Clab2-Report-u7156387

Task1

Q2&3)

```
img = imread('Task1/Harris-5.jpg')
img = gray2rgb(img)
#converted rgb2gray for faster computation
imggray = rgb2gray(img)

plt.imshow(imggray, cmap="gray")
plt.axis("off")
plt.show()
```

The below function returns harris_response

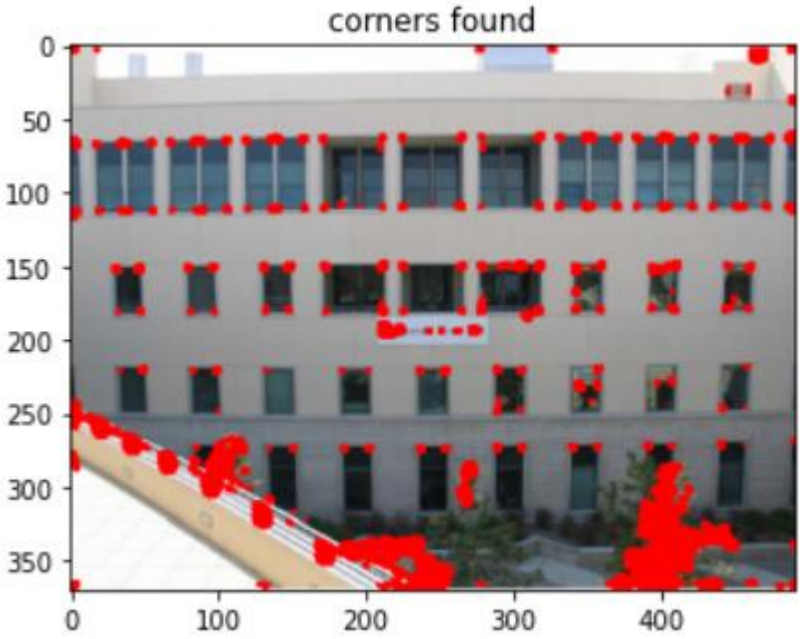
```
def func_harris_r():
    # taking the contanst value which ranges from [0.4,0.6]  to be 0.5 after hit & trial
    k = 0.05

    # computing determinant
    detA = Ix2 * Iy2 - Ixy ** 2
    # computing trace
    traceA = Ix2 + Iy2

    # Calculating the harris response after substituting the values for the determinant, trace and k
    harris_response = detA - k * traceA ** 2
    return harris_response
harris_response_ = func_harris_r()
```

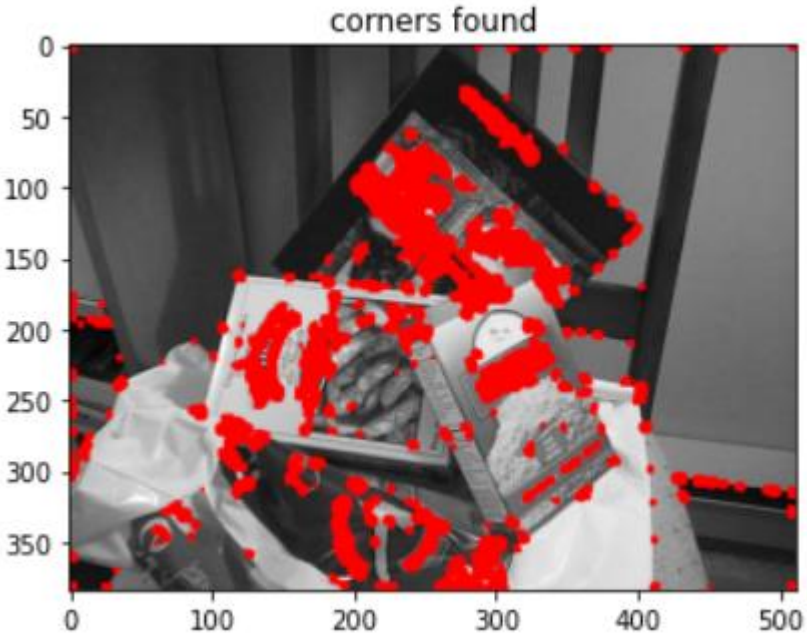
The below function perform non-maximum suppression and thresholding and finally I display ouput image.

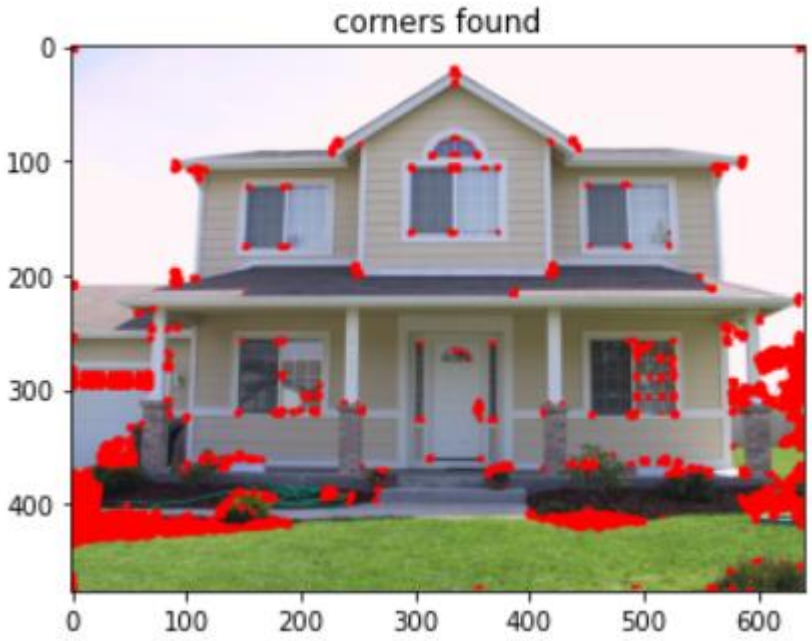
```
def non_max_supp(harris_response):
    img_copy_for_corners = np.copy(img)
    #Nonmaximum Supression
    # Using sigma to supress points
    harris_response = (harris_response>sigma*abs(np.mean(harris_response)))*harris_response
    harris_response = cv2.dilate(harris_response,None)
    # Using threshold to supress points
    for rowindex, response in enumerate(harris_response):
        for colindex, r in enumerate(response):
            if r > thresh*harris_response.max():
                img_copy_for_corners[rowindex, colindex] = [255,0,0]
    return img_copy_for_corners
img_copy_for_corners = non_max_supp(harris_response_)
plt.title("corners found")
plt.imshow(img_copy_for_corners,cmap='gray')
plt.show()
```

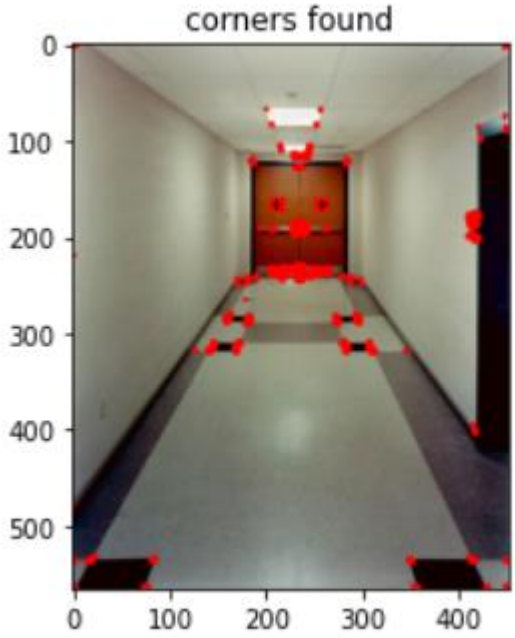
Q4)

Testing with four images for corner detection


corners found
0
50
100
150
200
250
300
350
0
100
200
300
400


corners found
0
50
100
150
200
250
300
350
0
100
200
300
400
500



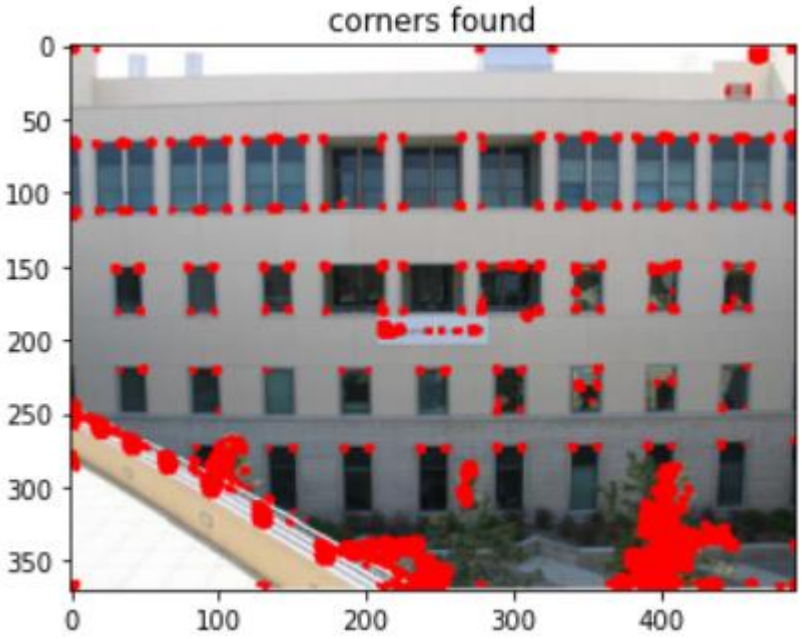


Apart from too closed corners my corner detection algorithm is able to successfully identify the corners.

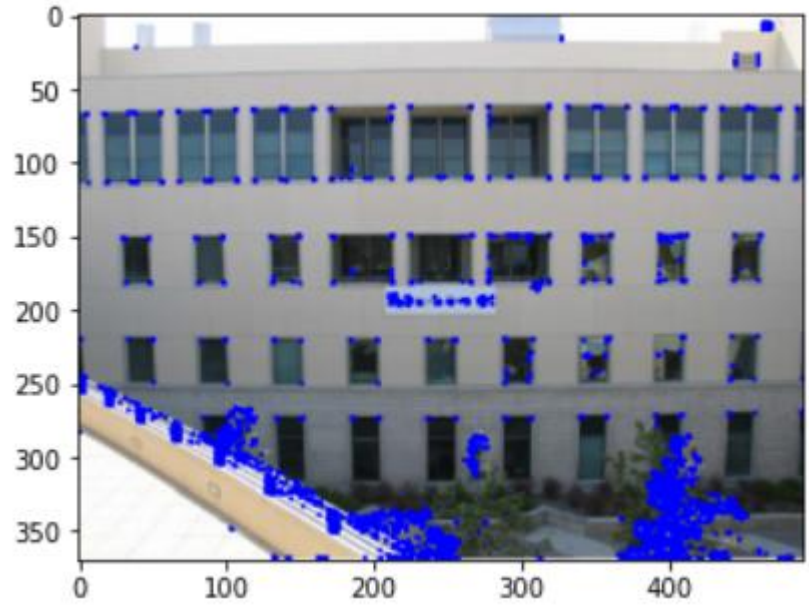
Q5

Red – my algorithm, Blue – cv2

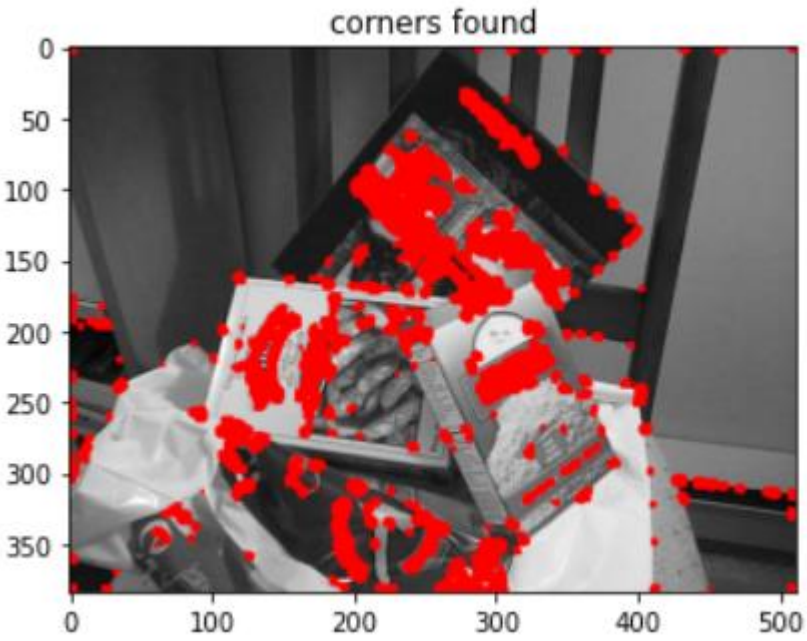
Harris 1



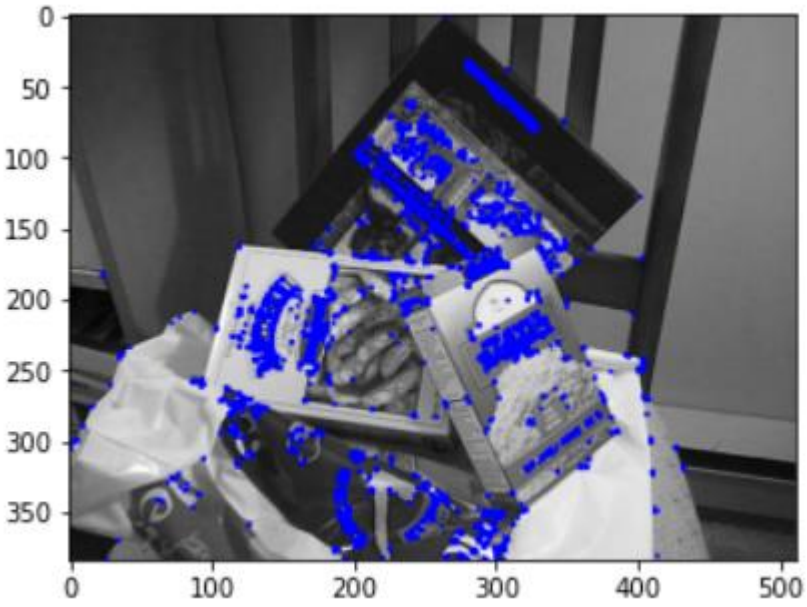
Using cv2



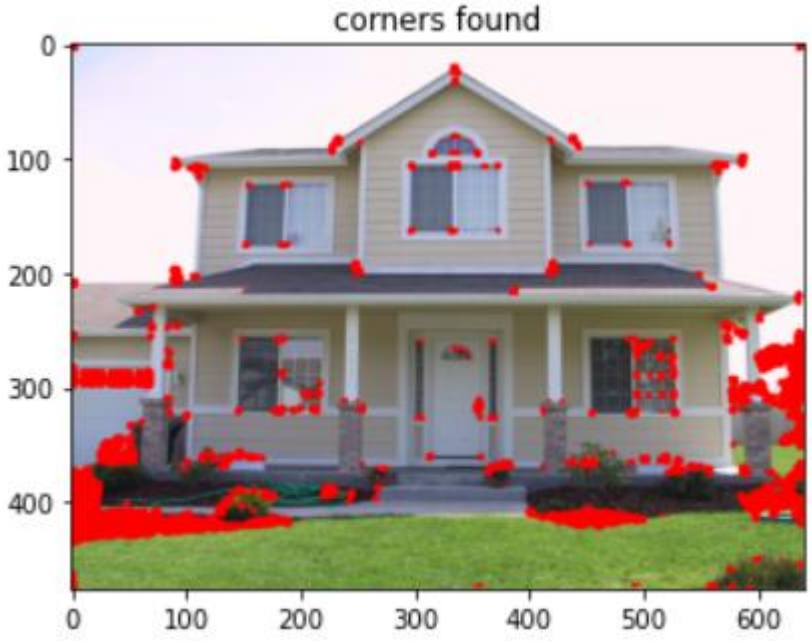
Harris 2



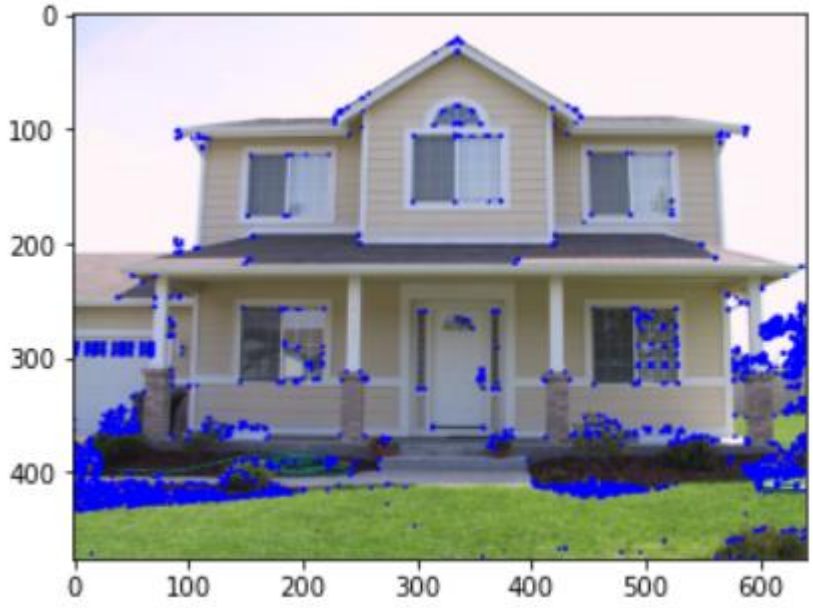
Using cv2



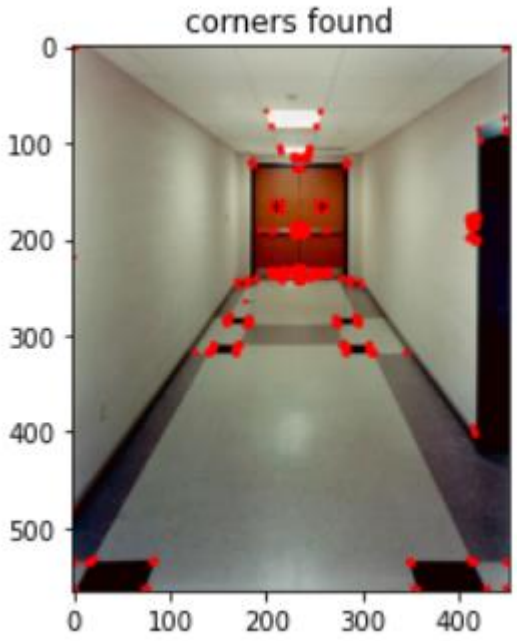
Harris 3


corners found
0
100
200
300
400
0
100
200
300
400
500
600

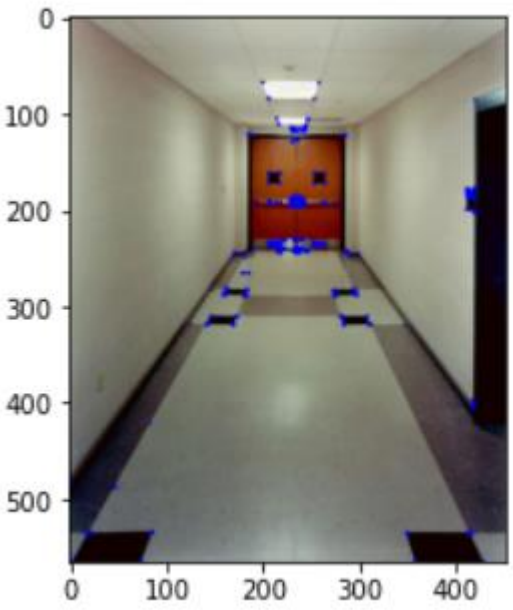
Using cv2


0
100
200
300
400
0
100
200
300
400
500
600

Harris 4



Using cv2

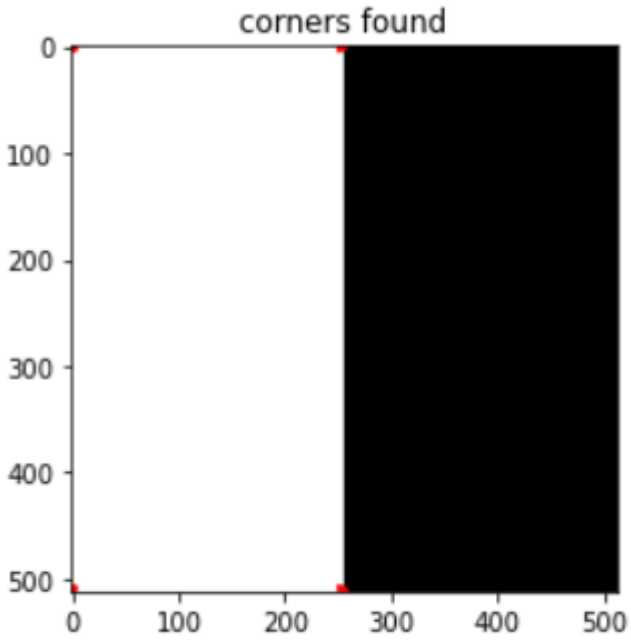


We can clearly observe cv2 algorithm and my algorithm are at par, if we ignore the size of dots on the corners.

The reason why harris corner detection is so important because of its ability tosuccesfully distinguish between corners and edges. Though it has been quite successful but the algorithm suffers from noise and thereby sometimes gives wrong result. If non-maximum suppression is not properly done, we get many invalid points. But, overall the algorithm is quite successful for detecting noise free images. In above images we can notice that the corner detection is not affected by illumination.

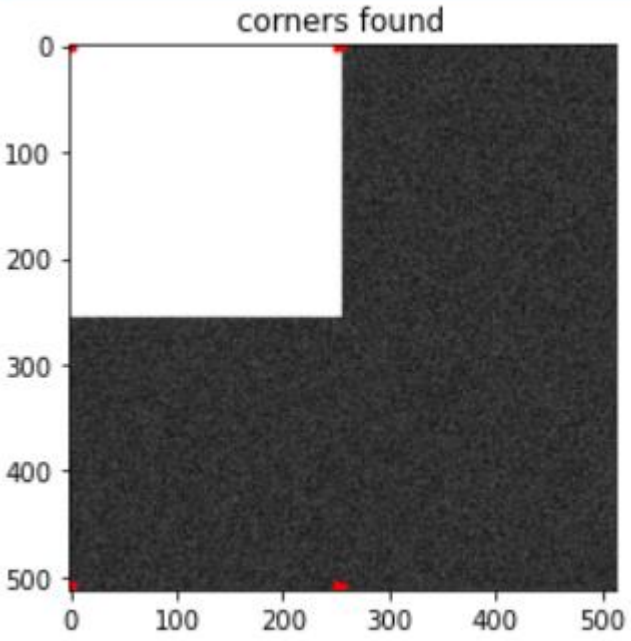
Q6)

Harris -5



We can see we find the corners for the white side only, the corners in the black side are lost due to non max suppression. Also at the right the corner detector have failed due to no further distinction in pixel values.

Q7)



Though the top two corners at white are identified correctly, but due to noises we see that some random points are identified as corners. Because of this the image can be filter for removing noise. We can try using gaussian and bilateral filter.

Task 2

Q1

```
def get_initial_centroids(data, k, seed=None):
    """As initial centroids, select k data points at random."""
    if seed is not None:  # a good way to get consistent results
        np.random.seed(seed)
    n = data.shape[0]  # the total number of data points

    # Choose K indices from the [0, N] set.
    rand_indices = np.random.randint(0, n, k)

    # Since several entries would be nonzero due to averaging, keep centroids in a dense format.
    # As long as at least one document in a cluster contains a term, the cluster is considered complete.
    # It will have a nonzero weight in the centroid's TF-IDF vector.
    centroids = data[rand_indices, :]

    return centroids


def my_centroid_pairwise_dist(X, centroids):
    # using the imported pairwise_distances function from sklearn.metrics to calculate pairwise dist.
    return pairwise_distances(X, centroids, metric="euclidean")



def my_assign_clusters(data, centroids):

    # Calculate the distances between each data point and the set of centroids using the following formula:
    distances_from_centroids = my_centroid_pairwise_dist(data, centroids)

    # Calculate each data point's cluster assignment:
    cluster_assignment = np.argmin(distances_from_centroids, axis=1)

    return cluster_assignment


def my_revise_centroids(data, k, cluster_assignment):
    new_centroids = []
    for i in range(k):
        # All data points that belong to cluster I should be selected.
        member_data_points = data[cluster_assignment == i]
        # Calculate the data points' mean.
        centroid = member_data_points.mean(axis=0)
        new_centroids.append(centroid)
    new_centroids = np.array(new_centroids)

    return new_centroids
```

```python
def my_kmeans(data, k, initial_centroids, maxiter=450):
    """This function applies k-means to a collection of data and initial centroids.
    maxiter: The maximum number of iterations that can be performed.(450 is the default)"""
    centroids = initial_centroids[:]
    prev_cluster_assignment = None

    for itr in range(maxiter):

        # 1. Assign clusters based on the centroids that are closest to each other.
        cluster_assignment = my_assign_clusters(data, centroids)

        # 2. For each of the k clusters, compute a new centroid by averaging all data points.
        #    That cluster was given a certain number of points.
        centroids = my_revise_centroids(data, k, cluster_assignment)

        # Check for convergence and move on to the next step if none of the assignments have changed.
        if (
            prev_cluster_assignment is not None
            and (prev_cluster_assignment == cluster_assignment).all()
        ):
            break

        if prev_cluster_assignment is not None:
            num_changed = np.sum(prev_cluster_assignment != cluster_assignment)


        prev_cluster_assignment = cluster_assignment[:]

    return centroids, cluster_assignment
```

Q2

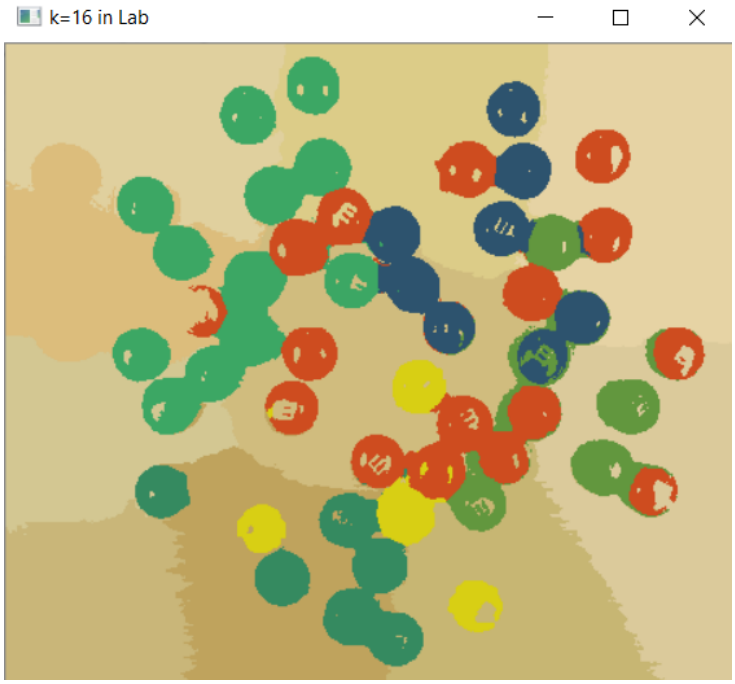
(1)

If k=3 is used, the algorithm will find three clusters in the picture. Likewise for k=8, and for k=16.

Reshaping the image into a two-dimensional (2D) array of pixels with L,a,b,x,y and visualising.

We can see only 3 clusters identified.

We can observe 8 clusters have been identified.

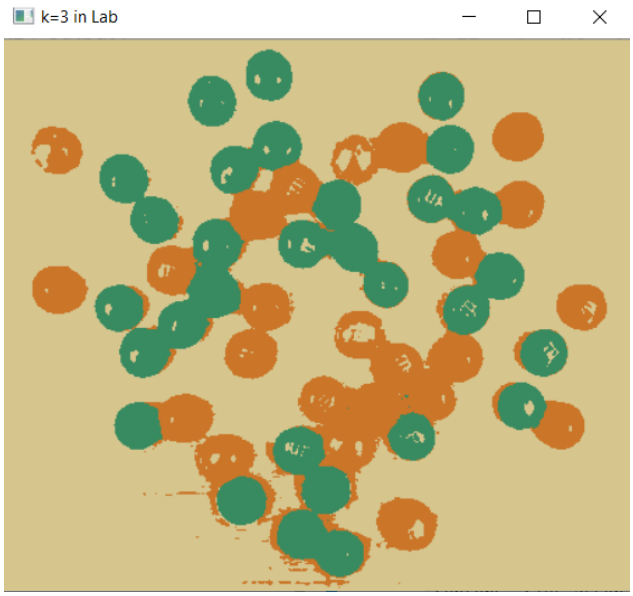


We can observe 16 clusters have been identified.

(2)

K=3

Without x,y co-ordinates

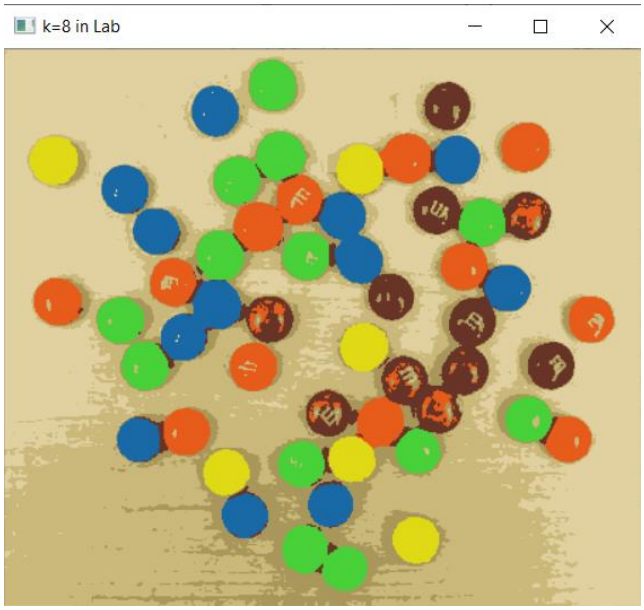


With x,y co-ordinates

When we include x-y positions we see more detailed cluster identification.

K=8

Without x,y co-ordinates



With x,y co-ordinates

It can be further observed when increasing k first of all important clusters are identified when we don't use x-y co-ordinates.

Q3)

The K-means algorithm is sensitive to how the centroids or mean points are initialised. As a result, if a centroid is set to be a far-way point, it may end up with no points associated with it, whereas more than one cluster may be connected to a single centroid. Therefore, there is need for a method which can overcome the initialization problem. The method is known as Kmean++, everything except the initialization is same between the algorithms. The initialization algorithm has the below steps: -

- Choose the first centroid at random from the data points.
- Calculate the distance between each data point and the closest, previously selected centroid.
- Choose the next centroid from the data points with a probability equal to its distance from the closest, previously chosen centroid.
- Steps 2 and 3 should be repeated until all k centroids have been sampled.

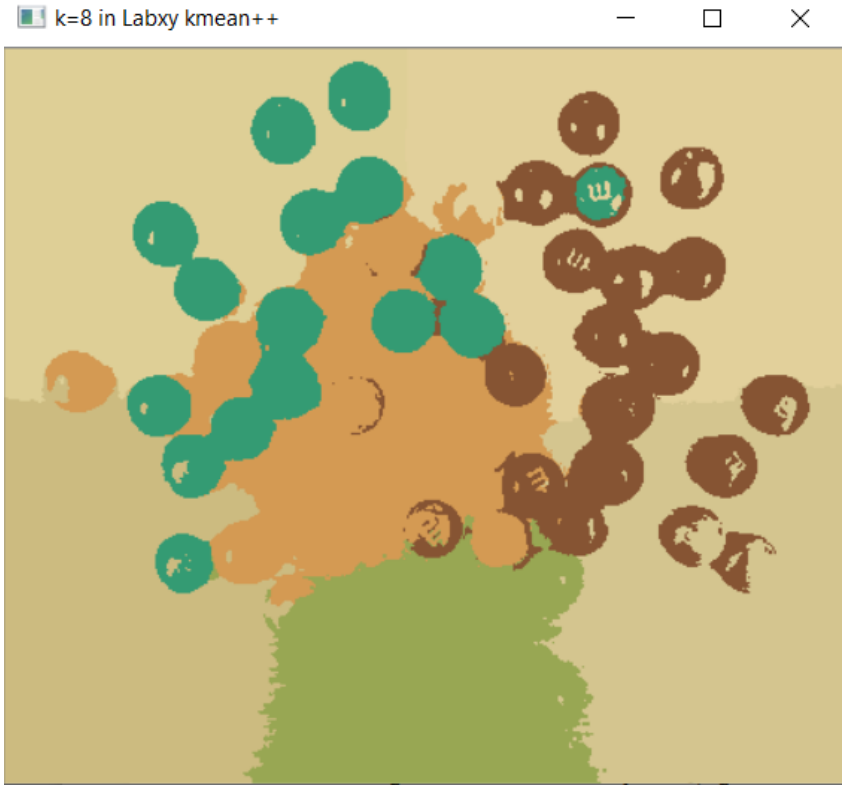
```python
# This function calculates euclidean distance.
def distance(p1, p2):
    return np.sum((p1 - p2)**2)
```

```python
# The procedure for initialization
def my_initialization(data, k):
    '''
    Centroid initializtion for K-means++
    inputs:
        data - a numpy list having shape of data points (177694, 5)
        k - the quantity of clusters
    '''
    ## initialize the centroids list and add
    ## a randomly selected data point to the list
    centroids = []
    centroids.append(data[np.random.randint(data.shape[0]), :])


    ## Calculating remaining k - 1 centroids
    for c_id in range(k - 1):

        ## initialize a list to store distances of data
        ## points from nearest centroid
        dist = []
        for i in range(data.shape[0]):
            point = data[i, :]
            d = sys.maxsize


            ## Calculate the distance between 'point' and each previously
            ## selected centroid and save the shortest distance.
            for j in range(len(centroids)):
                temp_dist = distance(point, centroids[j])
                d = min(d, temp_dist)
            dist.append(d)

        ## As our next centroid, choose the data point with the greatest distance.
        dist = np.array(dist)
        next_centroid = data[np.argmax(dist), :]
        centroids.append(next_centroid)
        dist = []
    return centroids
```

By kmean++

With x,y co-ordinates by kmean

By kmean++



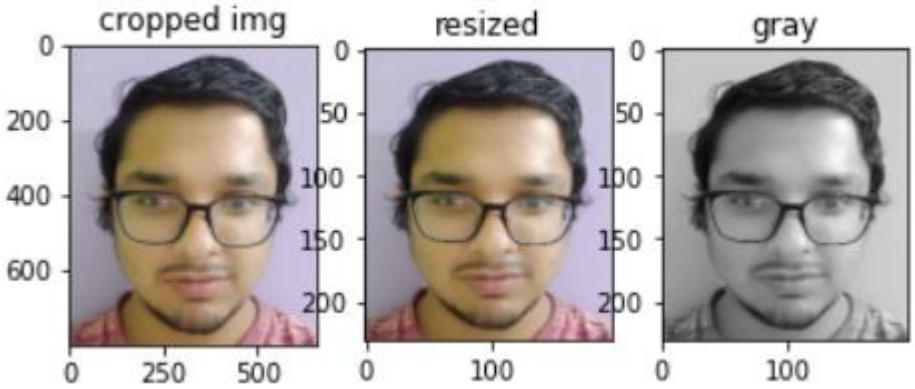
With x,y co-ordinates by kmean

It can be observed as the k value increases it is difficult to spot the difference between the two algorithms because of a greater number of clusters.

Task 3

Q1

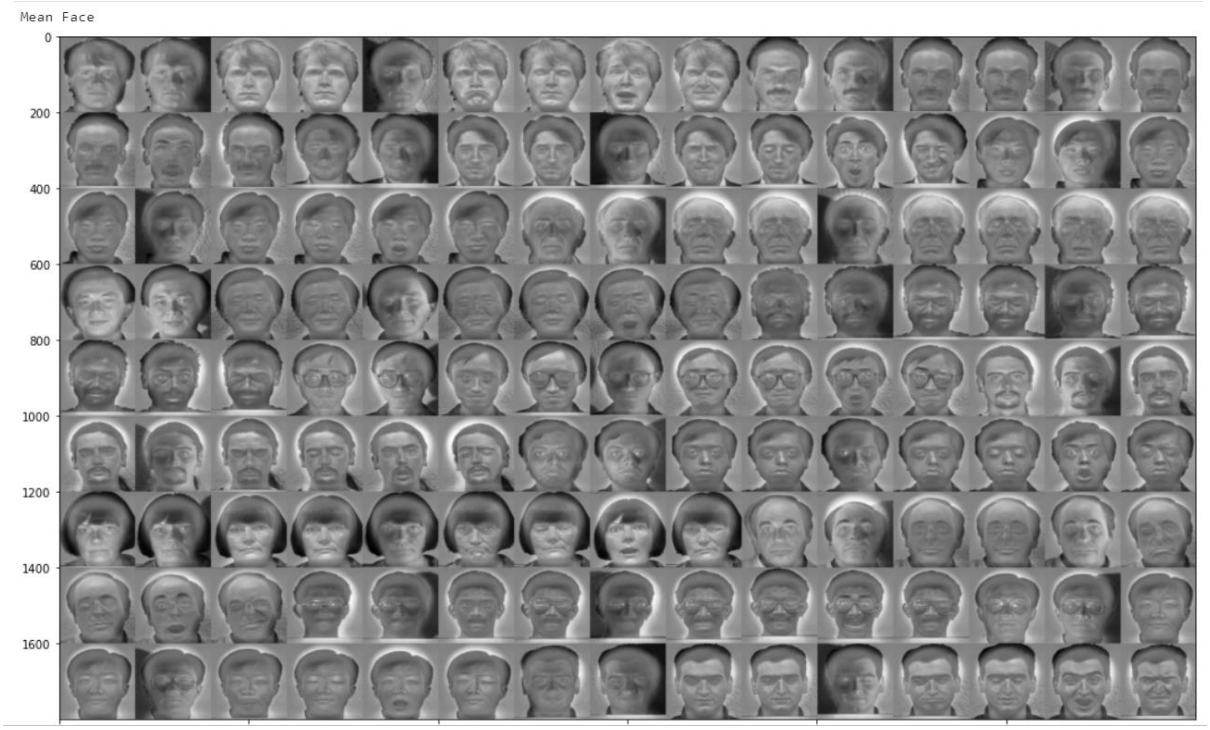
If images are properly aligned with respect to eyes, chin, mouth, then the computation time will decrease and we can identify the face with less error percentage, since the mean of images will be better, and will highly affect PCA and eigen face calculation.



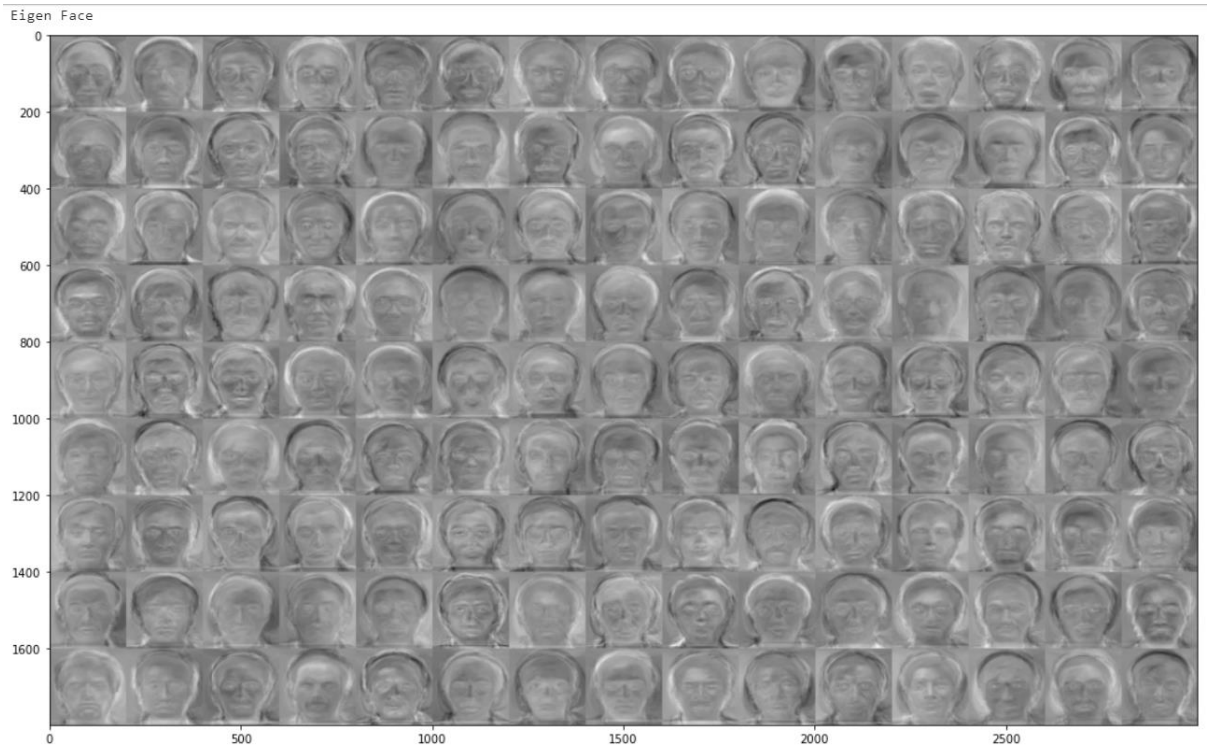
Q2

(1)

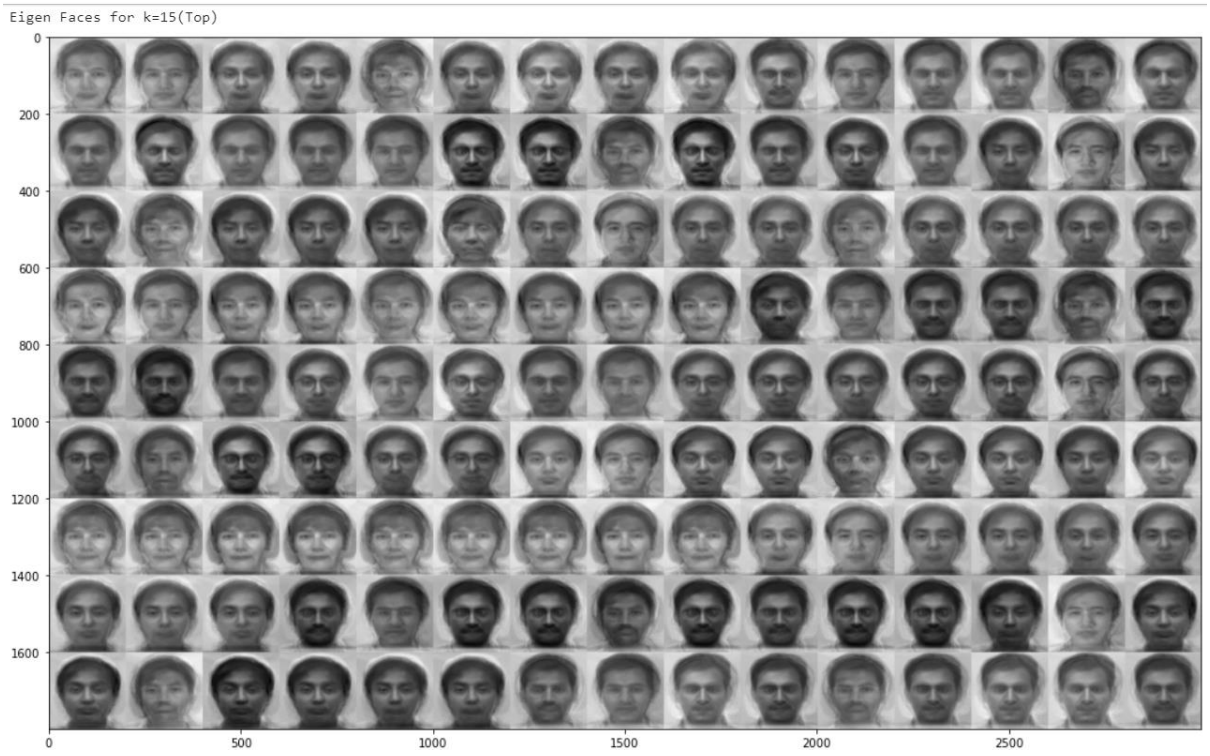
(2)



The covariance matrix of the probability distribution over the high-dimensional vector space of face images is used for faster calculation of eigenvectors. Eigen Face is just a set of eigen vectors.

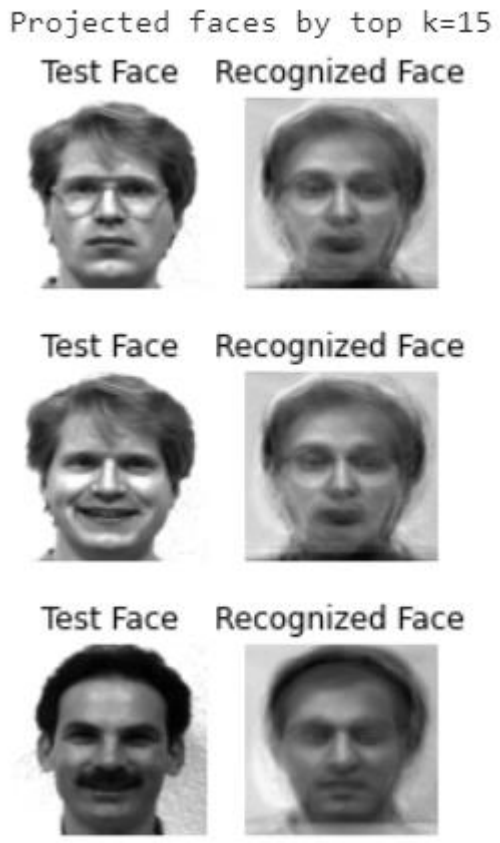


(3)



(4)

I failed to identify top 3 face from the dataset but the reconized face is the projected face for top 15 eigen faces.



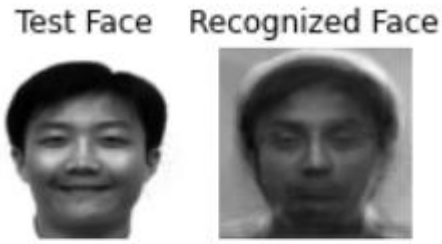
Test Face Recognized Face

Test Face Recognized Face

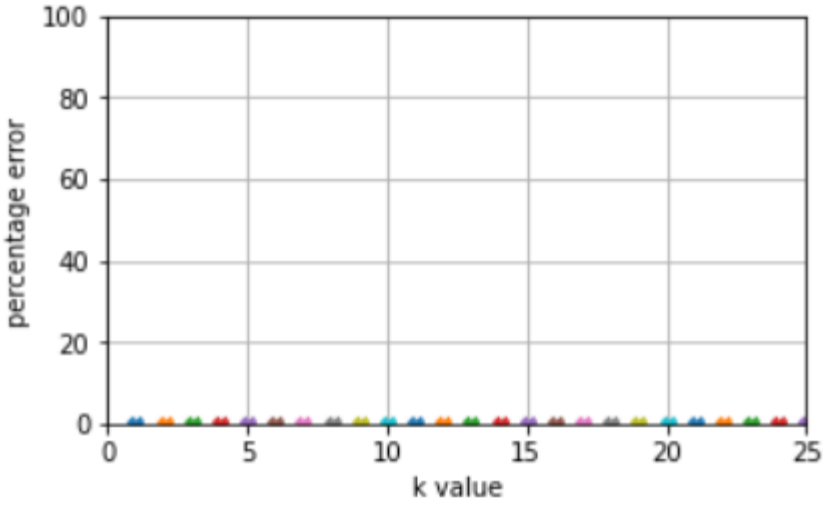
Test Face Recognized Face

Test Face Recognized Face

Test Face Recognized Face

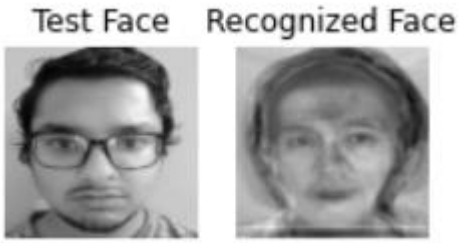


The percentage error here determines how many images that did not have face were recognized by the algorithm. The rate is zero since there was not a single image without a face in the test set.



(5)

I failed to identify top 3 face from the dataset but the recognized face is the projected face for top 15 eigen faces.



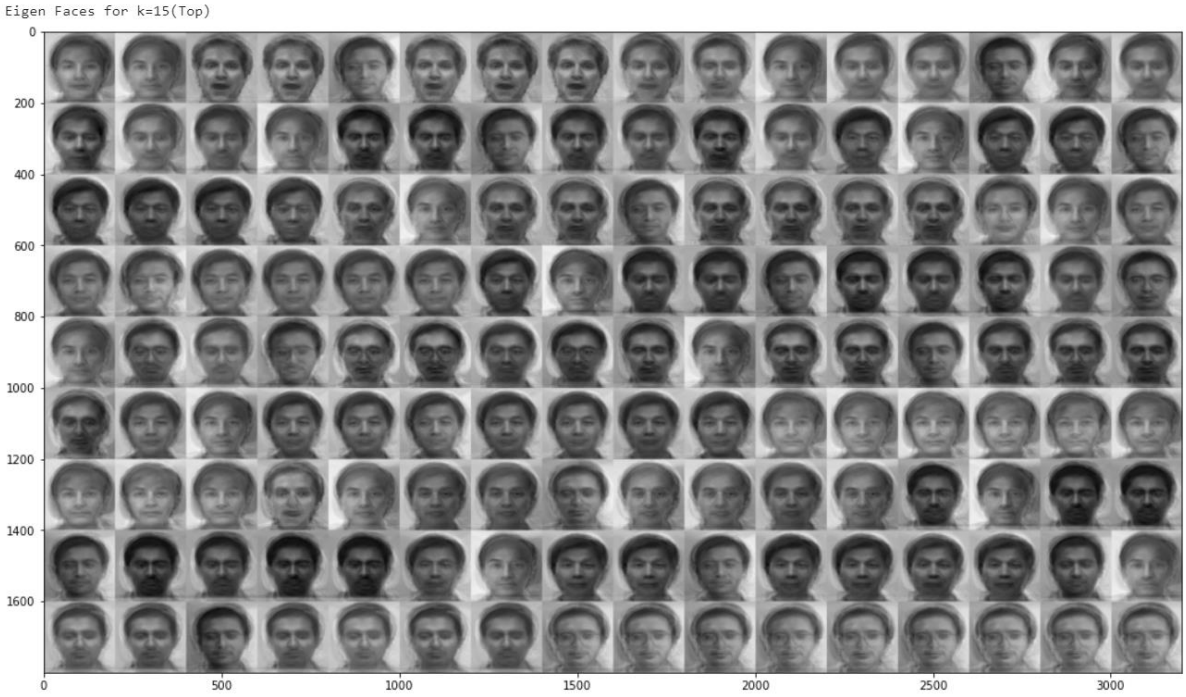
(6)

Total images = 144

```
print(len(image))
```

```
144
```

Below image shows eigen faces for k = 15.



I failed to identify top 3 face from the dataset but the reconized face is the projected face for top 15 eigen faces.

Summary of work performed for task 3

The images were resized to 200*200 before training. A total of 135 images and later 144 images were trained. The Eigenvectors of the covariance matrix of the training images were obtained using Principal Component Analysis (PCA). By choosing the top k = 15 Eigenfaces, the training faces were reconstructed (The eigenvectors that correspond to the largest eigenvalues are called eigenvectors). Then the test of images was performed. The closest training image was found by using Eucledian distance to recognise the face images. Error rate was also analysed using graph.