Clab-2 Report

ENGN6528

Shreya Chawla
u7195872

25/04/2021

Task-1 Tasks Harris Corner Detector. (6 marks):

For Python users:

1. The completed code for Task 1 is in “harris.py” in “Task 1” folder. All the results and visualizations obtained are saved in “Task 1” folder as well.
2. The “harris.py” python script consists of my_harris() function, non_maximum_suppression() function, plot_corner_img() function to plot corners, and functions originally defined in Lab2 code provided - conv2() and fspecial() functions.
   The following is the code for “Task: Complete the Harris Cornerness” (Fig 1). The arguments are grayscale image “im”, “sigma” used to find window ‘g’, threshold “thresh”, “k” parameter in cornerness formula. It returns an array containing corners’ x, y coordinates and response value.

```
'''
my_harris : Function to detect Haris corners on the given image
'''
def my_harris(im, sigma, thresh, k):
    # Find x and y gradient
    dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    dy = dx.transpose()  # 'dy' is transpose of 'dx'
    Ix = conv2(im, dx)
    Iy = conv2(im, dy)

    # Find corners in image
    # 'g' is the window function which will be used to find R (response). It is a Gaussian window function.
    # Dimensions of 'g' is neighborhood to be considered.
    g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma)
    Iy2 = conv2(np.power(Iy, 2), g)  # y and y
    Ix2 = conv2(np.power(Ix, 2), g)  # x and x
    Ixy = conv2(Ix * Iy, g)  # x and y

    # Find if interest point using cornerness
    # Determinant
    det = Ix2 * Iy2 - Ixy * Ixy
    # Trace
    trace = Ix2 + Iy2
    # R is the cornerness or response
    R = det - k * trace * trace

    # Threshold based on max response value
    thresh *= np.amax(R)  # Threshold for an optimal value, it may vary depending on the image.

    # List of corners detected
    corners = []

    # If resonse > threshold then it is a corner else skip
    for row, response in enumerate(R):
        for col, r in enumerate(response):
            if r > thresh:
                corners.append([row, col, r])
    print('\nNumber of corners detected by my Harris corner detector = ', len(corners))
    return corners

```

Fig 1: Task - Complete the Harris Cornerness

The following is the code for “Task: Perform non-maximum thresholding and return Nx2 matrix of x and y coordinates”. (Fig 2)

The arguments for this function are array of “corners” (a 3-D array containing x, y coordinates of corners and their cornerness score) and distance “dist”, which is the neighborhood on which suppression is performed.

```
'''
non_maximum_suppression : Function performs non-maximum suppression corners at distance dist
'''
def non_maximum_suppression(corners, dist):
    # If no corners found then return that empty list
    if len(corners) == 0:
        return corners

    # Sort corners based on their cornerness value
    corners = sorted(corners, key=lambda c: c[2], reverse=True)
    # List of corners that are not suppressed
    chosen_corners = list()
    chosen_corners.append(corners[0][: -1])

    #  Compare corners to see if there are more corners in vicinity of the current corner
    # If not in the neighborhood of dist distance then that corner is chosen
    for corner in corners:
        for chosen in chosen_corners:
            if abs(corner[0] - chosen[0]) < dist and abs(corner[1] - chosen[1]) < dist:
                break
        else:
            chosen_corners.append(corner[: -1])
    print('Number of corners detected after non maximum suppression = ', len(chosen_corners))

    return chosen_corners

```

Fig 2: Task - Perform non-maximum thresholding and return Nx2 matrix of x and y coordinates

3. The following comments in code corresponding to block #5 have been added as can be seen in Fig 3. (It is a part of Fig 1 according to my function definition).

```

    # Find corners in image
    # 'g' is the window function which will be used to find R (response). It is a Gaussian window function.
    # Dimensions of 'g' is neighborhood to be considered.
    g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma)
```

Fig 3: Comments on the codes in block #5

Comments added to code corresponding to block #7 are in Fig 1 and Fig 7 and explained in 2nd part of Task 1.

The following comments have been added to the fspecial function which has been used in block #5. (Fig 4)

```
def fspecial(shape=(3, 3), sigma=0.5):  # Default values of shape of window and sigma for Gaussian
    m, n = [(ss - 1.) / 2. for ss in shape]
    y, x = np.ogrid[-m:m + 1, -n:n + 1]  # Creates a multidimensional meshgrid
    h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))  # Fourier tranform of x
    # finfo() gives machine limits for floating point types, eps is the difference between 1.0 and the next smallest
    # representable float larger than 1.0. Thus np.finfo(h.dtype).eps would be the smallest float in magnitude.
    h[h < np.finfo(h.dtype).eps * h.max()] = 0
    sumh = h.sum()  # Sum of h array (which are > 0 in previous step)
    if sumh != 0:
        h /= sumh  # Normalize h if sum > 0
    return h
```

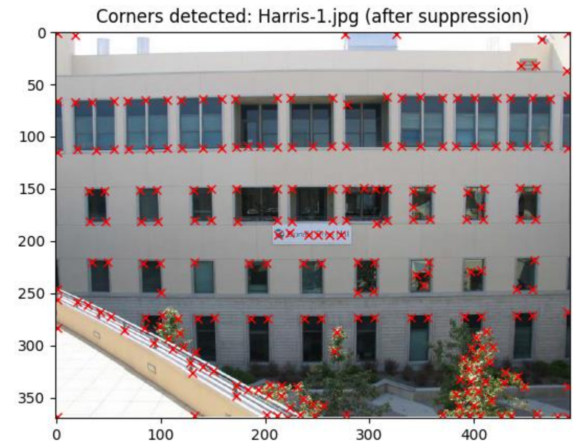
Fig 4: Comments on the codes in block #3 used in block #5

4. The results observed for the 4 test images (Harris-[1,2,3,4].jpg) are shown in Table 1 below. Red crosses have been used to display corners. Also, zoomed images for better observing corners detected in “Harris-3.jpg” and “Harris-4.jpg” are Fig 5 and Fig 6 respectively.

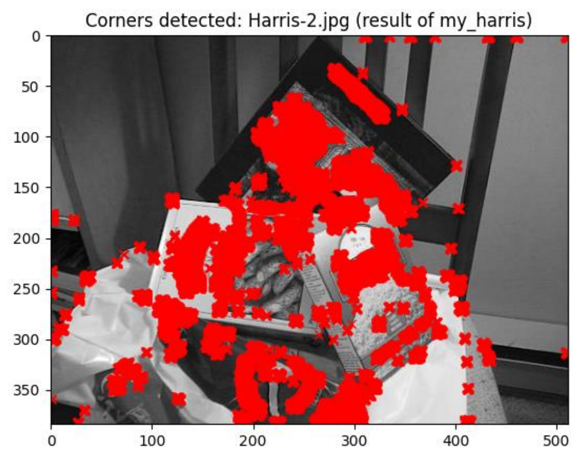
Result of my_harris() (Before suppression)

Result of non_maximum_suppression() (After suppression)

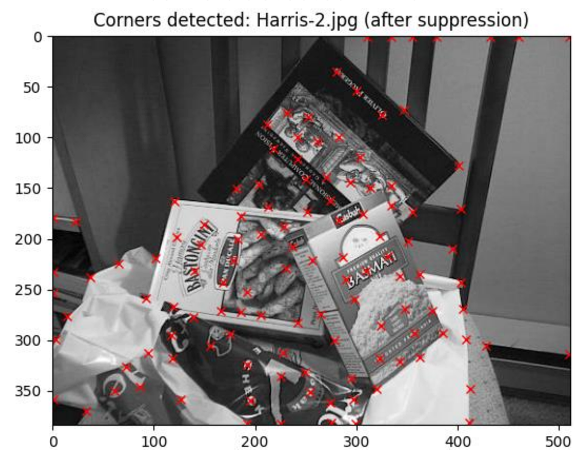
#corners detected = 10170



#corners detected = 228



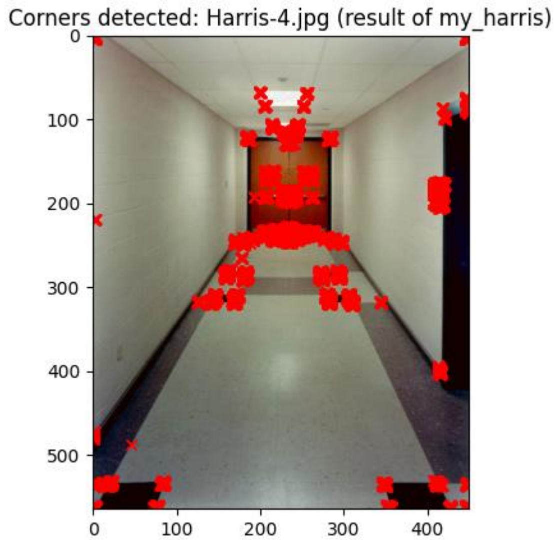
#corners detected = 18268



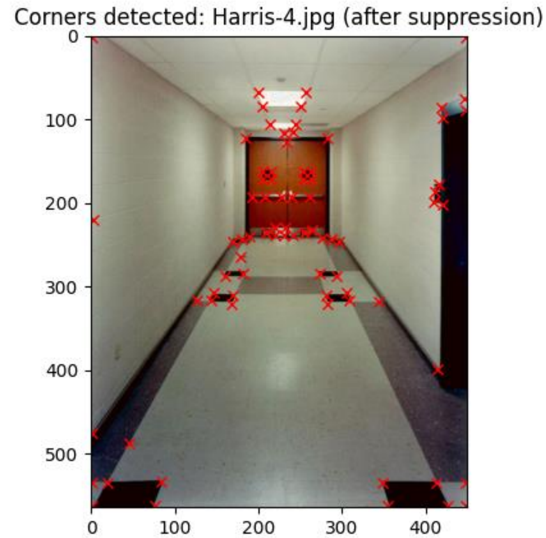
#corners detected = 119

#corners detected = 15264

#corners detected = 253



#corners detected = 4030



#corners detected = 78

Table 1: The left image corresponds to corners obtained after applying my_harris function (without suppression) and the right image corresponds to corners obtained after non_maximum_suppression function is performed.

The my_harris and non_maximun_suppression functions are able to detect many corners in the image as clearly visible in the zoomed images below. (Fig 5 and Fig 6).

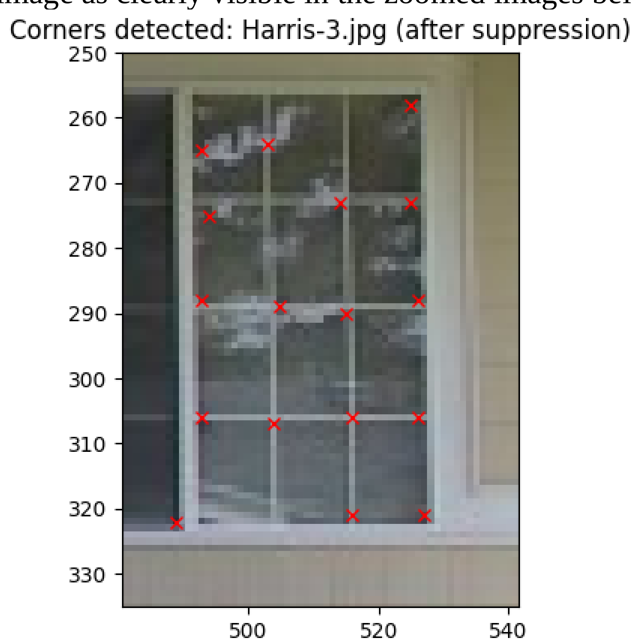


Fig 5: Zoomed Harris corners detected in Harris-3.jpg

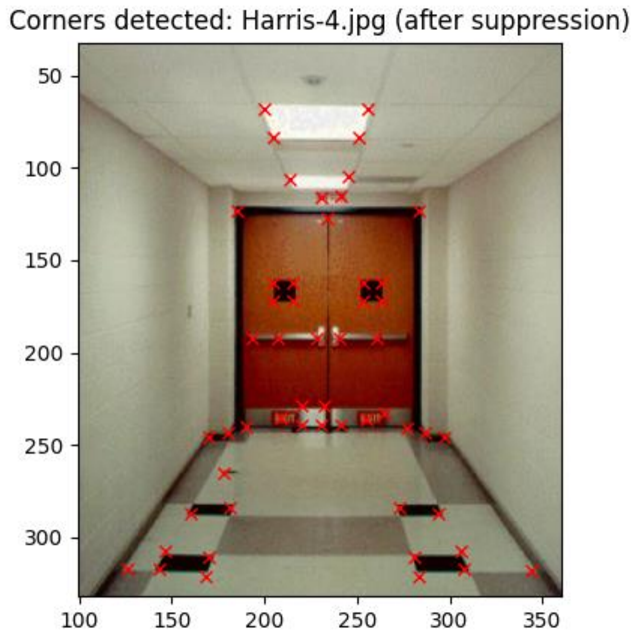


Fig 6: Zoomed Harris corners detected in Harris-4.jpg

5. The result of in-built cv2. cornerHarris() function on "Harris-1.jpg" is shown in Fig 7. Upon comparison, although the results are similar, the inbuilt function detects more corners than my_harris(). This could be due to different values of parameters used in both the functions although the arguments for both functions have same value.

Harris corner is affected by scale of image and is also dependent on the rotation of image. The threshold and the value of 'k' in cornerness response score are also some factors that affect the performance of Harris corner detection.

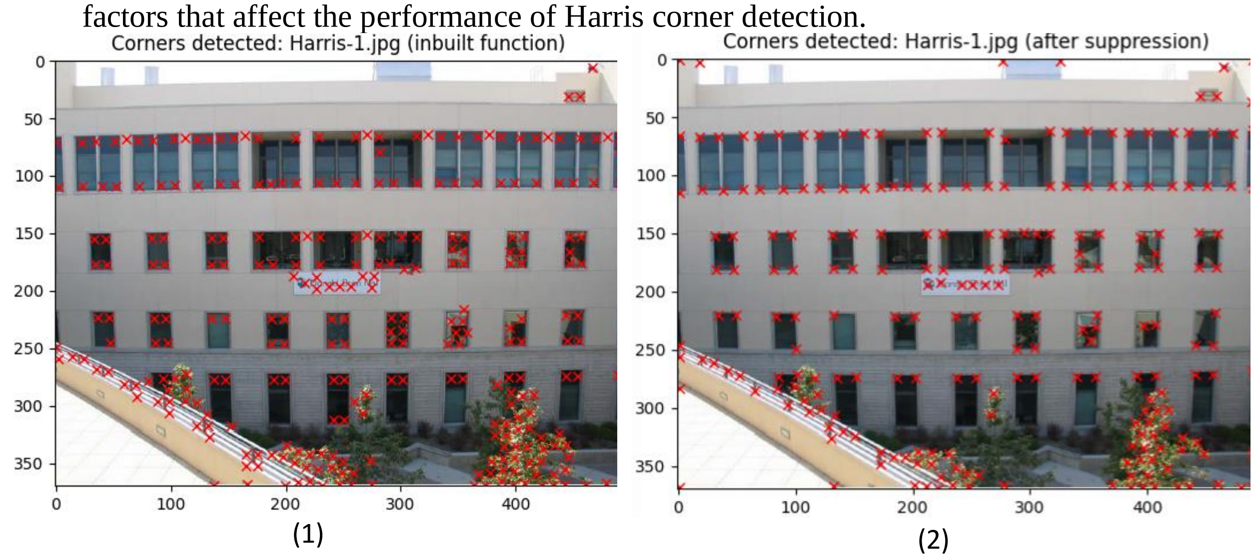


Fig 7: Comparison between Harris corners detected in "Harris-1.jpg" using cv2's inbuilt function (1) (left) and detected by my_harris (2) (right)

6. In ‘Harris-5.jpg’, we cannot get any corner as the corner response score (R) is less than threshold for all windows in image. That is because there is an edge in this image (i.e. R<0) which cannot be detected by Harris corner technique. Fig 8 shows the results visualized below.

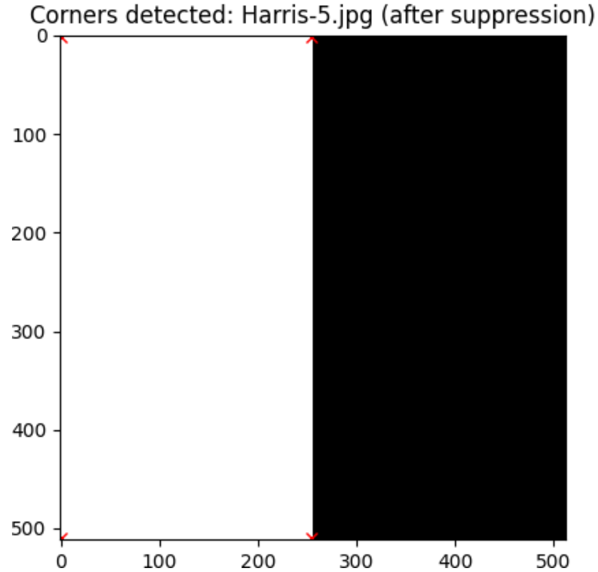


Fig 8: Harris corners for “Harris-5.jpg”

7. The visualization of results for “Harris-6.jpg” are in Fig 9. “Harris-6.jpg” contains a lot of noise hence it gets a lot of false positives in detecting corner. A smoothening filter like Bilateral filter can smoothen the noise while keeping the corner (and edges) sharp. This can help improve the corner detection.

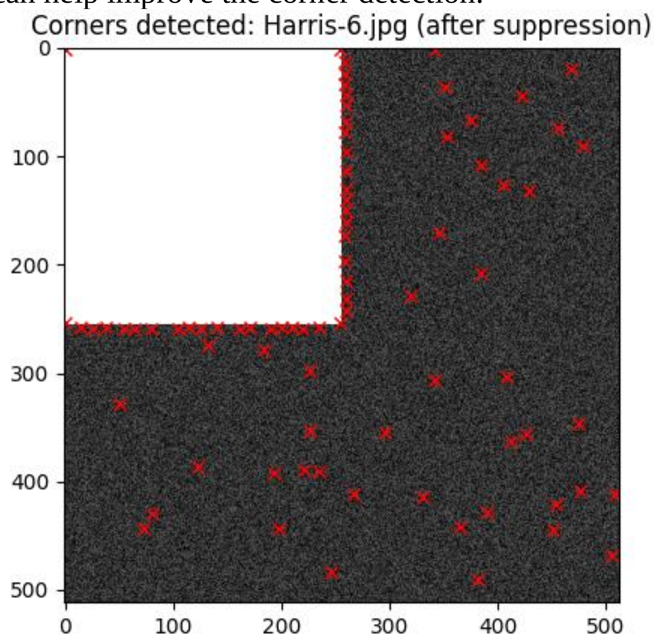


Fig 9: Harris corners for “Harris-6.jpg”

Task-2: K-Means Clustering and Color Image Segmentation. (5 marks)

In this task, you are asked to implement your own K-means clustering algorithm for colour image segmentation, and test it on the following **two images** as shown in Fig.1 (you can download them from wattle). Please first convert one image (in PNG-type (Portable Network Graphics)) of 16bit to 8bit image before the following process.

Fig. 1

1. The implementation of *my_kmeans()* is in Fig 10 below. The input is the data points to be processed (data) and the number of clusters (K), and the output is several clusters data (clusters).

```
'''
my_kmeans function: Performs K-means algorithm on data and returns the K clusters
'''
def my_kmeans(data, K):
    start_time = time.time()
    # Step 1 Initialize the K centroids randomly
    centroids = np.random.random((K, data.shape[1]))
    print('Initialized centroids to: ', centroids)
    clusters = np.zeros_like(data)  # Keeps track of which datapoint belongs to which cluster
    diff = True  # Boolean which keeps track of when to stop - reached local maximum

    while diff:
        new_centroids = np.zeros_like(centroids)  # New centroids calculated are stored in this array

        # Step 2 Find Euclidean distance (L2 norm) of each point from each centroid and store it in a dictionary
        for i in range(data.shape[0]):
            dist = dict()  # Dictionary to store distance of each centroid from each point
            for j in range(K):
                dist.update({j: np.linalg.norm(data[i] - centroids[j])})
            # Step 3 Assign points to closest cluster (or centroid)
            index = [key for key in dist if dist[key] == min(dist.values())]
            if len(index) > 1:  # If more than 1 centroids are at minimum distance
                index = index[0]
            clusters[i] = centroids[index]

        # Step 4 Compute the new centroid
        for i in range(K):
            cluster = np.where((clusters == centroids[i]).all(axis=1))[0]
            if cluster.shape[0] > 0:
                new_centroids[i] = sum(data[cluster]) / cluster.shape[0]
            else:
                new_centroids[i] = centroids[i]

        # Step 5 If no difference between between new and old set of centroids, then stop, else repeat step 2 to 4
        if np.isclose(centroids, new_centroids, 1e-15).all():
            diff = False
            # print('Game over')
        else:
            print('diff = ', centroids - new_centroids)
            centroids = new_centroids
    end_time = time.time()
    print('Time taken for the k means algorithm to rum = ', end_time - start_time)
    return clusters
```

Fig 10: Code for my_kmeans function

2. The images in LAB color space are displayed in Fig .

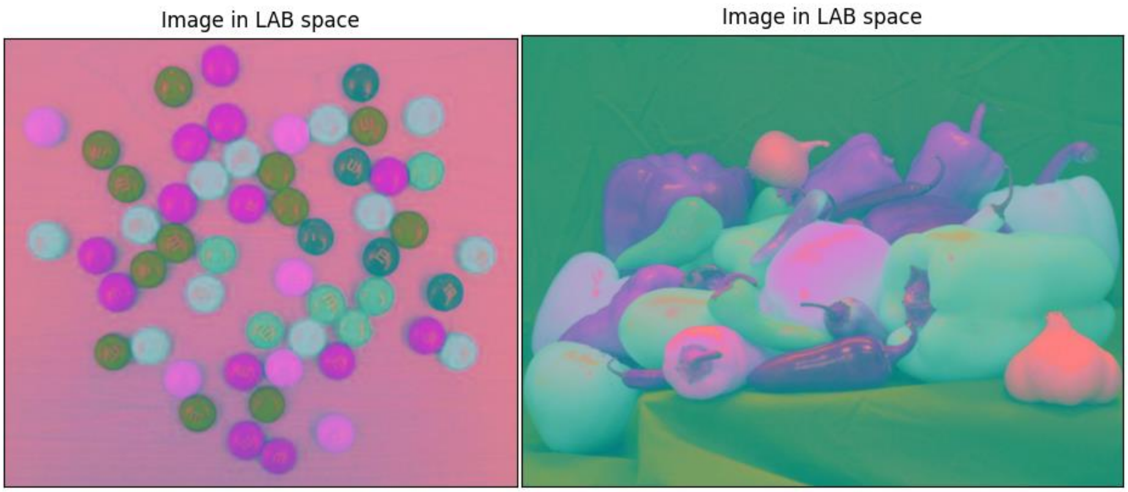


Fig 11: 'mandm.png' and 'pepper.png' in LAB color space

The boolean "with_coordinates" keeps track of weather to use 5D data or 3D. The 5-D vector that encodes: (1) L∗ - lightness of the color; (2) a∗ - the color position between red and green; (3) b∗ - the position between yellow and blue; (4) x, y - pixel coordinates. The code for the same is below in Fig 12.

```
with_coordinates = False  # Boolean for how the segmentation is to be carried out (with or without pixel coordinates)
clusters_k = np.array([])  # Stores all the clusters returned from my_kmeans method (K-means)
clusters_k_pp = np.array([])  # Stores all the clusters returned from my_kmeans_pp method (K-means++)
if with_coordinates:  # With pixel coordinates
    # Create 5-D image data: L, A, B, X, Y
    fiveD_data = list()
    for i in range(im.shape[0]):
        for j in range(im.shape[1]):
            fiveD_data.append(np.append(im[i, j], np.array([i, j])))
    fiveD_data = np.array(fiveD_data)
    fiveD_data = normalize(fiveD_data)
    clusters_k = my_kmeans(fiveD_data, K)
    # Take only first 3 values (L,A,B) to show image, drop x,y coordinate of centroid
    clusters_k = np.reshape(clusters_k[:, :3], im.shape)
    # Similarly for K-means++ clusters
    clusters_k_pp = my_kmeans_pp(fiveD_data, K)
    # Take only first 3 values (L,A,B) to show image, drop x,y coordinate of centroid
    clusters_k_pp = np.reshape(clusters_k_pp[:, :3], im.shape)

else:  # Without pixel coordinates
    # Create 3-D image data: L, A, B
    threeD_data = np.reshape(im, (im.shape[0] * im.shape[1], 3))
    threeD_data = normalize(threeD_data)
    clusters_k = my_kmeans(threeD_data, K)
    clusters_k_pp = my_kmeans_pp(threeD_data, K)
    clusters_k = clusters_k.reshape(im.shape)
    clusters_k_pp = clusters_k_pp.reshape(im.shape)
```

Fig 11: Code for dealing with coordinates or without

3. The key steps for K-means++ algorithm are initialization and convergence. The convergence of K-means++ algorithm is performed the same way as K-means algorithm. They only differ in initialization. Let D(x) denote the shortest distance from a data point in X to the closest center that was already chosen. The steps for the complete K-means++ algorithm are:
   1. Take one centroids c_1, uniformly sampled from X.
   2. Take a new centroids c_i, choosing $x \in X$ with probability $D(x)^2 / \Sigma_{x \in X} D(x)^2$.

3. Repeat step 2 until the number of centroids is k.
4. For each i ∈ {1, . . ., k}, set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$.
5. For each i ∈ {1, . . ., k}, c_i is set to be the center of mass of all points in cluster

   C_i: $c_i = 1/|Ci| \Sigma_{x \in Ci}$ x.

6. Repeat Steps 4 and 5 until the clusters C no longer change.

Steps 1 to 3 are the initialization steps which differ from K-means. The rest are the same steps.

The corresponding code is below (Fig 12)

```
'''
my_kmeans_pp function: Performs K-means++ algorithm on data and returns the K clusters
'''
def my_kmeans_pp(data, K):
    # Step 1 Initialize the K centroids randomly
    centroids = []
    # 1.1) Choose the 1st centroid uniformly sampling from the data points in data
    centroids.append(data[np.random.randint(data.shape[0])])
    # 1.2) For the next K-1 centroids, we need to choose them as far from the 1st as possible
    for c_id in range(K - 1):
        dist = []  # Shortest distance between point and closest centroid (D(x))
        for i in range(data.shape[0]):
            point = data[i, :]
            # For every point in X, choose the min distance between
            # np.inf and L2 norm (or euclidean distance between points and the centroids)
            d = np.inf
            for j in range(len(centroids)):
                temp_dist = np.linalg.norm(point - centroids[j])
                d = min(d, temp_dist)
            dist.append(d)

        # Select data point with maximum distance from the 1st centroid as the next centroid
        dist = np.array(dist)
        next_centroid = data[np.argmax(dist)]
        centroids.append(next_centroid)
        # 1.3) Repeat 1.1 and 1.2 until there are K clusters

    centroids = np.array(centroids)

    print('Initialized centroids to: \n', centroids)
    start_time = time.time()  # Keep track of convergence time
    clusters = np.zeros_like(data)  # Keeps track of which datapoint belongs to which cluster
    diff = True  # Boolean which keeps track of when to stop - reached local maximum

    while diff:
        new_centroids = np.zeros_like(centroids)  # New centroids calculated are stored in this array
        # Step 2 Find Euclidean distance (L2 norm) of each point from each centroid and store it in a dictionary
        for i in range(data.shape[0]):
            dist = dict()  # Dictionary to store distance of each centroid from each point
            for j in range(K):
                dist.update({j: np.linalg.norm(data[i] - centroids[j])})
            # Step 3 Assign points to closest cluster (or centroid)
            index = [key for key in dist if dist[key] == min(dist.values())]
            if len(index) > 1:  # If more than 1 centroids are at minimum distance
                index = index[1]
            clusters[i] = centroids[index]

        # Step 4 Compute the new centroid
        for i in range(K):
            cluster = np.where((clusters == centroids[i]).all(axis=1))[0]
            if cluster.shape[0] > 0:
                new_centroids[i] = sum(np.abs(data[cluster])) / cluster.shape[0]
            else:
                new_centroids[i] = centroids[i]

        # Step 5 If no difference between between new and old set of centroids, then stop, else repeat step 2 to 4
        if (centroids - new_centroids).sum() == 0:
            diff = False
        else:
            centroids = new_centroids
    end_time = time.time()
    print('Time taken for the k-means++ algorithm to converge = ', end_time - start_time)
    return clusters
```

Fig 12: Code for K-means++ algorithm

The convergence time of k-means is 6 seconds for k=3 without coordinates and that for k-means++ is 5 seconds. Thus k-means++ is faster than k-means.

Task-3: Face Recognition using Eigenface. (10 marks)

1. The script “my_images.py” converts image to gray scale, resizes them to standard size and stores them in folder “my_images”. The 10 images are labelled starting with “subject16” and of them one is test image called “subject16.test.jpg” being added to “test” folder and the rest to “train” folder of “my_images” directory.
   Image alignment is important as it improves the accuracy of face recognition using Eigenface. The images are directedly averaged and eigenvectors based on that average are used for eigenface. Hence, misalignment of image and background variations can lead to incorrect results.

2. Train an Eigen-face recognition system. Specifically, at least your face recognition system should be able to complete the following tasks:

   (1) All the 135 training images from Yale-Face are read and each image is represented as a single data point in a high dimensional space. All the data points into a big data matrix.
   (2) PCA has been performed on the data matrix X to find top k eigenvectors in the function pca() in “pca.py” (Fig 13). First the data (with each column as image) is normalized by subtracting mean from image. Then its covariance matrix is found. Next, eigenvectors and eigenvalues of the covariance matrix are calculated. The top k of Eigenvectors and Eigenvalues are returned with mean.

```

'''
pca function: Performs pca on matrix X and returns top k Eigenvalues, Eigenvectors, and mean of matrix X
'''
def pca(X, k):
    mean = X.mean(axis=0)
    # Subtract mean from images or datamatrix X
    X = X - mean

    n, m = X.shape
    if n > m:  # Use transpose trick when one of the dimensions of matrix is greater than other
        C = X.T @ X  # Covariance Matrix
        eigval, eigvec = np.linalg.eigh(C)
        eigvec = X @ eigvec
        for i in range(m):
            eigvec[:, i] = eigvec[:, i] / np.linalg.norm(eigvec[:, i])
    else:
        C = X @ X.T  # Covariance Matrix
        eigval, eigvec = np.linalg.eigh(C)

    # Sort eigenvectors descending by their eigenvalue
    idx = np.argsort(-eigval)
    eigval = eigval[idx]
    eigvec = eigvec[:, idx]
    # select only top k
    eigval = eigval[0: k].copy()
    eigvec = eigvec[:, 0: k].copy()

    return eigval, eigvec, mean

```

Fig 13: The function pca() performs PCA dimensionality reduction

The mean face obtained by averaging all the faces in training set is shown in image below. (Fig 14)

Mean Face

Fig 14: Mean face of all training images

Let a matrix S of dimension n x k be such that $k < n$. Then the non-zero eigenvalues of matrix SS^T are equal to the eigenvalues of matrix S^TS. And if the eigenvector of S^TS is v with eigenvalue λ, then the eigenvector of SS^T is Sv with same eigenvalue λ. It is then normalized (Theorem 1.4 from Week 06 Lect 09 Face at page 59).
The reason why this method works faster is that the matrix S^TS is a smaller k x k matrix compared to the larger SS^T matrix of n x n dimension. Using this theorem, we can find the eigenvectors of the large data matrix more quickly. This has been used in the code (Fig 13 – code lines: 44 to 50).

(3) The top 10 eigenfaces are plotted in image below. (Fig 15)

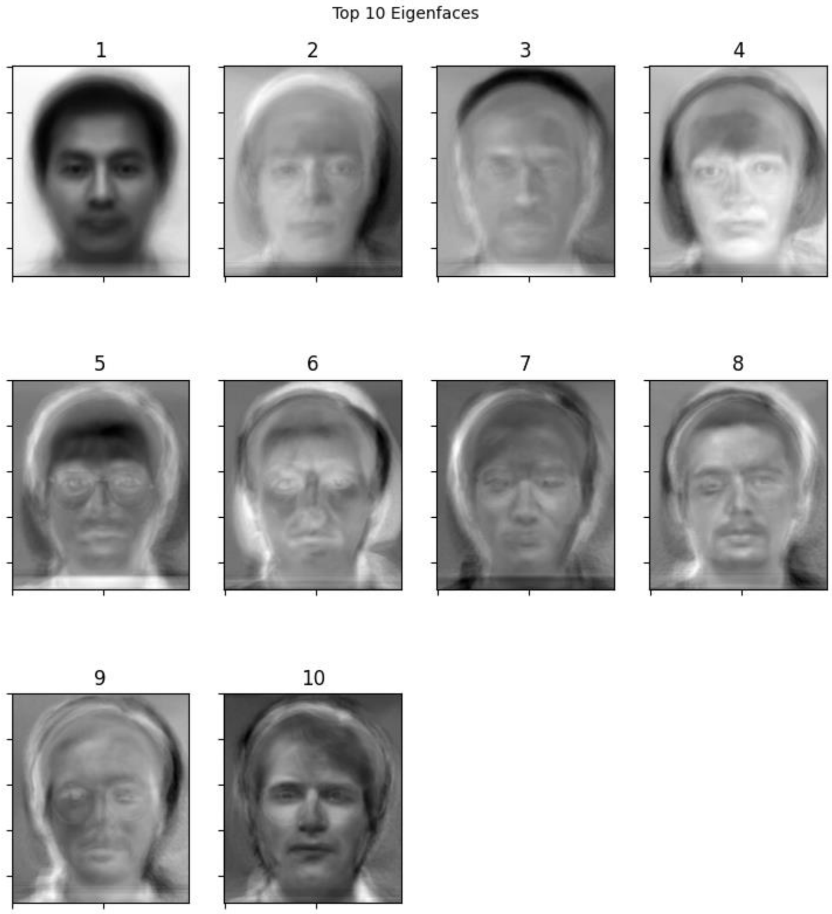
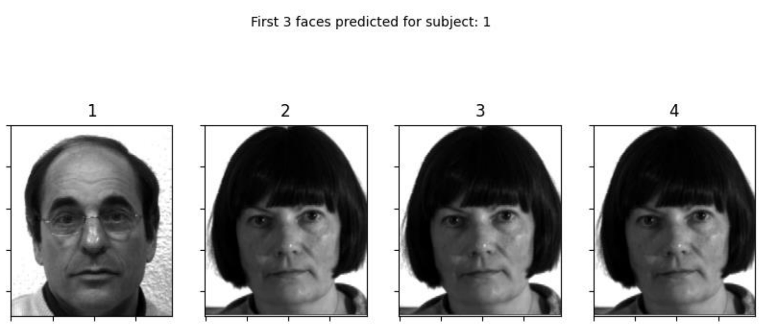


Fig 15: Top k (k=10) Eigenfaces

(4) For each of the 10 test images in the Yale-Face dataset, their projection onto the basis spanned by the top k eigenfaces is determined. This projection is used as a feature to perform nearest-neighbour search over all 135 faces and find out the top three face images that are most similar to the reference one are found. These top 3 faces are shown below in Table 2.

Table 2: Top 3 faces detected (2,3,4) to be closest to test face (1)

Subject 01



Subject 02

First 3 faces predicted for subject: 2

Subject 03

First 3 faces predicted for subject: 3

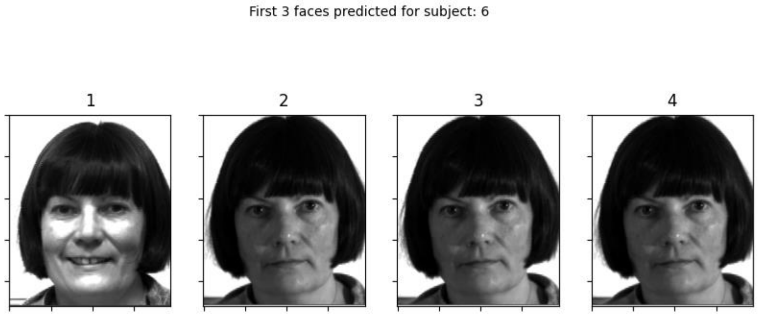
Subject 04

First 3 faces predicted for subject: 4

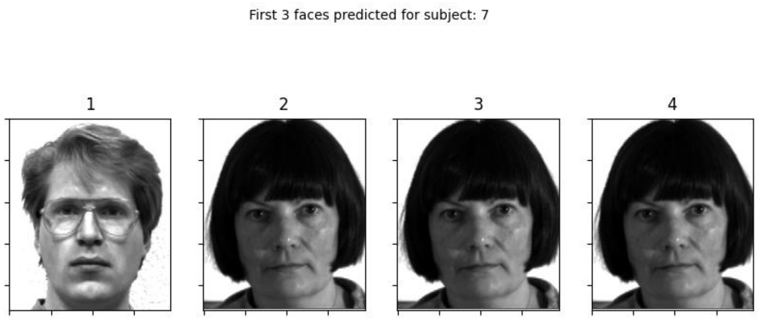
Subject 05

First 3 faces predicted for subject: 5

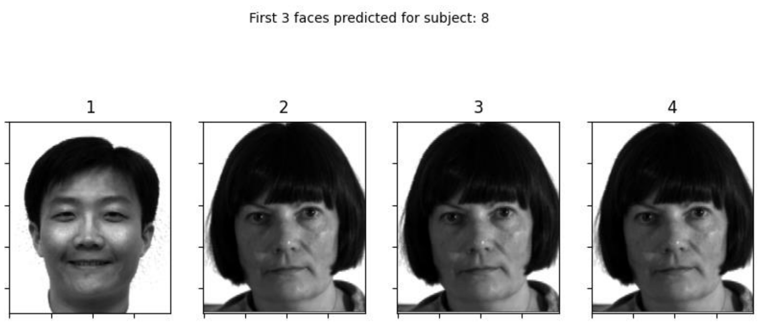
Subject 06



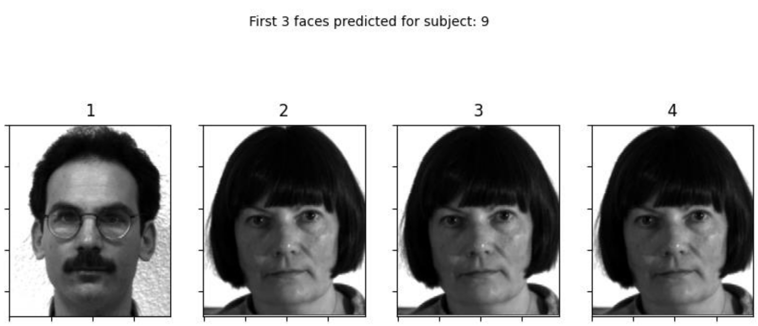
Subject 07



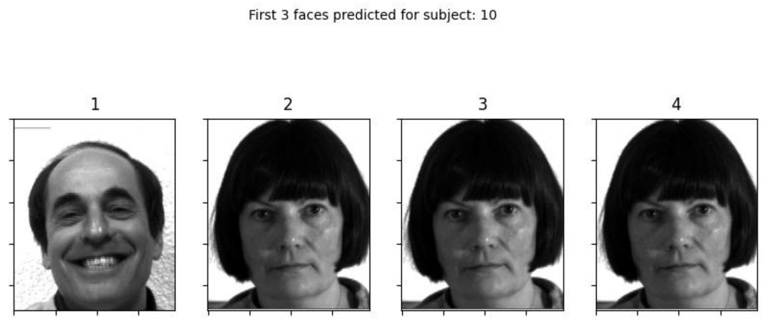
Subject 08



Subject 09



Subject 10



Accuracy is described in table below. (Table 3)

Table 3: Top-1 and Top-3 accuracies are listed below.

Accuracy (top-1)	Accuracy (top-3)
10%	30%

This is very poor accuracy. One possible reason for such low accuracy is the difference in lighting conditions in image and these images have varying background lighting.

(1) The result of running my test image through my face recognition system is in Fig 16.

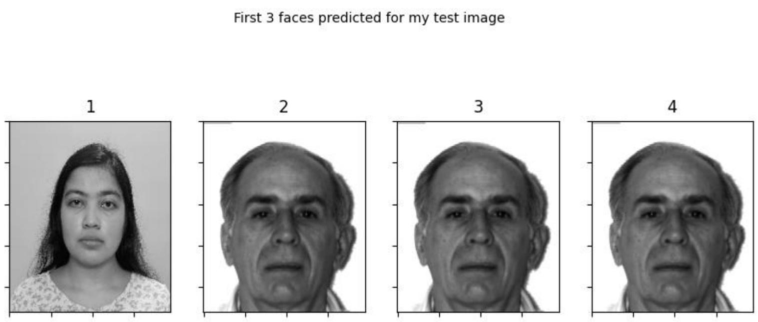


Fig 16: Top 3 faces resulting from comparison of my test face with training set images

(2) The previous experiment is repeated by pre-adding the other 9 additional images of my face into the training set (a total of 144 training images). The top 3 faces that are the closest to my test face are displayed below (Fig 17).

Fig 17: Top 3 faces resulting from comparison of my test face with the training set images including my image