# ASSIGNMENT-2: CLAB-2 REPORT

**- Tanmay Negi**

**- u6741351**

**Task 1 : Harris Corner Detector**

This task mainly aims on getting familiar with the inside workings of Harris Corner detection algorithm. Harris corner use second order moments (or autocorrelation) matrix of intensity gradients, equations of which were provided. The task is divided into two parts, Part 1 demanded to compute the cornerness matrix 'R' while Part 2 demanded to implement non-maximal suppression on R to select local maxima .

**Documentation:**

For **Part 1**, I build a function that uses the equations of provided auto-correlation matrix

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$ to calculate R = det(M) – k (trace M)^2

**Function : corner(Image)**

**@param** Image Input grayscale image

**@output->** "cornerness" matrix R

-→ **@output.shape** == @param Image.shape

-> **@output return type** : ndarray

For **Part 2**, I build a function that perform maximal suppression on calculated matrix "R"

**Function: suppression(R , k_size=3)**

**@param** R , "R" matrix from previous function

**@param** k_size , window size to find local maxima from, default value = 3

**@ output** -> A N*2 matrix containing [row,column] of N pixels classified as corner

**Results:**

Here's the result when I used the threshold of 0.01*max(R) on each pixel ,'R' being the "cornerness" matrix.

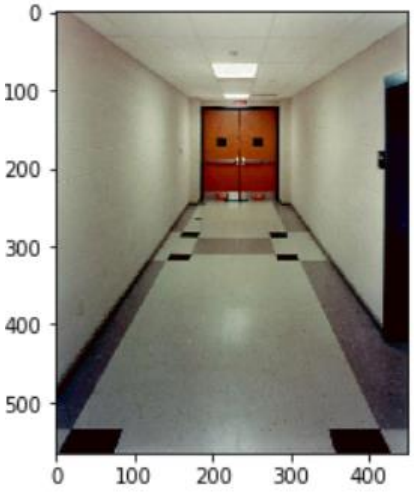



*Fig 1.1 input_image_1*

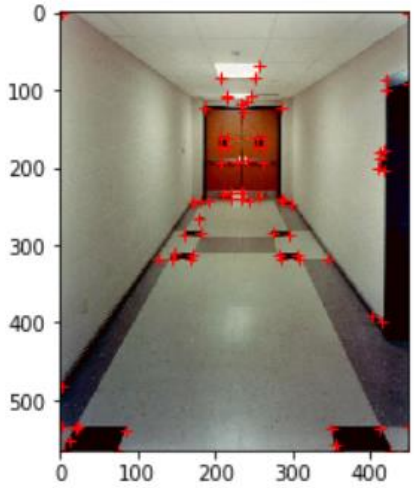



*Fig 1.2 output_image_1*

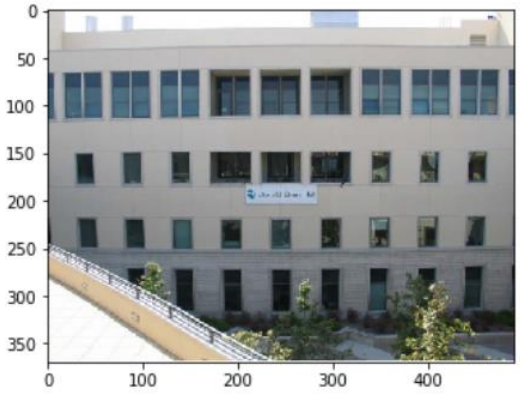



*Fig 1.3 input_imae_2*

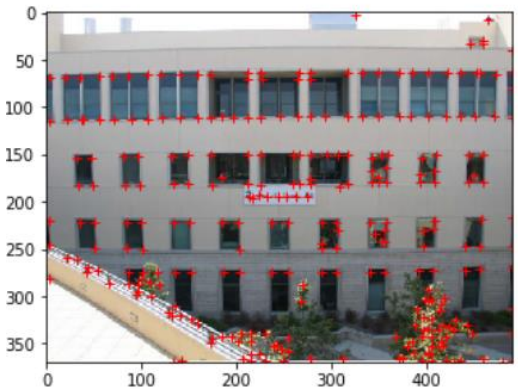



*Fig 1.4 output_image_2*


*Fig 1.5 input_image_3*

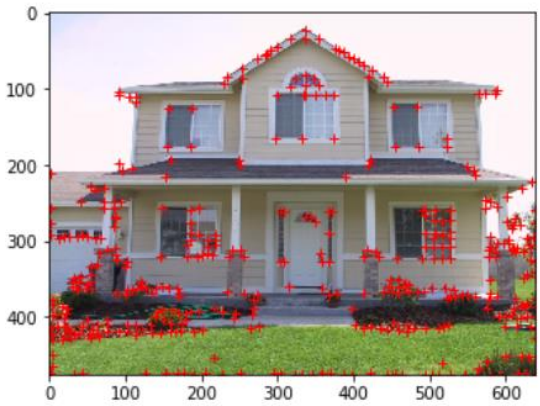



*Fig 1.6 output_image_3*

*Fig 1.6 input_image_4*

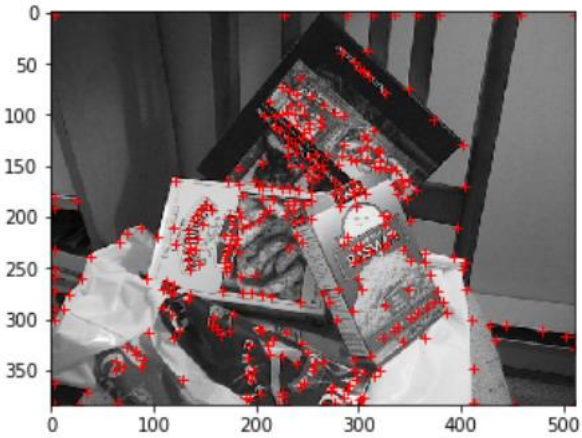


*Fig 1.7 output_image_4*

And here's a brief comparison with and without non-maximal suppression, along with thresholding.

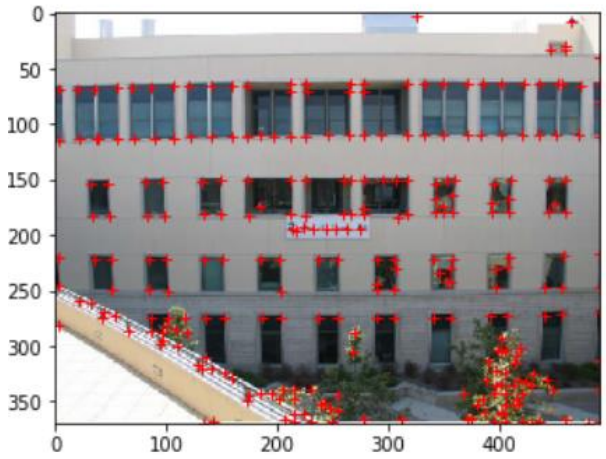


*Fig 1.7 with suppression*

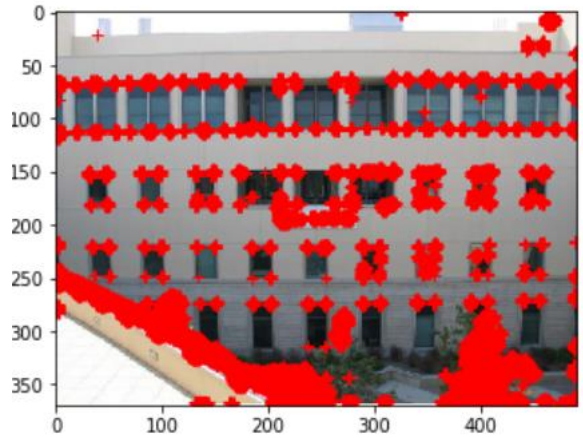


*Fig 1.8 without suppression*

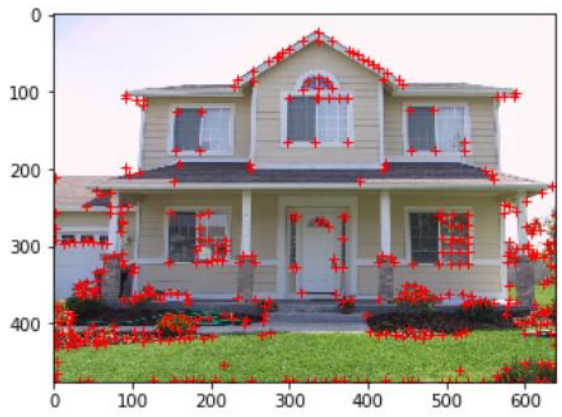


*Fig 1.9 with suppression*

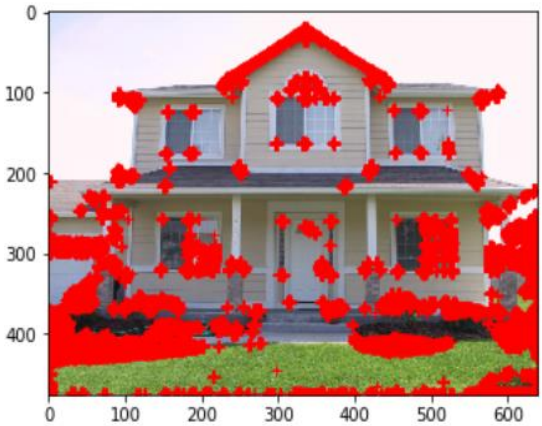


*Fig 1.10 without suppression*

As we can see, performing suppression does remove redundant features.

**# 1.5**

Here's the comparison with inbuilt python function for Harris Corner, **cv2.cornerHarris()**

*Fig 1.10 custom function*

*Fig 1.11 inbuilt function*

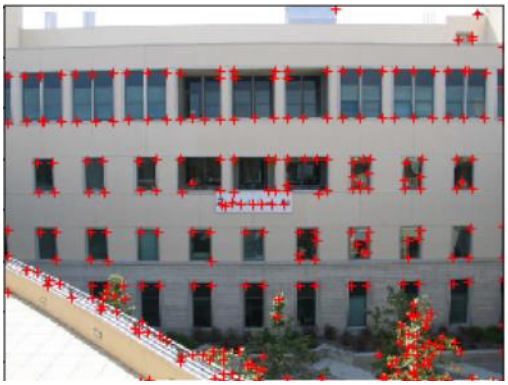


*Fig 1.12 custom function*

*Fig 1.13 inbuilt function*

*Fig 1.14 custom function*

*Fig 1.15 inbuilt function*

*Fig 1.16 custom function*

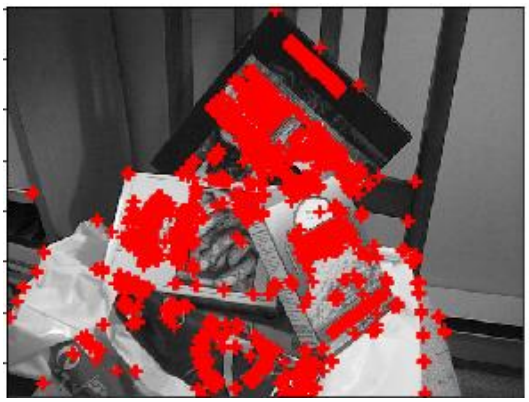


*Fig 1.17 inbuilt function*

# factors affecting performance of Harris Corner detection:

1. Thresholding on cornerness matrix: It should neither be too small as then it would pick redundant features similar to which we saw in Fig 1.8 , 1.10 , nor it should be too large as it would flush out some useful information
2. Window Size while performing suppression: As discussed in my custom function suppression default k_size is 3, when varying this parameter similar affect was seen on the performance, higher k_size would result in less marked corners as algorithm was highly selective, similar was true when decreasing this size, also if we change the stride of this kernel, similar effects were seen

**Task 2:**

This task is aimed to perform image segmenting via k-means clustering on each pixels of the image represented as vector. I have built a function for extracting pixel vectors from the image, and then a function for k-means.

**Documentation:**

Here's my function for extracting features.

Function => **def pixel_to_vec ( image )**

**@param** image, a RGB image of shape (n, m, 3)

**@return** a ndarray of shape (n*m, 5), where each row contains information [l, a, b, x, y] of corresponding pixel

'l': lightness of the color

'a': - the color position between red and green

'b': - the position between yellow and blue

'x', 'y': - pixel coordinates

Here's my function for **k-mean** for clustering.

Function => **def my_kmeans(df , n_clus=2 , itr = 10)**

**@param** df , a 2-d array of shape (n, m)

'n' => total number of vectors

'm' => features of vector

**@param** n_clus , number of clusters , default = 2

**@param** itr, number of iterations to perform, default = 10

**@return** ndarray of shape ( n , ) labelling 'n' vectors to corresponding cluster

**Results:**

To analyze the result we will be using following two images

*Image 1*

*Image 2*

**n_clusters = 2 , iterations = 10 , without pixel coordinates**

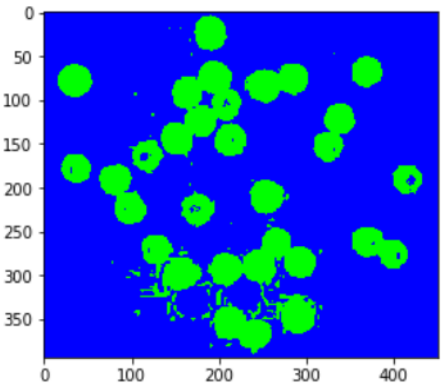



*Fig 2.1 my_kmeans( image1, 2 , 10)*

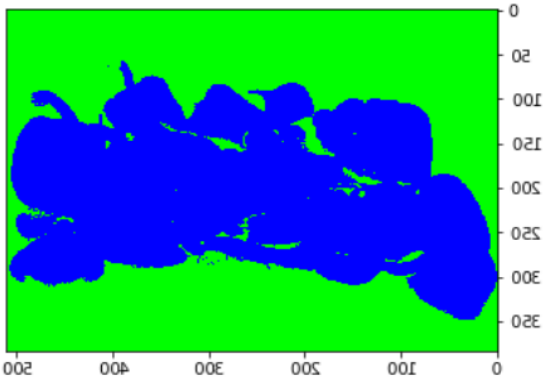


*Fig 2.2 my_kmeans(image2, 2 , 10)*

**n_clusters = 3 , iterations = 10 , without pixel coordinates**

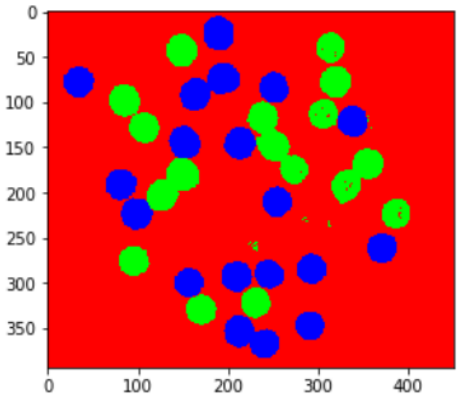



*Fig 2.3 my_kmeans(image1, 3, 10)*

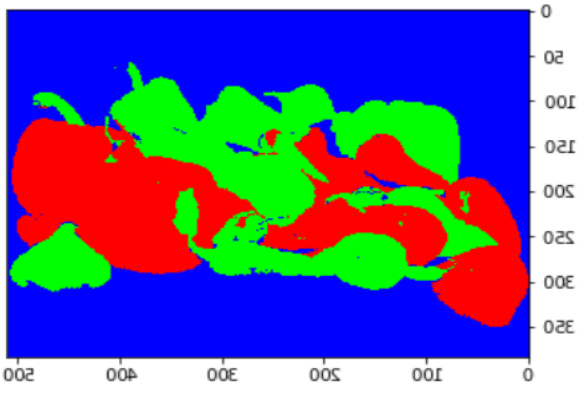


*Fig 2.4 my_kmeans(image2, 3, 10)*

**n_clusters = 2 , iterations = 10 , with pixel coordinates**

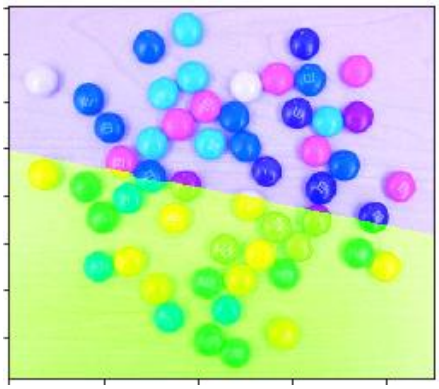


*Fig 2.5 my_kmeans(image1, 2, 10)*

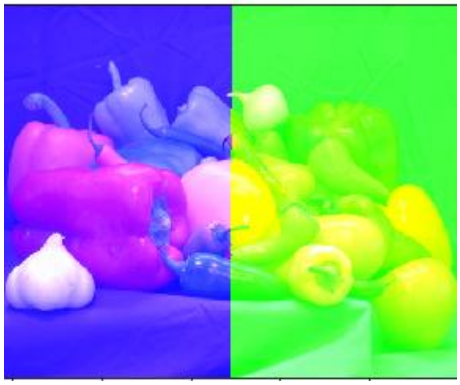


*Fig 2.4 my_kmeans(image2, 2, 10)*

**n_clusters = 3 , iterations = 10 , with pixel coordinates**

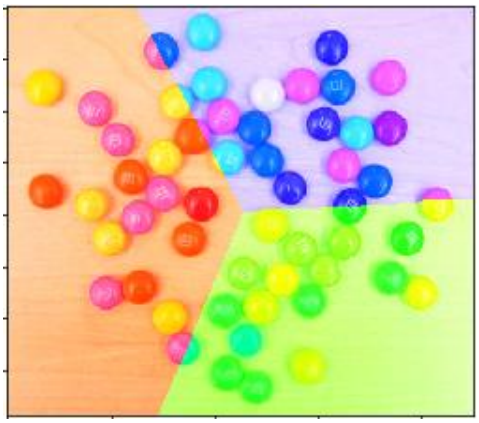


*Fig 2.6 my_kmeans(image1, 3, 10)*

*Fig 2.7 my_kmeans(image2, 3, 10)*

**# 2.3 kmeans vs kmeans++**

Given the vectors to be clustered, in **k-means** the initial k-centroids are randomly chosen whereas in **k-means ++** the initial clustered chosen are maximally separated to avoid local minima. In general k-means

++ provides better accuracy as mentioned, due to it's ability to avoid local minima. The paper further provides monte-carlo simulation result that showed **k-means ++** , is both faster and provides a better performance.

**# My implementation of k-means++**

Though at this stage while report is being written, I was not able to implement a successful k-means++ algorithm as every time I ran into stack-over flow / memory error while finding next k-1 clusters from initial random seed provided. So if comparing the speed of convergence of my **kmeans++** at this stage, it's infinite, and accuracy is zero.

**Task 3:**

**Documentation**

The main aim of this task was is to build a face recognizer using eigen-face method. First, I extracted top 10 eigenfaces from given 135(231*195) training images. Then I converted each image from test and training set as 1*10 vector, containing 10 projection weights along these 10 eigenfaces.

Then to find similarity I calculated the distance between each test image projection vector with projection vectors of corresponding 135 train images. Higher the similarity, shorter the distance.

**Note:**

For calculating eigenfaces I have documented 2 methods in code, first one uses normal eigenvectors decomposition on calculated covariance matrix from normalized training tensor, second uses direct SVD decomposition of training tensor for calculating eigenfaces thus eliminating the need for covariance matrix calculation.

Following results are based on method 1.

**Result and Analysis:**

**# 3.2.2**

Here's the mean face of the training dataset.

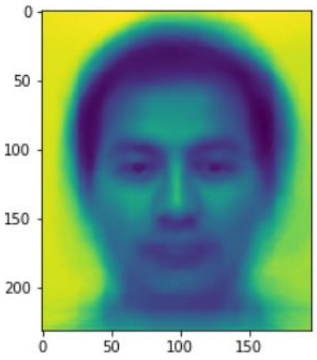



*Fig 3.1 mean face of 135 samples*

**# Computationally cheap way of finding eigenfaces**

Note that each of our (normalized) image is of size 45045 ( 231*195) pixels.

Let 'T' be the matrix of shape 45045*135 , containing 135 normalized images

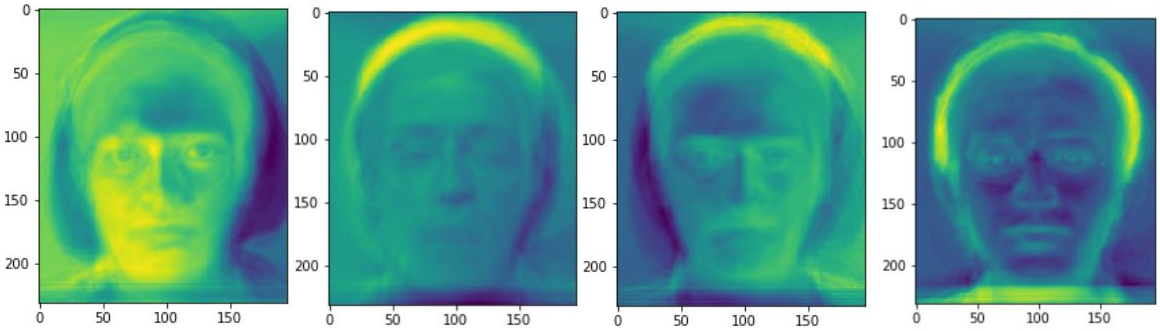
We can calculate covariance matrix as dot-product(T , T.transpose) and then perform eigenvector decomposition on it. But the resulting covariance matrix of shape (45045*45045), will be very large, one way around this is to perform eigen decomposition on dot_product(T.transpose , T), the resulting matrix will be of the shape 135*135 ( there are 135 training images).

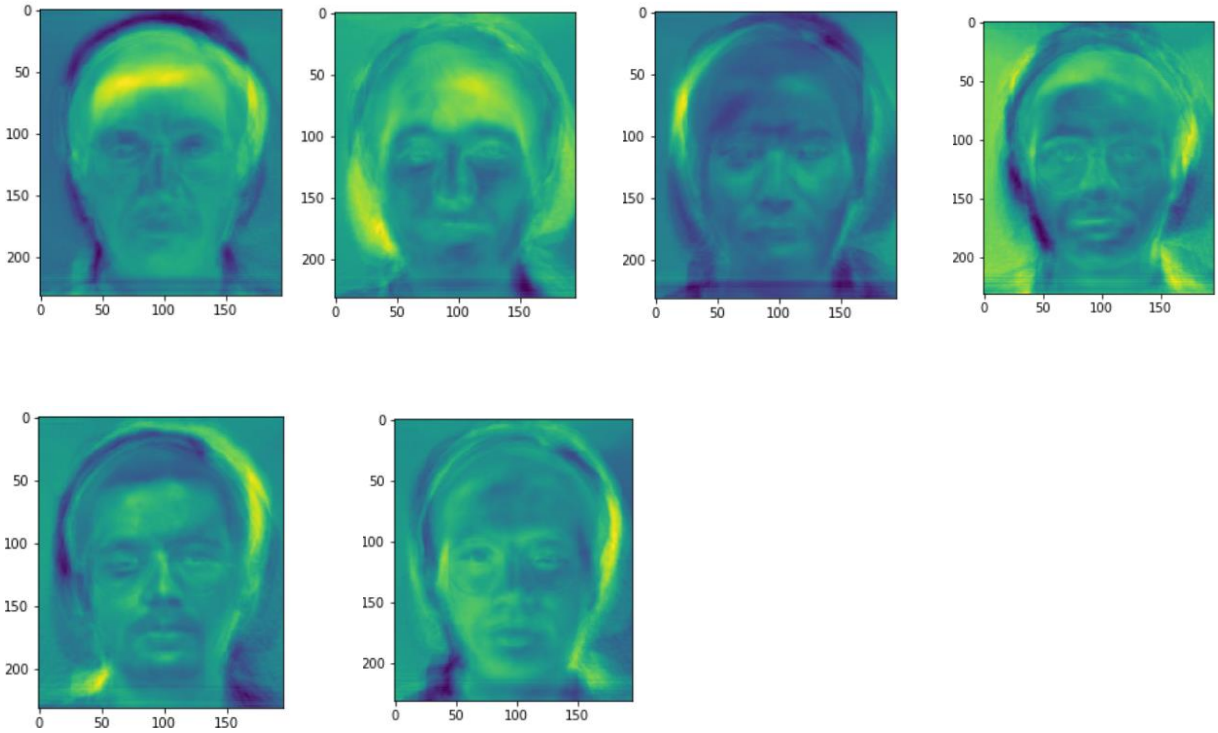
Here's a short article describing this equation in detail.

Or we can directly use SVD method on T to get eigenfaces, then we don't have to calculate covariance matrix at all. This was done by using numpy.linalg.svd() function, which was allowed to use.

**# 3.2.3**

Here are the top 10 eigenfaces that were computes from 135 training images. In theory this should represent the most variance in the training set.





**# 3.2.4**

Here's the results when I tested my model on 10 provided test images

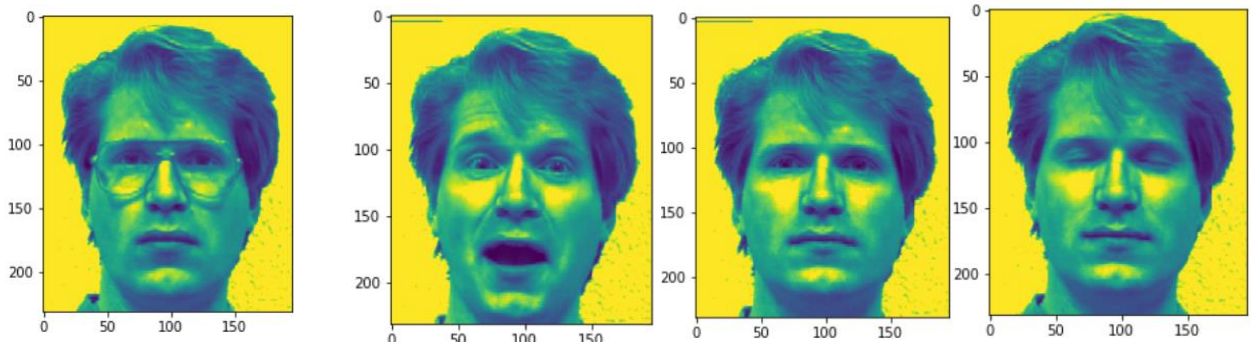


*Fig 3.12 Test_image 1* *Corresponding top 3 matched images from train_set*

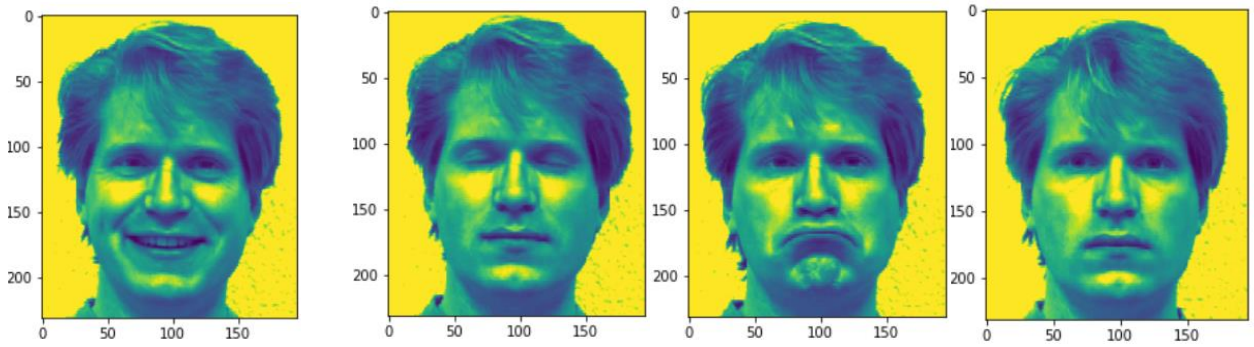


*Fig 3.13 Test_image 2* *Corresponding top 3 matches from train_set*

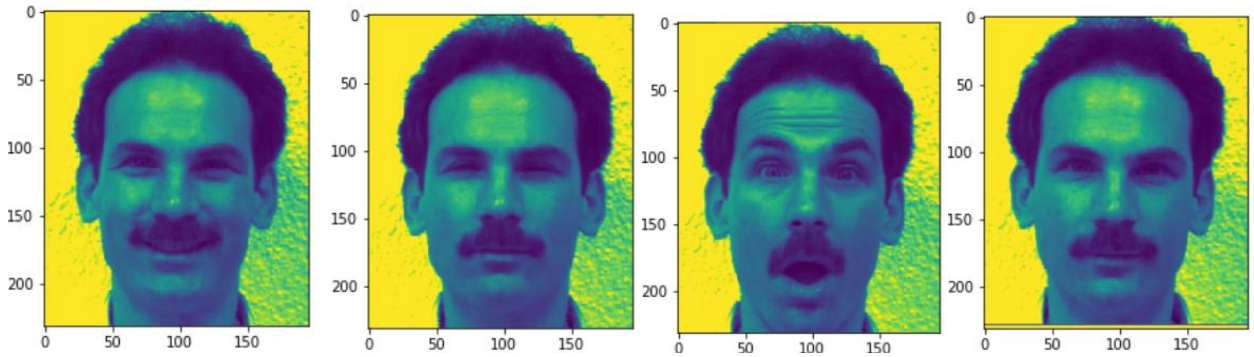


*Fig 3.14 Test_Image 3* *Corresponding top 3 matches from train_set*

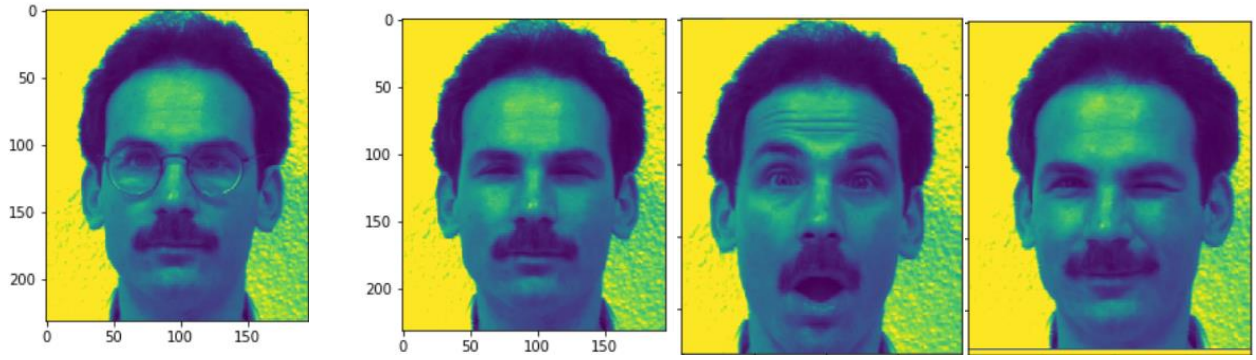


*Fig 3.15 Test_image 4* *Corresponding top 3 matches from train_set*

*Fig 3.16 Test_image 5* *Corresponding top 3 matches from train_set*

*Fig 3.17 Test_image 6* *Corresponding top 3 matches from train_set*

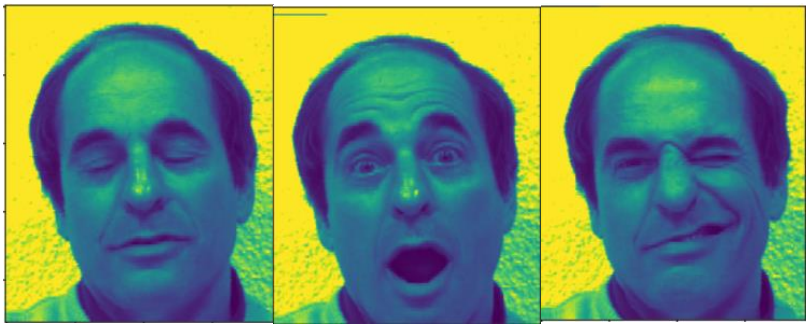


*Fig 3.18 Test_image 7* *Corresponding top 3 matches from train_set*

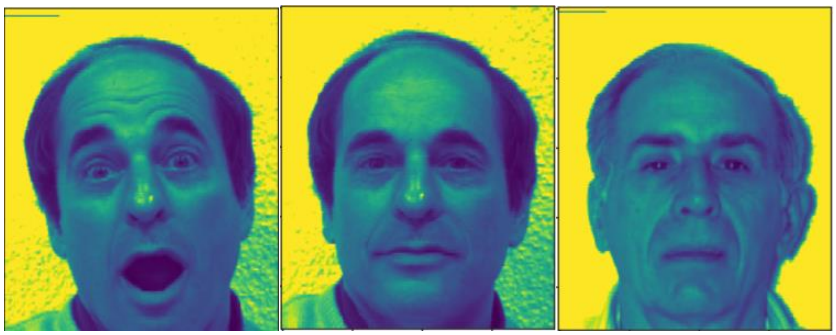
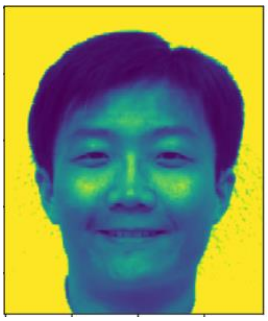


*Fig 3.19 Test_image 8* *Corresponding top 3 matches from train_set*



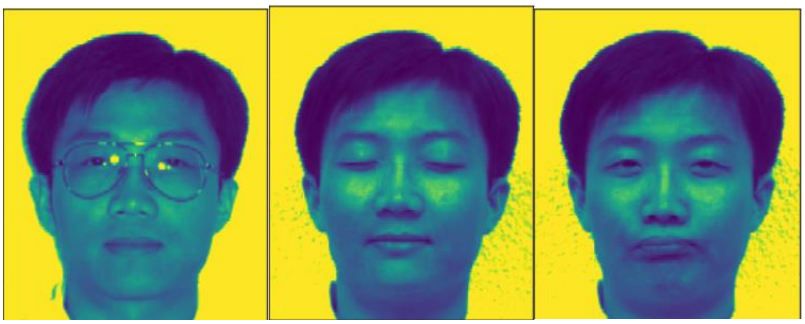


*Fig 3.20 Test_image 9* *Corresponding top 3 matches from train_set*

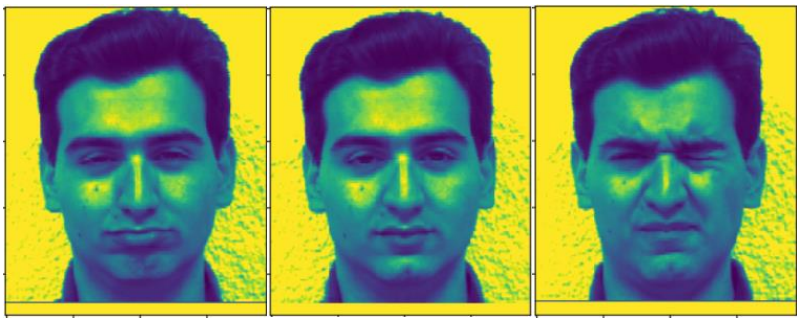


*Fig 3.21 Test_image 10* *Corresponding top 3 matches from train_set*

**Accuracy of recognizer:**

The recognizer worked well in most of the cases except for test image 5 and 8 (as you can see in fig 3.16 and 3.19).

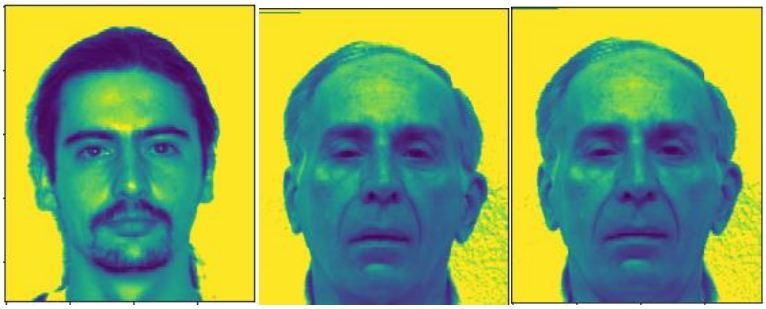
Assigning equal weightage to each of top 3 images the accuracy is 90% ( 27 / 30 ).

Note that the accuracy can be further increased by increasing the number of eigenfaces or eigen-dimension for representing each image, but we must take in account its trade-off with the memory. Here we have a feature-space of 10 eigenfaces and each image being represented as 1*10 vector, comprising 10 weights towards 10 eigenfaces axis, we can increase the number of eigenfaces but accordingly so our image representation bits will also increase.

**# 3.2.5**

Here's the result of recognizer on one of my image, note at this point model is trained just on original 135 training images.

*Fig 3.22 my Test_image*



*Corresponding top 3 matches from original train_set*

**# 3.2.6**

Here's the result when I trained the model with additional of 9 images with are provided in 'my_data/train' directory , along with original 135 images.

*Fig 3.23 my Test_image*

*Corresponding top 3 matches from new train_set*