to separate the blue cluster from the orange cluster.

## Observations and Discussion

---

### grid search on the hyper-parameters of the mixed RBF kernel

---

Recall that our mixed RBF kernel used is calculated as:

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

with  $S(x)$  being the **spatial** information, i.e. the **coordinate** of the pixel  $x$  (a 2d vector in `coords`) and  $C(x)$  being the **color** information, i.e. the **RGB** values of the pixel  $x$  (a 3d vector in `image`).

The hyper-parameters regarding the spatial kernel and the color kernel, namely  $\gamma_s$  and  $\gamma_c$ , are denoted as `gamma_s` and `gamma_c` in my code and are default to **0.001**.

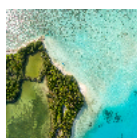
We would like to try out the values **0.01** and **0.0001** for these hyper-parameters as well, which is 10 times bigger and smaller.

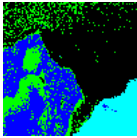
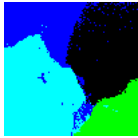
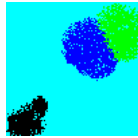
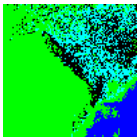
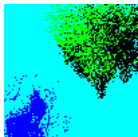
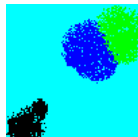
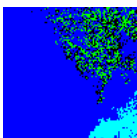
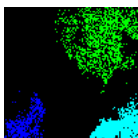
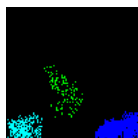
Please note that we only do the grid searching of  $\gamma_s$  and  $\gamma_c$  in **kernel k-means**, since in spectral clustering, tuning these parameters also changes the kernel, so the weighting matrix  $W$  and thus the (normalized) graph Laplacian matrix  $L$  ( $L_{sym}$ ) will also be changed. This means that we need to redo eigen decomposition every time we change the values of `gamma_s` and `gamma_c`, and this will cause enormous amount of time (*about **10~20 min a decomposition***).

The grid searching clustering results are listed as the following tables:

#### image 1

ground truth:

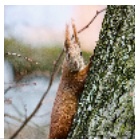


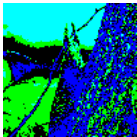
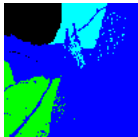
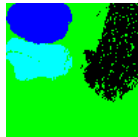
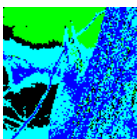
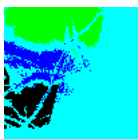
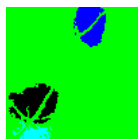
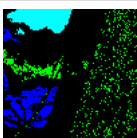
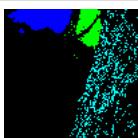

$\gamma_c \downarrow \gamma_s \rightarrow$	0.0001	0.001(default)	0.01
0.0001			
0.001(default)			
0.01			

The *only* hyper-parameters combination that converges within 20 epochs is  $(\gamma_s = 0.01, \gamma_c = 0.01)$ . Although one can see that this combination actually has bad clustering results. The best combination which yields a clustering result closest to the original image is  $(\gamma_c = 0.0001, \gamma_s = 0.0001)$ . However, I don't think we should set these hyper-parameters too small, because it might cause overfitting.

## image 2

ground truth:



$\gamma_c \downarrow \gamma_s \rightarrow$	0.0001	0.001(default)	0.01
0.0001			
0.001(default)			
0.01			

There are two hyper-parameters combination that converges within 20 epochs:  $(\gamma_s = 0.001, \gamma_c = 0.00001)$  and  $(\gamma_s = 0.01, \gamma_c = 0.0001)$ . The set  $(\gamma_s = 0.001, \gamma_c = 0.00001)$  actually generates quite a good clustering result.

However, the best combination which yields a clustering result closest to the original image still comes up to

( $\gamma_c = 0.0001$ ,  $\gamma_s = 0.0001$ ). Yet again, I don't think we should set these hyper-parameters too small, because it might cause overfitting.

Lastly, you can see that with small value of  $\gamma_c$ , the **texture** of the tree trunk actually clustered really well. So, focusing on the color information does help in rebuilding the image by clustering.

## A little self-study on methods to determine cluster number $K$

---

To choose the best cluster number  $K$ , there are 2 common ways. They are the **elbow method** and the **silhouette method**, respectively.

### elbow method

The concept of the method is based on the **sum of the squared errors, SSE**, which can be calculated as

$$SSE = \sum_{i=1}^K \sum_{\vec{x} \in C_i} \|\vec{x} - \vec{\mu}_i\|^2$$

where  $\vec{x}$  is some data point and  $\vec{\mu}_i$  is the center of cluster  $i$ ,  $C_i$ . We can draw a **line chart**, with  $K$  being the x-axis and  $SSE$  being the y-axis. This chart is typically called **scree plot**.

By observing scree plot, we can sometimes find a turning point (called **inflection point**) where the decrease in  $SSE$  slows down suddenly. Practically, we pick the point as our candidate cluster number  $K$ .

### silhouette method

The concept of the method is based on the initial yet most crucial target of cluster analysis: **Make the points within a cluster closer, while the points between different clusters stay further.**

To do this, we define 2 different measurements:

1.  $a_i$  is a measure of how dissimilar point  $i$  is to other points in the same cluster. Thus, a small value of  $a_i$  is preferred. In fact, this value is just the mean distance between point  $i$  and all other data points in the same cluster. Say, if  $i \in C_I$  cluster then,

$$a_i = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j)$$

The reason why we subtract 1 from  $|C_I|$  is that we don't calculate the distance between  $i$  and itself,  $d(i, i)$ , which is just 0.

2.  $b_i$  denotes the mean dissimilarity of point  $i$  to some cluster  $C_J$  as the mean of the distance from  $i$  to all points in  $C_J$ ,  $C_I \neq C_J$ . Thus, this value should be as large as possible.

Furthermore, we define  $b_i$  as the **minimum** of mean distance of  $i$  to all points in any other cluster. That is,

$$b_i = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

Then, we can define a **silhouette value** for each data point  $i$ , as

$$s_i = \frac{b_i - a_i}{\max\{a_i, b_i\}}, \text{ if } |C_I| > 1, 0 \text{ otherwise}$$

Silhouette method is simply calculate the **silhouette values** of all data points, and then sum them up (or equivalently, take average). The resulted values indicates how good the cluster number  $K$  is. The **bigger** the value, the better the current cluster number  $K$  is.