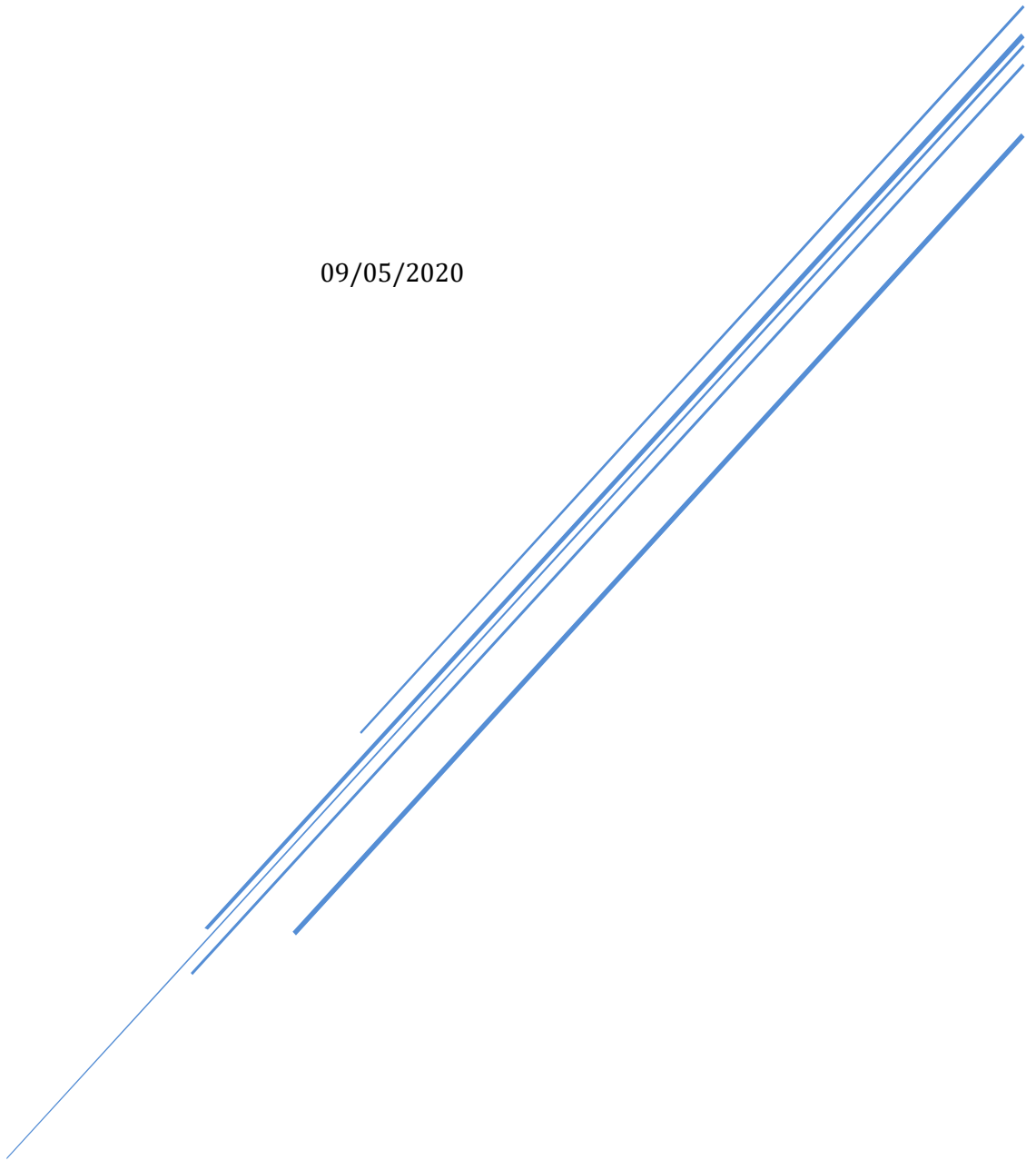


# COMPUTER VISION CLAB2 REPORT

ENGN6528

09/05/2020



u7043565  
Tanya Dixit

## CONTENTS

---

Task – 1: Harris corner detector .....	2
Task 2 – K-Means clustering and color image segmentation.....	11
Task 3 – Face recognition using eigenface .....	18

# TASK – 1: HARRIS CORNER DETECTOR

---

## Documentation

Following is the source code for Harris Corner Detection. The new code is highlighted with a black border and the rest of the code is included for completeness. New variables 'k' and 'window' are defined. k is used for calculating cornerness and 'window' is the size of the window to calculate max over while using non-max suppression to find final corner points.

```
: import numpy as np
import cv2
import matplotlib.pyplot as plt
```

```
filename = 'Harris_3.jpg'
```

```
def conv2(img, conv_filter):
    # flip the filter
    f_siz_1, f_size_2 = conv_filter.shape
    conv_filter = conv_filter[range(f_siz_1 - 1, -1, -1), :][:, range(f_siz_1 - 1, -1, -1)]
    pad = (conv_filter.shape[0] - 1) // 2
    result = np.zeros((img.shape))
    print(img.shape)
    img = np.pad(img, ((pad, pad), (pad, pad)), 'constant', constant_values=(0, 0))
    filter_size = conv_filter.shape[0]
    for r in np.arange(img.shape[0] - filter_size + 1):
        for c in np.arange(img.shape[1] - filter_size + 1):
            curr_region = img[r:r + filter_size, c:c + filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result) # Summing the result of multiplication.
            result[r, c] = conv_sum # Saving the summation in the convolution layer feature map.

    return result
```

```
def fspecial(shape=(3, 3), sigma=0.5):
    m, n = [(ss - 1.) / 2. for ss in shape]
    y, x = np.ogrid[-m:m + 1, -n:n + 1]
    h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))
    h[h < np.finfo(h.dtype).eps * h.max()] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h
```

```

# Parameters, add more if needed
sigma = 2
thresh = 789213.28#1000
k = 0.05
window = 3

# Derivative masks
dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
dy = dx.transpose()
import matplotlib.pyplot as plt

bw = plt.imread(filename)
bw = cv2.cvtColor(bw, cv2.COLOR_BGR2GRAY)
bw = np.array(bw * 255, dtype=int)
# compute x and y derivatives of image
print(bw.shape)
print(dx.shape)
Ix = conv2(bw, dx)
Iy = conv2(bw, dy)

```

```

#The below line creates a gaussian blur filter in a way that it emphasizes the gradient when
#convolved. Basically we want to pass a window (g) onto Ix^2, Iy^2 and Ix*Iy
#so that we get the directions of maximal gradient in the neighbourhood of the point (as it is done)
#for every point
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)
Ix2 = conv2(np.power(Ix, 2), g)
Ixy = conv2(Ix * Iy, g)

```

```

#####
# Task: Compute the Harris Cornerness
#####

det = Ix2*Iy2 - Ixy ** 2 #calculate the determinant of the M matrix
trace = Ix2 + Iy2 #calculate trace of the M matrix

R = det - k * trace**2 #cornerness R for each pixel is det - k*trace^2
#This tells us whether a particular pixel is a corner or not

```

```
#####
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
#####

#Now we have cornerness for each pixel, we apply non-max suppression to get the corners
output = set() #save corner points in a set so that repeating points are not marked twice in the image
for i in range(R.shape[0] - window): #iterate over each pixel in R
    for j in range(R.shape[1] - window):
        mat = R[i:i+window,j:j+window] #take a window of size window
        if np.max(mat) > thresh: #if the max in that window (mat) is greater than threshold
            #print(np.max(mat))
            (x, y) = np.unravel_index(np.argmax(mat, axis=None), mat.shape) #find out which x,y have the max value
            #print(mat.shape)
            #x, y = np.argmax(mat)
            output.add((x+i, y+j)) #save that x,y in the list of corners
        else:
            continue
```

The rest is display code and can be referred from Task1.py

## Observations and Results

### Img 1 (Harris\_1.jpg)

For sigma=2, thresh=789213.28, k=0.05, window=3 (Please refer to the code for the hyperparameters)

In Fig 1.1, we display the result of custom Harris corner detector written in Task1.py

Right below it, we display the result of cv2.cornerHarris in Fig 1.2 for comparison.

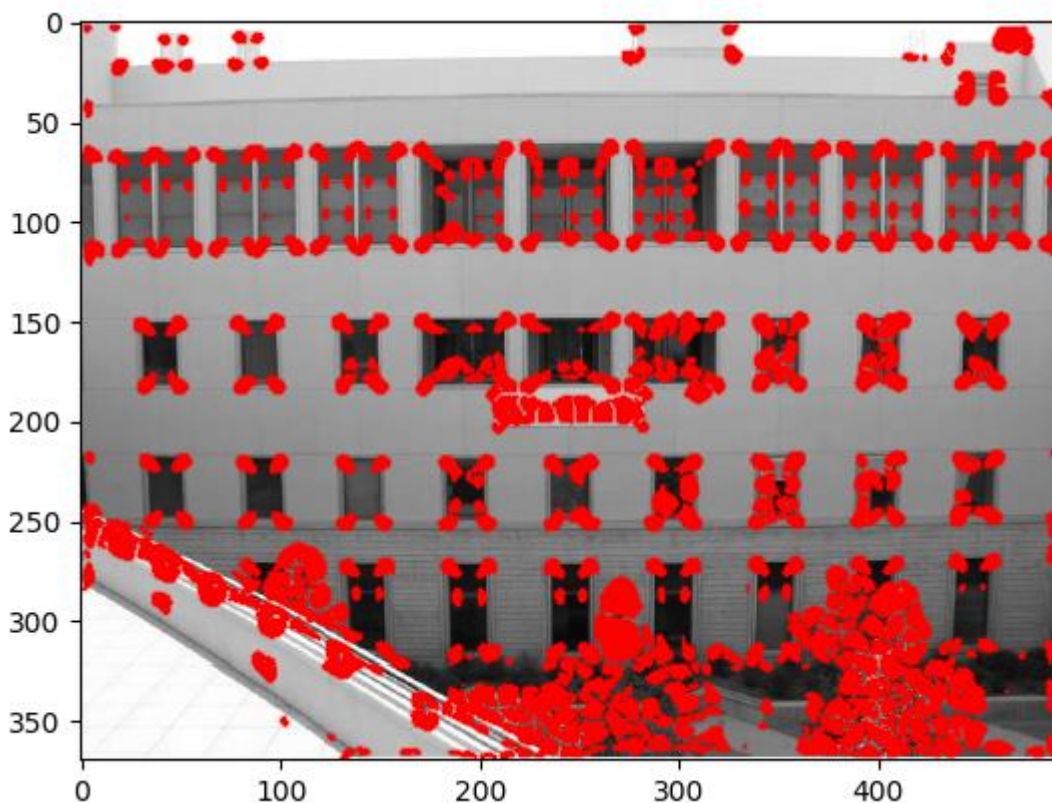


FIGURE 1.1 HARRIS CORNER DETECTOR RESULT ON IMG 1 USING TASK1.PY (OUR CODE)



FIGURE 1.2 HARRIS CORNER DETECTOR RESULT USING CV2.CORNERHARRIS ON IMG 1

For  $\sigma=2$ ,  $\text{thresh}=789213.28$ ,  $k=0.1$ ,  $\text{window}=5$

In Fig 1.3 and 1.4 we display the result of custom and cv2 library Harris corner detector respectively. We change the hyperparameters as per the heading and observe the results.

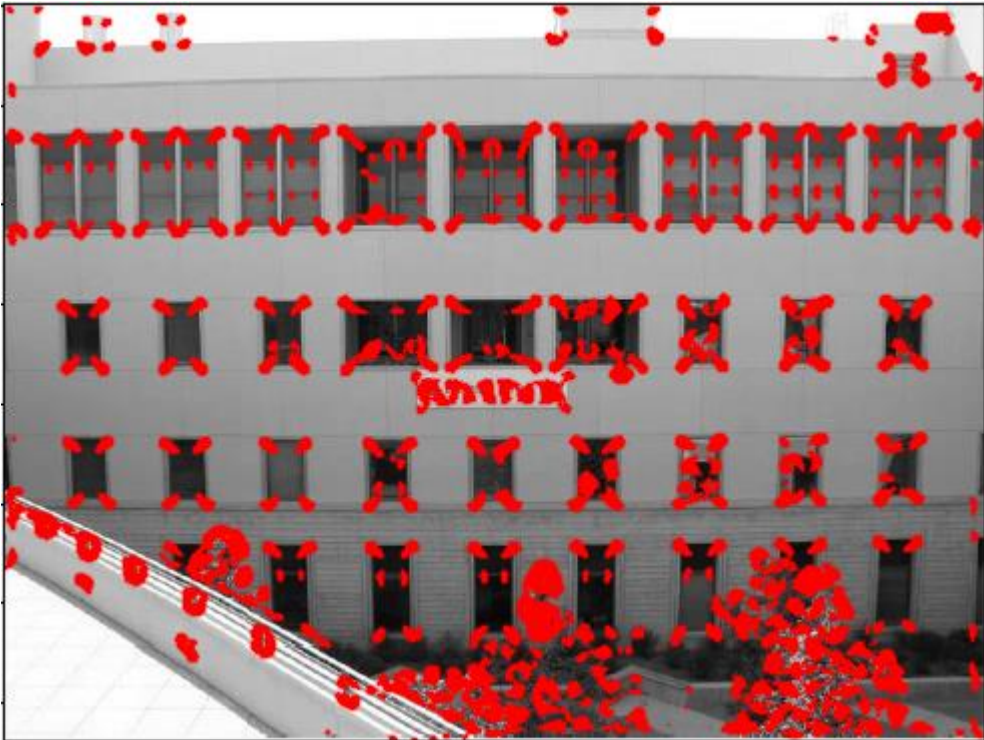


FIGURE 1.3 RESULT OF CUSTOM HARRIS CORNER DETECTOR WITH  $K=0.1$  AND WINDOW SIZE=5





FIGURE 1.4 RESULT OF CV2 HARRIS CORNER DETECTOR WITH  $K=0.1$  AND WINDOW SIZE=5

For  $\sigma=2$ ,  $\text{thresh}=1000$ ,  $k=0.05$ ,  $\text{window}=3$

In Fig 1.5 and 1.6 we display the result of custom and cv2 library Harris corner detector respectively. We change the hyperparameters as per the above heading and observe the results.

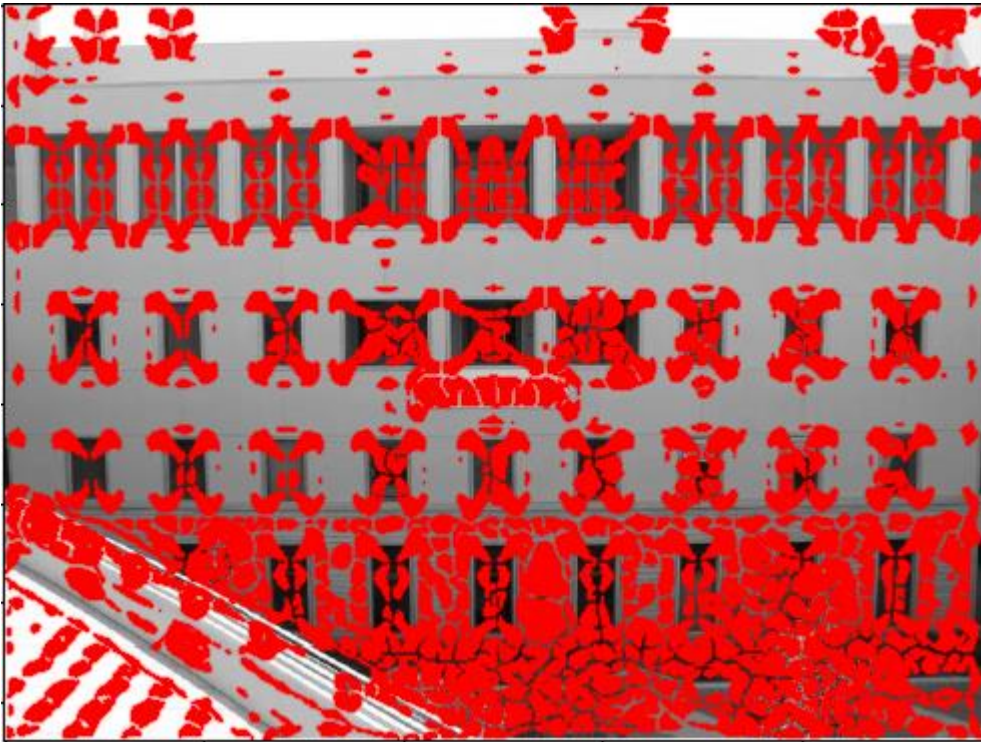


FIGURE 1.5 RESULT OF CUSTOM HARRIS CORNER DETECTOR WITH VERY LOW THRESHOLD (1000)

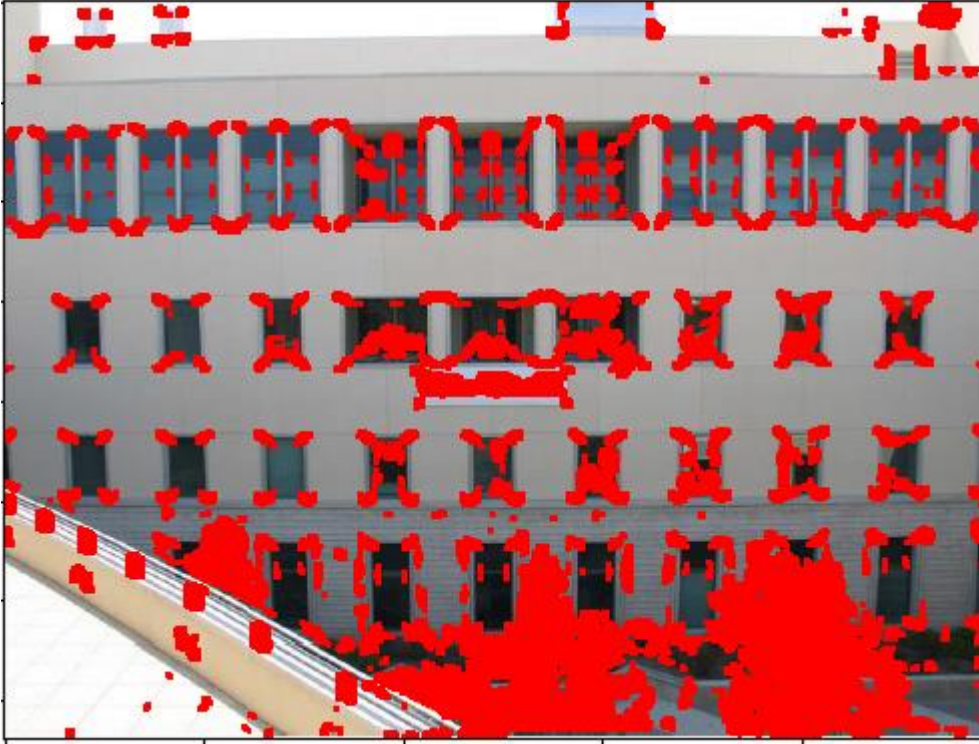


FIGURE 1.6 RESULT OF CV2 HARRIS CORNER DETECTOR WITH VERY LOW THRESHOLD (1000)

**Img 2 (Harris\_3.jpg)**

From this point on, we use default parameters ( $\sigma=2$ ,  $\text{thresh}=789213.28$ ,  $k=0.05$ ,  $\text{window}=3$ )

**Observations**

In Fig 1.7 and 1.8 we display the result of custom and cv2 library Harris corner detector respectively on Harris\_3.jpg.



FIGURE 1.7 RESULT OF CUSTOM HARRIS CORNER DETECTOR ON HARRIS\_3.JPG





FIGURE 1.8 RESULT OF CV2 HARRIS CORNER DETECTOR ON HARRIS\_3.JPG

**Img 3 (Harris\_4.jpg)**

**Observations**

In Fig 1.9 and 1.10 we display the result of custom and cv2 library Harris corner detector respectively on Harris\_4.png.

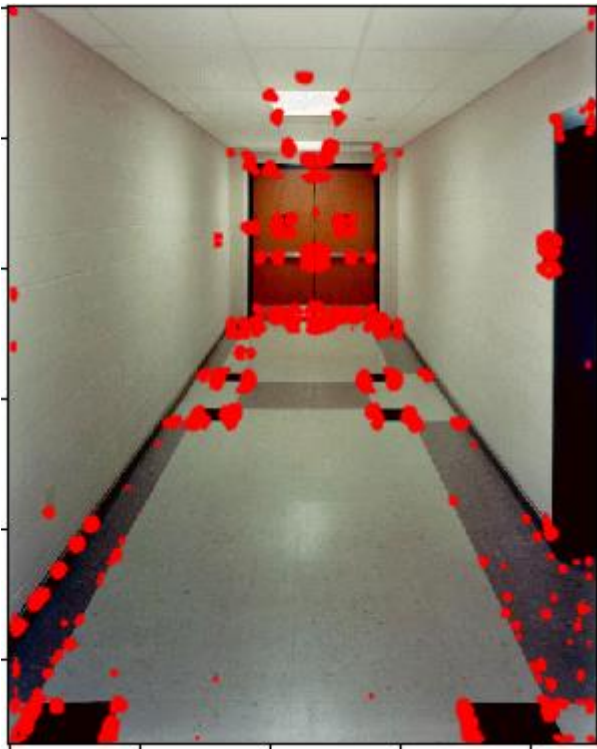


FIGURE 1.9 RESULT OF CUSTOM HARRIS CORNER DETECTOR ON HARRIS\_4.JPG



FIGURE 1.10 RESULT OF CV2 HARRIS CORNER DETECTOR ON HARRIS\_4.JPG

#### **Img 4 (Harris\_1.pgm)**

##### **Observations**

In Fig 1.11 and 1.12 we display the result of custom and cv2 library Harris corner detector respectively on Harris\_1.pgm.

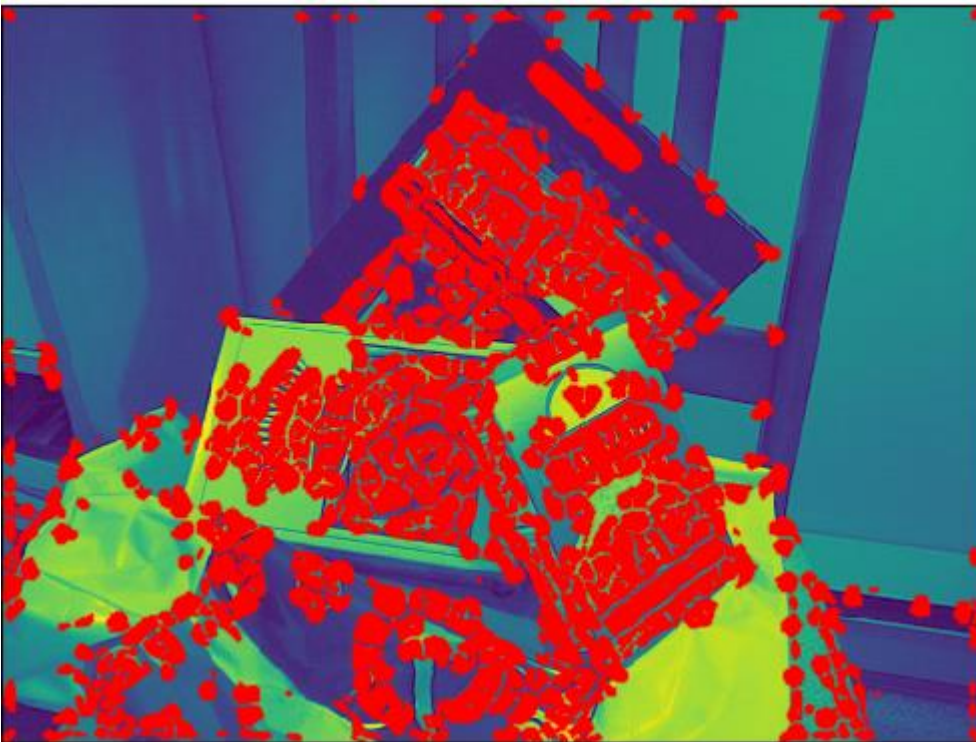


FIGURE 1.11 RESULT OF CUSTOM HARRIS CORNER DETECTOR ON HARRIS\_1.PGM



FIGURE 1.12 RESULT OF CV2 HARRIS CORNER DETECTOR ON HARRIS\_1.PGM

### **Analysis**

We observe that the parameters like threshold,  $k$  and window size play a role in how many image pixels will be classified as corner points. If the threshold is too low, a lot of points, even edges may be classified as corners which we don't want. Also, we observe that our algorithm is detecting more points as corners compared to the built-in function. This could be due to differences in implementation (dilation or not), or possibly due to the display differences (we might be displaying the markers differently, or marker size might be different). In all, we see that our algorithm mostly correctly classifies corners and doesn't give too many false positives. The only problem is areas like bush, trees etc, where well-defined corners are not present. In fact, sometimes our custom algorithm performs better in terms of detecting more corners that the cv2 function doesn't classify as corners.

Also, the window size matters. If the window size is too low, we generally get more corner points as seen from Fig. 1.1 vs Fig. 1.3.

## TASK 2 – K-MEANS CLUSTERING AND COLOR IMAGE SEGMENTATION

---

### 2.1 Code snippets with comments

#### Documentation

Please refer to the code snippets for the complete K Means implementation. Here, not all display code is added to avoid unnecessary clutter. Please refer to Task2.py for the complete code.

```
num_features=3 #5 #Do we want to take only L, a, b or L, a, b, x, and y
data_matrix = np.zeros((im.shape[0]*im.shape[1], num_features))
t=0
#create the data matrix by filling in L, a, b, x, and y values. It creates a matrix with rows=widthxheight
#and columns=5 or columns=3 matrix
for u in range(im.shape[0]):
    for v in range(im.shape[1]):
        data_matrix[t][0] = im[u][v][0]
        data_matrix[t][1] = im[u][v][1]
        data_matrix[t][2] = im[u][v][2]
        #data_matrix[t][3] = u
        #data_matrix[t][4] = v
        t+=1
```

```
#init centroids randomly from points in the data matrix
```

```
def init_centroids(data, k):
    c = np.zeros((k, data.shape[1]))
    indices = random.sample(range(0, data.shape[0]), k)
    c = data[indices]
    return c
```

```
def e_step(c, data):
    belong_vectors = np.zeros((data.shape)) #initialize a matrix that tells us which point belongs to which cluster
    for i in range(data.shape[0]):           #iterate over all the points
        min_dist = 999999999999              #initialize unusually high min distance for each point/iteration
        for j in range(c.shape[0]):          #iterate over all the centroids
            if min_dist > np.linalg.norm(data[i]-c[j]):
                belong_vectors[i] = c[j]     #now fill this matrix with the clutser closest to each point
                min_dist = np.linalg.norm(data[i]-c[j]) #update min distance for that point

    #print(belong_vectors)
    return belong_vectors
```

```
def m_step(belong_vectors, data, c):
    new_c = np.zeros(c.shape)               #In order to update centroids in m step, we need an empty matrix
    for i in range(c.shape[0]):              #iterate over centroids
        sum_points = np.zeros((1, c.shape[1]))
        count_points = 0
        for j in range(data.shape[0]):       #iterate over all points in data matrix
            if (belong_vectors[j] == c[i]).all(): #if the data point belongs to the cluster
                sum_points+=data[j]           #add it to the sum
                #print(sum_points.shape)
                count_points+=1               #for keeping count of points in a cluster

        new_c[i] = sum_points/count_points    #take the average of all points in that cluster
                                              #and assign result to the centroid (that's the definition of a centroid)
    return new_c                             #return new centroids
```



```
def my_kmeans(data, num_centroids, iterations): #finally the kmeans algorithm
    c = init_centroids(data, num_centroids) #init centroids
    last_c = c #assign the last iteration centroids to c for now

    for i in range(iterations):
        belong_vectors = e_step(c, data) #E step
        c = m_step(belong_vectors, data, c) #M step
        if (c == last_c).all(): #if new centroids are equal to last iteration centroids, break the loop
            break
        last_c = c #reassign

    return belong_vectors, c
```

```
b, c = my_kmeans(data_matrix, 20, 30)
#b contains the centroid information for all the points
```

```
final_image = np.zeros((im.shape[0], im.shape[1], 3))
t=0
#reconstruct the segmented image in a way that each point gets assigned to its cluster
for u in range(im.shape[0]):
    for v in range(im.shape[1]):
        final_image[u][v][0] = b[t][0] #we use t because our b vector was flattened in the same way
        final_image[u][v][1] = b[t][1] #and now we fill our image using the centroid values for each point
        final_image[u][v][2] = b[t][2]
        t+=1
```

```
final_image = np.uint8(final_image)
```

## **2.2 K Means using different clusters and with/without pixel coordinates**

### **Documentation**

In this part, K-means was applied to the two images given and segmentation results were compared for different K, and for with and without pixel coordinates. For changing K, simply a different K was passed to the function “my\_kmeans”

To add or remove pixel coordinates, the data dimension was changed from 5 to 3 (containing on L, a, b in case of 3). These experiments were performed in a Jupyter Notebook, and the relevant code is present in Task2.py with elaborate comments.

### **Observations and Results**

Here are the observations and results of various experiments conducted using the algorithm Kmeans.

#### **Original Images**

In Fig. 2.1, the original images are displayed (for completeness) on which we run this algorithm.



FIGURE 2.1 ORIGINAL IMAGES FOR INPUT TO SEGMENTATION ALGORITHM

(1) Using L, a, and b: The images in Fig 2.2, 2.3 and 2.4 are segmented results using only L, a, and b and **x, y are not considered.**

### Using K=3

In Fig. 2.2, we can see the segmentation results using K=3 clusters on the two original images and use 3-D representations of the images as input.

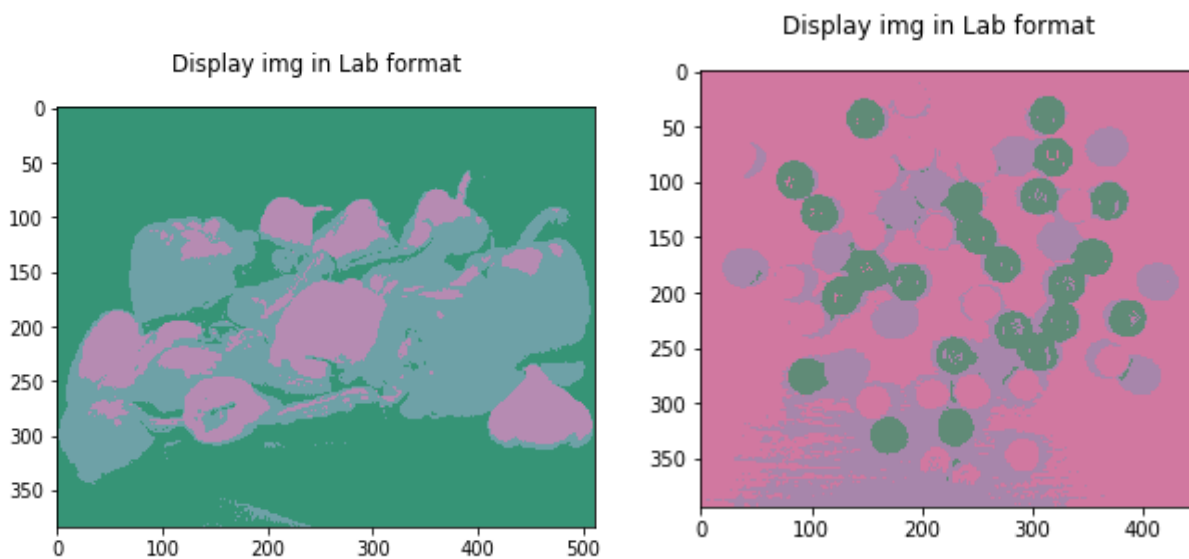


FIGURE 2.2 SEGMENTATION RESULTS OF KMEANS USING K=3 AND USING 3-D REPRESENTATIONS

### K=7

In Fig. 2.3, we display the segmentation results using K=7 clusters on the two original images and use 3-D representations of the images as input.

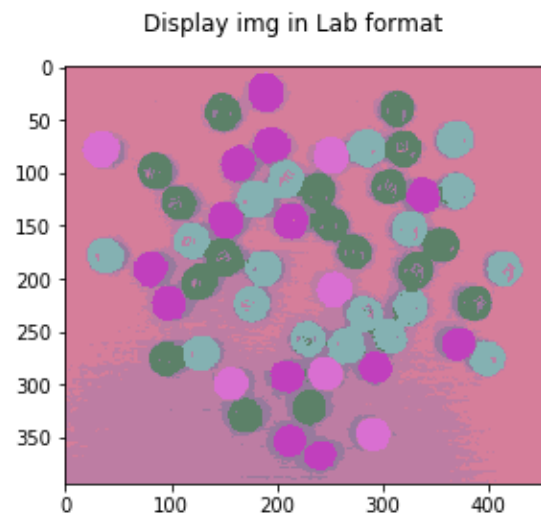
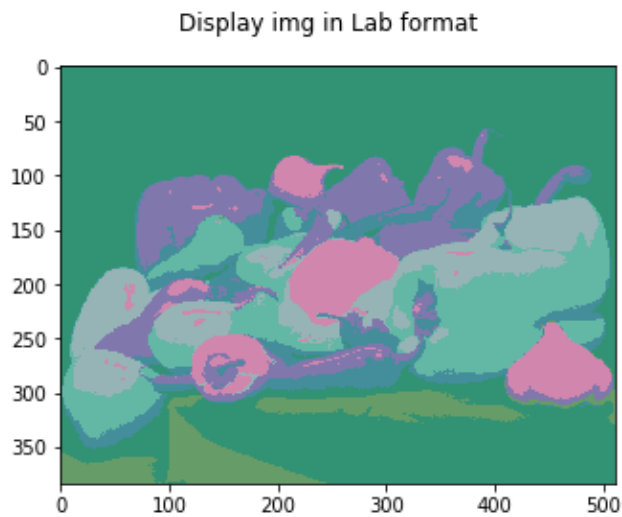


FIGURE 2.3 SEGMENTATION RESULTS OF KMEANS USING K=12 AND USING 3-D REPRESENTATIONS

### K=20

In Fig. 2.4, we display the segmentation results using K=20 clusters on the two original images and use 3-D representations of the images as input.

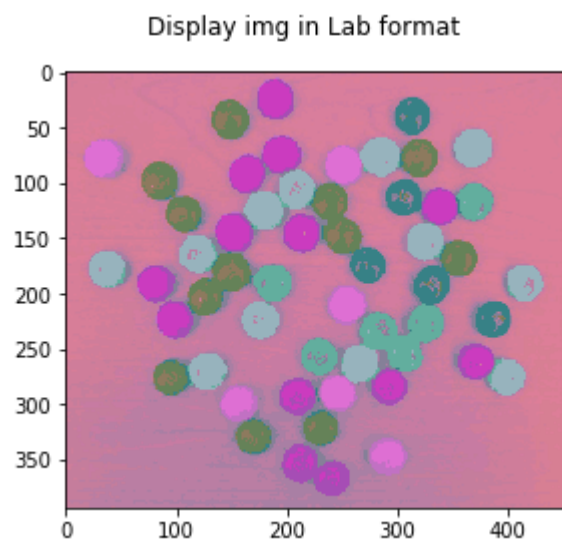
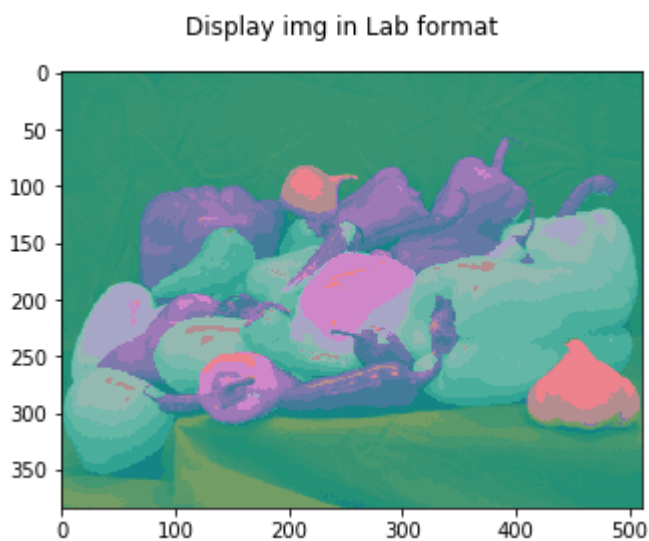


FIGURE 2.4 SEGMENTATION RESULTS OF KMEANS USING K=20 AND USING 3-D REPRESENTATIONS

### (2) Using L, a, b, x, and y:

### K=12

In Fig. 2.5, we display the segmentation results using K=12 clusters on the two original images and use 5-D representations of the images as input

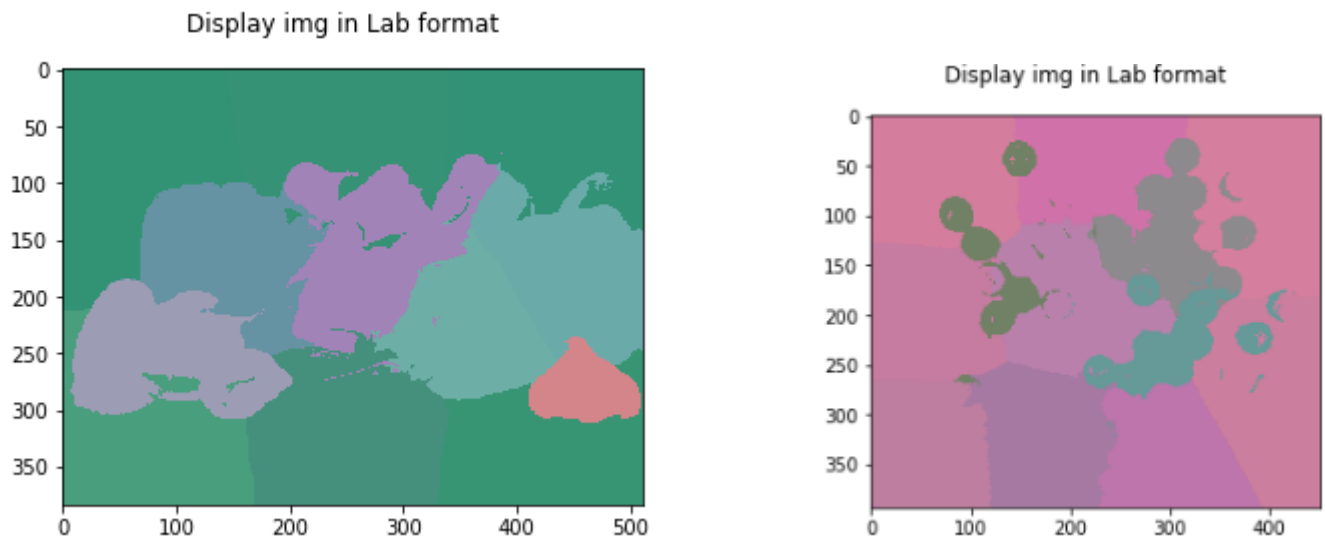


FIGURE 2.5 SEGMENTATION RESULTS OF KMEANS USING K=12 AND USING 5-D REPRESENTATIONS

### K=20

In Fig. 2.6, we display the segmentation results using K=20 clusters on the two original images and use 5-D representations of the images as input

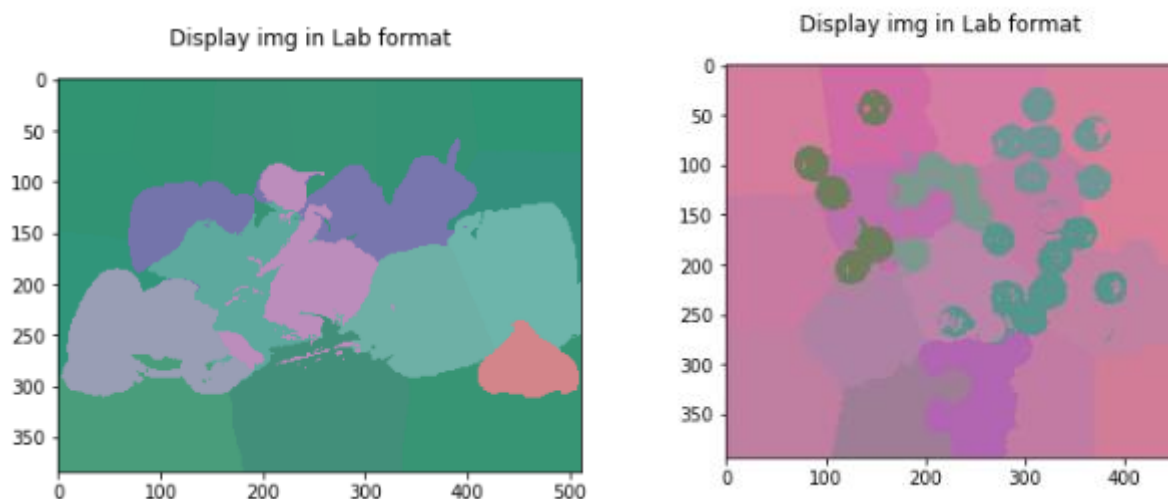


FIGURE 2.5 SEGMENTATION RESULTS OF KMEANS USING K=20 AND USING 5-D REPRESENTATIONS

### **Analysis**

From the observations in the above figures, we see that using 5-D representations have a detrimental impact on the segmentation output, whereas 3-D representations give highly accurate and visually verifiable output. This could be due to the fact that it's not necessary that nearby items belong to the same cluster because in an image, the x,y coordinates just tells the position whereas humans generally perform segmentation based on colours and visual similarity. Also, when we increase the cluster number, x,y give better performance than when cluster numbers are low.



## 2.3 Kmeans ++

### Key Steps:

1. Initialize the first centroid randomly
2. For the remaining k-1 centroids, follow this
  - 2.1 initialize a list of distances to store distances from nearest centroid
  - 2.2 for every point
    - 2.2.1 compute distance of this point from the previous centroids
    - 2.2.2 store the minimum distance in the list of distances
  - 2.3 Select the next centroid as the data point with maximum distance from its nearest centroid
  - 2.4 Repeat

The code snippet is displayed here for reference, but the complete code is in Task2.py. Please refer to the python file for the complete code and comments.

```
#distance square for Kmeans plus plus
def distance(p1, p2):
    return np.sum((p1 - p2)**2)

def init_centroids_plus_plus(data, k):

    #centroid list, we keep adding to it as we find more centroids till we reach k
    c = []
    #add the first centroid randomly from the data
    c.append(data[np.random.randint(data.shape[0]), :])
    print(c)

    #Loop over rest of the k and add centroids one by one
    for c_id in range(0,k-1):
        dist = []
        for i in range(data.shape[0]): #for each data point, we find distance to closest centroid
            point = data[i, :]
            d = sys.maxsize
            for j in range(len(c)):
                temp_dist = distance(point, c[j]) #only take centroids already in c
                d = min(d, temp_dist) #take min distance as we want closest centroid
            dist.append(d)
        dist = np.array(dist)
        next_c = data[np.argmax(dist), :] #find the point which has highest distance from its closest centroid
        c.append(next_c)
        dist = []

    return c
```

## Observations and Results

Using 5-D representation, the results with Kmeans++ are:

### K=12

In Fig. 2.7, we display the Kmeans++ segmentation results using K=12 clusters on the two original images and use 5-D representations of the images as input.

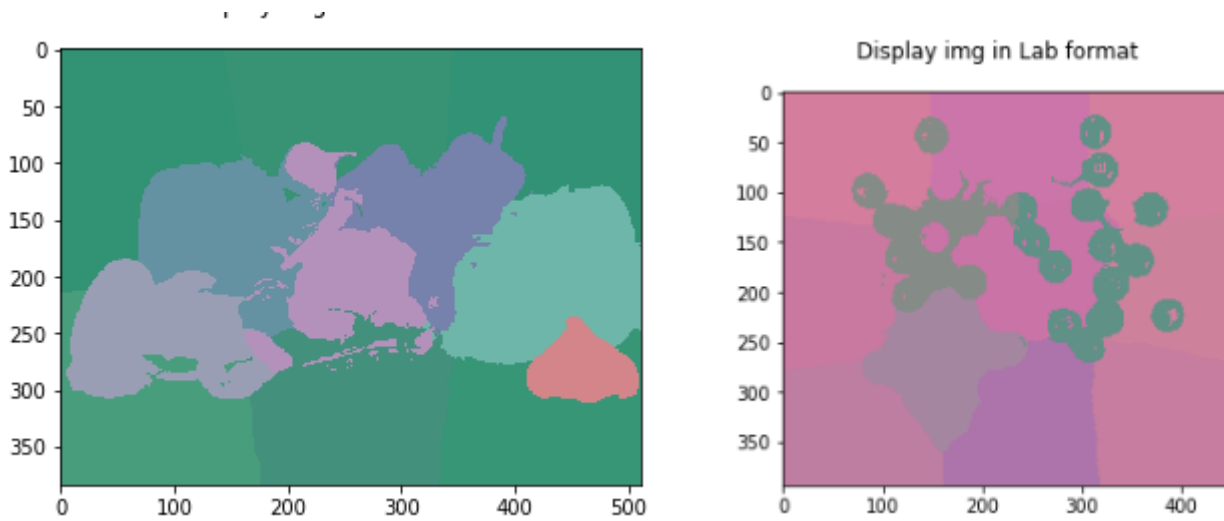


FIGURE 2.6 SEGMENTATION RESULTS OF KMEANS++ USING K=12 AND USING 5-D REPRESENTATIONS

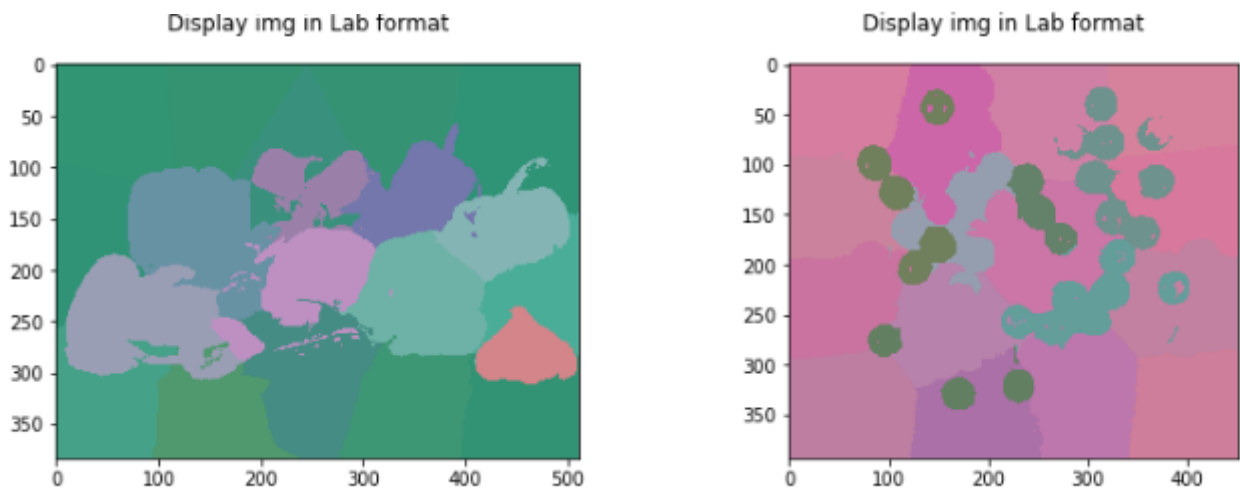


FIGURE 2.7 SEGMENTATION RESULTS OF KMEANS++ USING K=20 AND USING 5-D REPRESENTATIONS

### Analysis

We observe from Fig 2.6 and 2.7 contrasted with Fig 2.4 and 2.5 respectively that with the same number of clusters, Kmeans++ performs better than normal Kmeans. It also converges faster and gives better results in less iterations as observed from running the code. The reason is that it forces the centroids to be farther apart and that's why overcomes the problems of regular Kmeans where the initialization is random and some clusters might end up having 0 points.

## TASK 3 – FACE RECOGNITION USING EIGENFACE

---

3.1 Alignment is necessary for eigenfaces because we calculate mean and mean is dependent on alignment. If images are misaligned, the mean and eigenfaces will be distorted and when we reconstruct images, they will have aberrations. Also, if they are misaligned, the face recognition won't work properly as the extracted features won't be reliable and subject to changes in the training set (aligned or not).

### 3.2 PCA and Mean face

#### Documentation

To perform PCA, the mean of the data matrix was calculated and subtracted from the data. Then the data covariance matrix was calculated. But before doing that, the matrix was transposed. The reason is that the eigenvalues of a matrix and its transpose are equal, that's why we do the eigen decomposition of  $A.T@A$  as can be referred from Task3.py and from the below code snippet.

```
def eigen_faces(A_pp):
    A, Q_norms, A_means = preprocess(A_pp) #returns centered data and mu

    w,v = np.linalg.eigh(A.T@A)             #eigendecomposition of A.T@A
    F = A@v                                 #calculate the eigenvectors of the original data cov matrix (A@A.T)

    F = F/np.linalg.norm(F, axis = 0)      #normalize the eigenvectors, these are our eigenfaces
    C = A.T@F                              #coefficients of each image in terms of eigenvector basis

    D = np.diag(w, k=0)                    #eigenvalues

    return C, F, D, Q_norms, A_means

coeff, F, D, Q_norms, A_means = eigen_faces(all_data)
```

FIGURE 3.1 CODE SNIPPET FOR PERFORMING EIGENDECOMPOSITION AND RETURNING EIGENVECTORS AND MEAN

Also, the eigenvectors of the original data matrix can be calculated by taking a projection of the data  $A$  onto the eigenvectors of  $A^T A$  as mentioned in the code snippet above.

## Observations and Results

In Fig 3.2, we display the mean vector of all the data matrix (all images)

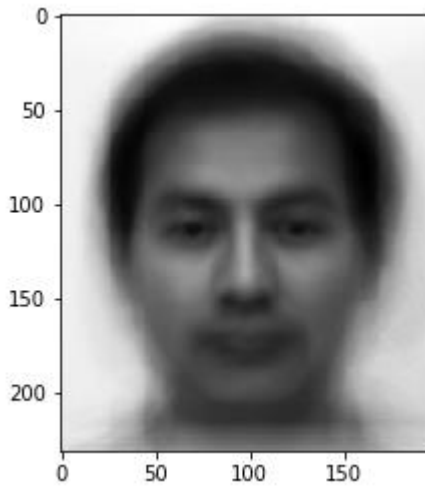
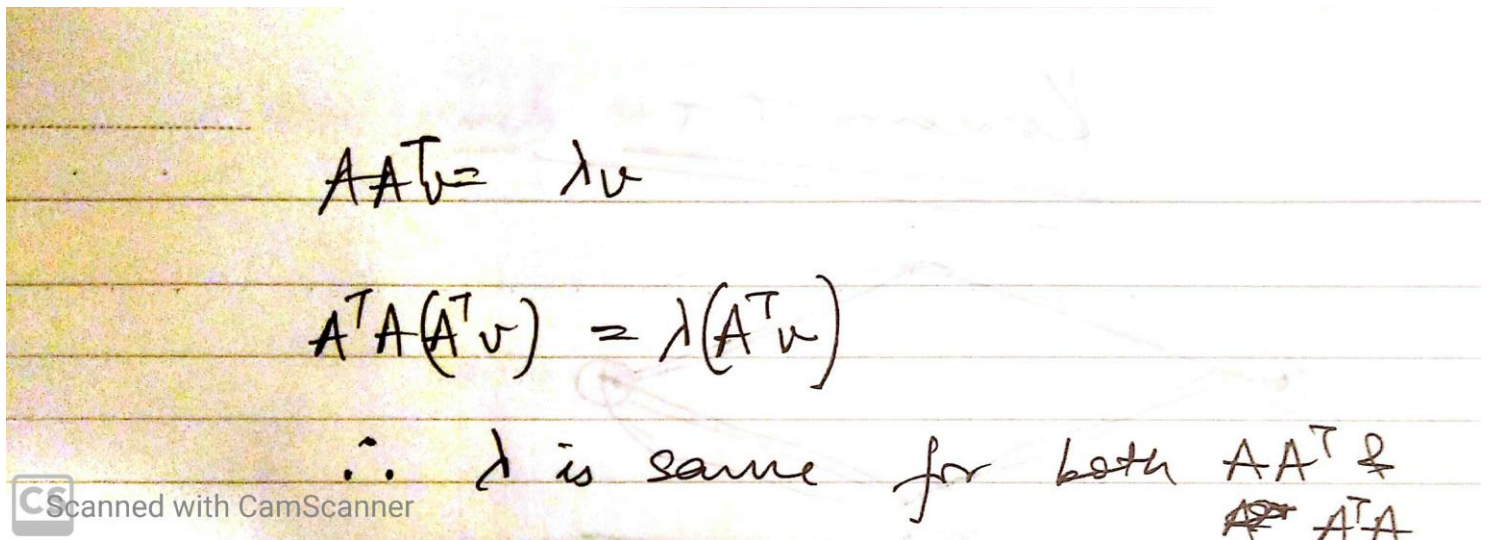


FIGURE 3.2 MEAN VECTOR OF ALL THE IMAGES IN YALE TRAINING SET

## Analysis

We observe that the mean image looks like what it's called – an average of all the faces in the image. Also, the reason we are able to find the eigenvectors without eigen-decomposition of whole matrix of  $45045 \times 45045$  is in Fig 3.3 and 3.4.


$$A A^T v = \lambda v$$
$$A^T A (A^T v) = \lambda (A^T v)$$

$\therefore \lambda$  is same for both  $A A^T$  &  $A^T A$

FIGURE 3.3 EXPLANATION WHY EIGENVALUES OF  $A^T A$  IS SAME AS  $A A^T$



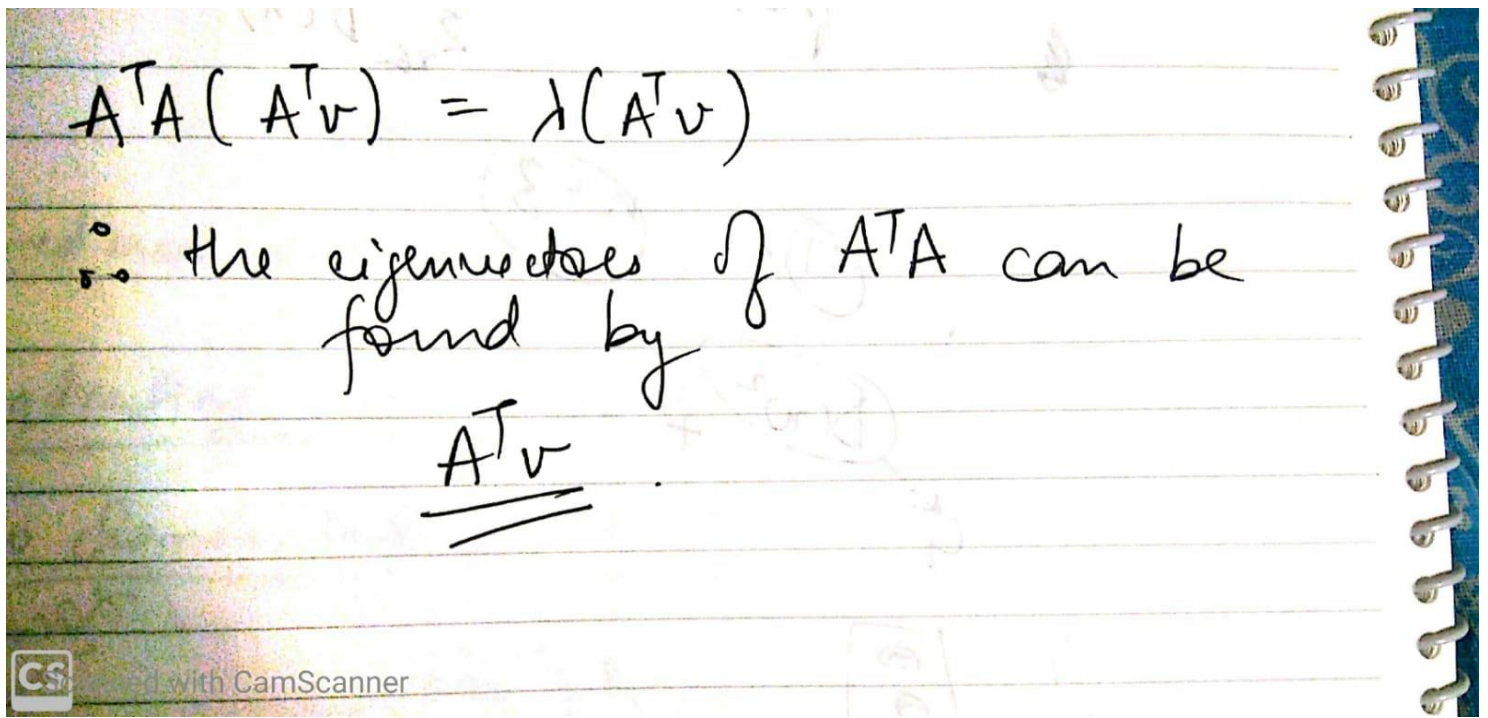


FIGURE 3.4 EXPLANATION HOW EIGENVECTORS OF  $A^T A$  CAN BE FOUND BY  $A^T v$

### 3.3 Top-k eigenfaces

#### **Documentation**

Top-15 eigenvectors are found by taking the largest 15 eigenvectors of the original matrix ([A@A.T](#)) and reshaping them and displaying them. Note that the eigenvectors of [A.T@A](#) will be 135x135, but of [A@A.T](#) will be 45045x135. So we take the largest 15 out of these 135 eigenvectors and display them. They look like faces (ghost faces though :P) and that's why we call them eigenfaces. They are actually eigenvectors.

#### **Observations and Results**

Fig 3.5, 3.6, 3.7 and 3.8 display the top 15 eigenfaces.

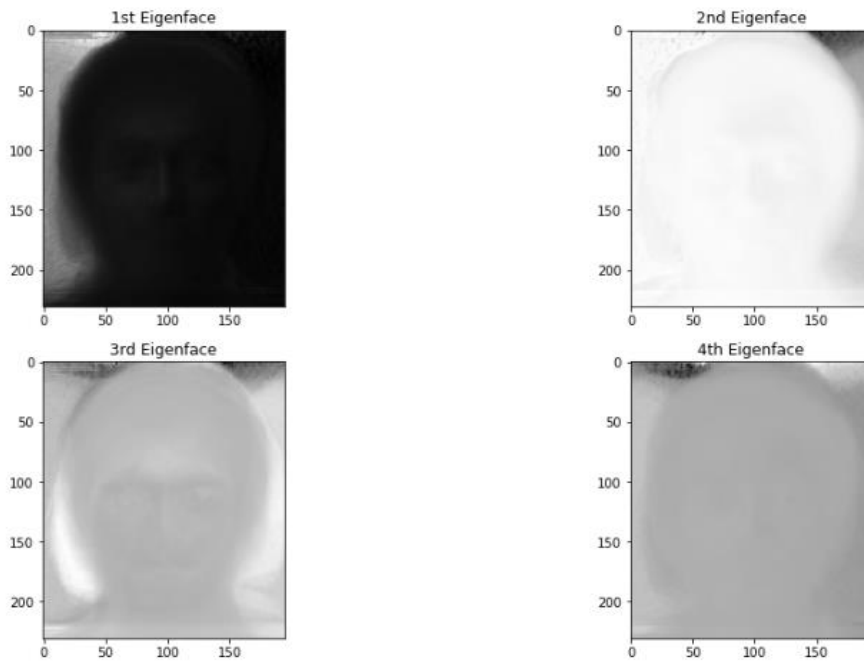


FIGURE 3.5 TOP 4 EIGENVECTORS (EIGENFACES)

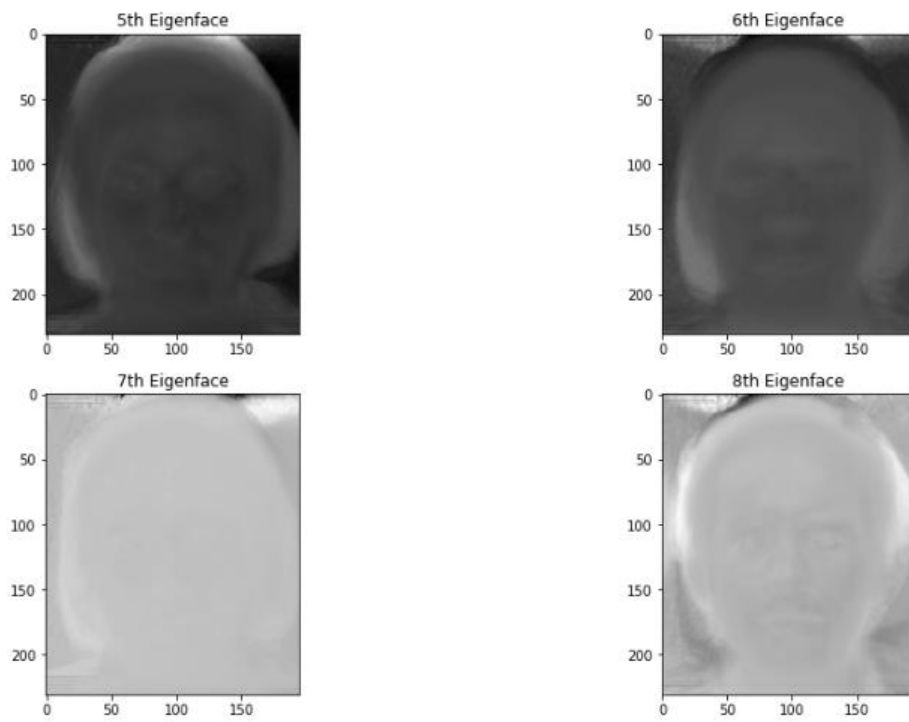


FIGURE 3.6 5<sup>TH</sup>-8<sup>TH</sup> EIGENVECTORS (EIGENFACES)

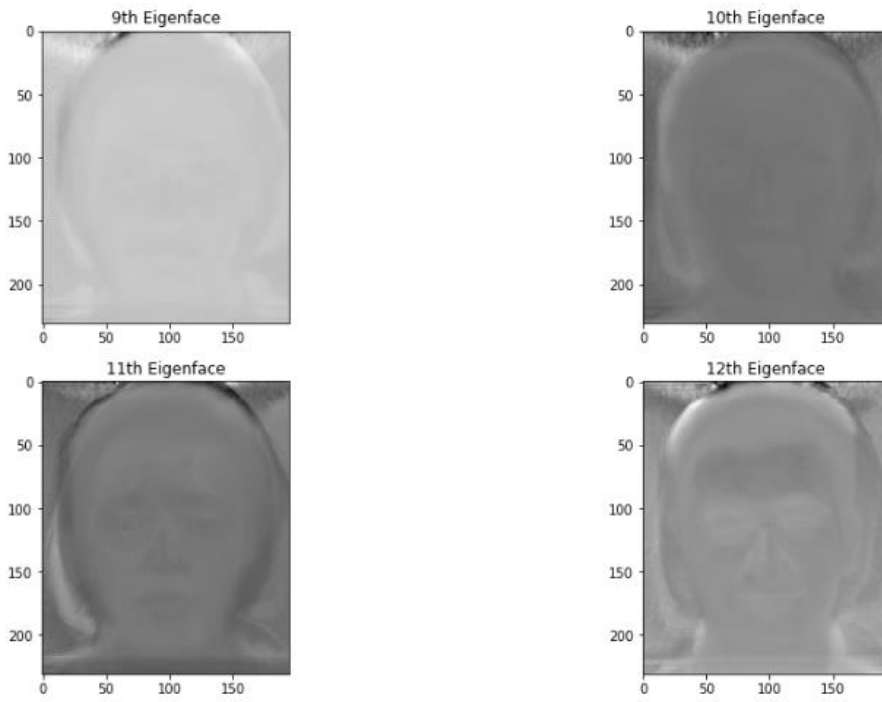


FIGURE 3.7 9<sup>TH</sup>-12<sup>TH</sup> EIGENVECTORS (EIGENFACES)

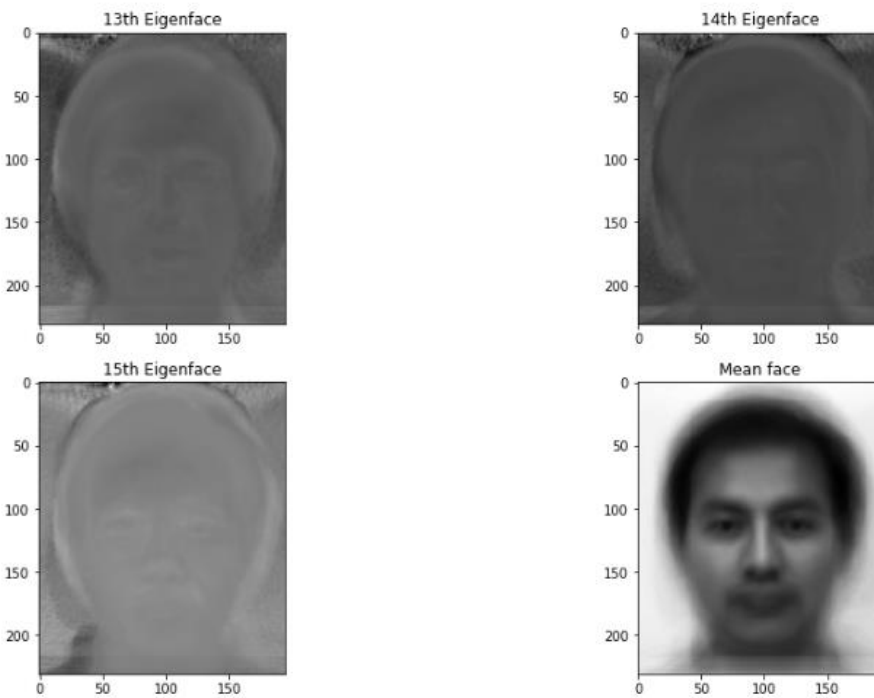


FIGURE 3.8 13<sup>TH</sup>-15<sup>TH</sup> EIGENVECTORS (EIGENFACES) AND THE MEAN FACE FOR COMPLETENESS

## Analysis

The eigenvectors displayed form a basis on which when we project our original data, we get the coefficients of each image for each eigenvector. Basically, our image is a linear combination of these eigenfaces added to the mean face. Every image in the training dataset can be found/reconstructed using  $K$  or more or all of these eigenvectors (eigenfaces). This is confirmed from the displayed images as different eigenvectors are capturing different features. Some are capturing brightness/darkness, while some are capturing more hair/less hair, and we can confirm visually

that they are excellent features for this dataset. Note that we display the top-15 eigenvectors here, meaning the ones with the 15 highest eigenvalues and the ones that capture the maximal variance in their direction.

3.4 Find Similar Faces in Training Set

Documentation

We find the projection onto the 15 top eigenvectors of our test image as well as our training data. Then use Euclidean norm and find the top-3 nearest training images for the test image. Refer to Task3.py for the code details.

Observations and Results

From Fig 3.9 – Fig. 3.18, we observe the top-3 matches for each of the ten test images in the Yale dataset.

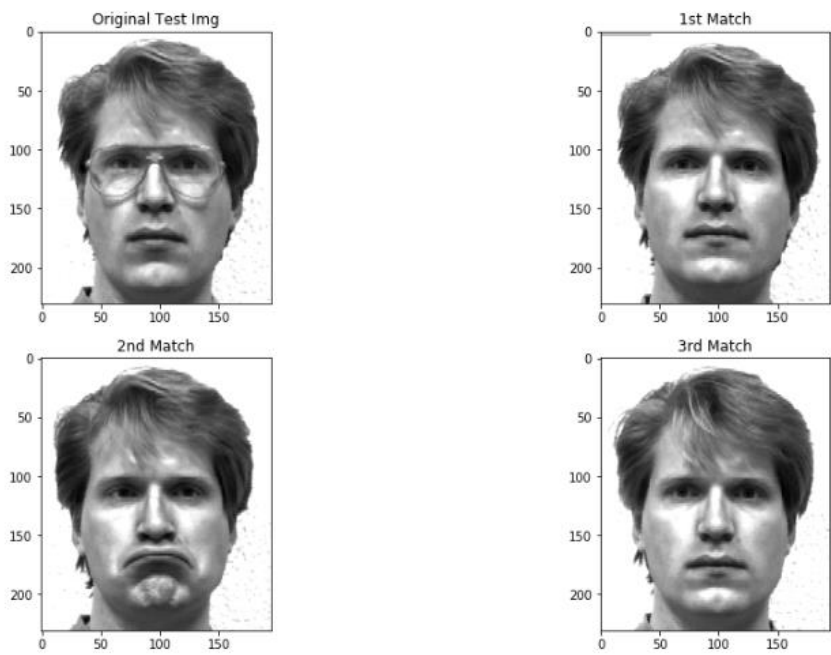


FIGURE 3.9 SUBJECT01.GLASSES TEST IMAGE AND TOP-3 MATCHES

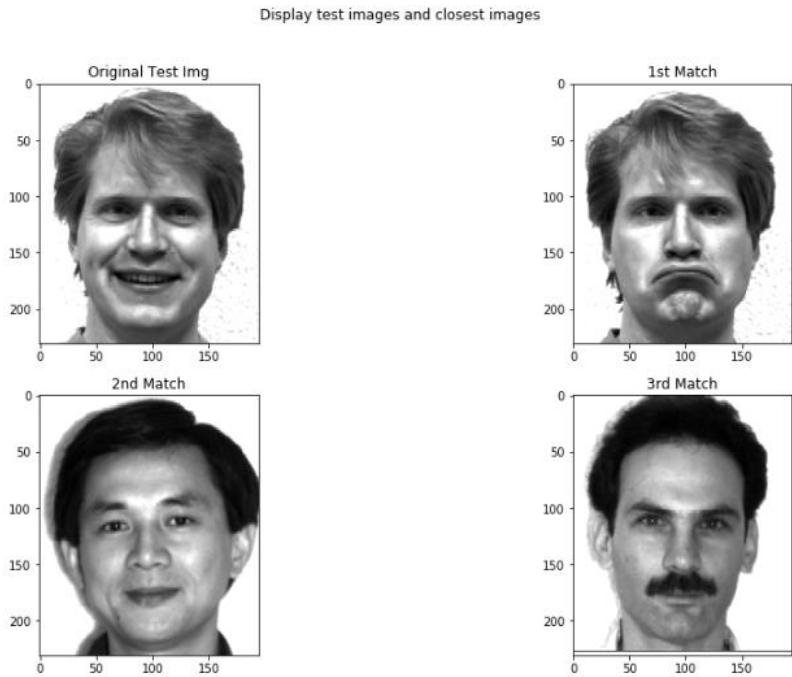


FIGURE 3.10 SUBJECT02.HAPPY TEST IMAGE AND TOP-3 MATCHES



Display test images and closest images

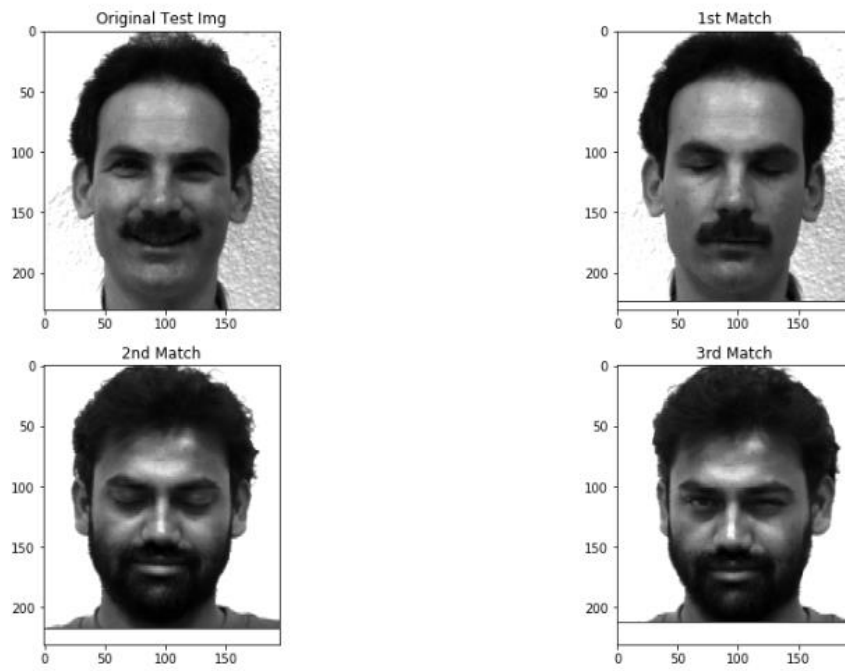


FIGURE 3.11 SUBJECT03.HAPPY TEST IMAGE AND TOP-3 MATCHES

Display test images and closest images

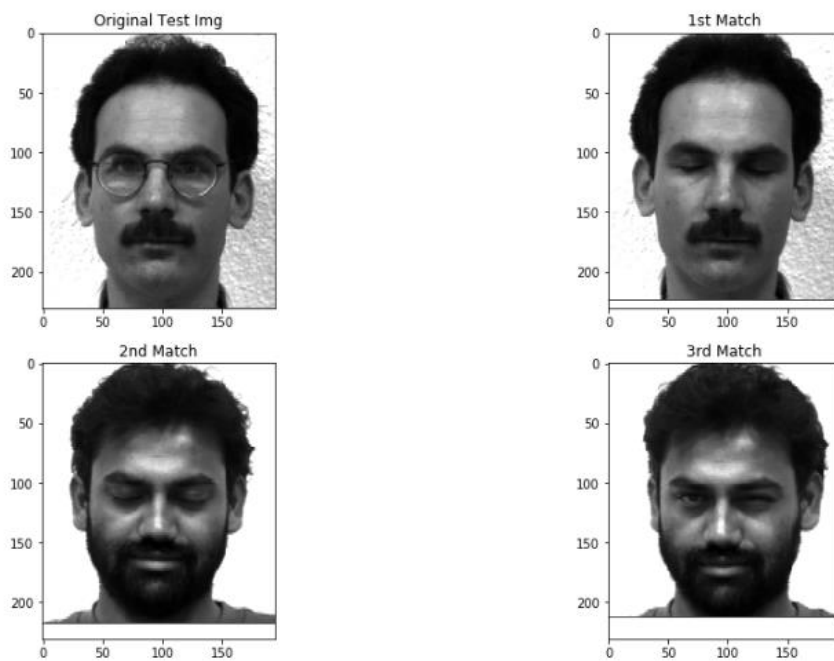


FIGURE 3.12 SUBJECT04.GLASSES TEST IMAGE AND TOP-3 MATCHES

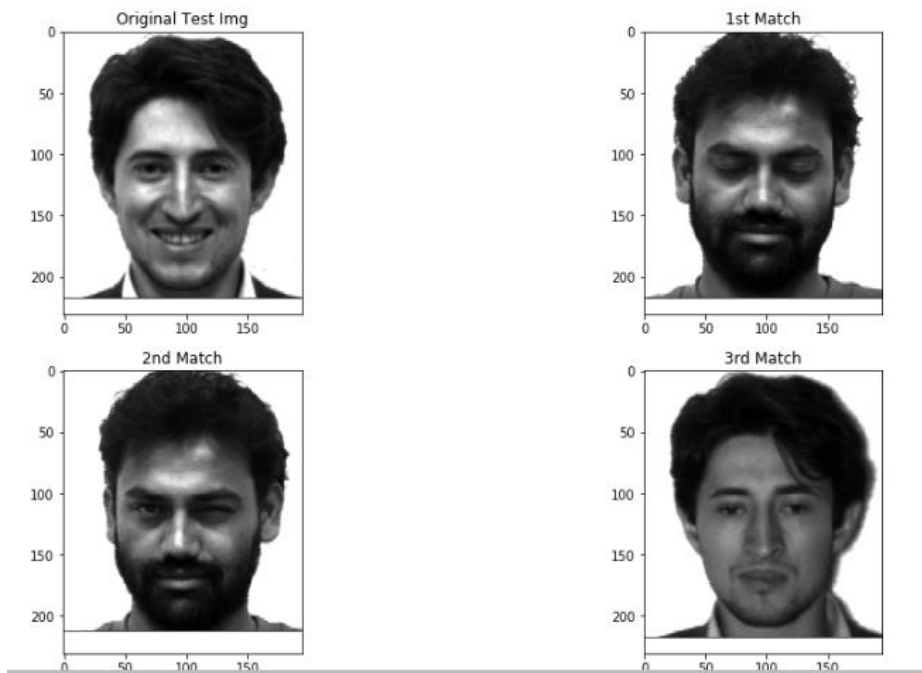


FIGURE 3.13 SUBJECT05.HAPPY TEST IMAGE AND TOP-3 MATCHES



FIGURE 3.14 SUBJECT06.HAPPY TEST IMAGE AND TOP-3 MATCHES

Display test images and closest images

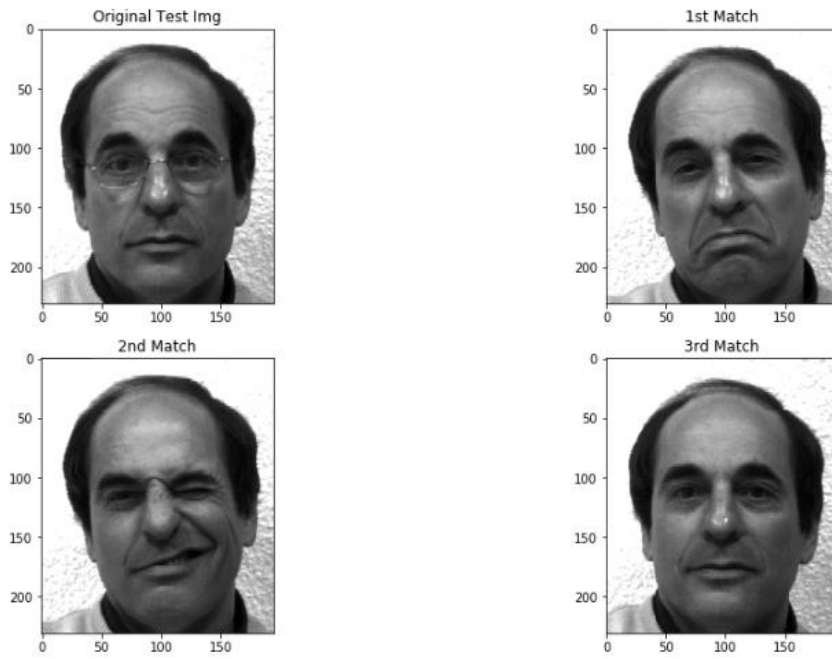


FIGURE 3.15 SUBJECT07.GLASSES TEST IMAGE AND TOP-3 MATCHES

Display test images and closest images

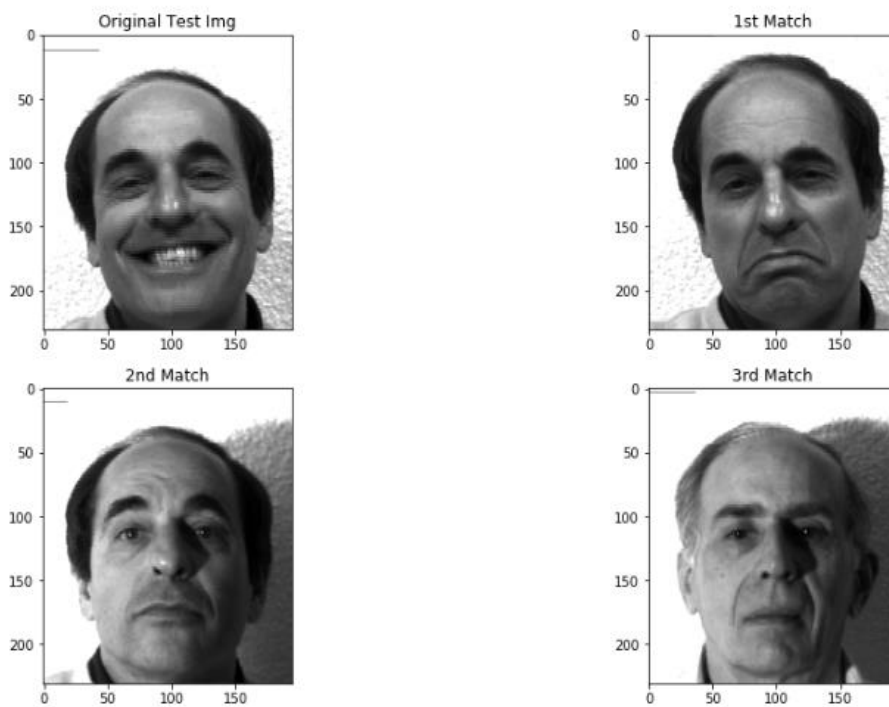


FIGURE 3.16 SUBJECT07.HAPPY TEST IMAGE AND TOP-3 MATCHES

Display test images and closest images

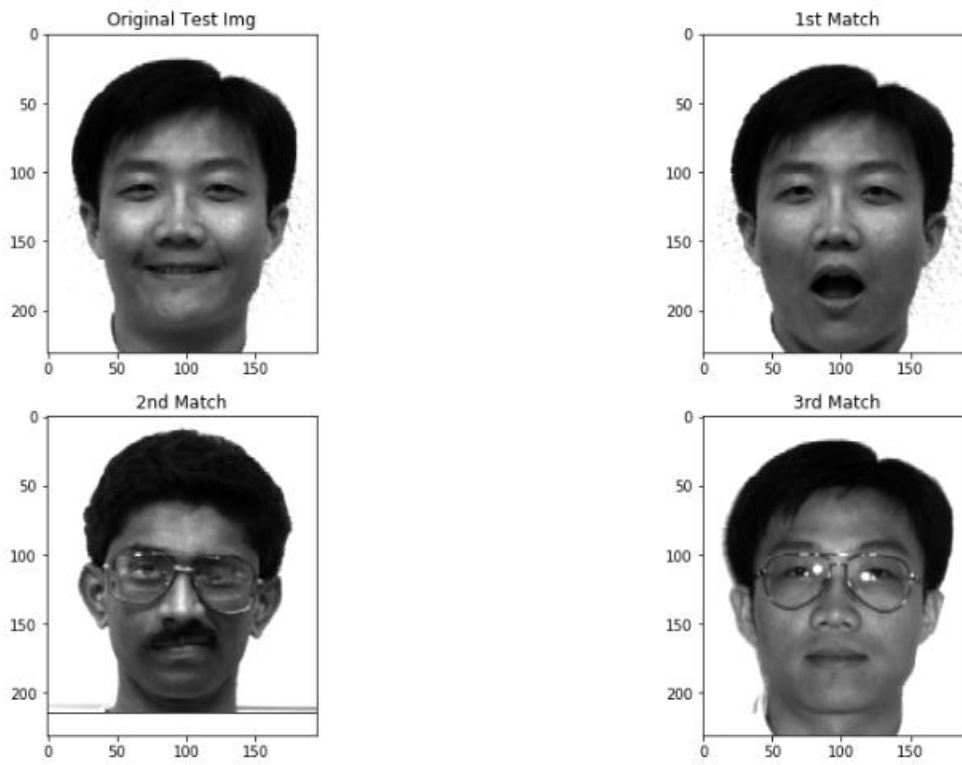


FIGURE 3.17 SUBJECT09.HAPPY TEST IMAGE AND TOP-3 MATCHES

Display test images and closest images

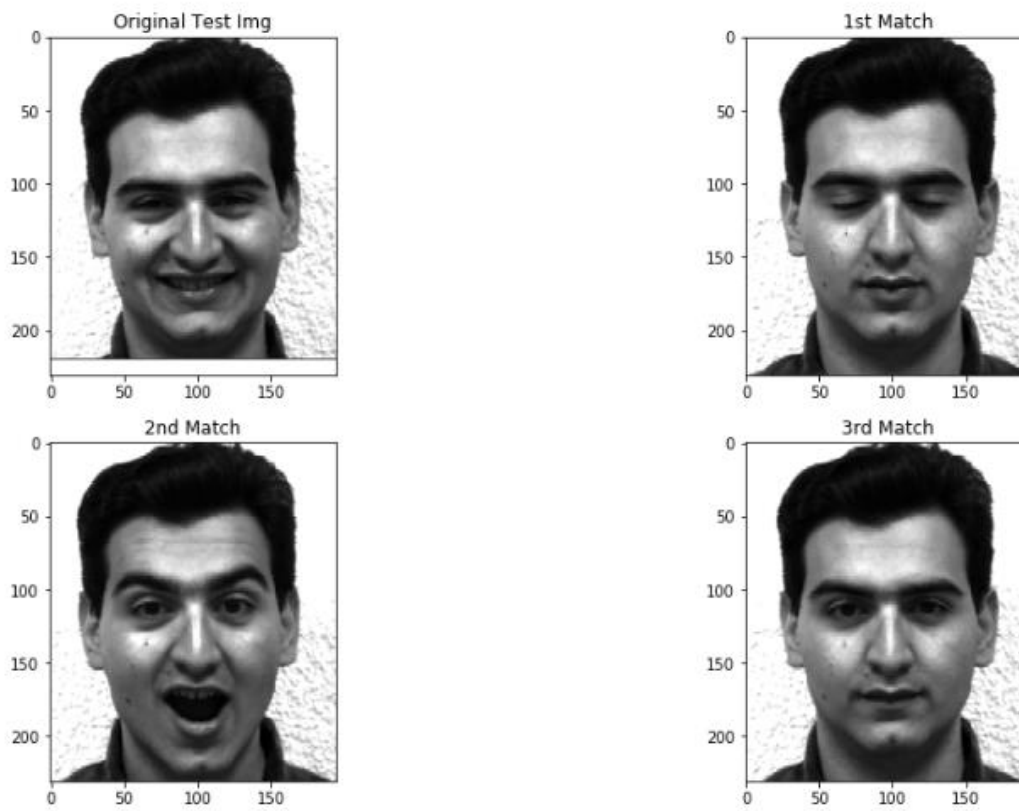


FIGURE 3.18 SUBJECT10.HAPPY TEST IMAGE AND TOP-3 MATCHES

## Analysis

We observe that the top matches are from the same person as the test image. This confirms that the Face Recognition engine we designed works. We took  $K=15$  to find matches. When we tried with  $k=10$ , the matches were not very accurate. This shows that the accuracy depends on the eigenvectors we take.

Percentage of information we retain: eigenvalues of top-15/sum of all eigenvalues = 0.8720157505997722%.

Let's calculate accuracy for top-1 match = 9/10 correct = 90%

Accuracy for top-2 match = 14/20 correct = 70%

Accuracy for top-3 match = 20/30 correct = 66.67%

### 3.5 My Image without My Images in Training Set

#### Observations

Fig 3.19 shows the top 3 matches for my image.

Display test images and closest images

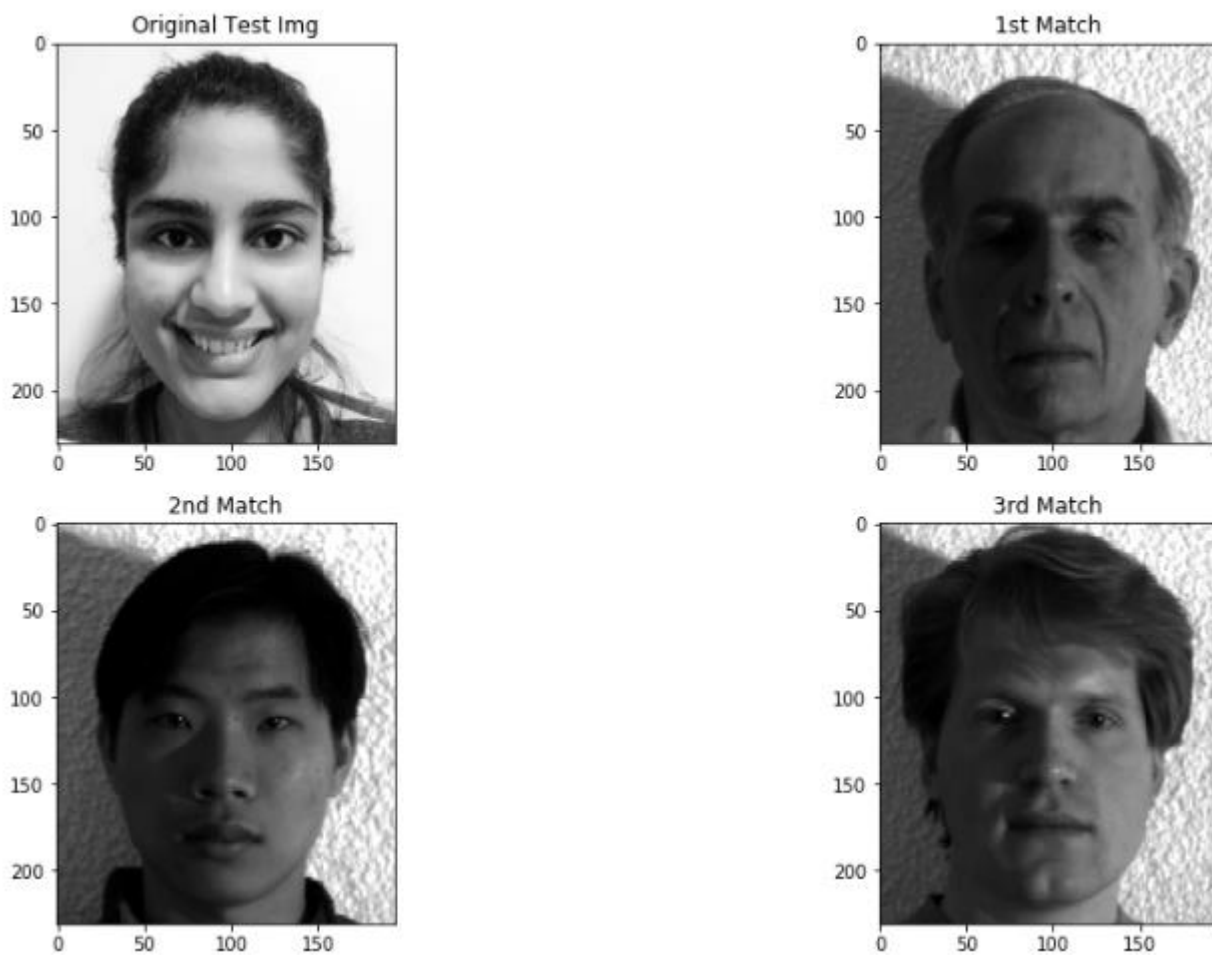


FIGURE 3.19 SUBJECT.NEW.01 TEST IMAGE AND TOP-3 MATCHES (MY IMAGE)

## Analysis

We can see the matches are pretty random some taking my smile into account and matching it with the images and that's how I get the first match. Some taking my nose shape into account (probably) or face shape. But since we don't have training images that are my faces, we don't get a proper match.



### 3.6 Face recognition with updates training set

#### Observations and Results

In Fig. 3.20, we see the matches to my image after incorporating my other 9 images in the training set and then calculating the nearest neighbours.

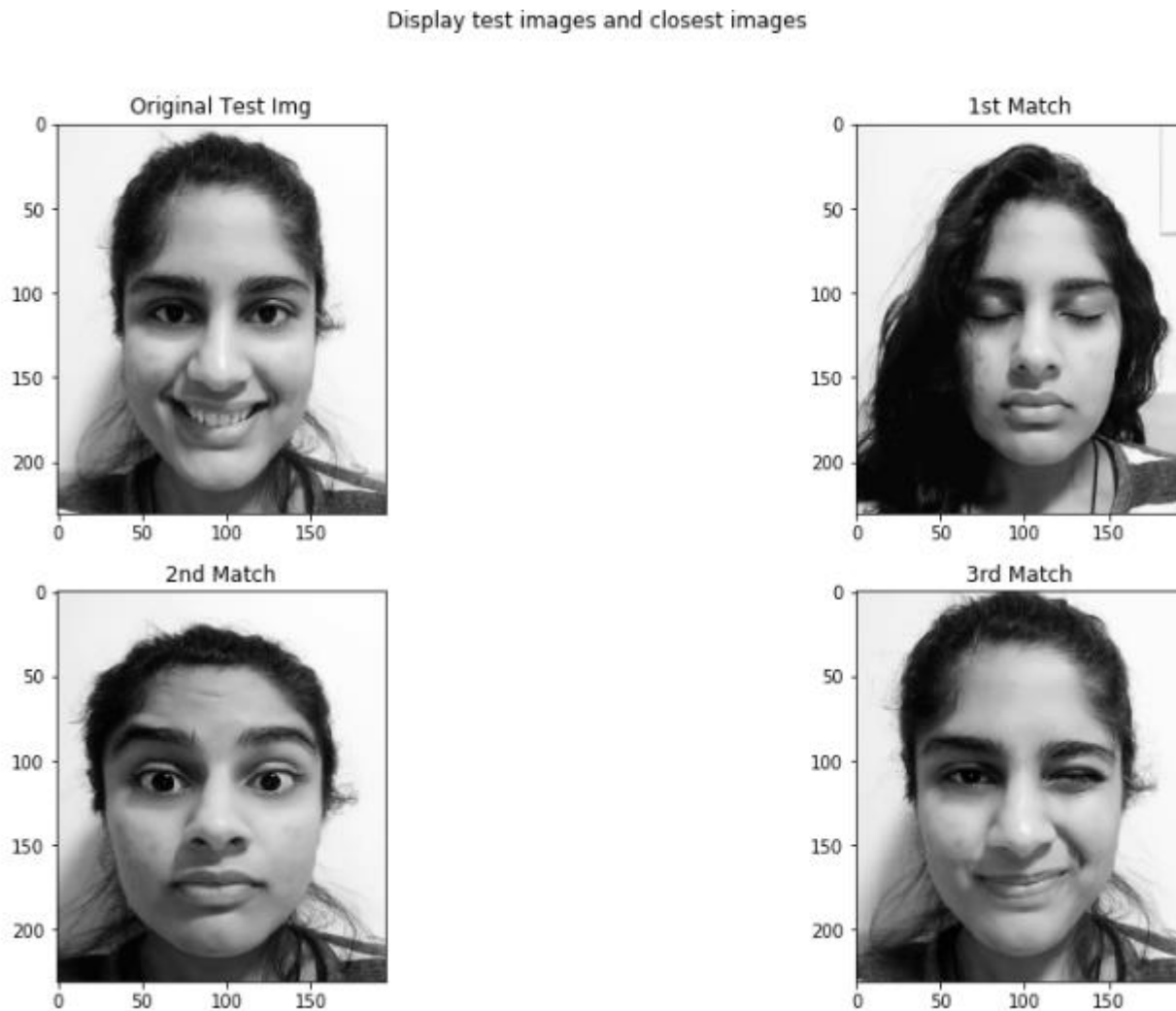


FIGURE 3.20 SUBJECT.NEW.01 TEST IMAGE AND TOP-3 MATCHES (WITH ADDED TRAINING DATA)

#### Analysis

We see the matches are good now and we don't get any spurious results. This further confirms that this system works. It also makes one fact apparent that if the training set doesn't contain images of a person, we get the closest images based on some facial features, but will never get a match because of obvious reasons. But however, if we were using GANs to generate images, we might be able to generate new images. (just a side-note).

Thank you for taking the time to read this report.