Monash University: Assessment Cover Sheet

Student name	Tan	Jin **Chun**	
School/Campus	**Monash University Malaysia**	**Student's I.D. number**	32194471
Unit name	ECE4076 - Computer vision - S1 2023		
Lecturer's name	**Dr.Maxine Tan**	**Tutor's name**	
Assignment name	Lab 3 Results Document Submission	**Group Assignment: No** **Note, each student must attach a coversheet**	

Lab/Tute Class: Friday Lab Session	**Lab/Tute Time: 10a.m - 12p.m**	**Word Count:**
Due date: 07-05-2023	**Submit Date: 07-05-2023**	**Extension granted** ☐

If an extension of work is granted, specify date and provide the signature of the lecturer/tutor. Alternatively, attach an email printout or handwritten and signed notice from your lecturer/tutor verifying an extension has been granted.

Extension granted until (date): ..../......./........... Signature of lecturer/tutor: ....

Late submissions policy	**Days late**	**Penalty applied**
Penalties apply to late submissions and may vary between faculties. Please refer to your faculty's late assessment policy for details.		

Patient/client confidentiality: Where a patient/client case study is undertaken a signed Consent Form must be obtained.

Intentional plagiarism or collusion amounts to cheating under Part 7 of the Monash University (Council) Regulations

Plagiarism: Plagiarism means to take and use another person's ideas and or manner of expressing them and to pass these off as one's own by failing to give appropriate acknowledgement. This includes material from any source, staff, students or the Internet - published and unpublished works.

Collusion: Collusion means unauthorised collaboration on assessable written, oral or practical work with another person. Where there are reasonable grounds for believing that intentional plagiarism or collusion has occurred, this will be reported to the Associate Dean (Education) or nominee, who may disallow the work concerned by prohibiting assessment or refer the matter to the Faculty Discipline Panel for a hearing.

Student Statement:

- I have read the university's Student Academic Integrity Policy and Procedures
- I understand the consequences of engaging in plagiarism and collusion as described in Part 7 of the Monash University (Council) Regulations (academic misconduct).
- I have taken proper care to safeguard this work and made all reasonable efforts to ensure it could not be copied.
- No part of this assignment has been previously submitted as part of another unit/course.
- I acknowledge and agree that the assessor of this assignment may, for the purposes of assessment, reproduce the assignment and:

  i. provide it to another member of faculty and any external marker; and/or

  ii. submit to a text matching/originality checking software; and/or

  iii. submit it to a text matching/originality checking software which may then retain a copy of the assignment on its database for the purpose of future plagiarism checking.

- I certify that I have not plagiarised the work of others or participated in unauthorised collaboration or otherwise breached the academic integrity requirements in the Student Academic Integrity Policy.

Date: ..07../..05../..2023...... Signature:....Tan Jin Chun.................................. *

Privacy Statement:

For information about how the University deals with your personal information go to http://privacy.monash.edu.au/guidelines/collection-personal-information.html#enrol

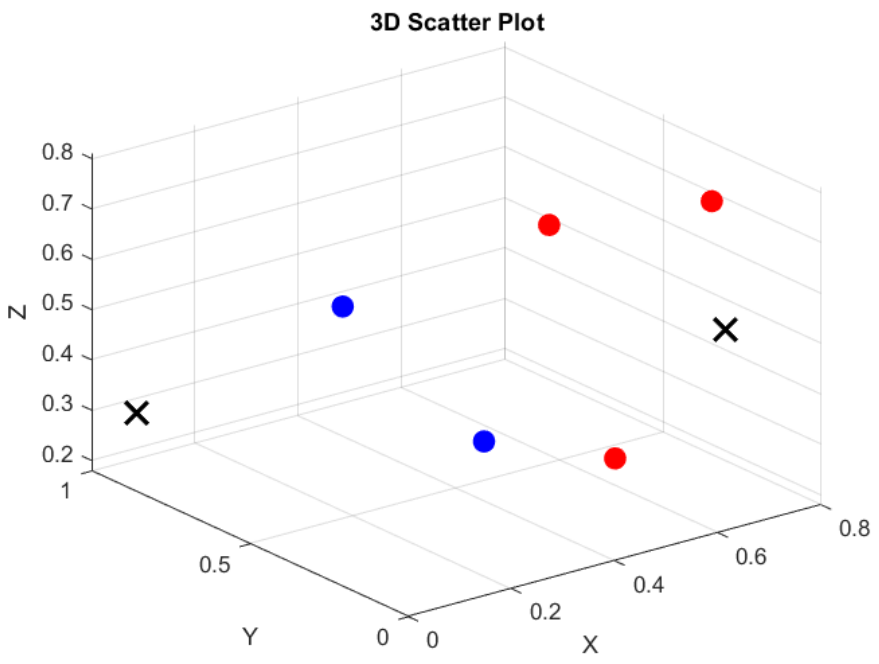
ECE4076 lab 3 results documents

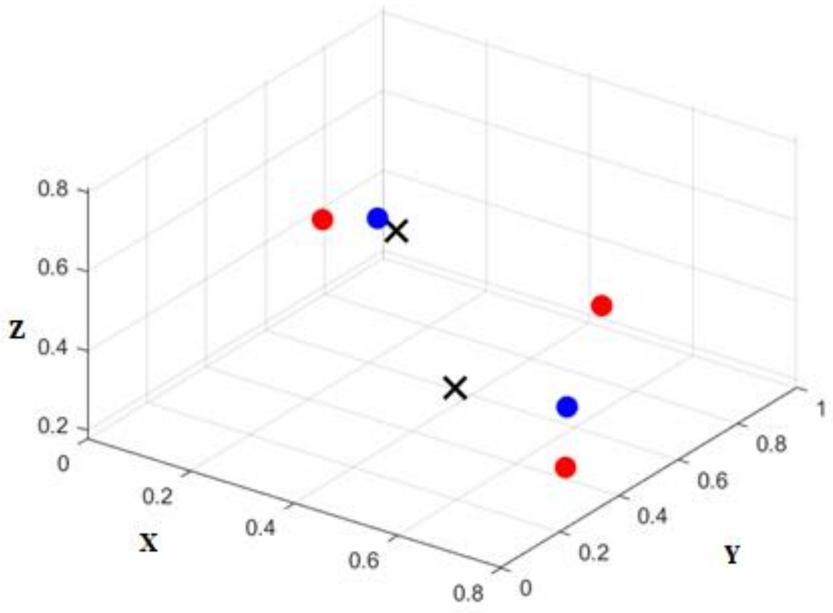
Name: **Tan Jin Chun (32194471)**

Task 1 (1 mark)

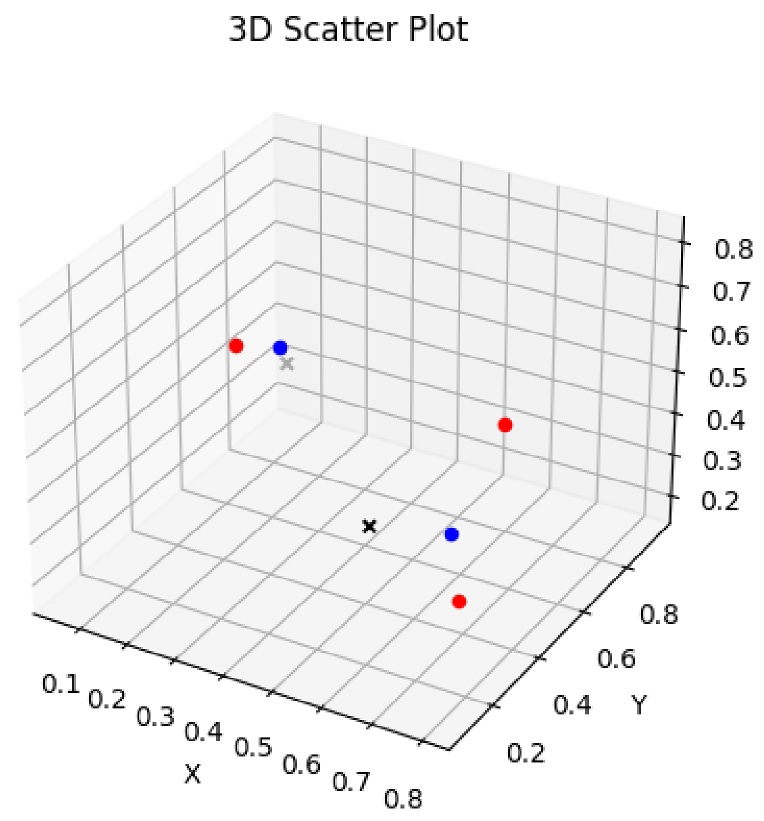
Insert the 3D scatter plot assigning each datapoint to its closest centroid using two colors:

Plotted With MATLAB





Plotted with Python



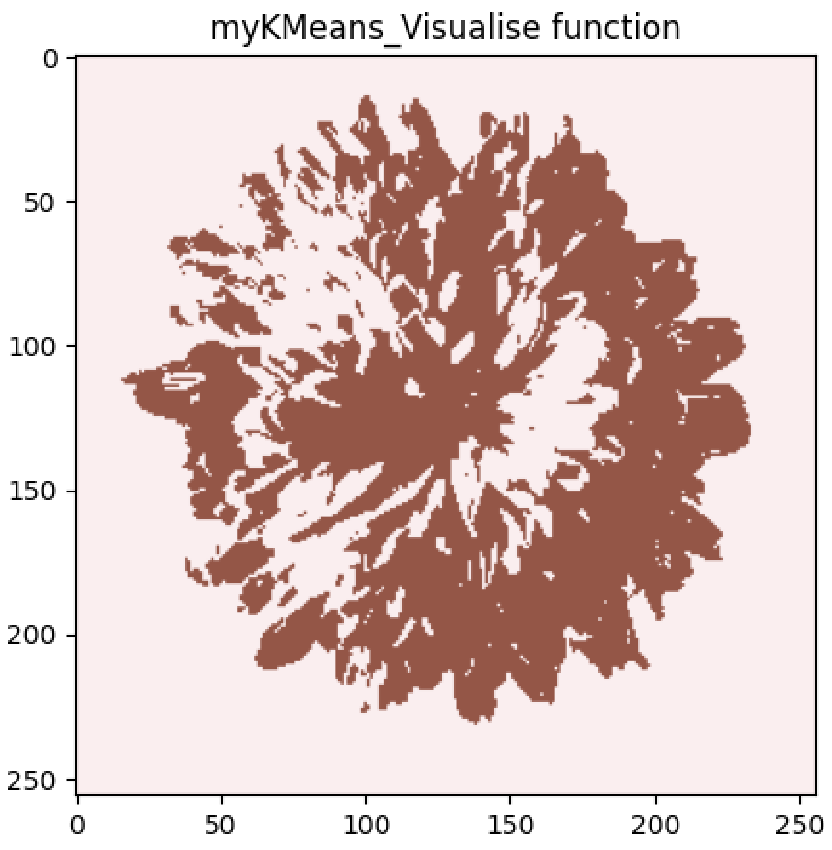
What do you observe here? Are the data points assigned to their closest centroid?

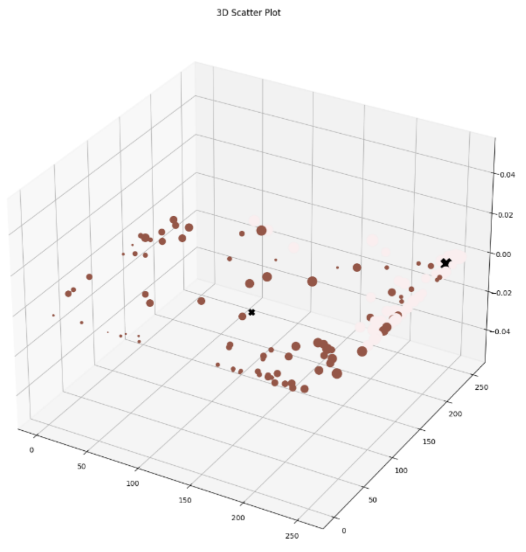
As we can see from the above 3D Scatter Plots, the data points are assigned to their closest centroid. There will be 2 blue points assigned to the first centroid and 3 red points assigned to the second centroids.

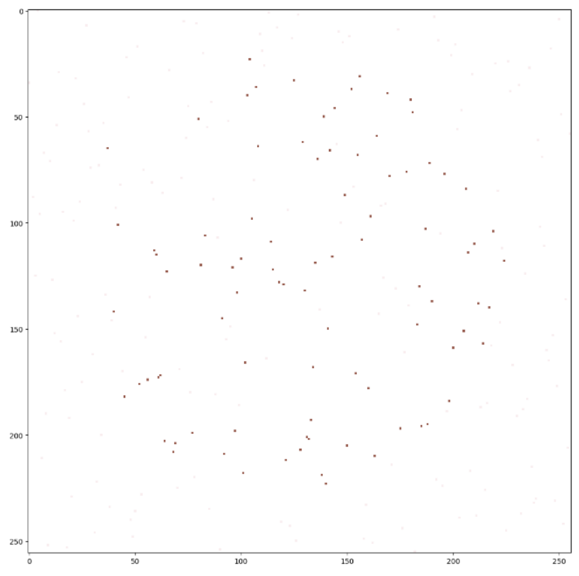
Task 2 (2 marks)

Have some fun with your k-Means Clustering implementation! Run the program several times with different random seeds to see if you always converge to the same solution. Try changing k from 4 to other numbers (from 2 to 10) and see how this affects the output and the repeatability of the program. Report and discuss your observations.

When k = 2

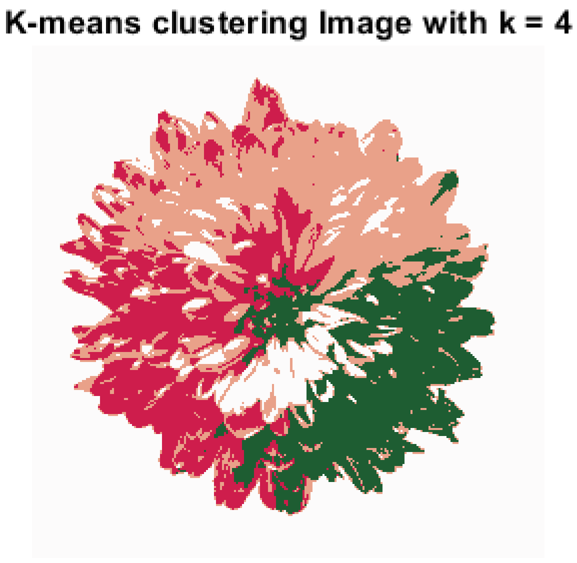


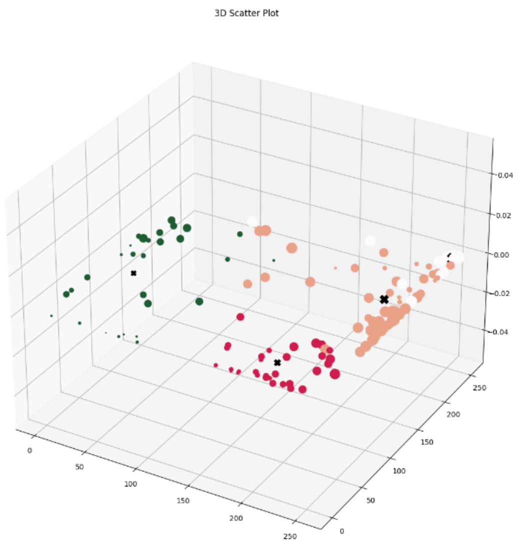


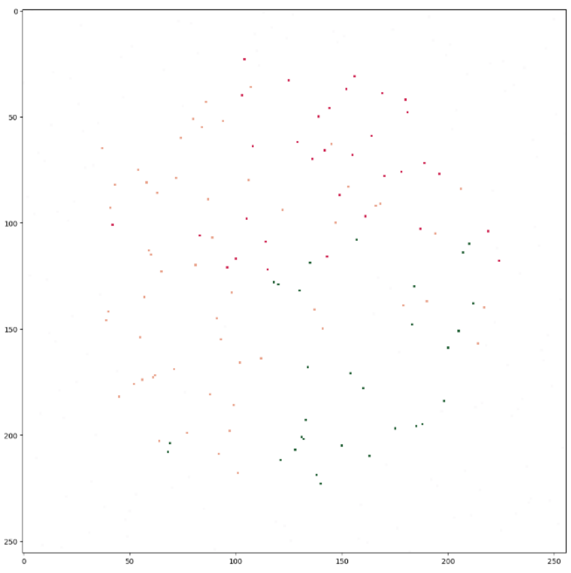


Insert the final result of your clustered sharon.jpg image here (where each pixel is colour coded with the newly computed mean) for k = 4 means. Also, insert the final output of the 3D scatter plot of the data points and centroids for k = 4 means. Also, include the corresponding iteration number and the loss:

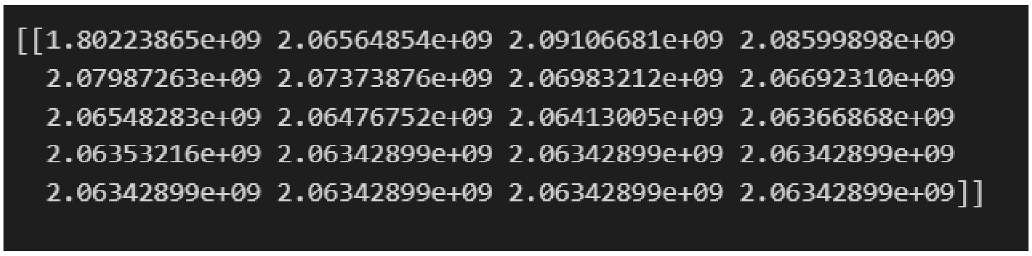
When k = 4







Iteration number: 20
Loss:



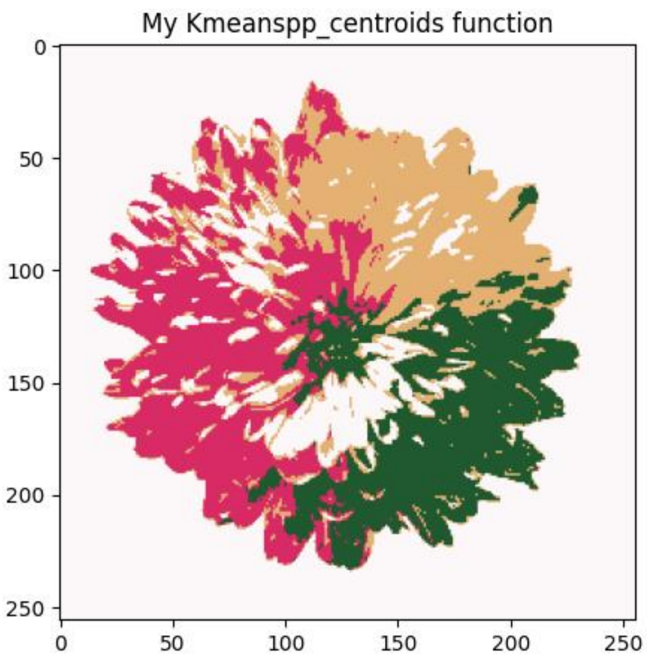
```
[[1.80223865e+09 2.06564854e+09 2.09106681e+09 2.08599898e+09
  2.07987263e+09 2.07373876e+09 2.06983212e+09 2.06692310e+09
  2.06548283e+09 2.06476752e+09 2.06413005e+09 2.06366868e+09
  2.06353216e+09 2.06342899e+09 2.06342899e+09 2.06342899e+09
  2.06342899e+09 2.06342899e+09 2.06342899e+09 2.06342899e+09]]
```

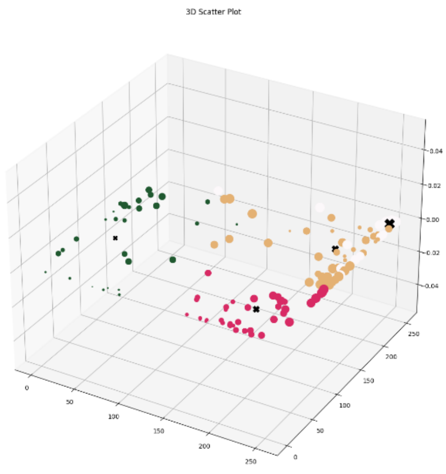
Observe how the centroids move with each iteration. Did you see any pattern in the movement of each centroid? Report and discuss your observations!

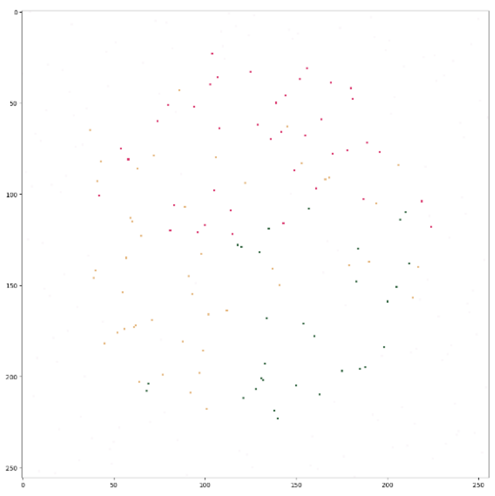
The centroids can be seen moving slowly to each of their respective clusters. Some of the centroids will also remain at its original position. Initially, the centroids are far from the optimal position but eventually it will reach a stable position (converges to a stable position).

Task 3 (1 mark)

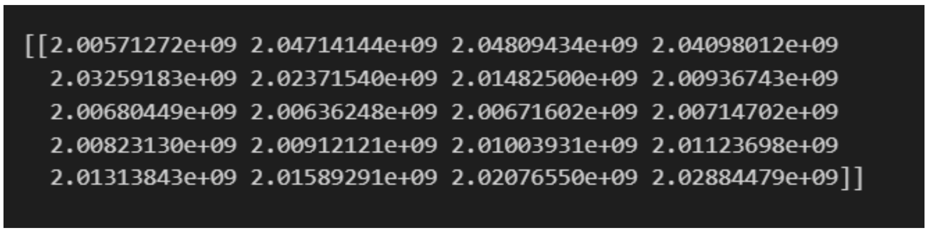
Insert final clustered sharon.jpg image with k-Means++ initialization here (where each pixel is colour coded with the newly computed mean) for k = 4 means:







Loss:

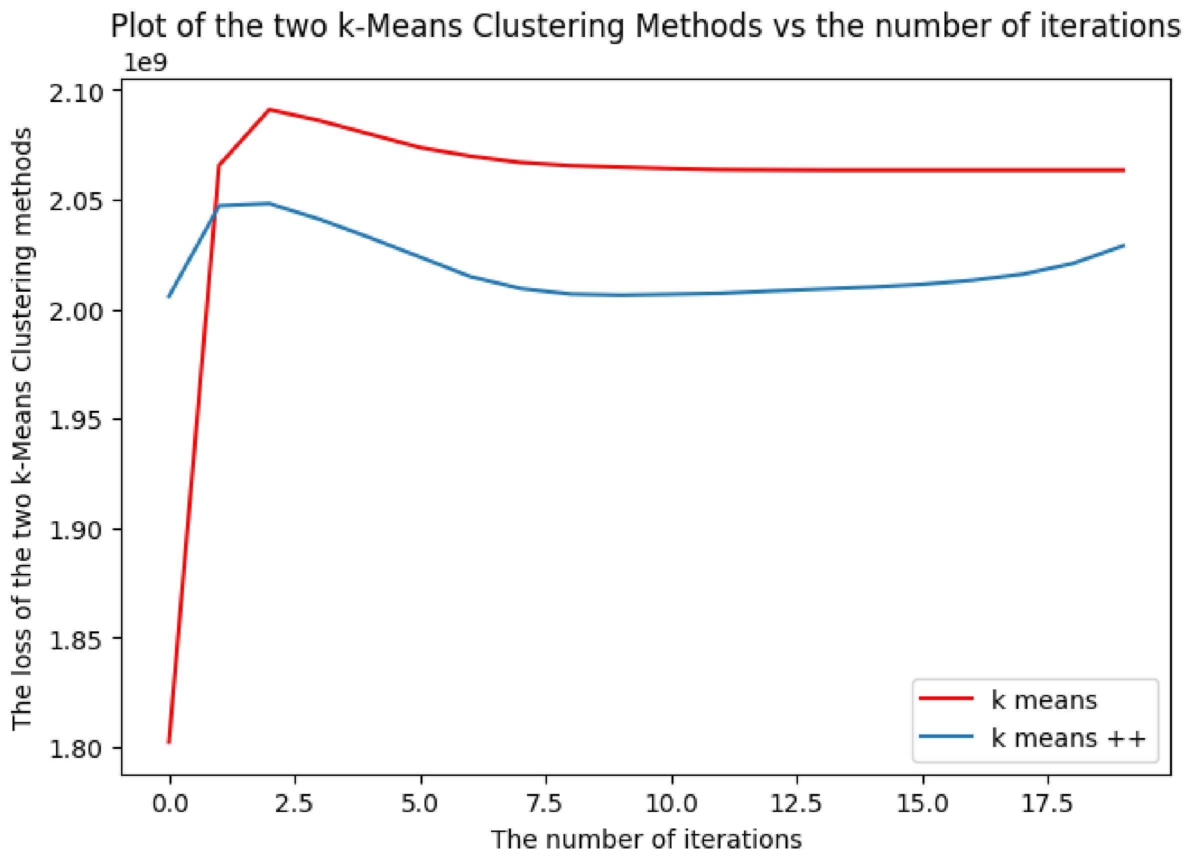


```
[[2.00571272e+09 2.04714144e+09 2.04809434e+09 2.04098012e+09
  2.03259183e+09 2.02371540e+09 2.01482500e+09 2.00936743e+09
  2.00680449e+09 2.00636248e+09 2.00671602e+09 2.00714702e+09
  2.00823130e+09 2.00912121e+09 2.01003931e+09 2.01123698e+09
  2.01313843e+09 2.01589291e+09 2.02076550e+09 2.02884479e+09]]
```

Were you able to obtain the same results (the same clustered sharon image) as you did with the random initialization, or do they differ significantly? Were you able to obtain the results faster than with the random initialization? Report your findings, and explain why you think this happens!

I was able to obtain the same results (the same clustered Sharon image) as I did with the random initialization. Due to randomization of point, they may differ slightly but nonetheless the same. I was able to obtain the results slightly faster than the random initialization. I suspect that the result was produced faster as the loss decreased faster.

Include the plot of the loss of the two k-Means Clustering methods over the same number of iterations:



From your results, discuss the main differences between random initialization and k-Means++ initialization. What do you notice regarding the position of the initial centroids? Which one do you think is better, and why? What do you observe regarding the loss and convergence of both methods? Discuss the pros and cons of the two methods.

From my results, the main differences between the random initialization and the k-means++ initialization is that the position of the initial centroids for the random initialization will lead to the poor placement of the centroids. This would lead to a slower convergence and potentially undesirable results. K-Means++ initialization will place the initial centroids more optimally and will cover the data points more uniformly. This will lead to a better clustering result.

From the graph that we have plotted, we can see that the K-means++ Initialization will converge much faster when compared to the random initialization and there is a lower final loss in k-Means++.

My conclusion is that the k-Means++ algorithm is better due to the reason that I have stated above.

Pros & Cons

Random Initialization

Pros

1) Easy to implement

2) Computationally Less Expensive

Cons

1) Slower convergence

2) Potentially Higher Loss

k-Means++ Initialization

Pros

1) Faster Convergence

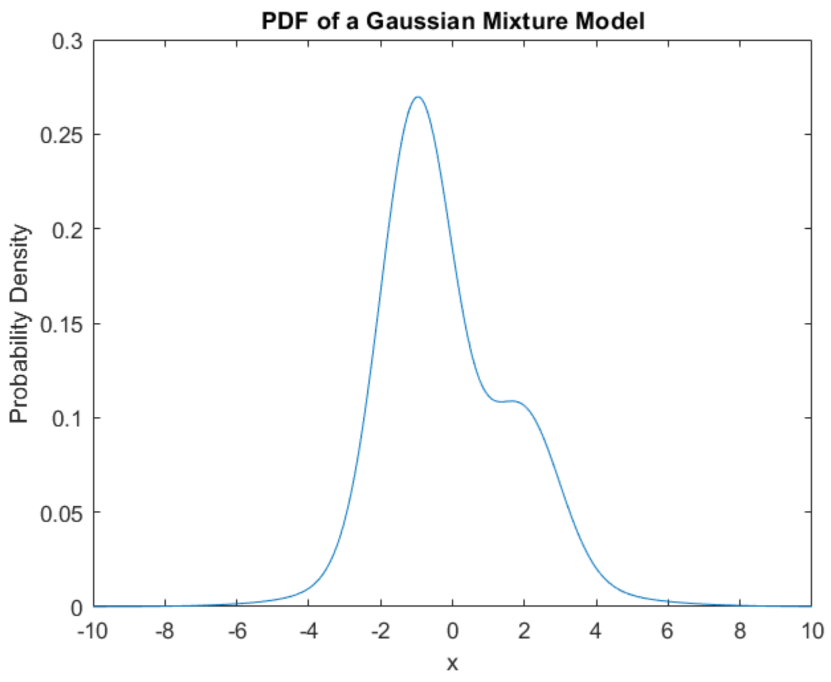
2) Lower Final Loss

Cons

1) Computationally More Expensive

Task 4 (1 mark)

Insert the plot the pdf of the GMM between [-10,10]:



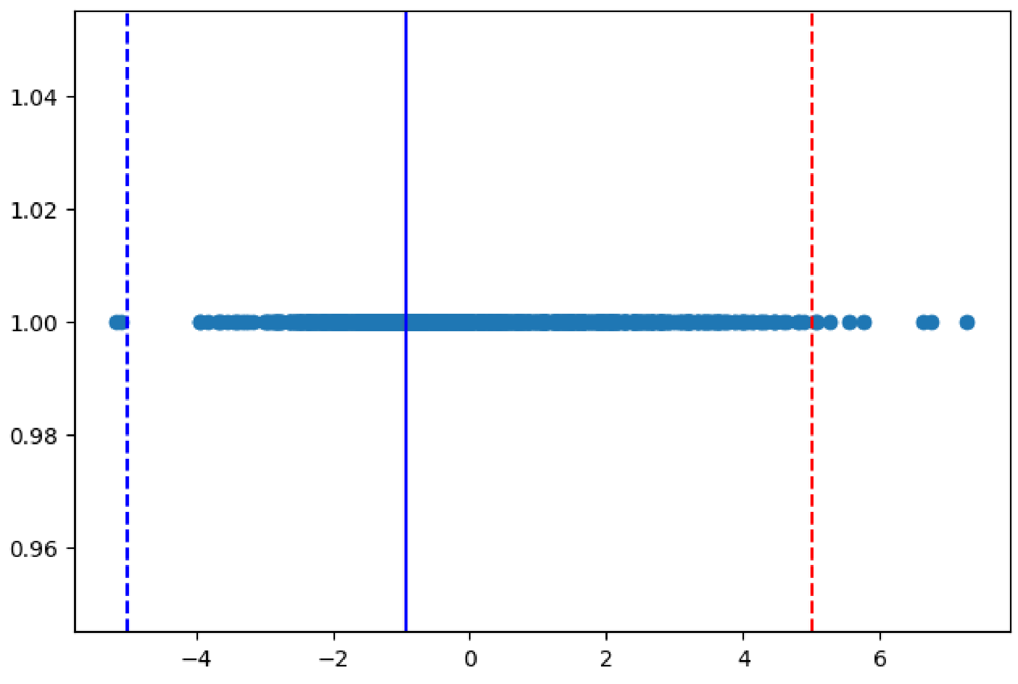
Explain what the plot displayed in the image 'Task4_GMM_01_what_is_a_pdf.png' shows.

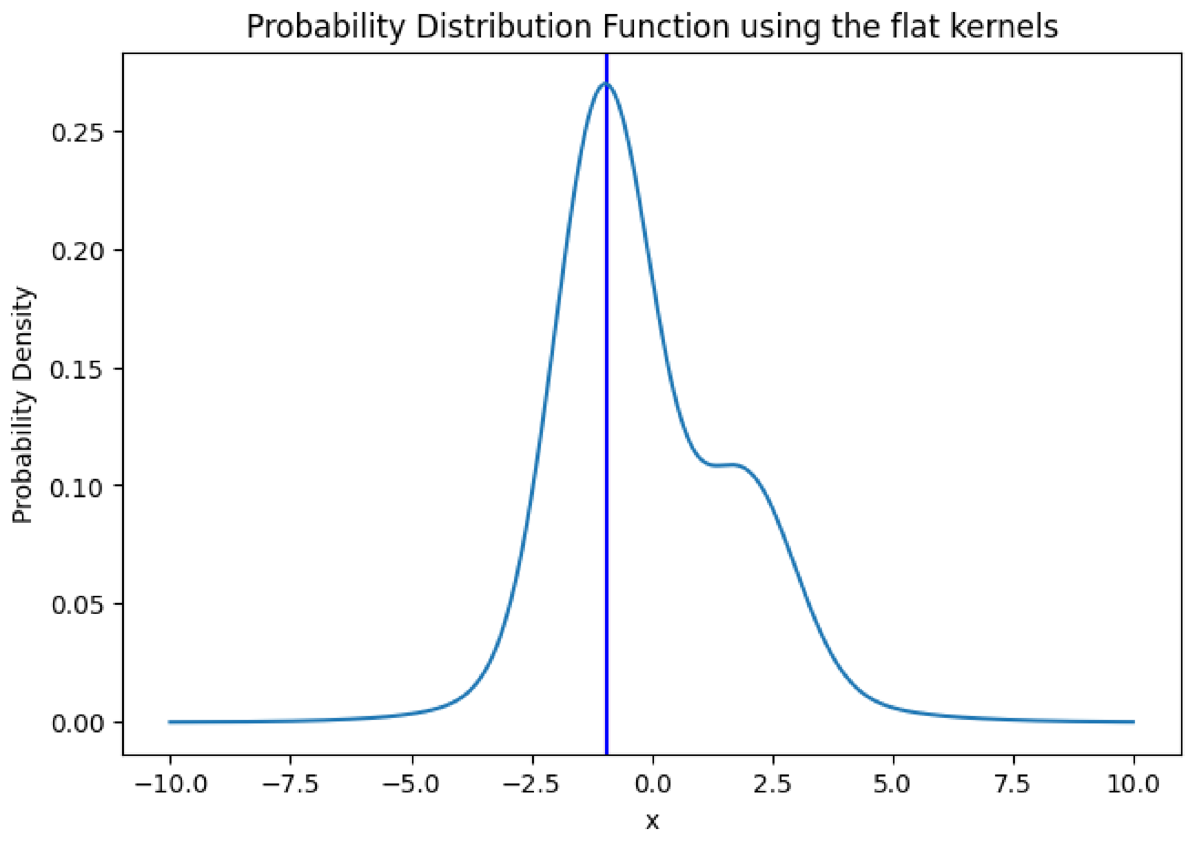
As we can see on the plot displayed in the image 'Task4_GMM_01_what_is_a_pdf .png', we can observe that the majority of the red points lies in the region where the probability density of the graph is high. We can also observe that there are only a very small number of outliers in the given graph.

Explain what you understand from the KDE code snippet given. Also explain what you understand from the KDE plot given in Task4_GMM_02_kde.png.

Based on the KDE code snippet given, I have deduced that the KDE will estimate the pdf of the samples based on the positions of the samples acquired by random. Based on the graph that we have created, the pdf predictions are very closely match to our made predictions

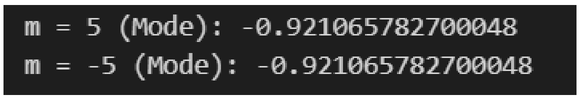
Task 5 (2 marks)
Insert the plot of the starting points(s), calculated modes and PDF using the flat kernel. Also, give the computed centers for starting from points 5 and -5 after convergence.





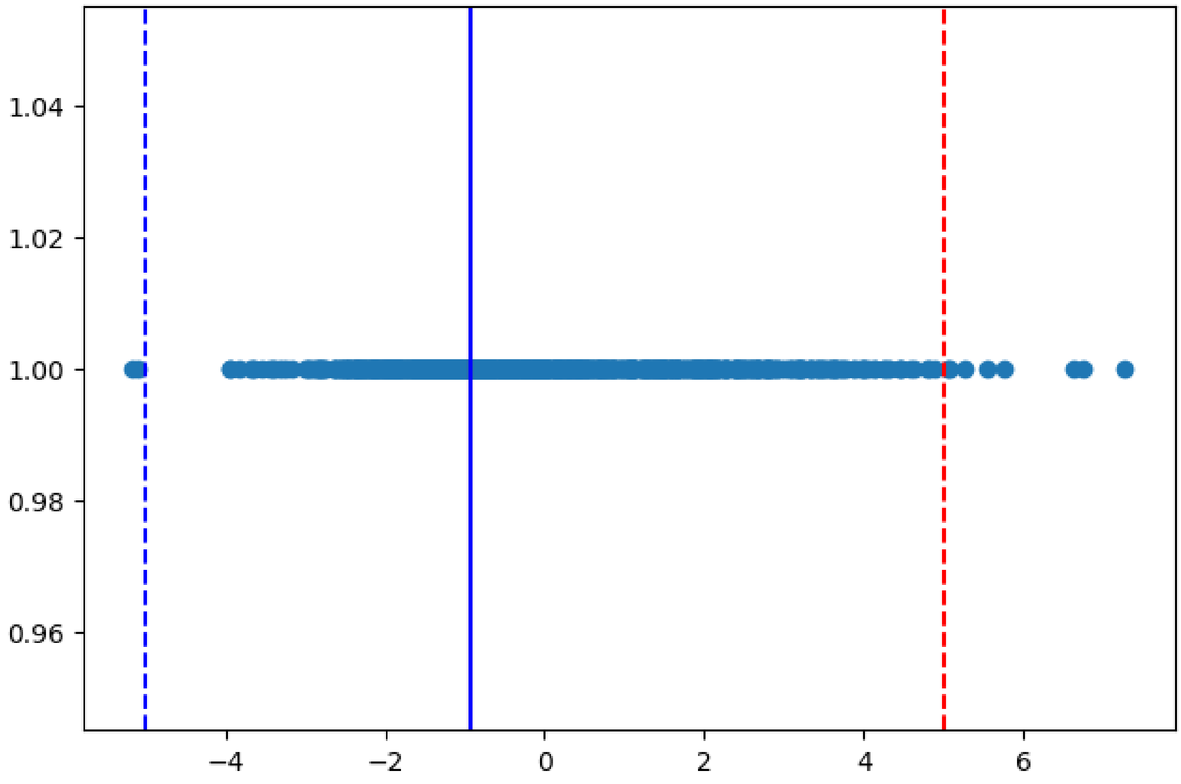
Computed center for starting from point 5 after convergence: -0.921

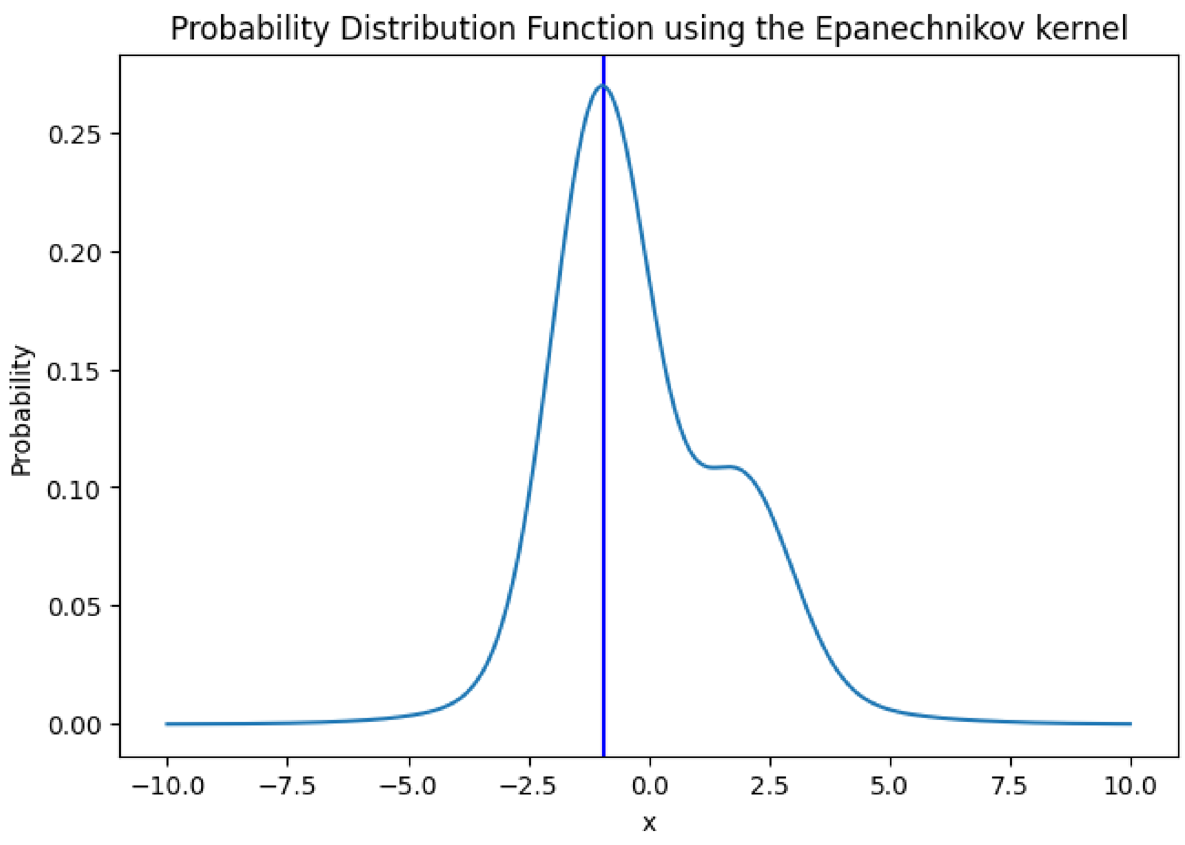
Computed center for starting from point -5 after convergence: -0.921



```
m = 5 (Mode): -0.921065782700048
m = -5 (Mode): -0.921065782700048
```

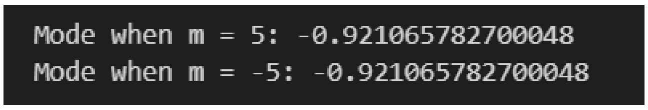
Insert the plot of the starting points(s), calculated modes and PDF using the Epanechnikov kernel. Also, give the computed centers for starting from points 5 and -5 after convergence.





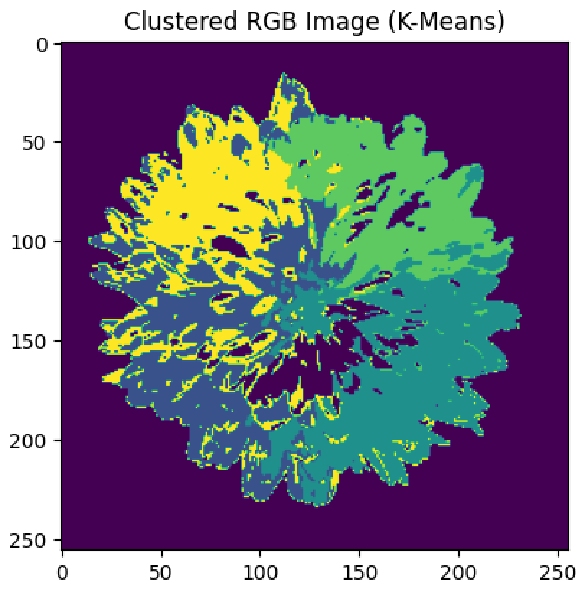
Computed center for starting from point 5 after convergence: -0.921

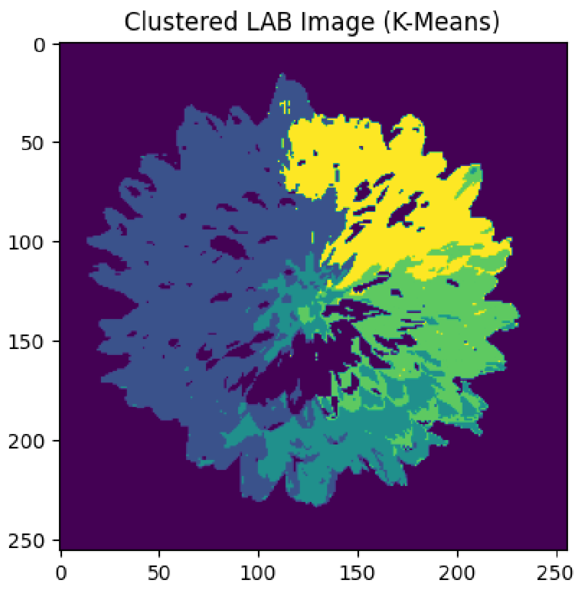
Computed center for starting from point -5 after convergence: -0.921



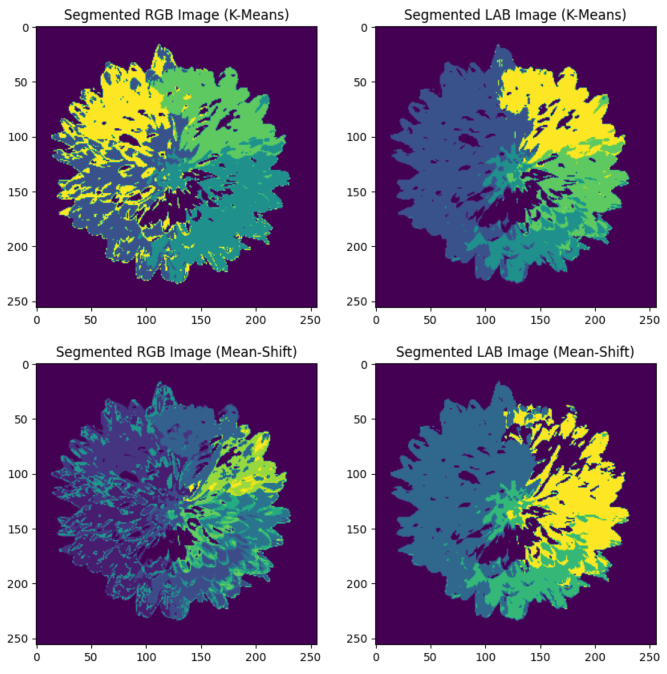
```
Mode when m = 5: -0.921065782700048
Mode when m = -5: -0.921065782700048
```

Insert the final results of your k-means clustered sharon.jpg image for 2 different spaces, RGB and LAB here (where each pixel is colour coded with the newly computed mean) for k = 4 means. Make sure to display all results in the RGB space (and convert appropriately – convert the LAB image result to RGB before displaying it):





Include visualisations for all 4 resulting clustered images: k-Means RGB & LAB, as well as mean shift RGB & LAB. Hint: As before, make sure to display all results in the RGB space (and convert appropriately):



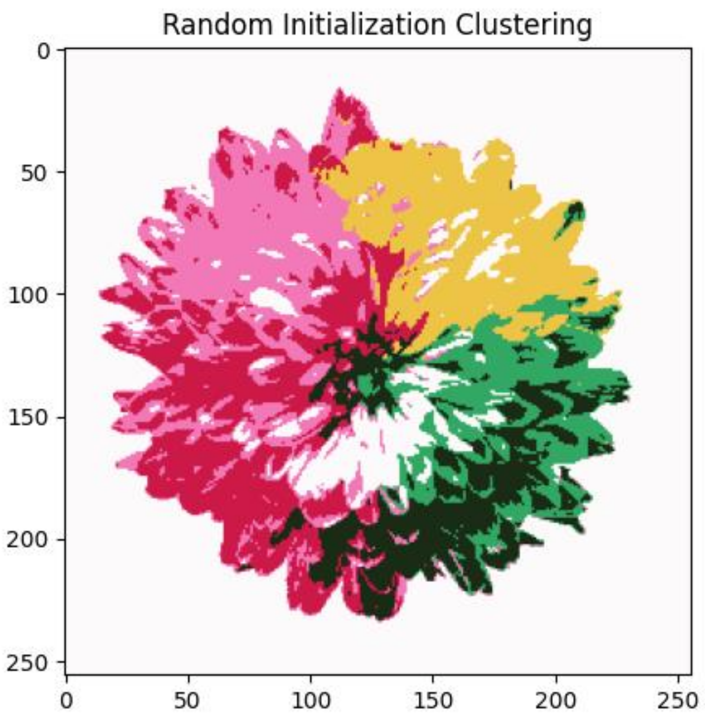
What do you notice about the mean-shift algorithm compared to the k-means for the RGB and LAB images?

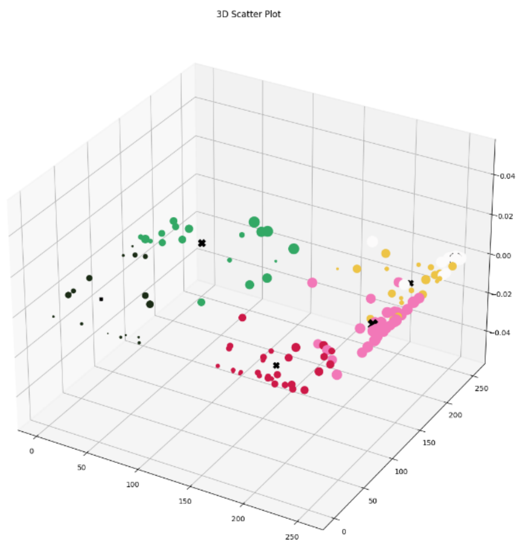
I have noticed that the mean-shift algorithm compared to the k-means for LAB images looks a bit similar. However, when comparing the mean-shift algorithm and the k-means for RGB images, the mean-shift algorithm has a clear discoloration on the image itself. This could mean that the mean-shift might be less sensitive to the initialization than k-means.

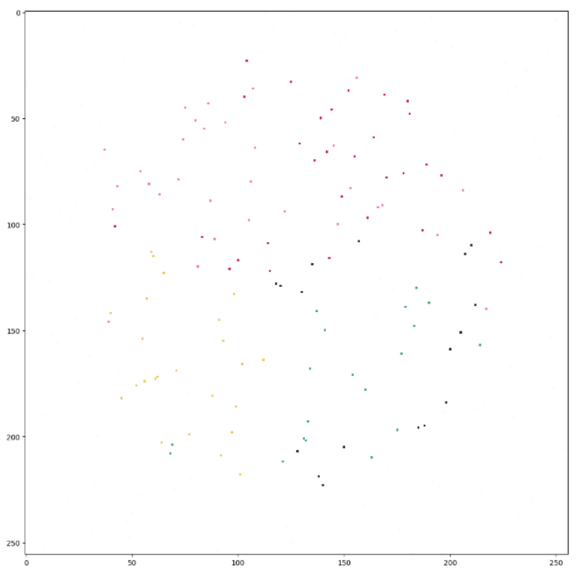
Task 6 (1 mark)

Insert plots of generated final centroids from each initialization method together with the final clustering result of the data points (by using corresponding centroid colours). For the two k-means methods, additionally visualise their ***initial*** *centroids after initialisation. You may want generate a side-by-side plot for easier comparison of the three methods (e.g. via subplots). Also make sure to clearly label your methods as well as axes in the plot.*

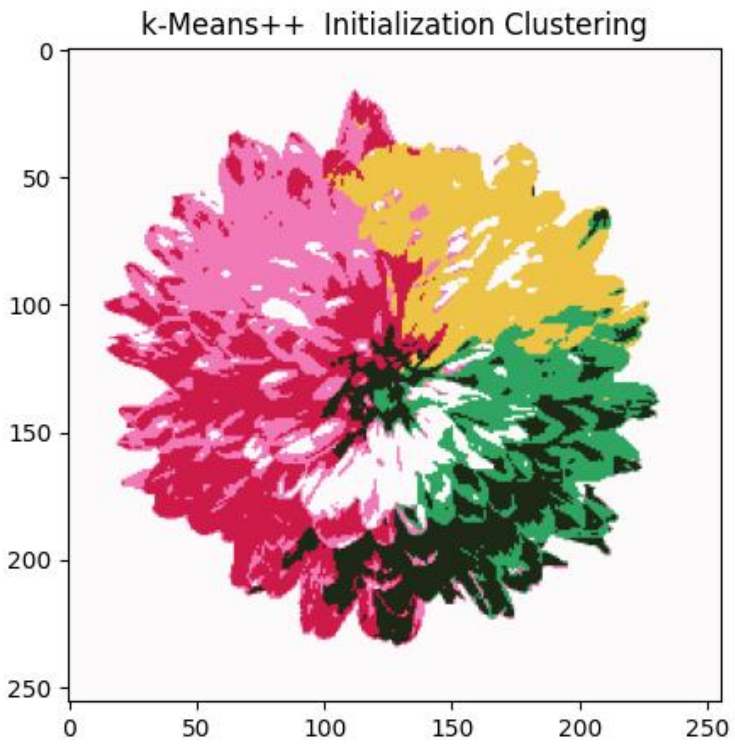
1) Random Initialization

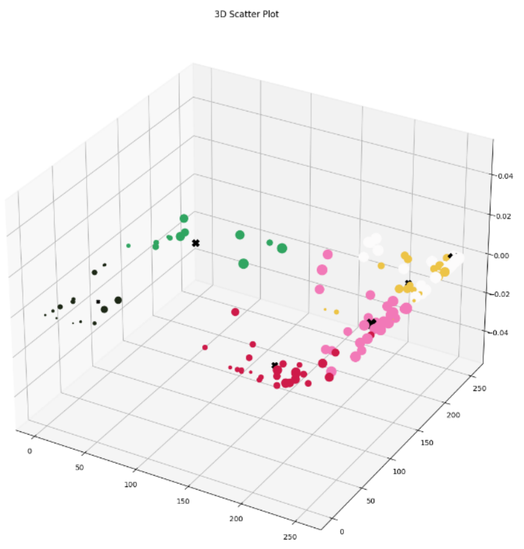


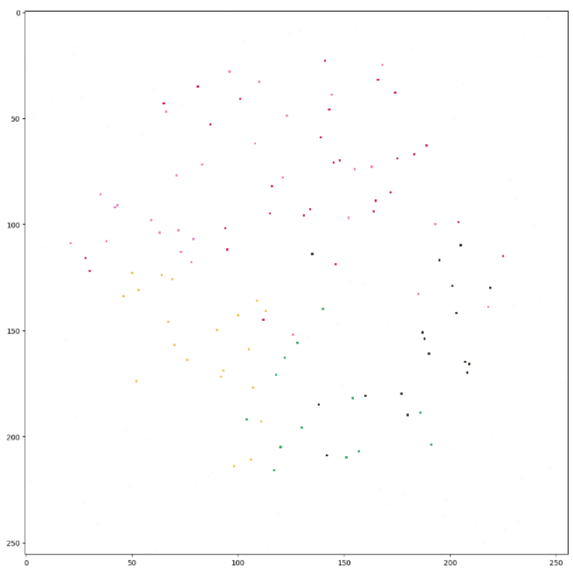




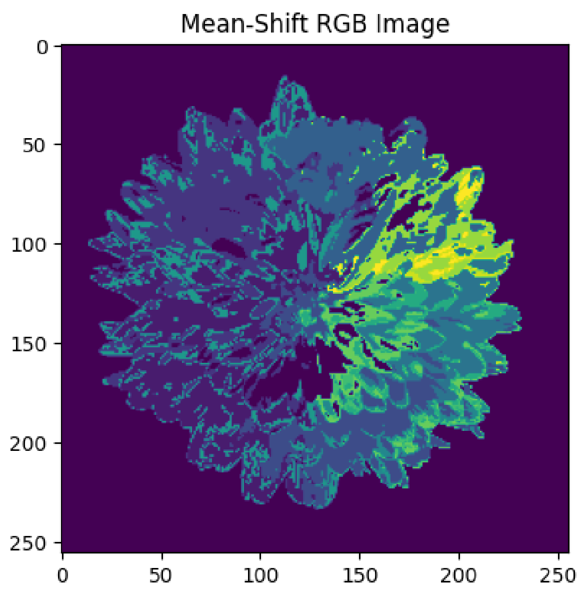
2) K-Means++ Initialization

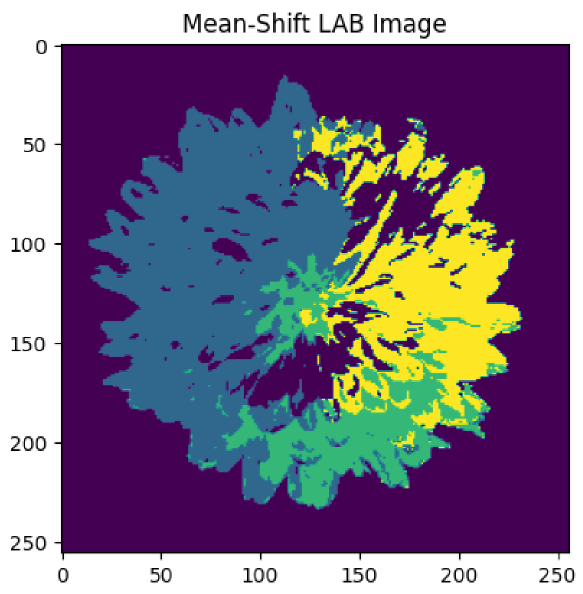






3) Mean Shift Clustering





What do you notice about the mean-shift clustering compared to the k-means algorithms? And how do the two k-means initializations differ?

As we can see from the above graph, The k-Means++ initialization method will result in better clustering, as it ensures a more optimal initial placement of centroids, leading to faster convergence which will lead to a better result. We can also notice that as the centroids move closer to the various clusters, the k-means changes. The means produced from the mean shift will move to each nearby mean.

The mean-shift algorithm does not require the specification of the number of clusters. It works by finding the densest regions in the feature space, making it suitable for cases where the number of clusters is unknown or not well-defined. It may also have difficulty detecting clusters with different densities or irregular shapes.

We can notice that the k-Means++ initialization often result in better clustering than random initialization as it minimizes the chances of poor centroid initialization. Mean-shift can adapt to the underlying structure of the data but it might be sensitive to the choice of bandwidth parameter and may not perform well when the densities of the clusters are very different.

In summary, k-Means with random initialization may not produce the desirable results due to the initial centroid placement but k-Means++ initialization will improve the clustering quality. Mean-shift is a flexible algorithm that can adapt to the data structure but the performance is sensitive to the choice of bandwidth parameter and may struggle with clusters of varying densities or irregular shapes.

Code for Task 1:

Paste your code with unanswered questions and comments in here.

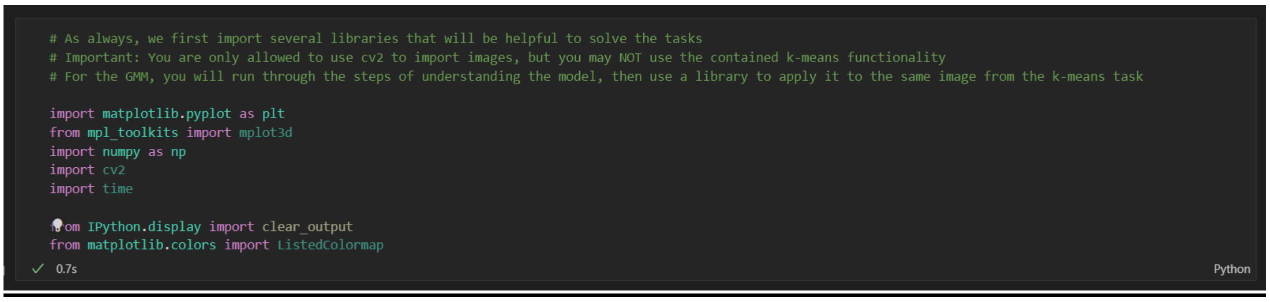
MATLAB CODE

```
% Written by Nigel Tan Jin Chun
% Last Modified: 26/4/2023
% Name of the file: Lab3_Task1
% Function:
% Building a helper function that will compute the squared
% distance from a set of data points to a set of centroids

clear all;clc;close all;

% The given small dataset of 5 data points
X = [0.67187976, 0.44254368, 0.17900127;
     0.55085456, 0.65891464, 0.18370379;
     0.79861987, 0.3439561, 0.68334744;
     0.36695437, 0.15391793, 0.81100023;
     0.22898267, 0.58062367, 0.5637733];

% The given two centroids
M = [0.66441854, 0.08332493, 0.54049661;
     0.05491067, 0.94606233, 0.29515262];

% Calling the function
D = transpose(dist2c(X, M));

% Displaying the matrix
disp(D);

% Assign each point to closest centroid
% D = transpose(D)
% [~, labels] = min(D, [], 2);
[~, labels] = min(D, [], 1);

% Define colors for the points
colors = ['r', 'b'];

% Create 3D scatter plot
figure(1);
% scatter3(X(:, 1), X(:, 2), X(:, 3), 100, labels, 'filled');
for i = 1:size(X, 1)
    scatter3(X(i, 1), X(i, 2), X(i, 3), 100, colors(labels(i)), 'filled'); hold
on;
end

% Define colors for centroids
centroid_colors = ['k', 'k'];

% Plot the centroids
hold on;
% scatter3(M(:, 1), M(:, 2), M(:, 3), 200, 'r', 'X', 'LineWidth',2);
for i = 1:size(M, 1)
    scatter3(M(i, 1), M(i, 2), M(i, 3), 200, centroid_colors(i), 'X',
'LineWidth',2);
    hold on;
end
hold off;
```

```
% Labelling the graph
xlabel('X')
ylabel('Y')
zlabel('Z')
```

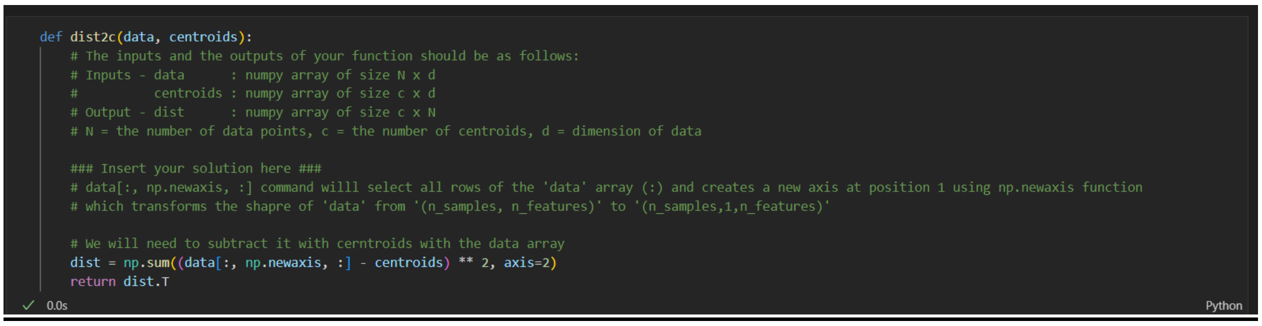
Python Code

Declaring and importing the libraries

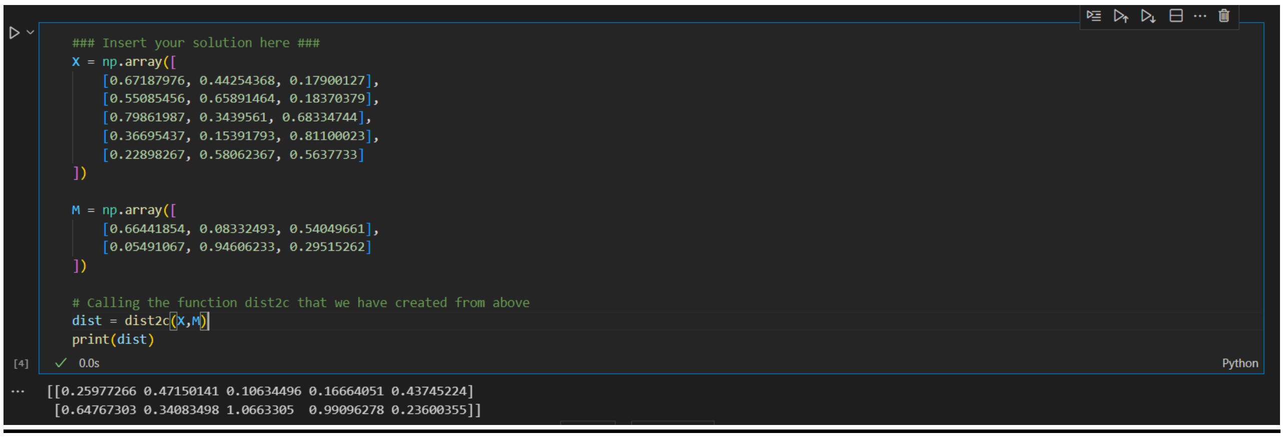


```python
# As always, we first import several libraries that will be helpful to solve the tasks
# Important: You are only allowed to use cv2 to import images, but you may NOT use the contained k-means functionality
# For the GMM, you will run through the steps of understanding the model, then use a library to apply it to the same image from the k-means task

import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np
import cv2
import time

from IPython.display import clear_output
from matplotlib.colors import ListedColormap
```

dist2c function

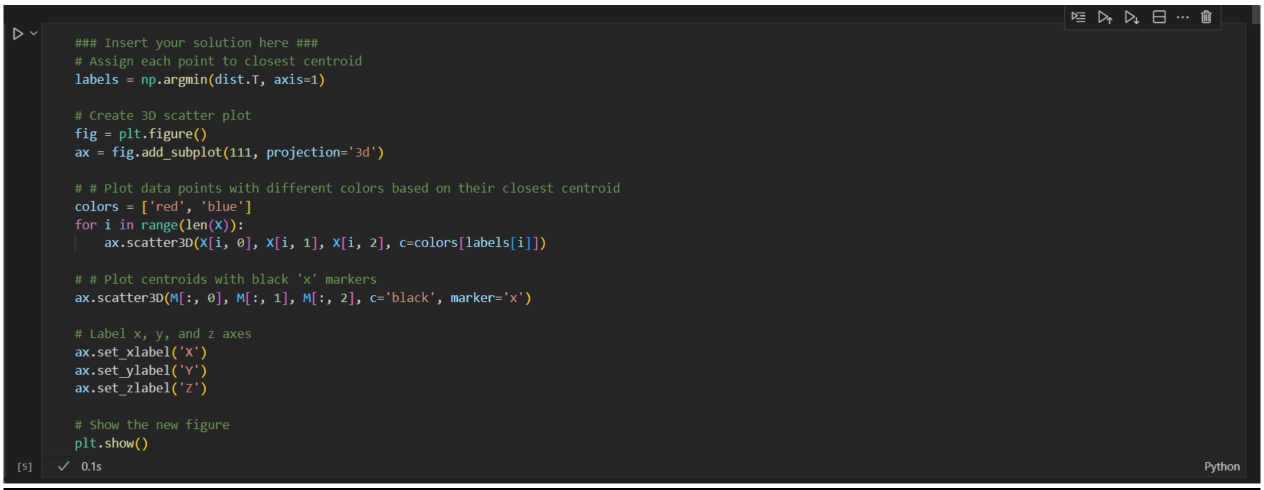


```python
def dist2c(data, centroids):
    # The inputs and the outputs of your function should be as follows:
    # Inputs - data       : numpy array of size N x d
    #           centroids : numpy array of size c x d
    # Output - dist       : numpy array of size c x N
    # N = the number of data points, c = the number of centroids, d = dimension of data

    ### Insert your solution here ###
    # data[:, np.newaxis, :] command willl select all rows of the 'data' array (:) and creates a new axis at position 1 using np.newaxis function
    # which transforms the shapre of 'data' from '(n_samples, n_features)' to '(n_samples,1,n_features)'

    # We will need to subtract it with cerntroids with the data array
    dist = np.sum((data[:, np.newaxis, :] - centroids) ** 2, axis=2)
    return dist.T
```



```python
### Insert your solution here ###
X = np.array([
    [0.67187976, 0.44254368, 0.17900127],
    [0.55085456, 0.65891464, 0.18370379],
    [0.79861987, 0.3439561, 0.68334744],
    [0.36695437, 0.15391793, 0.81100023],
    [0.22898267, 0.58062367, 0.5637733]
])

M = np.array([
    [0.66441854, 0.08332493, 0.54049661],
    [0.05491067, 0.94606233, 0.29515262]
])

# Calling the function dist2c that we have created from above
dist = dist2c(X,M)
print(dist)
```

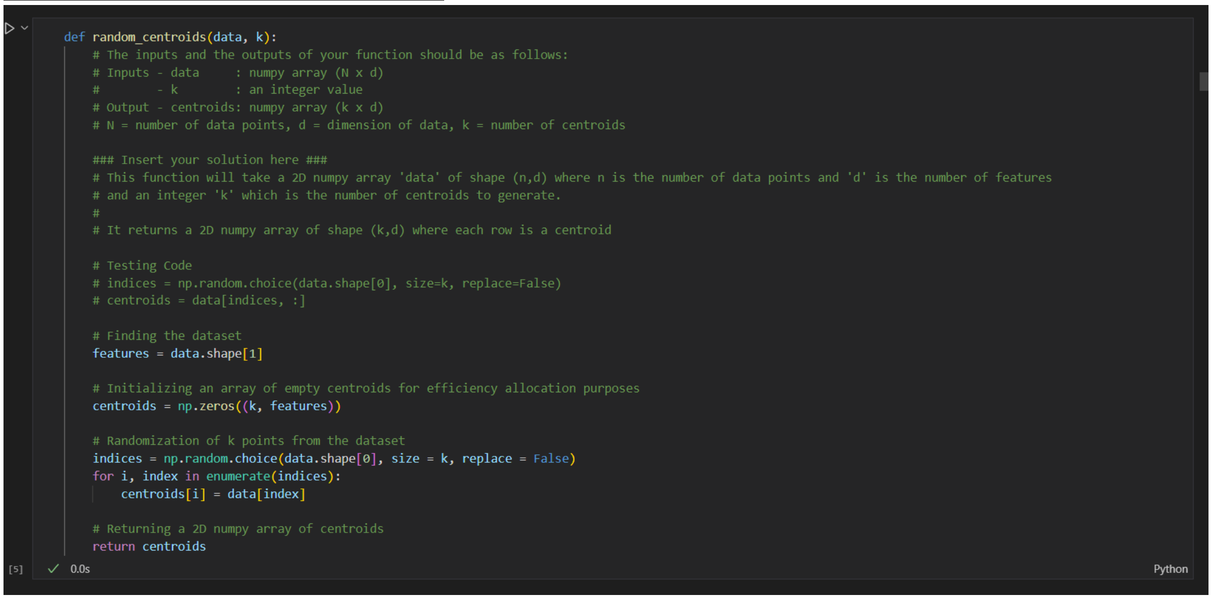
```
[[0.25977266 0.47150141 0.10634496 0.16664051 0.43745224]
 [0.64767303 0.34083498 1.0663305  0.99096278 0.23600355]]
```


```python
### Insert your solution here ###
# Assign each point to closest centroid
labels = np.argmin(dist.T, axis=1)

# Create 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# # Plot data points with different colors based on their closest centroid
colors = ['red', 'blue']
for i in range(len(X)):
    ax.scatter3D(X[i, 0], X[i, 1], X[i, 2], c=colors[labels[i]])

# # Plot centroids with black 'x' markers
ax.scatter3D(M[:, 0], M[:, 1], M[:, 2], c='black', marker='x')

# Label x, y, and z axes
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Show the new figure
plt.show()
```
[5] ✓ 0.1s Python

Code for Task 2:

Paste your code with unanswered questions and comments in here.

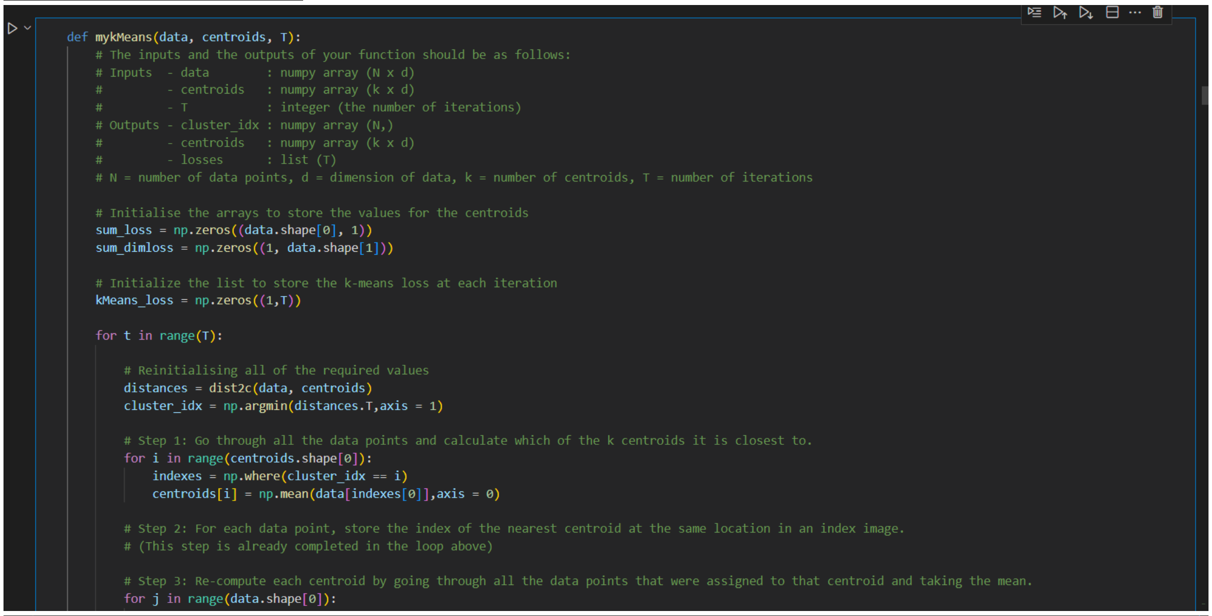
Random Centroids Function



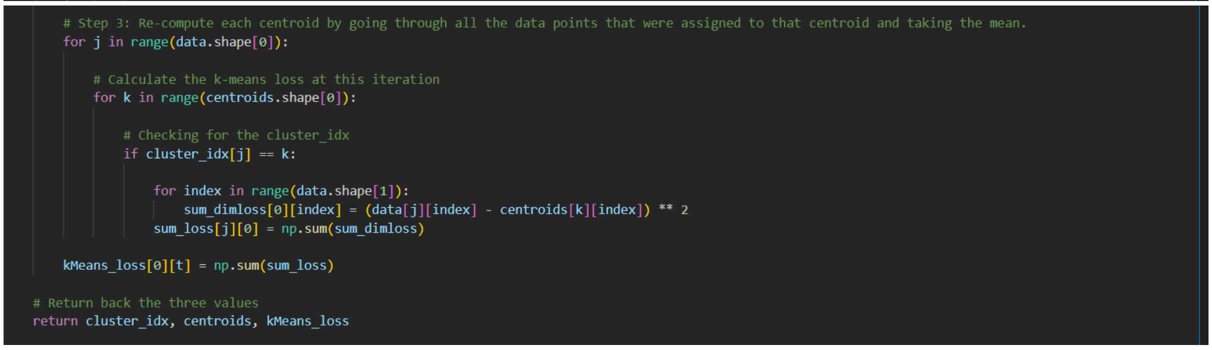
```python
def random_centroids(data, k):
    # The inputs and the outputs of your function should be as follows:
    # Inputs - data     : numpy array (N x d)
    #        - k        : an integer value
    # Output - centroids: numpy array (k x d)
    # N = number of data points, d = dimension of data, k = number of centroids

    ### Insert your solution here ###
    # This function will take a 2D numpy array 'data' of shape (n,d) where n is the number of data points and 'd' is the number of features
    # and an integer 'k' which is the number of centroids to generate.
    #
    # It returns a 2D numpy array of shape (k,d) where each row is a centroid

    # Testing Code
    # indices = np.random.choice(data.shape[0], size=k, replace=False)
    # centroids = data[indices, :]

    # Finding the dataset
    features = data.shape[1]

    # Initializing an array of empty centroids for efficiency allocation purposes
    centroids = np.zeros((k, features))

    # Randomization of k points from the dataset
    indices = np.random.choice(data.shape[0], size = k, replace = False)
    for i, index in enumerate(indices):
        centroids[i] = data[index]

    # Returning a 2D numpy array of centroids
    return centroids
```

[5] ✓ 0.0s Python

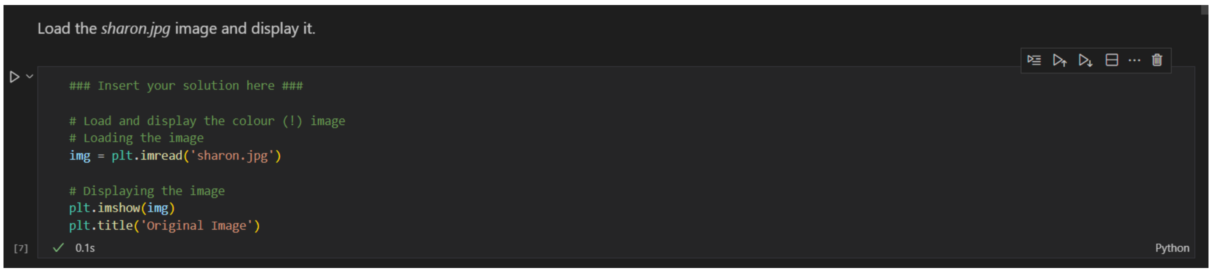
mykMeans function



```python
def mykMeans(data, centroids, T):
    # The inputs and the outputs of your function should be as follows:
    # Inputs  - data        : numpy array (N x d)
    #         - centroids   : numpy array (k x d)
    #         - T           : integer (the number of iterations)
    # Outputs - cluster_idx : numpy array (N,)
    #         - centroids   : numpy array (k x d)
    #         - losses      : list (T)
    # N = number of data points, d = dimension of data, k = number of centroids, T = number of iterations

    # Initialise the arrays to store the values for the centroids
    sum_loss = np.zeros((data.shape[0], 1))
    sum_dimloss = np.zeros((1, data.shape[1]))

    # Initialize the list to store the k-means loss at each iteration
    kMeans_loss = np.zeros((1,T))

    for t in range(T):

        # Reinitialising all of the required values
        distances = dist2c(data, centroids)
        cluster_idx = np.argmin(distances.T,axis = 1)

        # Step 1: Go through all the data points and calculate which of the k centroids it is closest to.
        for i in range(centroids.shape[0]):
            indexes = np.where(cluster_idx == i)
            centroids[i] = np.mean(data[indexes[0]],axis = 0)

        # Step 2: For each data point, store the index of the nearest centroid at the same location in an index image.
        # (This step is already completed in the loop above)

        # Step 3: Re-compute each centroid by going through all the data points that were assigned to that centroid and taking the mean.
        for j in range(data.shape[0]):
```

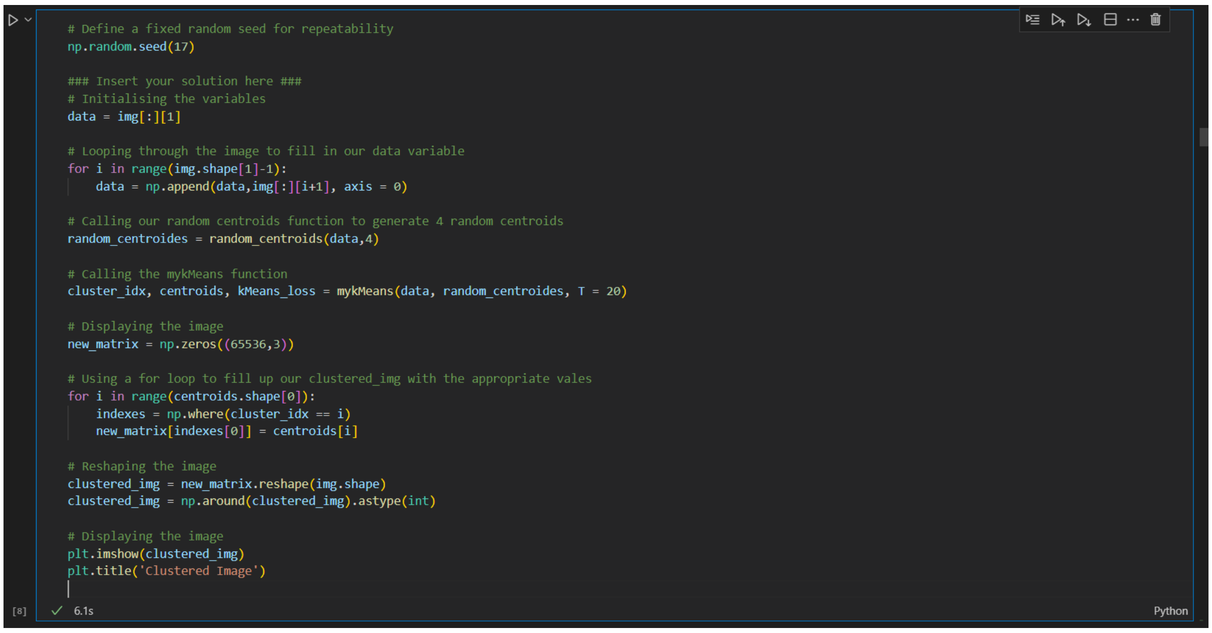


```python
    # Step 3: Re-compute each centroid by going through all the data points that were assigned to that centroid and taking the mean.
    for j in range(data.shape[0]):

        # Calculate the k-means loss at this iteration
        for k in range(centroids.shape[0]):

            # Checking for the cluster_idx
            if cluster_idx[j] == k:

                for index in range(data.shape[1]):
                    sum_dimloss[0][index] = (data[j][index] - centroids[k][index]) ** 2
                sum_loss[j][0] = np.sum(sum_dimloss)

    kMeans_loss[0][t] = np.sum(sum_loss)

# Return back the three values
return cluster_idx, centroids, kMeans_loss
```


Load the *sharon.jpg* image and display it.

```python
### Insert your solution here ###

# Load and display the colour (!) image
# Loading the image
img = plt.imread('sharon.jpg')

# Displaying the image
plt.imshow(img)
plt.title('Original Image')
```

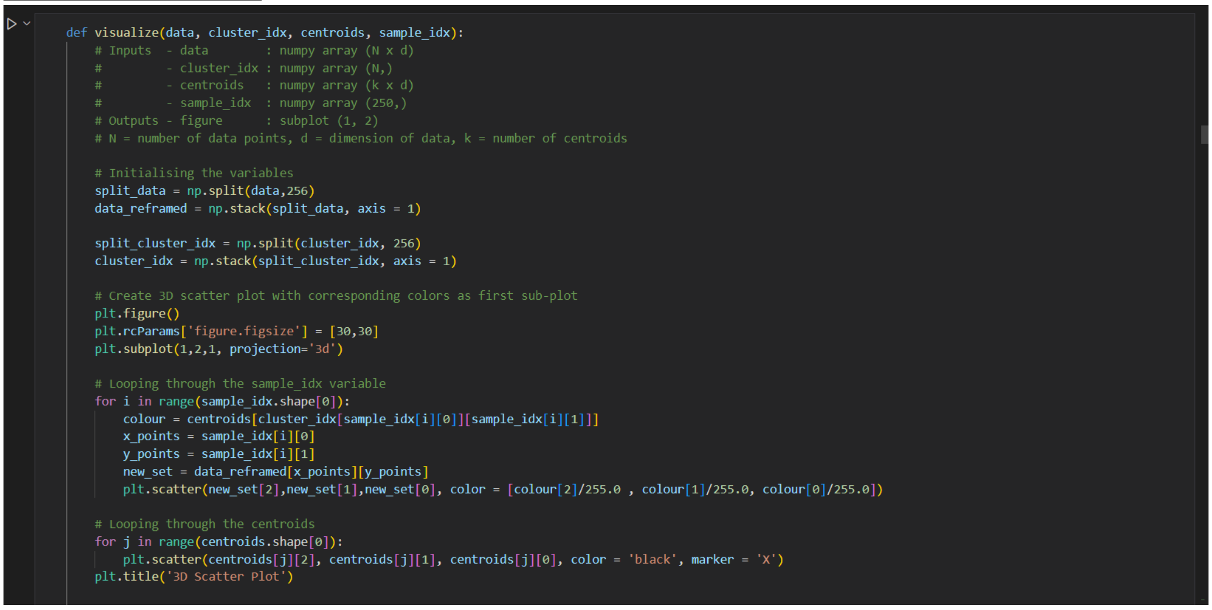
[7] ✓ 0.1s Python



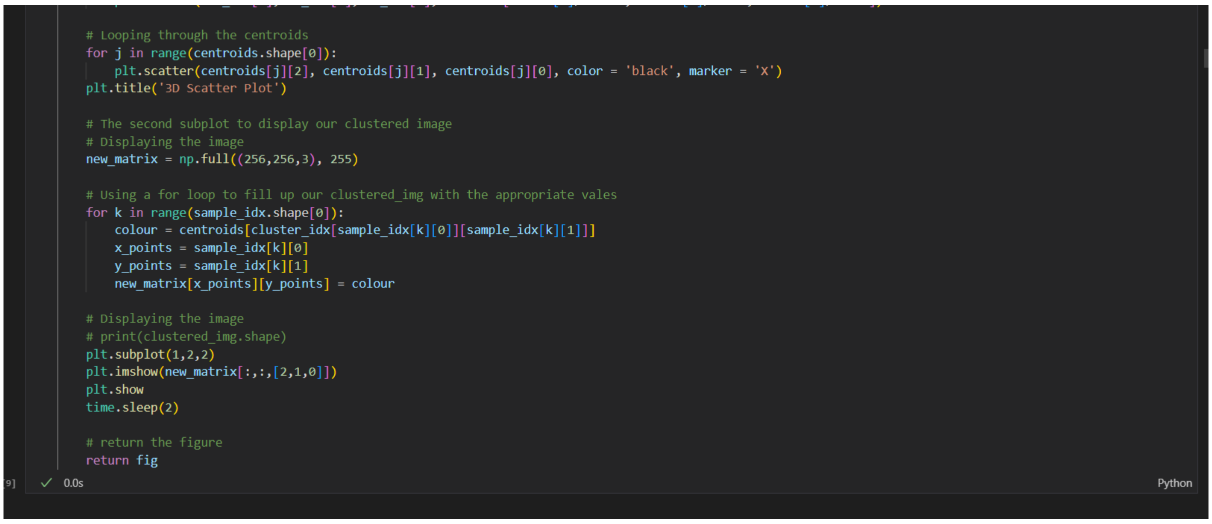
```python
# Define a fixed random seed for repeatability
np.random.seed(17)

### Insert your solution here ###
# Initialising the variables
data = img[:][1]

# Looping through the image to fill in our data variable
for i in range(img.shape[1]-1):
    data = np.append(data,img[:][i+1], axis = 0)

# Calling our random centroids function to generate 4 random centroids
random_centroides = random_centroids(data,4)

# Calling the mykMeans function
cluster_idx, centroids, kMeans_loss = mykMeans(data, random_centroides, T = 20)

# Displaying the image
new_matrix = np.zeros((65536,3))

# Using a for loop to fill up our clustered_img with the appropriate vales
for i in range(centroids.shape[0]):
    indexes = np.where(cluster_idx == i)
    new_matrix[indexes[0]] = centroids[i]

# Reshaping the image
clustered_img = new_matrix.reshape(img.shape)
clustered_img = np.around(clustered_img).astype(int)

# Displaying the image
plt.imshow(clustered_img)
plt.title('Clustered Image')

```

[8] ✓ 6.1s Python

<u>Visualize Function</u>



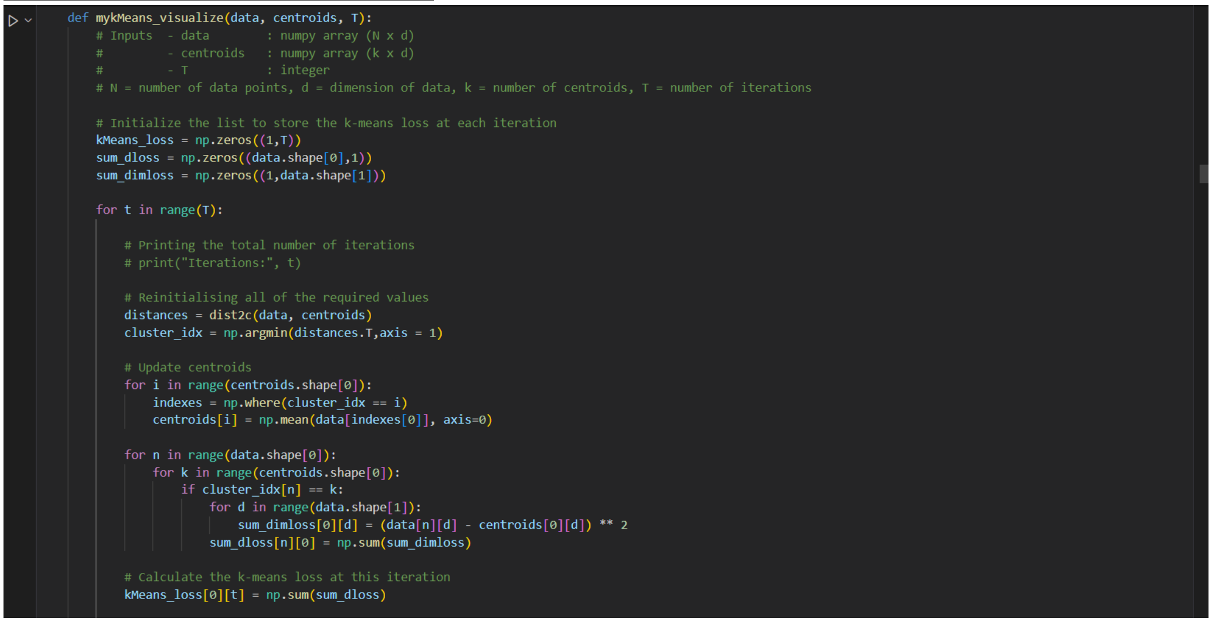
```python
def visualize(data, cluster_idx, centroids, sample_idx):
    # Inputs  - data        : numpy array (N x d)
    #         - cluster_idx : numpy array (N,)
    #         - centroids   : numpy array (k x d)
    #         - sample_idx  : numpy array (250,)
    # Outputs - figure      : subplot (1, 2)
    # N = number of data points, d = dimension of data, k = number of centroids

    # Initialising the variables
    split_data = np.split(data,256)
    data_reframed = np.stack(split_data, axis = 1)

    split_cluster_idx = np.split(cluster_idx, 256)
    cluster_idx = np.stack(split_cluster_idx, axis = 1)

    # Create 3D scatter plot with corresponding colors as first sub-plot
    plt.figure()
    plt.rcParams['figure.figsize'] = [30,30]
    plt.subplot(1,2,1, projection='3d')

    # Looping through the sample_idx variable
    for i in range(sample_idx.shape[0]):
        colour = centroids[cluster_idx[sample_idx[i][0]][sample_idx[i][1]]]
        x_points = sample_idx[i][0]
        y_points = sample_idx[i][1]
        new_set = data_reframed[x_points][y_points]
        plt.scatter(new_set[2],new_set[1],new_set[0], color = [colour[2]/255.0 , colour[1]/255.0, colour[0]/255.0])

    # Looping through the centroids
    for j in range(centroids.shape[0]):
        plt.scatter(centroids[j][2], centroids[j][1], centroids[j][0], color = 'black', marker = 'X')
    plt.title('3D Scatter Plot')
```



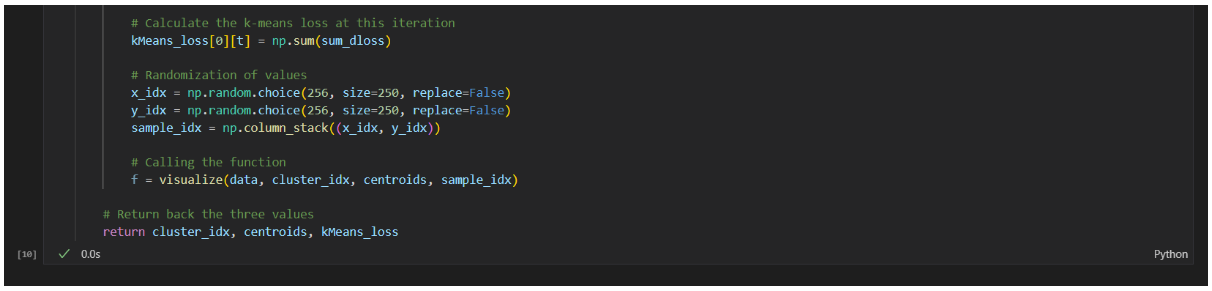
```
    # Looping through the centroids
    for j in range(centroids.shape[0]):
        plt.scatter(centroids[j][2], centroids[j][1], centroids[j][0], color = 'black', marker = 'X')
    plt.title('3D Scatter Plot')

    # The second subplot to display our clustered image
    # Displaying the image
    new_matrix = np.full((256,256,3), 255)

    # Using a for loop to fill up our clustered_img with the appropriate vales
    for k in range(sample_idx.shape[0]):
        colour = centroids[cluster_idx[sample_idx[k][0]][sample_idx[k][1]]]
        x_points = sample_idx[k][0]
        y_points = sample_idx[k][1]
        new_matrix[x_points][y_points] = colour

    # Displaying the image
    # print(clustered_img.shape)
    plt.subplot(1,2,2)
    plt.imshow(new_matrix[:,:,[2,1,0]])
    plt.show
    time.sleep(2)

    # return the figure
    return fig
```

[9] ✓ 0.0s Python

myK-Means Visualize Function

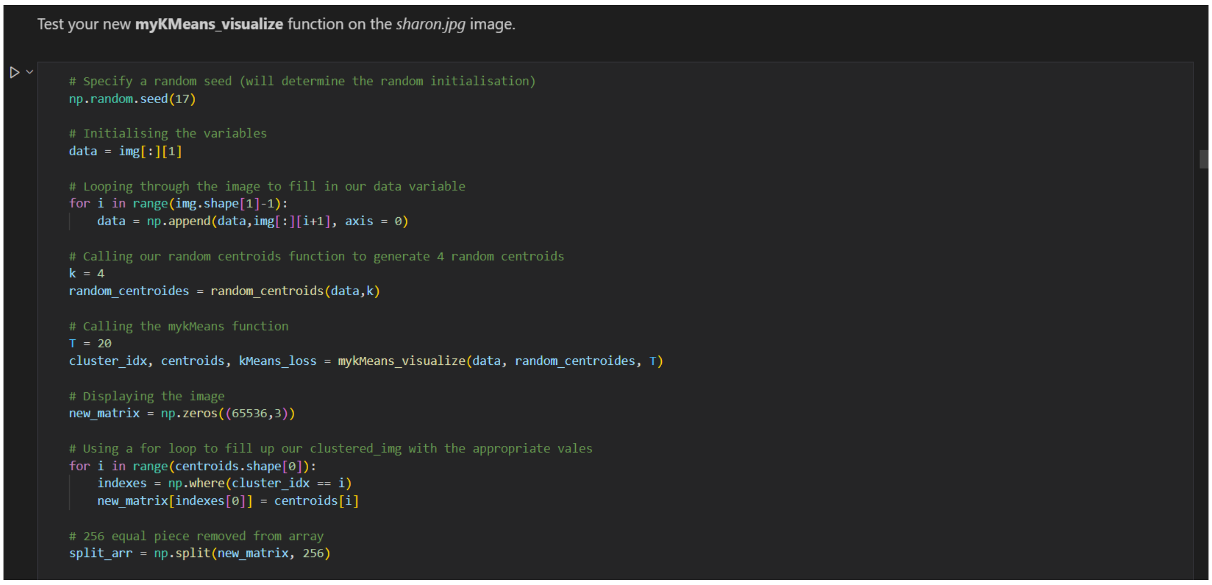


```
def mykMeans_visualize(data, centroids, T):
    # Inputs  - data        : numpy array (N x d)
    #         - centroids   : numpy array (k x d)
    #         - T           : integer
    # N = number of data points, d = dimension of data, k = number of centroids, T = number of iterations

    # Initialize the list to store the k-means loss at each iteration
    kMeans_loss = np.zeros((1,T))
    sum_dloss = np.zeros((data.shape[0],1))
    sum_dimloss = np.zeros((1,data.shape[1]))

    for t in range(T):

        # Printing the total number of iterations
        # print("Iterations:", t)

        # Reinitialising all of the required values
        distances = dist2c(data, centroids)
        cluster_idx = np.argmin(distances.T,axis = 1)

        # Update centroids
        for i in range(centroids.shape[0]):
            indexes = np.where(cluster_idx == i)
            centroids[i] = np.mean(data[indexes[0]], axis=0)

        for n in range(data.shape[0]):
            for k in range(centroids.shape[0]):
                if cluster_idx[n] == k:
                    for d in range(data.shape[1]):
                        sum_dimloss[0][d] = (data[n][d] - centroids[0][d]) ** 2
                    sum_dloss[n][0] = np.sum(sum_dimloss)

        # Calculate the k-means loss at this iteration
        kMeans_loss[0][t] = np.sum(sum_dloss)
```



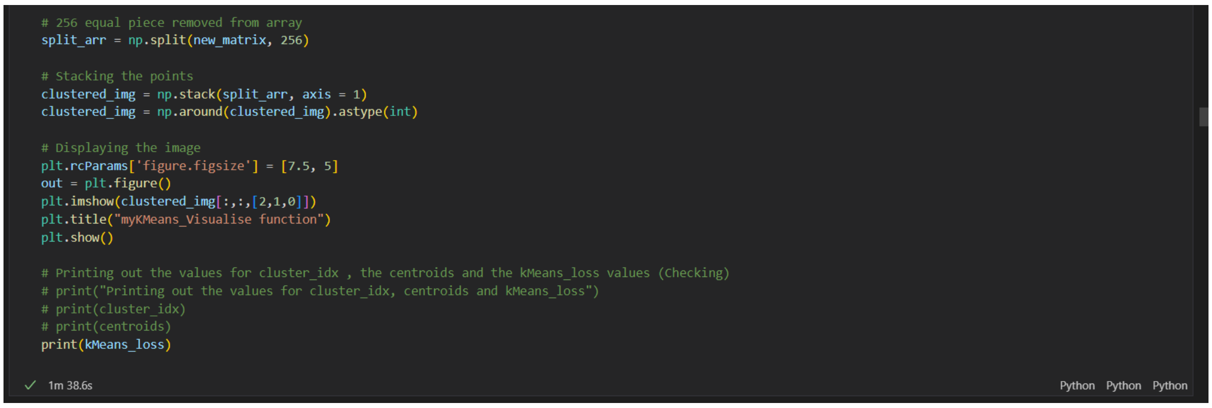
```
        # Calculate the k-means loss at this iteration
        kMeans_loss[0][t] = np.sum(sum_dloss)

        # Randomization of values
        x_idx = np.random.choice(256, size=250, replace=False)
        y_idx = np.random.choice(256, size=250, replace=False)
        sample_idx = np.column_stack((x_idx, y_idx))

        # Calling the function
        f = visualize(data, cluster_idx, centroids, sample_idx)

    # Return back the three values
    return cluster_idx, centroids, kMeans_loss
```

[10] ✓ 0.0s Python



Test your new **myKMeans_visualize** function on the *sharon.jpg* image.

```python
# Specify a random seed (will determine the random initialisation)
np.random.seed(17)

# Initialising the variables
data = img[:][1]

# Looping through the image to fill in our data variable
for i in range(img.shape[1]-1):
    data = np.append(data,img[:][i+1], axis = 0)

# Calling our random centroids function to generate 4 random centroids
k = 4
random_centroides = random_centroids(data,k)

# Calling the mykMeans function
T = 20
cluster_idx, centroids, kMeans_loss = mykMeans_visualize(data, random_centroides, T)

# Displaying the image
new_matrix = np.zeros((65536,3))

# Using a for loop to fill up our clustered_img with the appropriate vales
for i in range(centroids.shape[0]):
    indexes = np.where(cluster_idx == i)
    new_matrix[indexes[0]] = centroids[i]

# 256 equal piece removed from array
split_arr = np.split(new_matrix, 256)
```



```python
# 256 equal piece removed from array
split_arr = np.split(new_matrix, 256)

# Stacking the points
clustered_img = np.stack(split_arr, axis = 1)
clustered_img = np.around(clustered_img).astype(int)

# Displaying the image
plt.rcParams['figure.figsize'] = [7.5, 5]
out = plt.figure()
plt.imshow(clustered_img[:,:,[2,1,0]])
plt.title("myKMeans_Visualise function")
plt.show()

# Printing out the values for cluster_idx , the centroids and the kMeans_loss values (Checking)
# print("Printing out the values for cluster_idx, centroids and kMeans_loss")
# print(cluster_idx)
# print(centroids)
print(kMeans_loss)
```

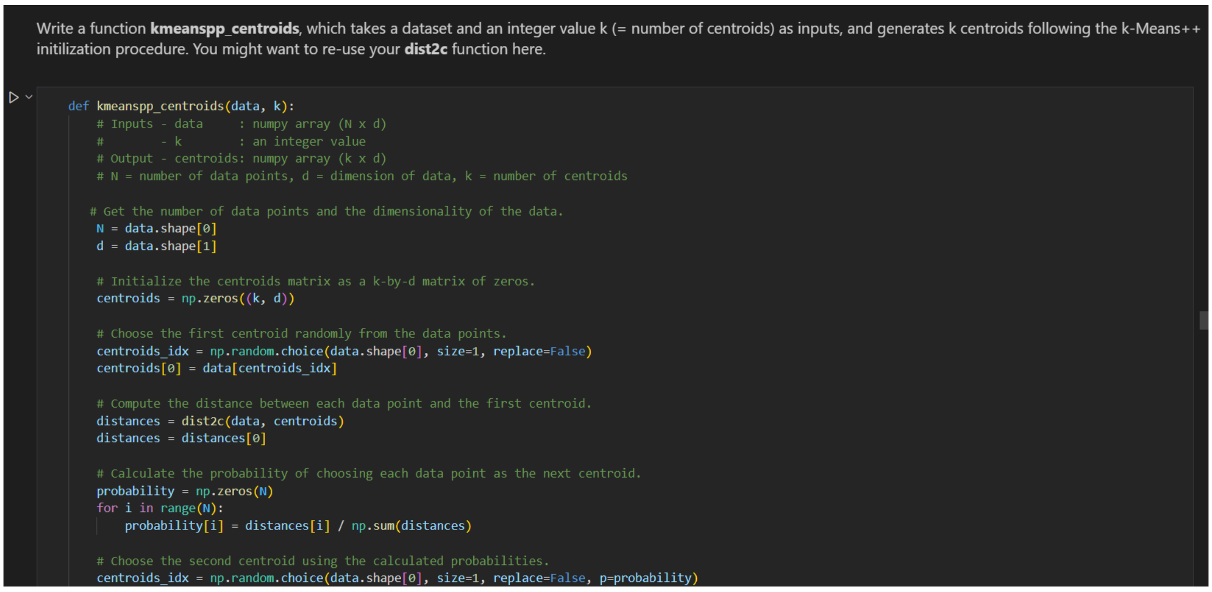
✓ 1m 38.6s

Python Python Python

Code for Task 3:

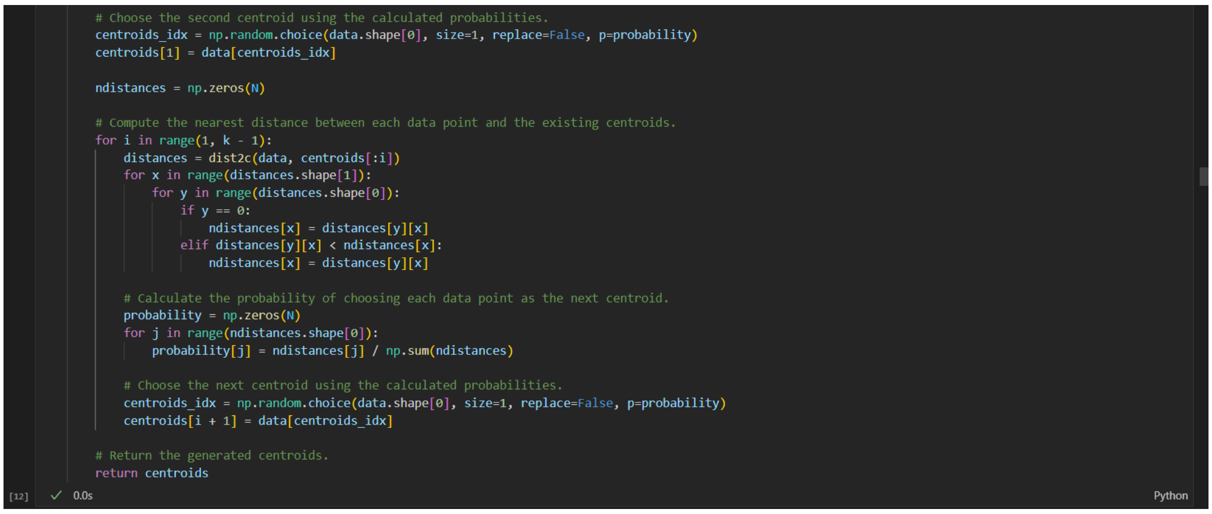
Paste your code with unanswered questions and comments in here.

Kmeanspp centroids function



Write a function **kmeanspp_centroids**, which takes a dataset and an integer value k (= number of centroids) as inputs, and generates k centroids following the k-Means++ initilization procedure. You might want to re-use your **dist2c** function here.

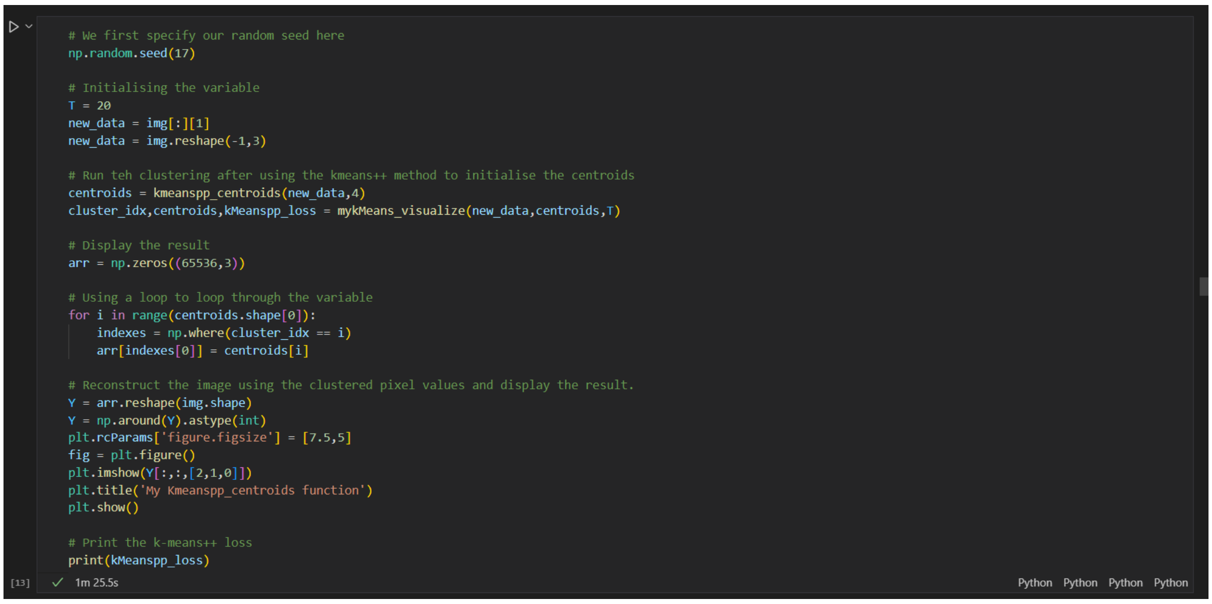
```python
def kmeanspp_centroids(data, k):
    # Inputs - data     : numpy array (N x d)
    #        - k        : an integer value
    # Output - centroids: numpy array (k x d)
    # N = number of data points, d = dimension of data, k = number of centroids

    # Get the number of data points and the dimensionality of the data.
    N = data.shape[0]
    d = data.shape[1]

    # Initialize the centroids matrix as a k-by-d matrix of zeros.
    centroids = np.zeros((k, d))

    # Choose the first centroid randomly from the data points.
    centroids_idx = np.random.choice(data.shape[0], size=1, replace=False)
    centroids[0] = data[centroids_idx]

    # Compute the distance between each data point and the first centroid.
    distances = dist2c(data, centroids)
    distances = distances[0]

    # Calculate the probability of choosing each data point as the next centroid.
    probability = np.zeros(N)
    for i in range(N):
        probability[i] = distances[i] / np.sum(distances)

    # Choose the second centroid using the calculated probabilities.
    centroids_idx = np.random.choice(data.shape[0], size=1, replace=False, p=probability)
```



```python
    # Choose the second centroid using the calculated probabilities.
    centroids_idx = np.random.choice(data.shape[0], size=1, replace=False, p=probability)
```

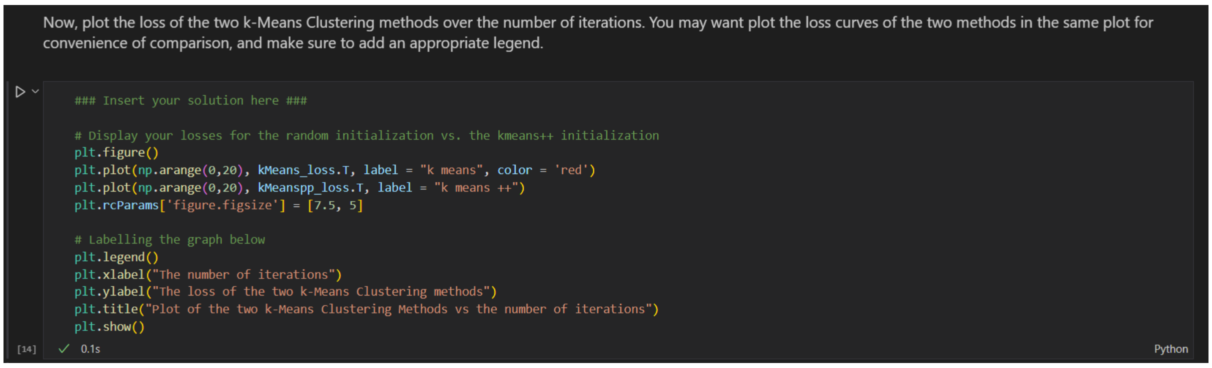
```python
    centroids[1] = data[centroids_idx]

    ndistances = np.zeros(N)

    # Compute the nearest distance between each data point and the existing centroids.
    for i in range(1, k - 1):
        distances = dist2c(data, centroids[:i])
        for x in range(distances.shape[1]):
            for y in range(distances.shape[0]):
                if y == 0:
                    ndistances[x] = distances[y][x]
                elif distances[y][x] < ndistances[x]:
                    ndistances[x] = distances[y][x]

        # Calculate the probability of choosing each data point as the next centroid.
        probability = np.zeros(N)
        for j in range(ndistances.shape[0]):
            probability[j] = ndistances[j] / np.sum(ndistances)

        # Choose the next centroid using the calculated probabilities.
        centroids_idx = np.random.choice(data.shape[0], size=1, replace=False, p=probability)
        centroids[i + 1] = data[centroids_idx]

    # Return the generated centroids.
    return centroids
```

[12] ✓ 0.0s Python



```python
# We first specify our random seed here
np.random.seed(17)

# Initialising the variable
T = 20
new_data = img[:][1]
new_data = img.reshape(-1,3)

# Run teh clustering after using the kmeans++ method to initialise the centroids
centroids = kmeanspp_centroids(new_data,4)
cluster_idx,centroids,kMeanspp_loss = mykMeans_visualize(new_data,centroids,T)

# Display the result
arr = np.zeros((65536,3))

# Using a loop to loop through the variable
for i in range(centroids.shape[0]):
    indexes = np.where(cluster_idx == i)
    arr[indexes[0]] = centroids[i]

# Reconstruct the image using the clustered pixel values and display the result.
Y = arr.reshape(img.shape)
Y = np.around(Y).astype(int)
plt.rcParams['figure.figsize'] = [7.5,5]
fig = plt.figure()
plt.imshow(Y[:,:,[2,1,0]])
plt.title('My Kmeanspp_centroids function')
plt.show()

# Print the k-means++ loss
print(kMeanspp_loss)
```

[13] ✓ 1m 25.5s Python Python Python Python



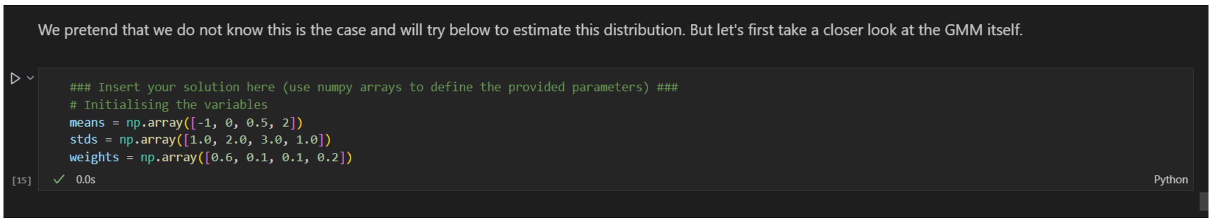
Now, plot the loss of the two k-Means Clustering methods over the number of iterations. You may want plot the loss curves of the two methods in the same plot for convenience of comparison, and make sure to add an appropriate legend.

```python
### Insert your solution here ###

# Display your losses for the random initialization vs. the kmeans++ initialization
plt.figure()
plt.plot(np.arange(0,20), kMeans_loss.T, label = "k means", color = 'red')
plt.plot(np.arange(0,20), kMeanspp_loss.T, label = "k means ++")
plt.rcParams['figure.figsize'] = [7.5, 5]

# Labelling the graph below
plt.legend()
plt.xlabel("The number of iterations")
plt.ylabel("The loss of the two k-Means Clustering methods")
plt.title("Plot of the two k-Means Clustering Methods vs the number of iterations")
plt.show()
```

[14] ✓ 0.1s Python

Code for Task 4:

Paste your code with unanswered questions and comments in here.

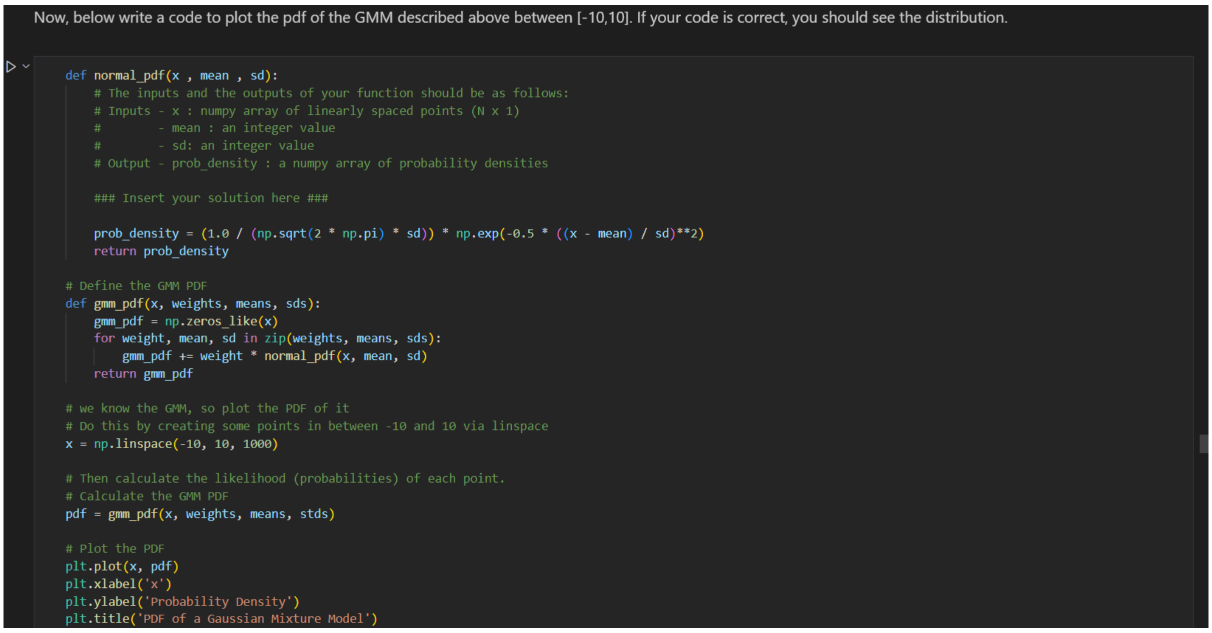


We pretend that we do not know this is the case and will try below to estimate this distribution. But let's first take a closer look at the GMM itself.

```
### Insert your solution here (use numpy arrays to define the provided parameters) ###
# Initialising the variables
means = np.array([-1, 0, 0.5, 2])
stds = np.array([1.0, 2.0, 3.0, 1.0])
weights = np.array([0.6, 0.1, 0.1, 0.2])
```

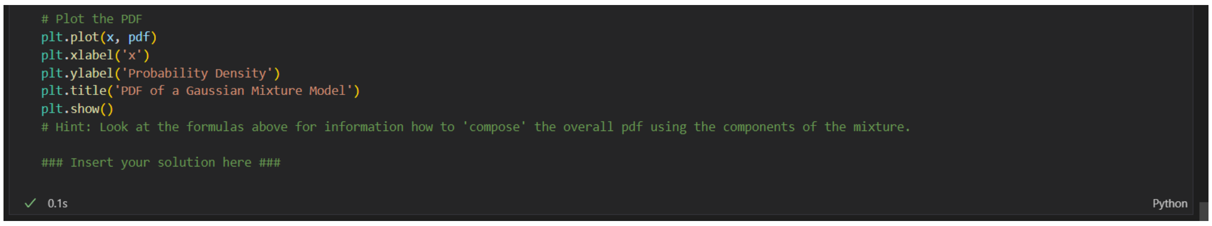
[15] ✓ 0.0s Python

normal_pdf function



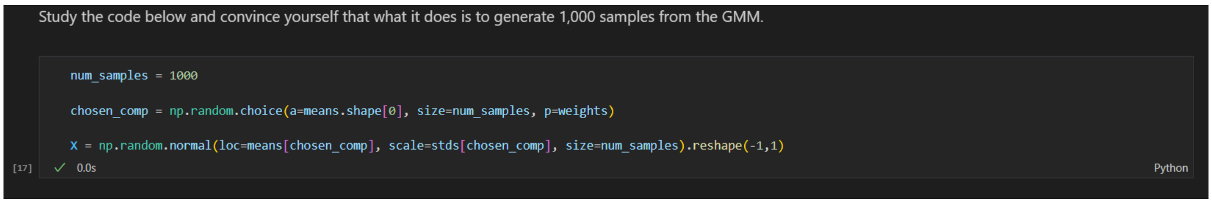
Now, below write a code to plot the pdf of the GMM described above between [-10,10]. If your code is correct, you should see the distribution.

```
def normal_pdf(x , mean , sd):
    # The inputs and the outputs of your function should be as follows:
    # Inputs - x : numpy array of linearly spaced points (N x 1)
    #        - mean : an integer value
    #        - sd: an integer value
    # Output - prob_density : a numpy array of probability densities

    ### Insert your solution here ###

    prob_density = (1.0 / (np.sqrt(2 * np.pi) * sd)) * np.exp(-0.5 * ((x - mean) / sd)**2)
    return prob_density

# Define the GMM PDF
def gmm_pdf(x, weights, means, sds):
    gmm_pdf = np.zeros_like(x)
    for weight, mean, sd in zip(weights, means, sds):
        gmm_pdf += weight * normal_pdf(x, mean, sd)
    return gmm_pdf

# we know the GMM, so plot the PDF of it
# Do this by creating some points in between -10 and 10 via linspace
x = np.linspace(-10, 10, 1000)

# Then calculate the likelihood (probabilities) of each point.
# Calculate the GMM PDF
pdf = gmm_pdf(x, weights, means, stds)

# Plot the PDF
plt.plot(x, pdf)
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.title('PDF of a Gaussian Mixture Model')
```



```
# Plot the PDF
plt.plot(x, pdf)
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.title('PDF of a Gaussian Mixture Model')
plt.show()
# Hint: Look at the formulas above for information how to 'compose' the overall pdf using the components of the mixture.

### Insert your solution here ###
```

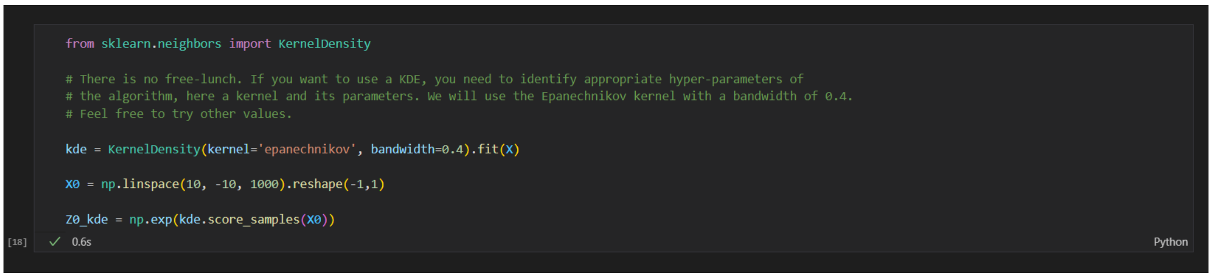
✓ 0.1s Python



Study the code below and convince yourself that what it does is to generate 1,000 samples from the GMM.

```
num_samples = 1000

chosen_comp = np.random.choice(a=means.shape[0], size=num_samples, p=weights)

X = np.random.normal(loc=means[chosen_comp], scale=stds[chosen_comp], size=num_samples).reshape(-1,1)
```

[17] ✓ 0.0s Python



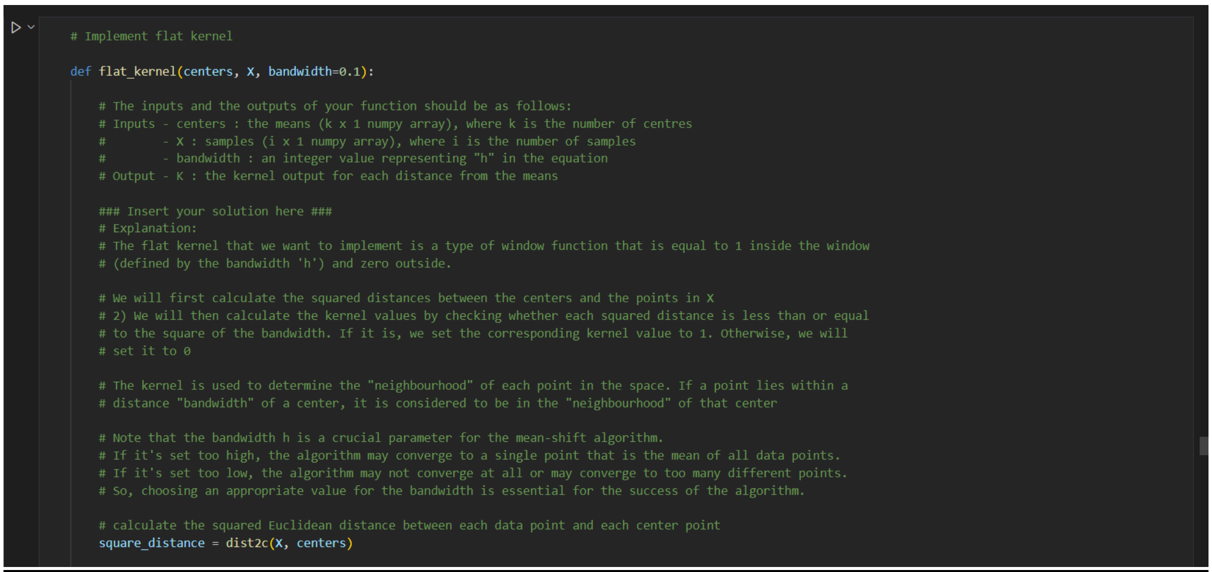
```
from sklearn.neighbors import KernelDensity

# There is no free-lunch. If you want to use a KDE, you need to identify appropriate hyper-parameters of
# the algorithm, here a kernel and its parameters. We will use the Epanechnikov kernel with a bandwidth of 0.4.
# Feel free to try other values.

kde = KernelDensity(kernel='epanechnikov', bandwidth=0.4).fit(X)

X0 = np.linspace(10, -10, 1000).reshape(-1,1)

Z0_kde = np.exp(kde.score_samples(X0))
```

[18] ✓ 0.6s Python

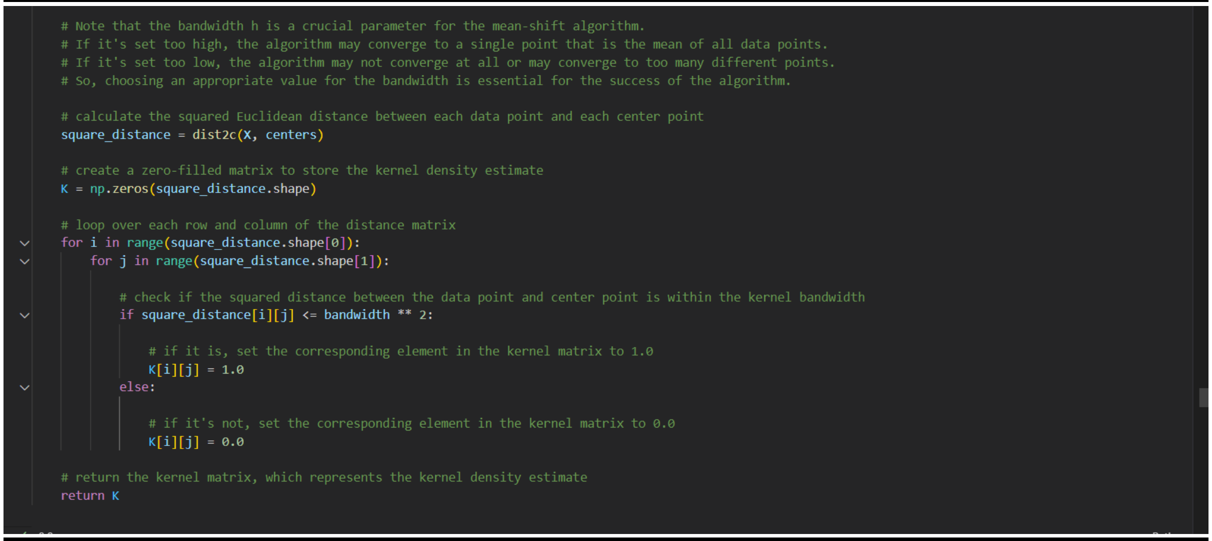
Code for Task 5:

Paste your code with unanswered questions and comments in here.

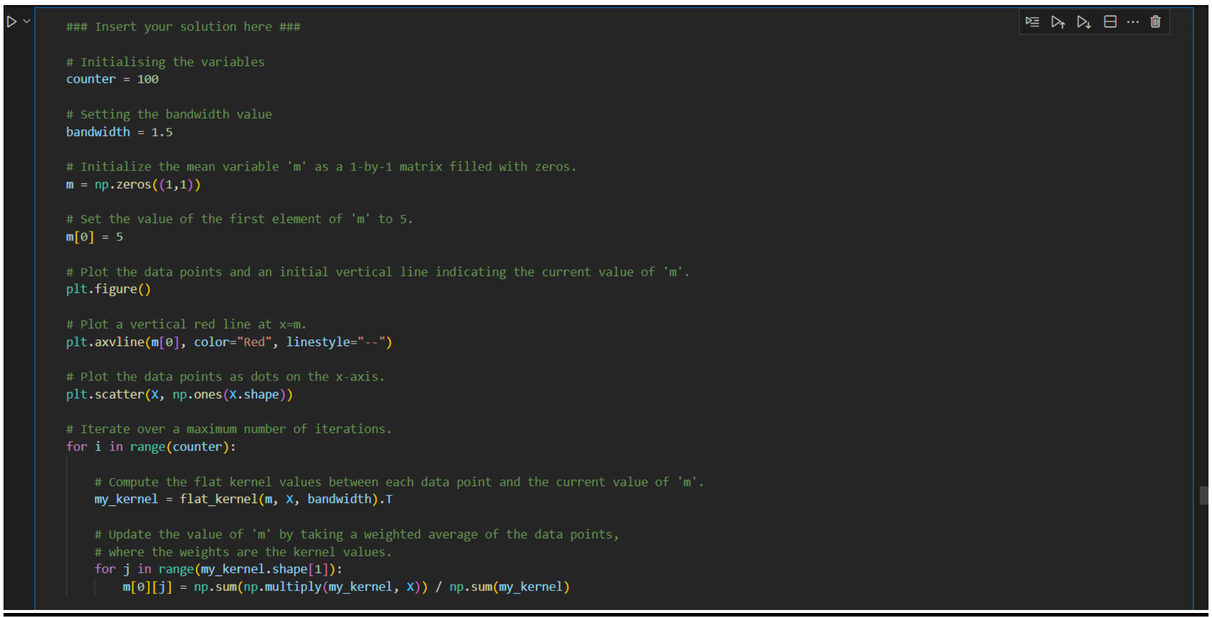
flat_kernel_function



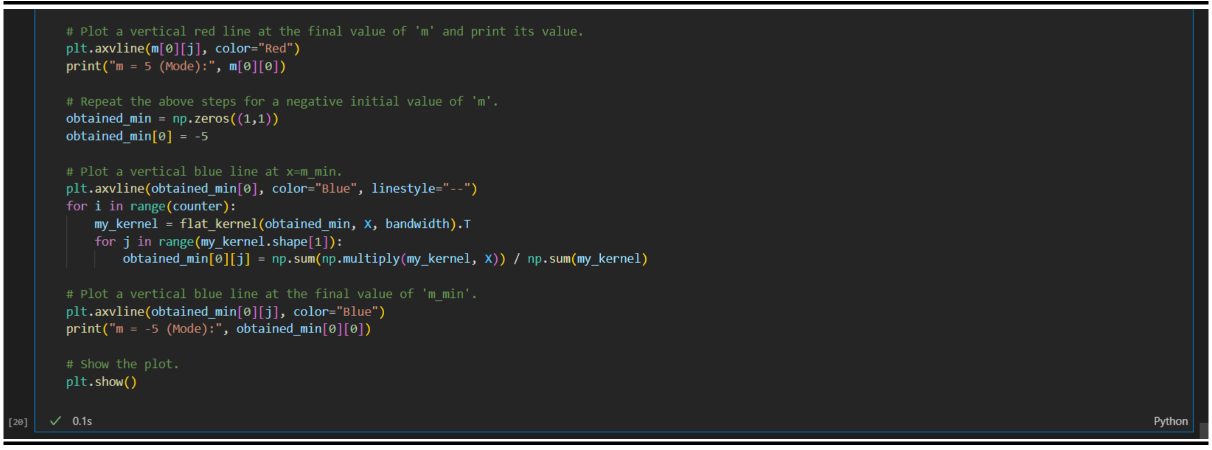
```python
# Implement flat kernel

def flat_kernel(centers, X, bandwidth=0.1):

    # The inputs and the outputs of your function should be as follows:
    # Inputs - centers : the means (k x 1 numpy array), where k is the number of centres
    #         - X : samples (i x 1 numpy array), where i is the number of samples
    #         - bandwidth : an integer value representing "h" in the equation
    # Output - K : the kernel output for each distance from the means

    ### Insert your solution here ###
    # Explanation:
    # The flat kernel that we want to implement is a type of window function that is equal to 1 inside the window
    # (defined by the bandwidth 'h') and zero outside.

    # We will first calculate the squared distances between the centers and the points in X
    # 2) We will then calculate the kernel values by checking whether each squared distance is less than or equal
    # to the square of the bandwidth. If it is, we set the corresponding kernel value to 1. Otherwise, we will
    # set it to 0

    # The kernel is used to determine the "neighbourhood" of each point in the space. If a point lies within a
    # distance "bandwidth" of a center, it is considered to be in the "neighbourhood" of that center

    # Note that the bandwidth h is a crucial parameter for the mean-shift algorithm.
    # If it's set too high, the algorithm may converge to a single point that is the mean of all data points.
    # If it's set too low, the algorithm may not converge at all or may converge to too many different points.
    # So, choosing an appropriate value for the bandwidth is essential for the success of the algorithm.

    # calculate the squared Euclidean distance between each data point and each center point
    square_distance = dist2c(X, centers)
```



```python
    # Note that the bandwidth h is a crucial parameter for the mean-shift algorithm.
    # If it's set too high, the algorithm may converge to a single point that is the mean of all data points.
    # If it's set too low, the algorithm may not converge at all or may converge to too many different points.
    # So, choosing an appropriate value for the bandwidth is essential for the success of the algorithm.

    # calculate the squared Euclidean distance between each data point and each center point
    square_distance = dist2c(X, centers)

    # create a zero-filled matrix to store the kernel density estimate
    K = np.zeros(square_distance.shape)

    # loop over each row and column of the distance matrix
    for i in range(square_distance.shape[0]):
        for j in range(square_distance.shape[1]):

            # check if the squared distance between the data point and center point is within the kernel bandwidth
            if square_distance[i][j] <= bandwidth ** 2:

                # if it is, set the corresponding element in the kernel matrix to 1.0
                K[i][j] = 1.0
            else:

                # if it's not, set the corresponding element in the kernel matrix to 0.0
                K[i][j] = 0.0

    # return the kernel matrix, which represents the kernel density estimate
    return K
```

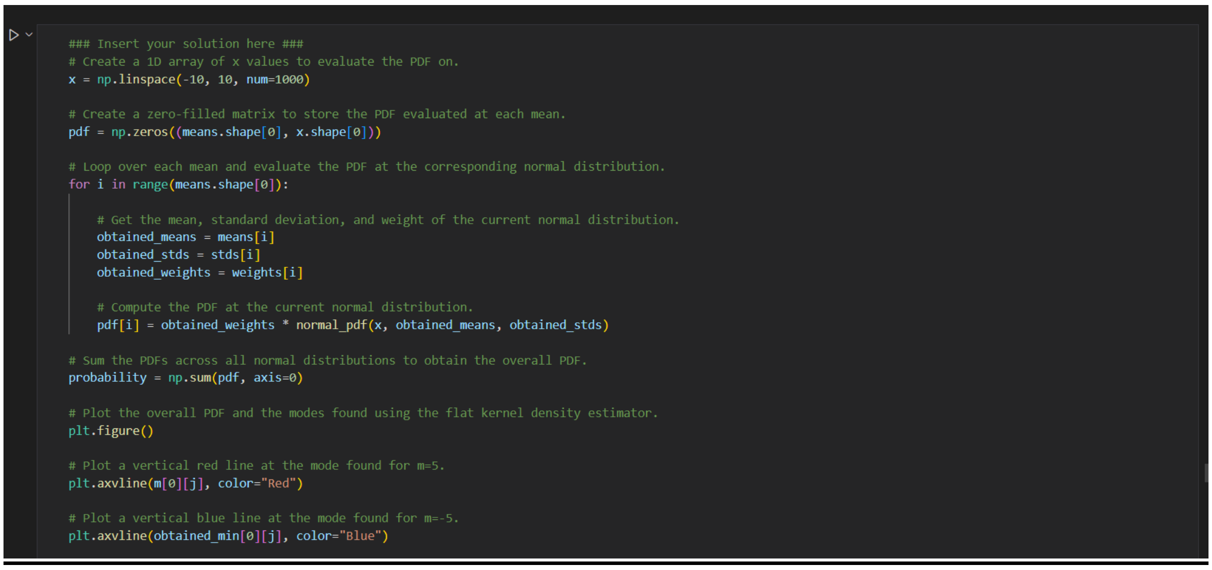


```python
### Insert your solution here ###

# Initialising the variables
counter = 100

# Setting the bandwidth value
bandwidth = 1.5

# Initialize the mean variable 'm' as a 1-by-1 matrix filled with zeros.
m = np.zeros((1,1))

# Set the value of the first element of 'm' to 5.
m[0] = 5

# Plot the data points and an initial vertical line indicating the current value of 'm'.
plt.figure()

# Plot a vertical red line at x=m.
plt.axvline(m[0], color="Red", linestyle="--")

# Plot the data points as dots on the x-axis.
plt.scatter(X, np.ones(X.shape))

# Iterate over a maximum number of iterations.
for i in range(counter):

    # Compute the flat kernel values between each data point and the current value of 'm'.
    my_kernel = flat_kernel(m, X, bandwidth).T

    # Update the value of 'm' by taking a weighted average of the data points,
    # where the weights are the kernel values.
    for j in range(my_kernel.shape[1]):
        m[0][j] = np.sum(np.multiply(my_kernel, X)) / np.sum(my_kernel)
```

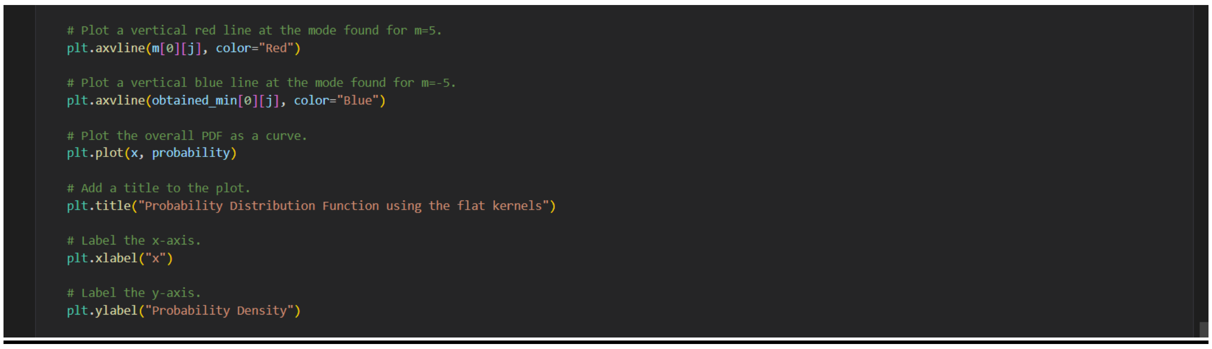


```python
# Plot a vertical red line at the final value of 'm' and print its value.
plt.axvline(m[0][j], color="Red")
print("m = 5 (Mode):", m[0][0])

# Repeat the above steps for a negative initial value of 'm'.
obtained_min = np.zeros((1,1))
obtained_min[0] = -5

# Plot a vertical blue line at x=m_min.
plt.axvline(obtained_min[0], color="Blue", linestyle="--")
for i in range(counter):
    my_kernel = flat_kernel(obtained_min, X, bandwidth).T
    for j in range(my_kernel.shape[1]):
        obtained_min[0][j] = np.sum(np.multiply(my_kernel, X)) / np.sum(my_kernel)

# Plot a vertical blue line at the final value of 'm_min'.
plt.axvline(obtained_min[0][j], color="Blue")
print("m = -5 (Mode):", obtained_min[0][0])

# Show the plot.
plt.show()
```

[20] ✓ 0.1s Python

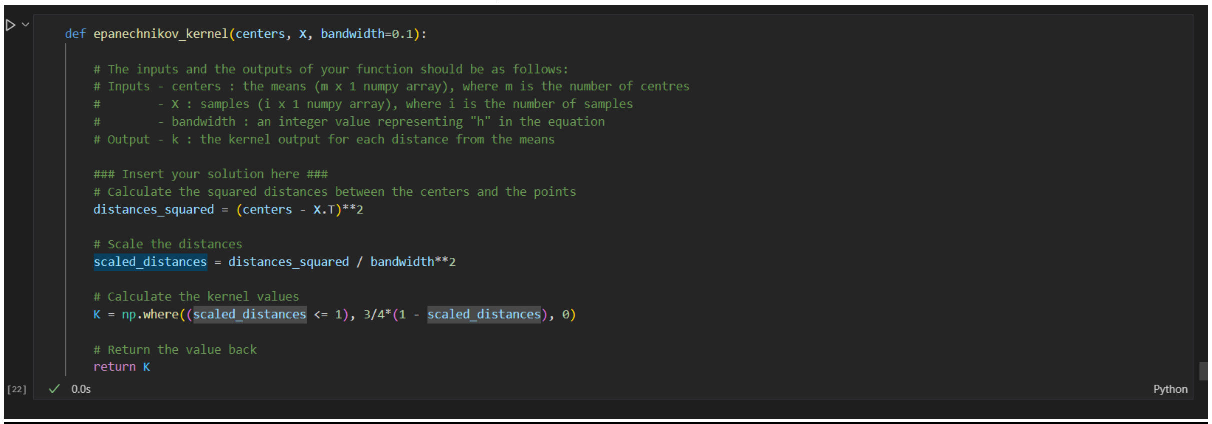


```python
### Insert your solution here ###
# Create a 1D array of x values to evaluate the PDF on.
x = np.linspace(-10, 10, num=1000)

# Create a zero-filled matrix to store the PDF evaluated at each mean.
pdf = np.zeros((means.shape[0], x.shape[0]))

# Loop over each mean and evaluate the PDF at the corresponding normal distribution.
for i in range(means.shape[0]):

    # Get the mean, standard deviation, and weight of the current normal distribution.
    obtained_means = means[i]
    obtained_stds = stds[i]
    obtained_weights = weights[i]

    # Compute the PDF at the current normal distribution.
    pdf[i] = obtained_weights * normal_pdf(x, obtained_means, obtained_stds)

# Sum the PDFs across all normal distributions to obtain the overall PDF.
probability = np.sum(pdf, axis=0)

# Plot the overall PDF and the modes found using the flat kernel density estimator.
plt.figure()

# Plot a vertical red line at the mode found for m=5.
plt.axvline(m[0][j], color="Red")

# Plot a vertical blue line at the mode found for m=-5.
plt.axvline(obtained_min[0][j], color="Blue")
```



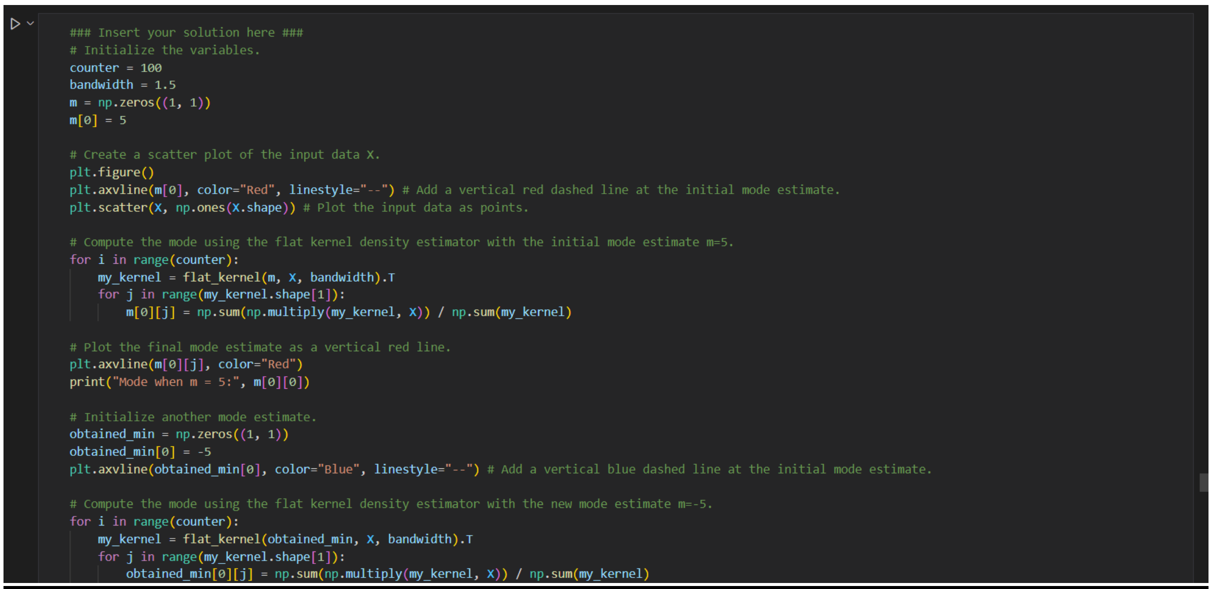
```python
# Plot a vertical red line at the mode found for m=5.
plt.axvline(m[0][j], color="Red")

# Plot a vertical blue line at the mode found for m=-5.
plt.axvline(obtained_min[0][j], color="Blue")

# Plot the overall PDF as a curve.
plt.plot(x, probability)

# Add a title to the plot.
plt.title("Probability Distribution Function using the flat kernels")

# Label the x-axis.
plt.xlabel("x")

# Label the y-axis.
plt.ylabel("Probability Density")
```

epanechnikov_kernel function

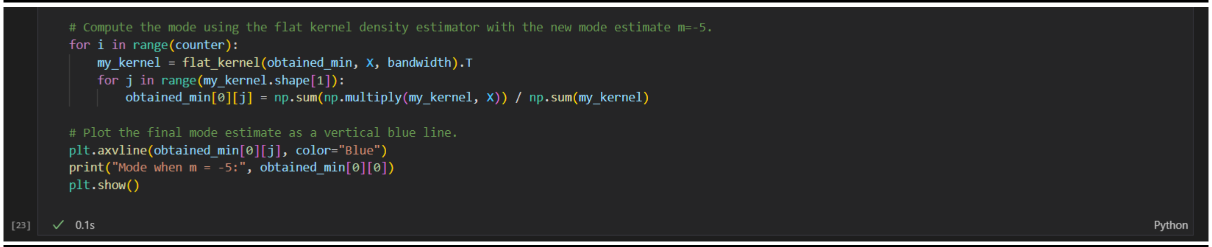


```python
def epanechnikov_kernel(centers, X, bandwidth=0.1):

    # The inputs and the outputs of your function should be as follows:
    # Inputs - centers : the means (m x 1 numpy array), where m is the number of centres
    #        - X : samples (i x 1 numpy array), where i is the number of samples
    #        - bandwidth : an integer value representing "h" in the equation
    # Output - k : the kernel output for each distance from the means

    ### Insert your solution here ###
    # Calculate the squared distances between the centers and the points
    distances_squared = (centers - X.T)**2

    # Scale the distances
    scaled_distances = distances_squared / bandwidth**2

    # Calculate the kernel values
    K = np.where((scaled_distances <= 1), 3/4*(1 - scaled_distances), 0)

    # Return the value back
    return K
```

```
[22]  ✓  0.0s                                                    Python
```

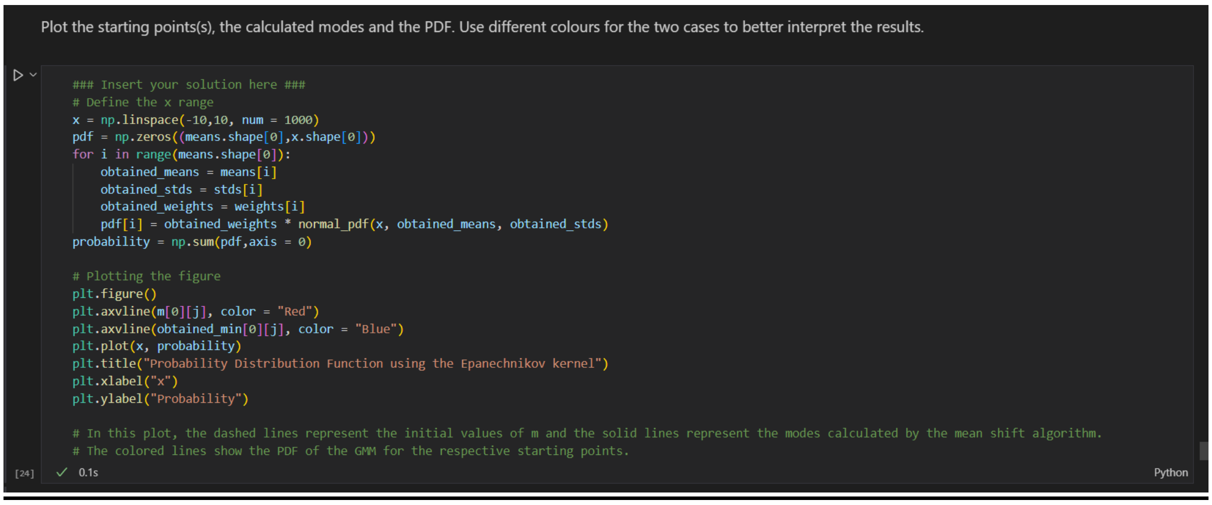


```python
### Insert your solution here ###
# Initialize the variables.
counter = 100
bandwidth = 1.5
m = np.zeros((1, 1))
m[0] = 5

# Create a scatter plot of the input data X.
plt.figure()
plt.axvline(m[0], color="Red", linestyle="--") # Add a vertical red dashed line at the initial mode estimate.
plt.scatter(X, np.ones(X.shape)) # Plot the input data as points.

# Compute the mode using the flat kernel density estimator with the initial mode estimate m=5.
for i in range(counter):
    my_kernel = flat_kernel(m, X, bandwidth).T
    for j in range(my_kernel.shape[1]):
        m[0][j] = np.sum(np.multiply(my_kernel, X)) / np.sum(my_kernel)

# Plot the final mode estimate as a vertical red line.
plt.axvline(m[0][j], color="Red")
print("Mode when m = 5:", m[0][0])

# Initialize another mode estimate.
obtained_min = np.zeros((1, 1))
obtained_min[0] = -5
plt.axvline(obtained_min[0], color="Blue", linestyle="--") # Add a vertical blue dashed line at the initial mode estimate.

# Compute the mode using the flat kernel density estimator with the new mode estimate m=-5.
for i in range(counter):
    my_kernel = flat_kernel(obtained_min, X, bandwidth).T
    for j in range(my_kernel.shape[1]):
        obtained_min[0][j] = np.sum(np.multiply(my_kernel, X)) / np.sum(my_kernel)
```



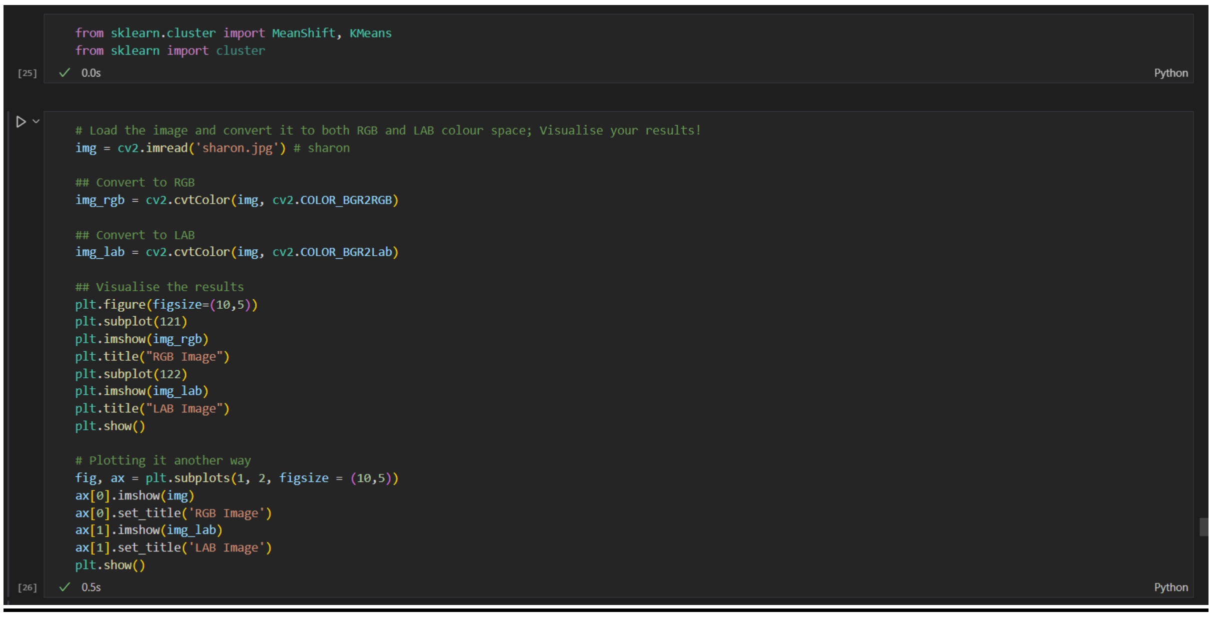
```python
# Compute the mode using the flat kernel density estimator with the new mode estimate m=-5.
for i in range(counter):
    my_kernel = flat_kernel(obtained_min, X, bandwidth).T
    for j in range(my_kernel.shape[1]):
        obtained_min[0][j] = np.sum(np.multiply(my_kernel, X)) / np.sum(my_kernel)

# Plot the final mode estimate as a vertical blue line.
plt.axvline(obtained_min[0][j], color="Blue")
print("Mode when m = -5:", obtained_min[0][0])
plt.show()
```

```
[23]  ✓  0.1s                                                    Python
```


Plot the starting points(s), the calculated modes and the PDF. Use different colours for the two cases to better interpret the results.

```python
### Insert your solution here ###
# Define the x range
x = np.linspace(-10,10, num = 1000)
pdf = np.zeros((means.shape[0],x.shape[0]))
for i in range(means.shape[0]):
    obtained_means = means[i]
    obtained_stds = stds[i]
    obtained_weights = weights[i]
    pdf[i] = obtained_weights * normal_pdf(x, obtained_means, obtained_stds)
probability = np.sum(pdf,axis = 0)

# Plotting the figure
plt.figure()
plt.axvline(m[0][j], color = "Red")
plt.axvline(obtained_min[0][j], color = "Blue")
plt.plot(x, probability)
plt.title("Probability Distribution Function using the Epanechnikov kernel")
plt.xlabel("x")
plt.ylabel("Probability")

# In this plot, the dashed lines represent the initial values of m and the solid lines represent the modes calculated by the mean shift algorithm.
# The colored lines show the PDF of the GMM for the respective starting points.
```

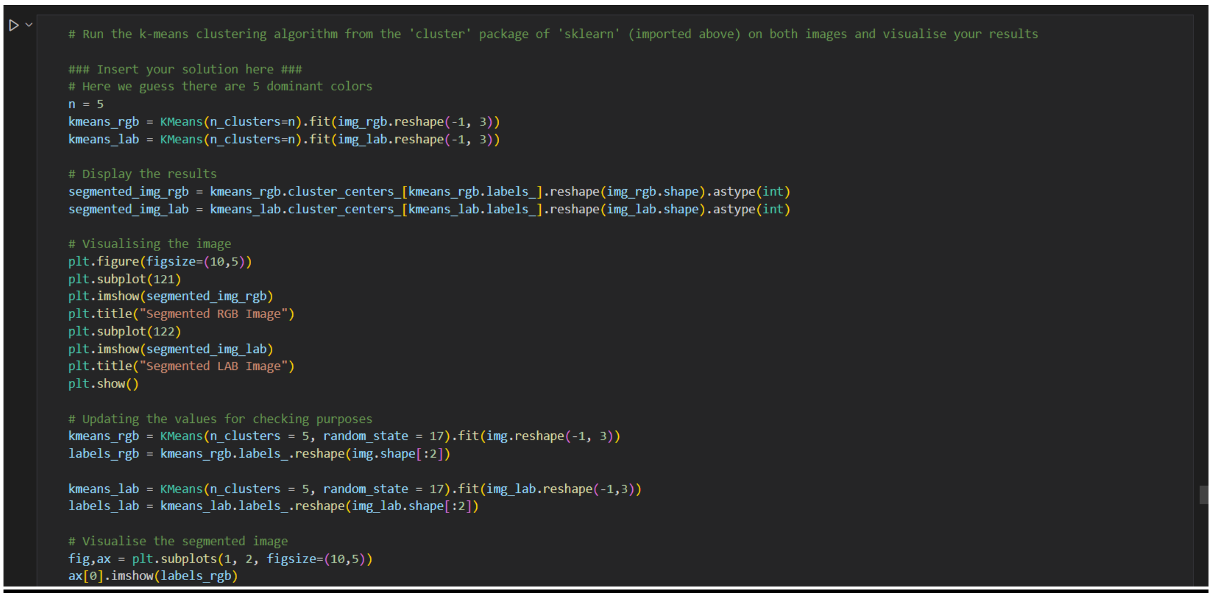
[24] ✓ 0.1s Python

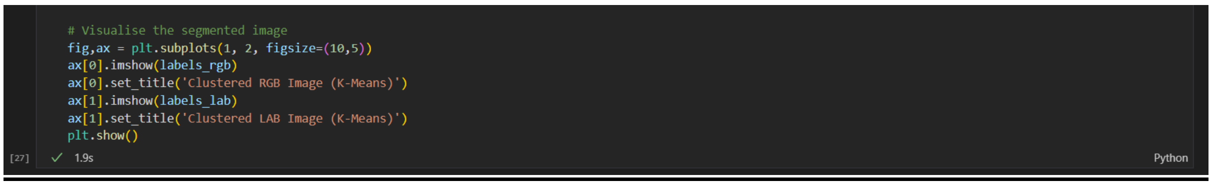

```python
from sklearn.cluster import MeanShift, KMeans
from sklearn import cluster
```

[25] ✓ 0.0s Python

```python
# Load the image and convert it to both RGB and LAB colour space; Visualise your results!
img = cv2.imread('sharon.jpg') # sharon

## Convert to RGB
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

## Convert to LAB
img_lab = cv2.cvtColor(img, cv2.COLOR_BGR2Lab)

## Visualise the results
plt.figure(figsize=(10,5))
plt.subplot(121)
plt.imshow(img_rgb)
plt.title("RGB Image")
plt.subplot(122)
plt.imshow(img_lab)
plt.title("LAB Image")
plt.show()

# Plotting it another way
fig, ax = plt.subplots(1, 2, figsize = (10,5))
ax[0].imshow(img)
ax[0].set_title('RGB Image')
ax[1].imshow(img_lab)
ax[1].set_title('LAB Image')
plt.show()
```

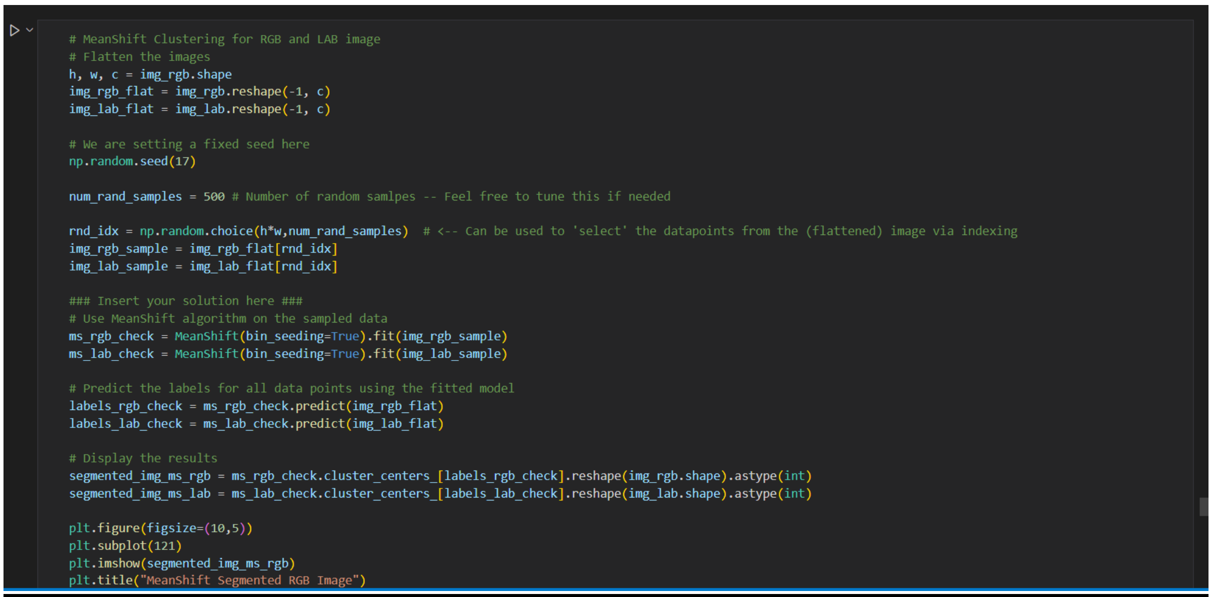
[26] ✓ 0.5s Python


```python
# Run the k-means clustering algorithm from the 'cluster' package of 'sklearn' (imported above) on both images and visualise your results

### Insert your solution here ###
# Here we guess there are 5 dominant colors
n = 5
kmeans_rgb = KMeans(n_clusters=n).fit(img_rgb.reshape(-1, 3))
kmeans_lab = KMeans(n_clusters=n).fit(img_lab.reshape(-1, 3))

# Display the results
segmented_img_rgb = kmeans_rgb.cluster_centers_[kmeans_rgb.labels_].reshape(img_rgb.shape).astype(int)
segmented_img_lab = kmeans_lab.cluster_centers_[kmeans_lab.labels_].reshape(img_lab.shape).astype(int)

# Visualising the image
plt.figure(figsize=(10,5))
plt.subplot(121)
plt.imshow(segmented_img_rgb)
plt.title("Segmented RGB Image")
plt.subplot(122)
plt.imshow(segmented_img_lab)
plt.title("Segmented LAB Image")
plt.show()

# Updating the values for checking purposes
kmeans_rgb = KMeans(n_clusters = 5, random_state = 17).fit(img.reshape(-1, 3))
labels_rgb = kmeans_rgb.labels_.reshape(img.shape[:2])

kmeans_lab = KMeans(n_clusters = 5, random_state = 17).fit(img_lab.reshape(-1,3))
labels_lab = kmeans_lab.labels_.reshape(img_lab.shape[:2])

# Visualise the segmented image
fig,ax = plt.subplots(1, 2, figsize=(10,5))
ax[0].imshow(labels_rgb)
```

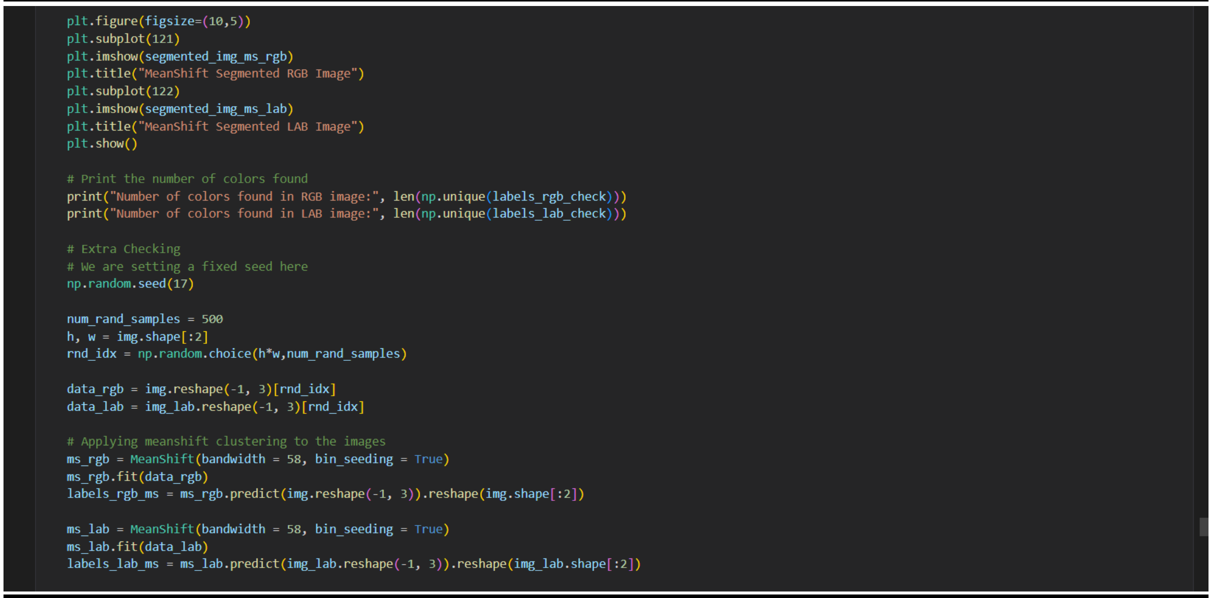


```python
# Visualise the segmented image
fig,ax = plt.subplots(1, 2, figsize=(10,5))
ax[0].imshow(labels_rgb)
ax[0].set_title('Clustered RGB Image (K-Means)')
ax[1].imshow(labels_lab)
ax[1].set_title('Clustered LAB Image (K-Means)')
plt.show()
```

```
[27]  ✓  1.9s                                                    Python
```

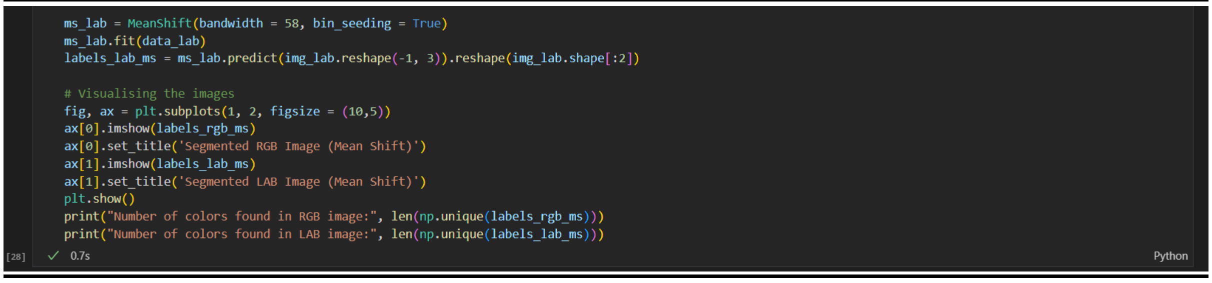


```python
# MeanShift Clustering for RGB and LAB image
# Flatten the images
h, w, c = img_rgb.shape
img_rgb_flat = img_rgb.reshape(-1, c)
img_lab_flat = img_lab.reshape(-1, c)

# We are setting a fixed seed here
np.random.seed(17)

num_rand_samples = 500 # Number of random samlpes -- Feel free to tune this if needed

rnd_idx = np.random.choice(h*w,num_rand_samples)  # <-- Can be used to 'select' the datapoints from the (flattened) image via indexing
img_rgb_sample = img_rgb_flat[rnd_idx]
img_lab_sample = img_lab_flat[rnd_idx]

### Insert your solution here ###
# Use MeanShift algorithm on the sampled data
ms_rgb_check = MeanShift(bin_seeding=True).fit(img_rgb_sample)
ms_lab_check = MeanShift(bin_seeding=True).fit(img_lab_sample)

# Predict the labels for all data points using the fitted model
labels_rgb_check = ms_rgb_check.predict(img_rgb_flat)
labels_lab_check = ms_lab_check.predict(img_lab_flat)

# Display the results
segmented_img_ms_rgb = ms_rgb_check.cluster_centers_[labels_rgb_check].reshape(img_rgb.shape).astype(int)
segmented_img_ms_lab = ms_lab_check.cluster_centers_[labels_lab_check].reshape(img_lab.shape).astype(int)

plt.figure(figsize=(10,5))
plt.subplot(121)
plt.imshow(segmented_img_ms_rgb)
plt.title("MeanShift Segmented RGB Image")
```



```python
plt.figure(figsize=(10,5))
plt.subplot(121)
plt.imshow(segmented_img_ms_rgb)
plt.title("MeanShift Segmented RGB Image")
```

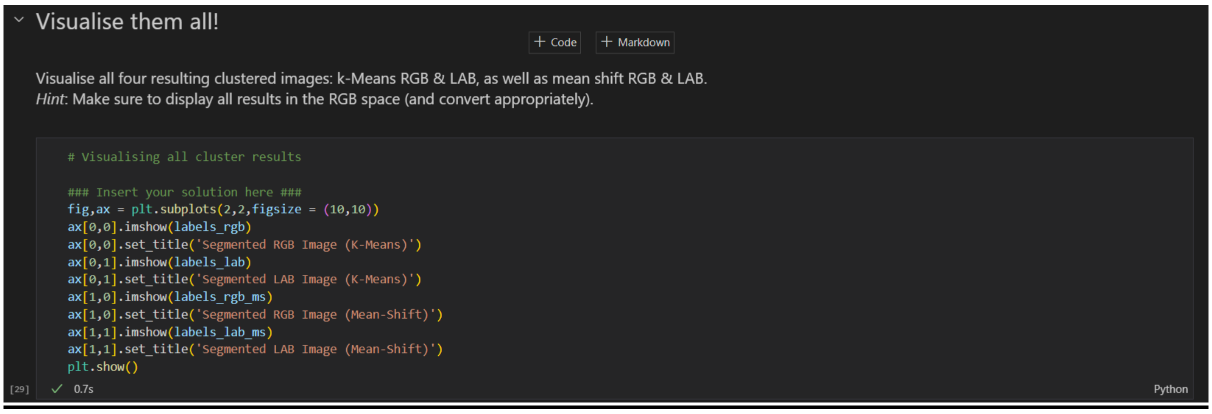
```python
plt.subplot(122)
plt.imshow(segmented_img_ms_lab)
plt.title("MeanShift Segmented LAB Image")
plt.show()

# Print the number of colors found
print("Number of colors found in RGB image:", len(np.unique(labels_rgb_check)))
print("Number of colors found in LAB image:", len(np.unique(labels_lab_check)))

# Extra Checking
# We are setting a fixed seed here
np.random.seed(17)

num_rand_samples = 500
h, w = img.shape[:2]
rnd_idx = np.random.choice(h*w,num_rand_samples)

data_rgb = img.reshape(-1, 3)[rnd_idx]
data_lab = img_lab.reshape(-1, 3)[rnd_idx]

# Applying meanshift clustering to the images
ms_rgb = MeanShift(bandwidth = 58, bin_seeding = True)
ms_rgb.fit(data_rgb)
labels_rgb_ms = ms_rgb.predict(img.reshape(-1, 3)).reshape(img.shape[:2])

ms_lab = MeanShift(bandwidth = 58, bin_seeding = True)
ms_lab.fit(data_lab)
labels_lab_ms = ms_lab.predict(img_lab.reshape(-1, 3)).reshape(img_lab.shape[:2])
```



```python
ms_lab = MeanShift(bandwidth = 58, bin_seeding = True)
ms_lab.fit(data_lab)
labels_lab_ms = ms_lab.predict(img_lab.reshape(-1, 3)).reshape(img_lab.shape[:2])
```

```python
# Visualising the images
fig, ax = plt.subplots(1, 2, figsize = (10,5))
ax[0].imshow(labels_rgb_ms)
ax[0].set_title('Segmented RGB Image (Mean Shift)')
ax[1].imshow(labels_lab_ms)
ax[1].set_title('Segmented LAB Image (Mean Shift)')
plt.show()
print("Number of colors found in RGB image:", len(np.unique(labels_rgb_ms)))
print("Number of colors found in LAB image:", len(np.unique(labels_lab_ms)))
```

```
[28]  ✓  0.7s                                                    Python
```



Visualise them all!

+ Code + Markdown

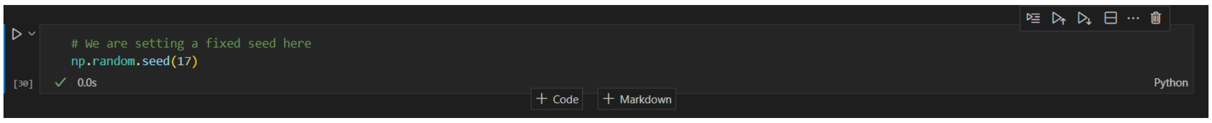
Visualise all four resulting clustered images: k-Means RGB & LAB, as well as mean shift RGB & LAB.
Hint: Make sure to display all results in the RGB space (and convert appropriately).

```python
# Visualising all cluster results

### Insert your solution here ###
fig,ax = plt.subplots(2,2,figsize = (10,10))
ax[0,0].imshow(labels_rgb)
ax[0,0].set_title('Segmented RGB Image (K-Means)')
ax[0,1].imshow(labels_lab)
ax[0,1].set_title('Segmented LAB Image (K-Means)')
ax[1,0].imshow(labels_rgb_ms)
ax[1,0].set_title('Segmented RGB Image (Mean-Shift)')
ax[1,1].imshow(labels_lab_ms)
ax[1,1].set_title('Segmented LAB Image (Mean-Shift)')
plt.show()
```

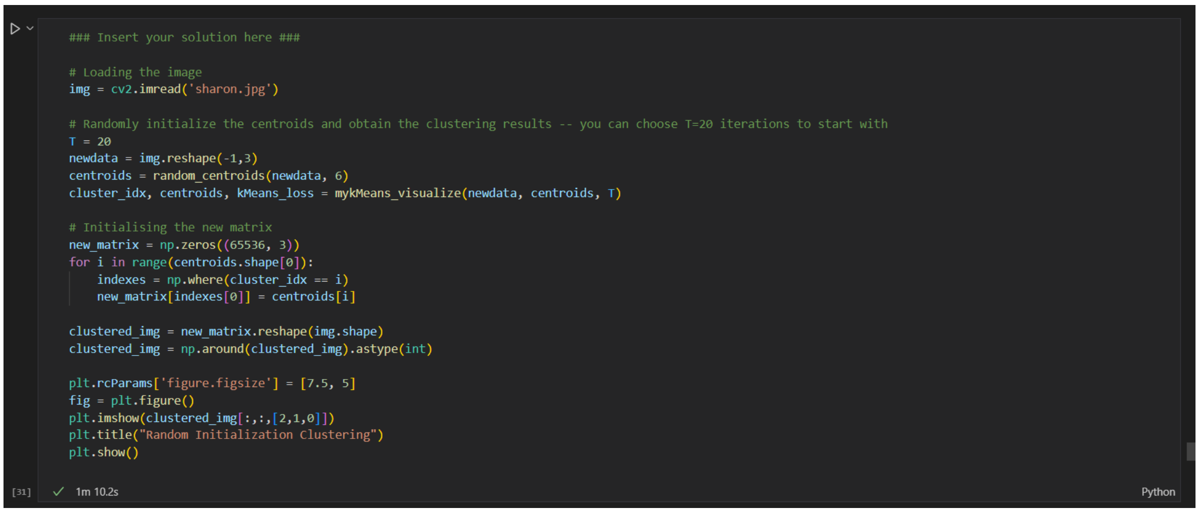
[29] ✓ 0.7s Python

Code for Task 6:

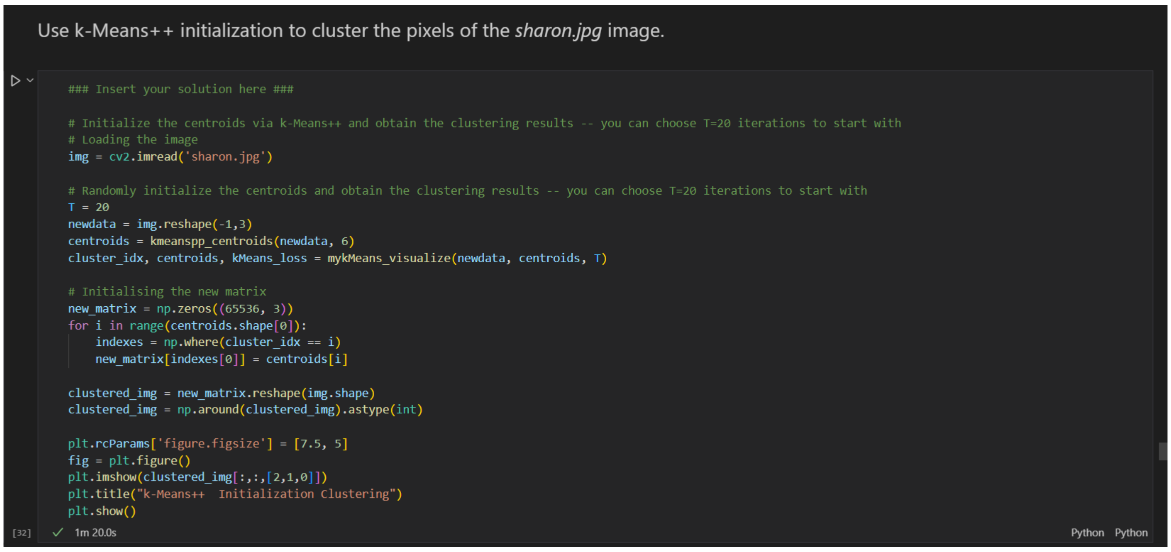
Paste your code with unanswered questions and comments in here.



```python
# We are setting a fixed seed here
np.random.seed(17)
```

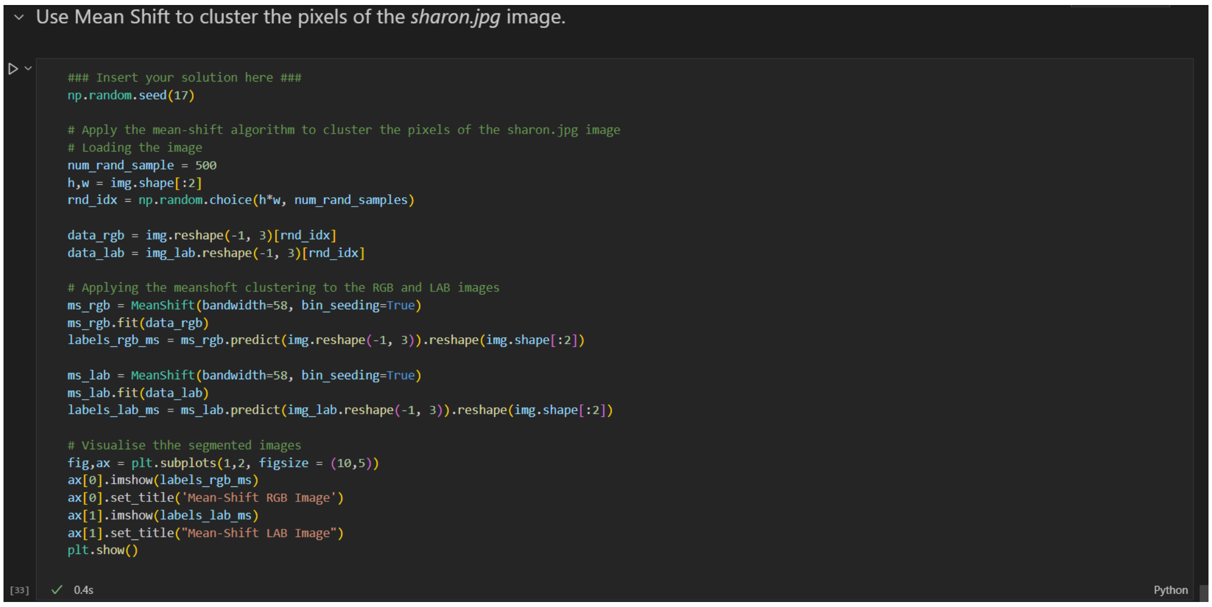


```python
### Insert your solution here ###

# Loading the image
img = cv2.imread('sharon.jpg')

# Randomly initialize the centroids and obtain the clustering results -- you can choose T=20 iterations to start with
T = 20
newdata = img.reshape(-1,3)
centroids = random_centroids(newdata, 6)
cluster_idx, centroids, kMeans_loss = mykMeans_visualize(newdata, centroids, T)

# Initialising the new matrix
new_matrix = np.zeros((65536, 3))
for i in range(centroids.shape[0]):
    indexes = np.where(cluster_idx == i)
    new_matrix[indexes[0]] = centroids[i]

clustered_img = new_matrix.reshape(img.shape)
clustered_img = np.around(clustered_img).astype(int)

plt.rcParams['figure.figsize'] = [7.5, 5]
fig = plt.figure()
plt.imshow(clustered_img[:,:,[2,1,0]])
plt.title("Random Initialization Clustering")
plt.show()
```



Use k-Means++ initialization to cluster the pixels of the *sharon.jpg* image.

```python
### Insert your solution here ###

# Initialize the centroids via k-Means++ and obtain the clustering results -- you can choose T=20 iterations to start with
# Loading the image
img = cv2.imread('sharon.jpg')

# Randomly initialize the centroids and obtain the clustering results -- you can choose T=20 iterations to start with
T = 20
newdata = img.reshape(-1,3)
centroids = kmeanspp_centroids(newdata, 6)
cluster_idx, centroids, kMeans_loss = mykMeans_visualize(newdata, centroids, T)

# Initialising the new matrix
new_matrix = np.zeros((65536, 3))
for i in range(centroids.shape[0]):
    indexes = np.where(cluster_idx == i)
    new_matrix[indexes[0]] = centroids[i]

clustered_img = new_matrix.reshape(img.shape)
clustered_img = np.around(clustered_img).astype(int)

plt.rcParams['figure.figsize'] = [7.5, 5]
fig = plt.figure()
plt.imshow(clustered_img[:,:,[2,1,0]])
plt.title("k-Means++  Initialization Clustering")
plt.show()
```


Use Mean Shift to cluster the pixels of the *sharon.jpg* image.

```python
### Insert your solution here ###
np.random.seed(17)

# Apply the mean-shift algorithm to cluster the pixels of the sharon.jpg image
# Loading the image
num_rand_sample = 500
h,w = img.shape[:2]
rnd_idx = np.random.choice(h*w, num_rand_samples)

data_rgb = img.reshape(-1, 3)[rnd_idx]
data_lab = img_lab.reshape(-1, 3)[rnd_idx]

# Applying the meanshoft clustering to the RGB and LAB images
ms_rgb = MeanShift(bandwidth=58, bin_seeding=True)
ms_rgb.fit(data_rgb)
labels_rgb_ms = ms_rgb.predict(img.reshape(-1, 3)).reshape(img.shape[:2])

ms_lab = MeanShift(bandwidth=58, bin_seeding=True)
ms_lab.fit(data_lab)
labels_lab_ms = ms_lab.predict(img_lab.reshape(-1, 3)).reshape(img.shape[:2])

# Visualise thhe segmented images
fig,ax = plt.subplots(1,2, figsize = (10,5))
ax[0].imshow(labels_rgb_ms)
ax[0].set_title('Mean-Shift RGB Image')
ax[1].imshow(labels_lab_ms)
ax[1].set_title("Mean-Shift LAB Image")
plt.show()
```

[33] ✓ 0.4s Python