Assignment 5 – Face Recognition

-Ashrith Chandramouli (aschandr@syr.edu)

Brief overview :

The script can be run using the command – 'python eigenfaces.py'

The script does not require any parameters and the paths to the input images are hardcoded.

Below libraries are necessary to run the script

```
import time
from math import fabs, hypot, inf, pi, sqrt
from os import listdir
from os.path import isfile, join

import matplotlib.pyplot as plt
import numpy as np
from numpy import linalg as LA
from scipy import ndimage, signal
from scipy.linalg import eigh as largest_eigh
from scipy.spatial import distance
from skimage import io
from sklearn import preprocessing
```

By running the script you will find the outputs for

- Face projections of trained images
- Face matching for face images

Other outputs are also available which have been commented.

- Output of eigenfaces
- Face matching for non-face images
- Image differences between original and reconstructed
- Frobenius norm plots

The output of eigenfaces has been commented in the init part of the Training class

All the other test cases are commented in the main stub. In the main stub, find_match performs face recognition and show_diff shows the image differences and plots the frobenius norm.

The result of the report contains my observations and explanations

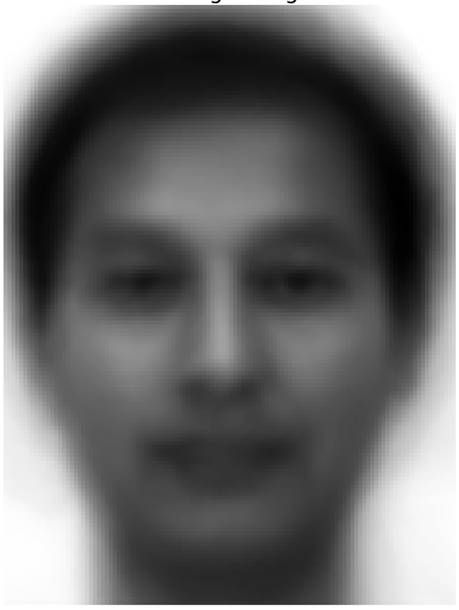
Part 1) EIGENFACES

a) Reading images from the training set and displaying average face

I first collect all the images, find the average image and then flatten it. Then I create the image matrix by flattening the read images and subtracting the mean from them. The code is as shown below

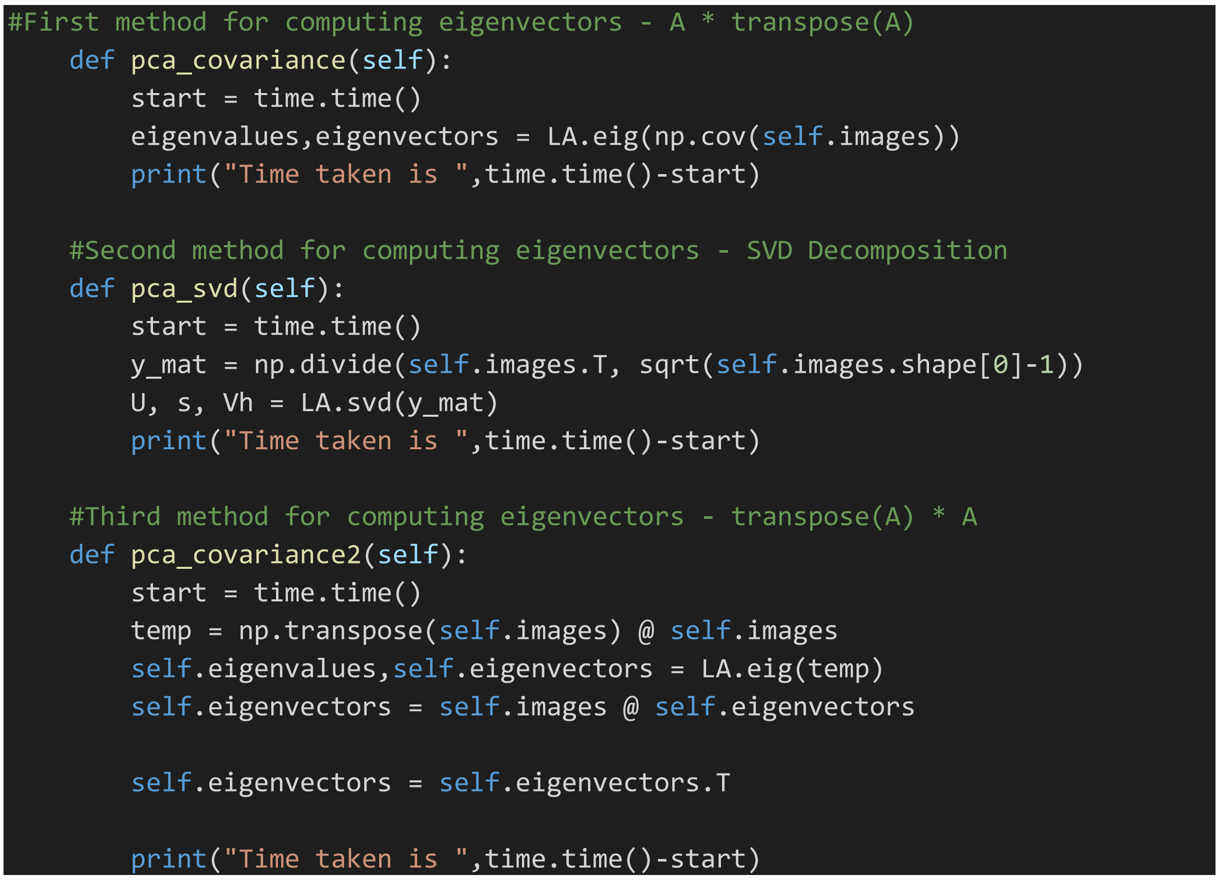
```
#Get normalized images (mean subtracted), flatten and create a matrix of image
s - A
    def read_images(self):
        self.images = []
        self.avg_img = np.zeros_like(io.imread(self.image_files[0]),dtype='int
32')
        #print(self.avg_img)
        for file in self.image_files:
            img = io.imread(file)
            #print(img)
            self.avg_img = self.avg_img + img
            self.images.append(img.flatten())

        self.avg_img = np.divide(self.avg_img,len(self.image_files))
        self.images = np.array(self.images,dtype='int32')
        flattened_avg = self.avg_img.flatten()
        for ind,val in enumerate(self.images):
            self.images[ind] = np.subtract(self.images[ind],flattened_avg)
        self.images = np.transpose(self.images)
```

Average image



b) Perform PCA in 3 ways

I have implemented 3 functions pca_covariance, pca_svd and pca_covariance2 which correspond to methods 1-3 respectively. The code for these methods is as shown below



```
#First method for computing eigenvectors - A * transpose(A)
    def pca_covariance(self):
        start = time.time()
        eigenvalues,eigenvectors = LA.eig(np.cov(self.images))
        print("Time taken is ",time.time()-start)

    #Second method for computing eigenvectors - SVD Decomposition
    def pca_svd(self):
        start = time.time()
        y_mat = np.divide(self.images.T, sqrt(self.images.shape[0]-1))
        U, s, Vh = LA.svd(y_mat)
        print("Time taken is ",time.time()-start)

    #Third method for computing eigenvectors - transpose(A) * A
    def pca_covariance2(self):
        start = time.time()
        temp = np.transpose(self.images) @ self.images
        self.eigenvalues,self.eigenvectors = LA.eig(temp)
        self.eigenvectors = self.images @ self.eigenvectors

        self.eigenvectors = self.eigenvectors.T

        print("Time taken is ",time.time()-start)
```

c) Compare the PCA measurements

On comparison, I found that the first method for finding the principal components was very slow and took about 4793 seconds or 79.88 minutes approximately an hour and 20 minutes. With the SVD method, calculating the principal components took about 105 seconds or roughly a minute and 45 seconds. With the third method, I was able to calculate the principal components in just 1 second. This shows that the third method was significantly faster than the other two

d) Find n significant eigenvectors

The ‘n’ value I’ve chosen here is 100 just so I get the maximum details per person. I first collectively sort both the eigenvalues and eigenvectors according to the absolute value of the eigenvalue since negative high eigenvalues are also available. I choose the last 100 of the corresponding eigenvectors as my eigenfaces. The code for that is as shown below

```
#Find top n eigenvectors that become the face space - eigenfaces
    def find_eigenfaces(self,n):
        eValues = np.array(self.eigenvalues, dtype='int32')
        eFaces = [y for x,y in sorted(zip(eValues,self.eigenvectors),key=lambd
a x:abs(x[0]))] #abs is used since there maybe high negative value eigenvalues
        self.eigenfaces = eFaces[-n:]
        self.eigenfaces = np.array(preprocessing.normalize(self.eigenfaces, ax
is=1, norm='l2'))
```

e) Finding weights

For finding weights or the face classes, I first calculate the weights of all the images of a person from the training set and I only take the average of those weights as the face class for the person. This is repeated for every person on the training set. I consider 9 images per person for the weights.

The code is as shown below

```
#Calculate the face class or the weight vectors for each image
    def find_weights(self,n):
        self.face_class = []
        weights = np.zeros((1,n))
        count = 0
        for img in self.image_files:
            image = io.imread(img)
            weight = self.eigenfaces @ np.transpose(np.subtract(image,self.avg
_img).flatten())
            weights = weights + np.array(weight)
            count += 1
            #Considering 8 images per person in training set
            if count == 9:
                self.face_class.append(np.divide(weights,9))
                weights = np.zeros((1,n))
                count = 0
```

f) Reconstruct images from weights

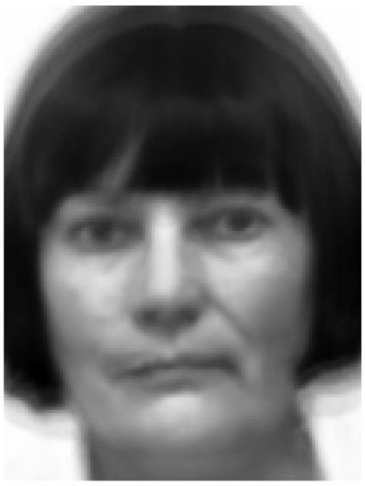
As I have calculated the average of 9 weights per person, I will not be able to display the original image. I will display the reconstructed images from those average weights. Due to this, any change of position, tilt, lighting or expression will be influenced in the output image and might not look exactly like the original image.

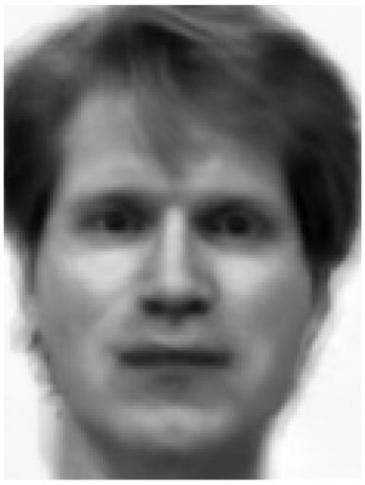
The code for reconstruction is as shown below

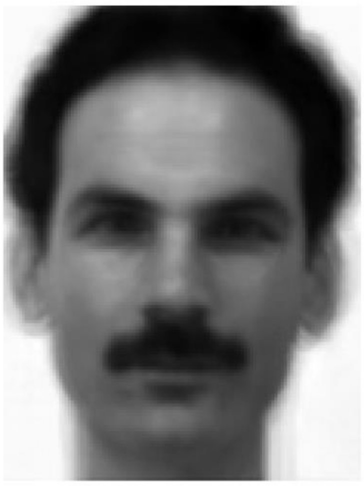
```
#Projections of each person's weights with the face space or the eigenfaces
    def show_face_class(self):
        dimX,dimY = self.avg_img.shape
        for weight in self.face_class:
```

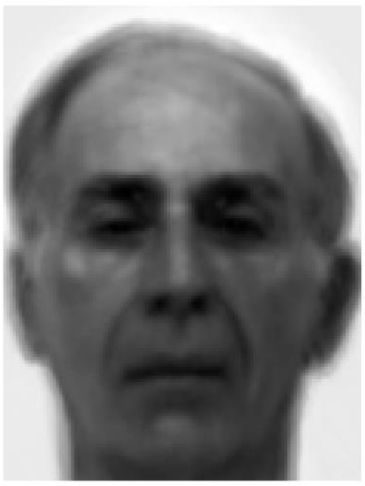
```
        img = np.dot(weight,self.eigenfaces)
        img = np.array(img,dtype='int32')
        img = np.reshape(img,(dimX,dimY))
        img = img + self.avg_img
        plt.axis('off')
        plt.imshow(img,cmap='gray')
        plt.show()
```

The projections are as shown below









g) Finding matches in the test set

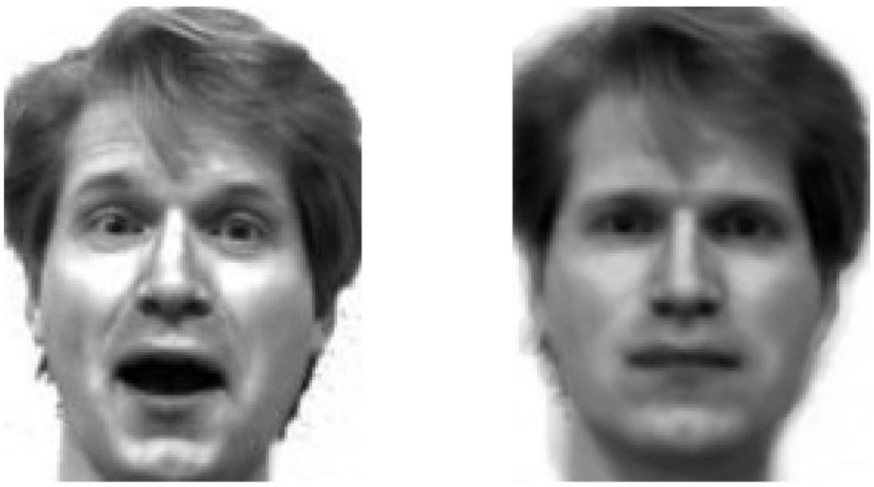
To find a match, I reconstruct the test image using the weights which I have calculated previously and find the Euclidean distance between the reconstructed image and the normalized original image (subtracted mean image). If the distance is higher than 8000, this is not considered to be a face and the message is shown appropriately. If it is lesser than the threshold, I check to see if it matches with any of the weight vectors representing a person. If the Euclidean distance between the test weights and weights of the face class are greater than the threshold, it means that the image was not recognized. If the image was recognized, the image is shown along with the original test image on a pyplot.

The code for this is as shown below

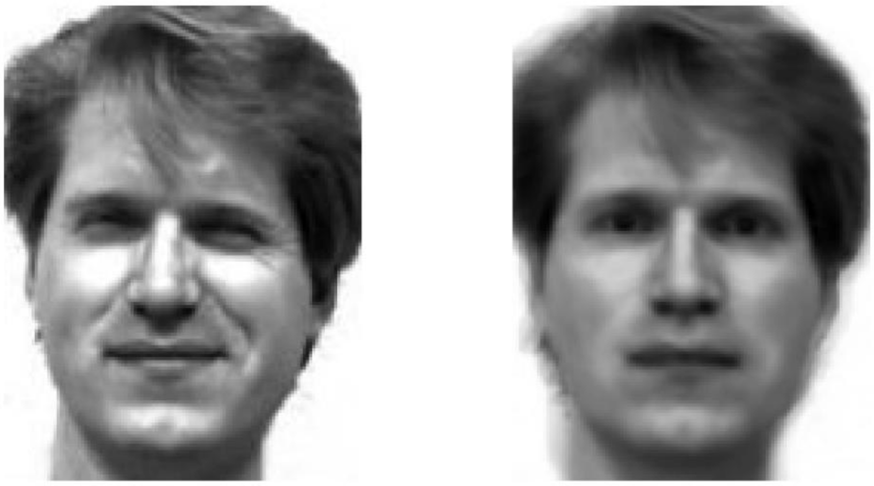
```python
#Class for test modules
class Testing:
    def __init__(self,module,img,face_thresh=8000,match_thresh=10000):
        self.eigenfaces = module.eigenfaces
        self.avg_img = module.avg_img
        self.face_class = module.face_class
        self.img = io.imread(img,as_gray=True)
        self.normalized_img = np.transpose(np.subtract(self.img,self.avg_img).
flatten())
        self.img.reshape((self.avg_img.shape[0],-1))
        self.projection()
        self.face_thresh = face_thresh
        self.match_thresh = match_thresh

    #Projection of input image onto the facespace to get the weights
    def projection(self):
        self.weights = self.eigenfaces @ self.normalized_img

    #Function for face recognition
    def find_match(self):
        index = 0

        #Detect if face or not
        projected_face = self.weights @ self.eigenfaces
        face_dist = distance.euclidean(self.normalized_img,projected_face)
        #print(face_dist)
        if face_dist > self.face_thresh:
            self.no_match(self.img,False)
            return

        #Detect if match or not
        min_distance = inf
        for ind,face in enumerate(self.face_class):
            dist = distance.euclidean(self.weights,face)
            if dist < self.match_thresh and dist < min_distance:
                index = ind
                min_distance = dist

        if min_distance == inf:
            self.no_match(self.img)
            return
        print(min_distance)
        #Display matched face
        matched_face = self.face_class[index] @ self.eigenfaces
        matched_face = np.reshape(matched_face,(self.avg_img.shape))
        matched_face = matched_face + self.avg_img
        self.show_face_match(self.img,matched_face)
    #Display function in case no match is found
    def no_match(self,img1,isFace = True):
        fig = plt.figure()
        if isFace:
            fig.suptitle("Face not recognized",fontsize=20)
        else:
            fig.suptitle("Face not detected",fontsize=20)
        plt.gray()
        plt.axis('off')
        plt.imshow(img1)
        plt.show()

    #Display function in case a match is found
    def show_face_match(self,img1, img2):
        fig = plt.figure()
        fig.suptitle("Matched!",fontsize=20)
        plt.subplot(1,2,1)
        plt.axis('off')
        plt.imshow(img1, cmap='gray')
        plt.subplot(1,2,2)
        plt.axis('off')
        plt.imshow(img2, cmap='gray')
        plt.show()
```

The resulting images are as shown below

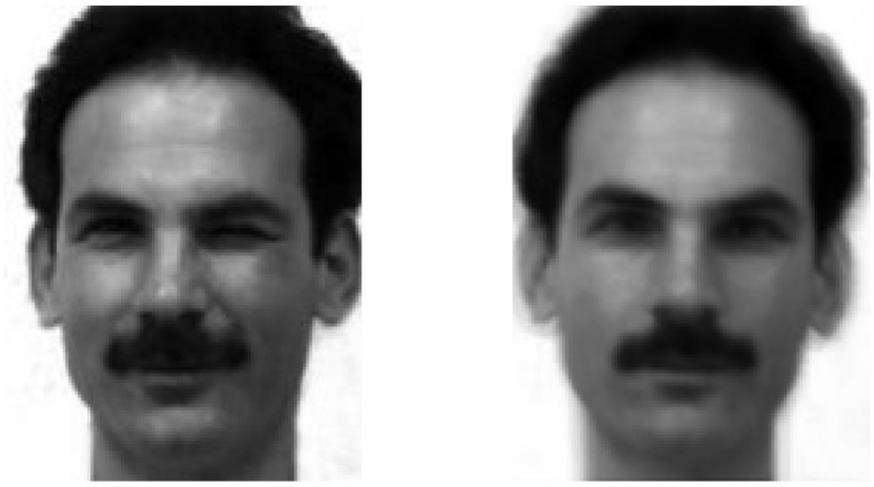
Matched!



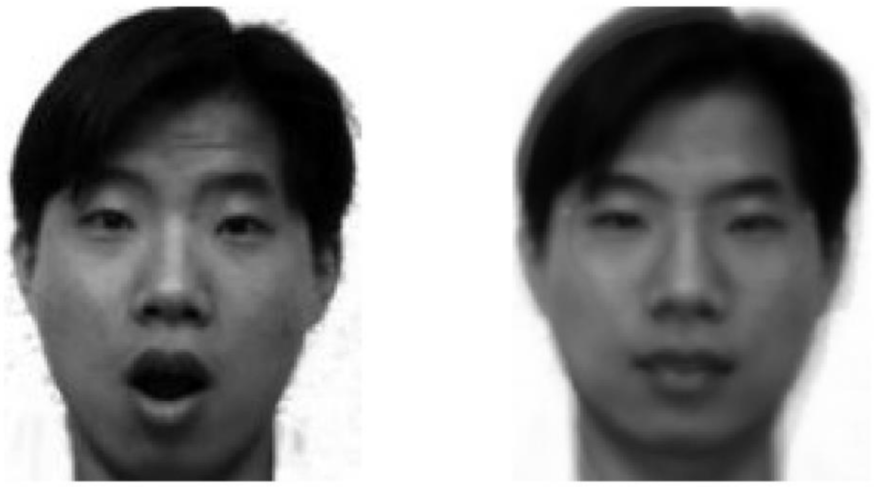
Matched!



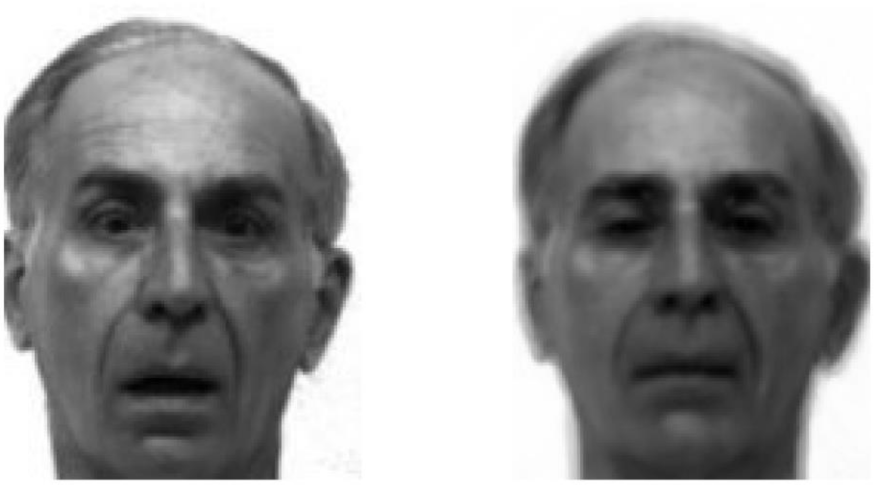
Matched!



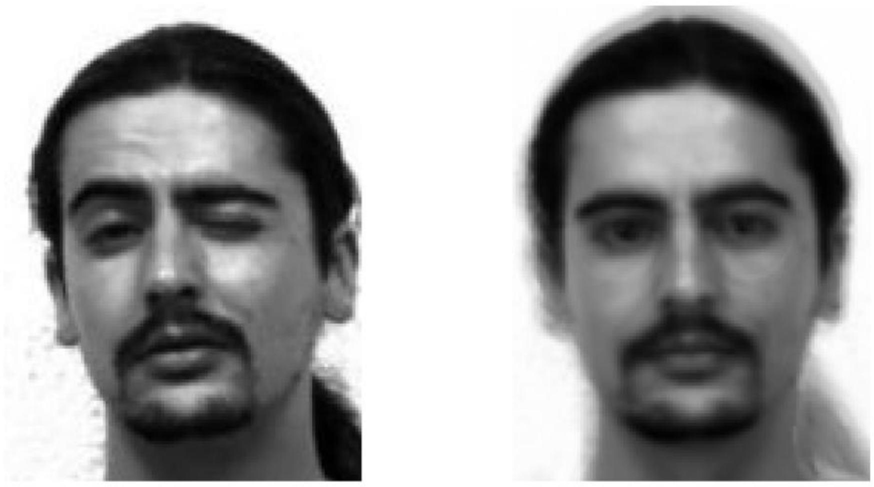
Matched!



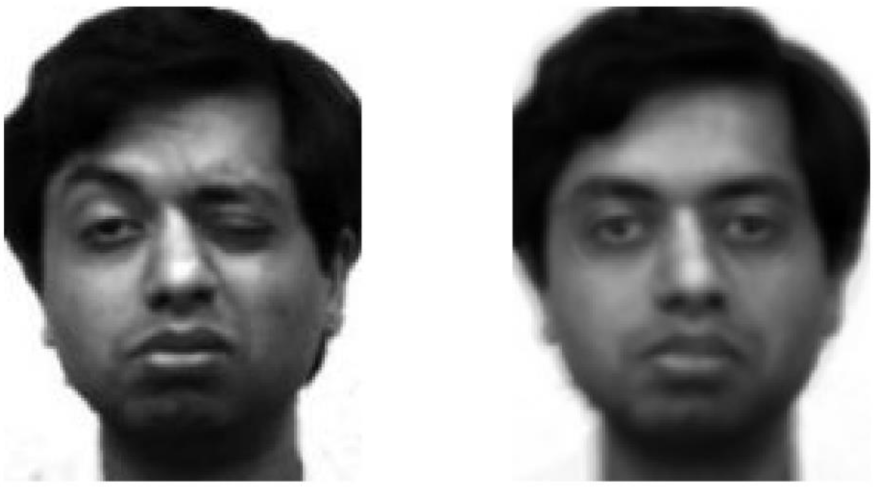
Matched!



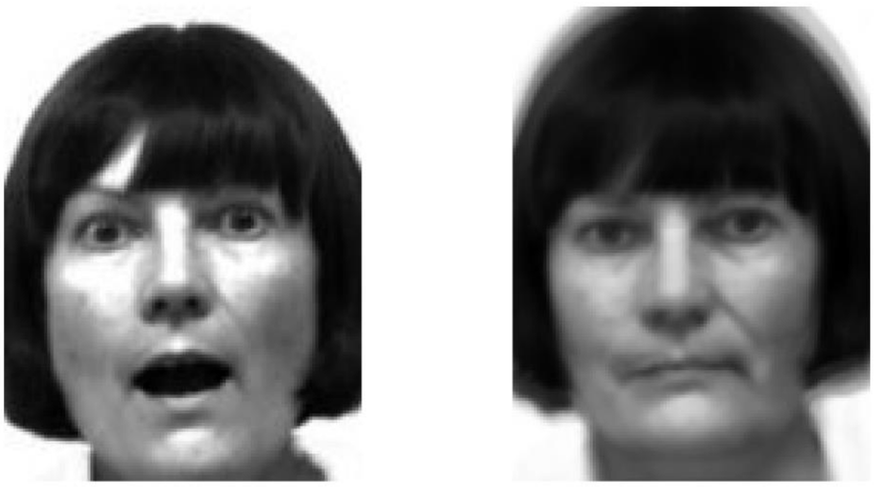
Matched!

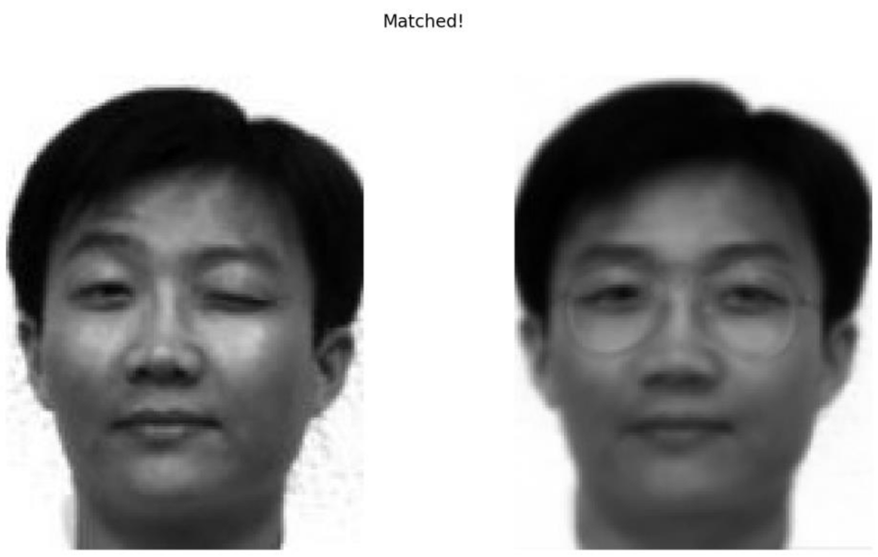


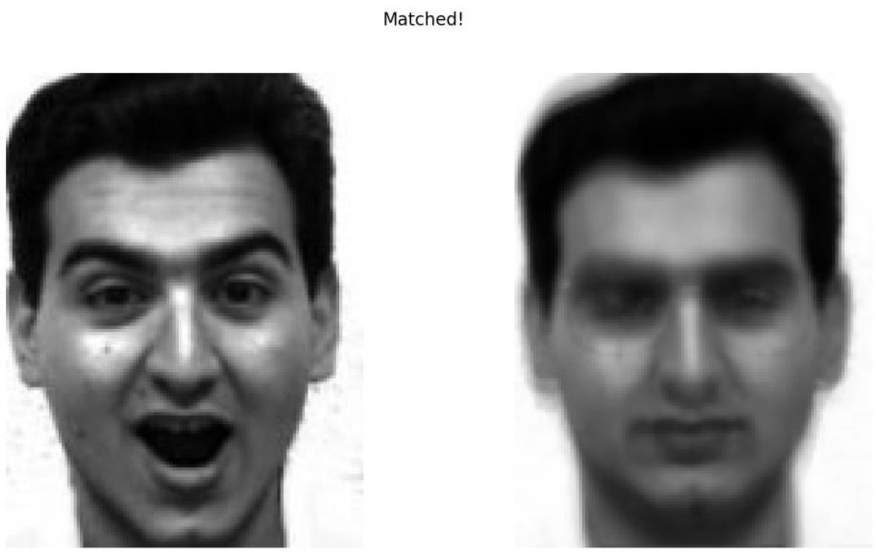
Matched!



Matched!







Part 2) Questions, evaluation and Comments

a) How many eigenfaces are necessary?
   In my case, I required a minimum of 10 eigenfaces to faithfully reproduce the original image and also recognize the images. Anything below that resulted in bad images or inaccurate recognitions

b) Calculate the recognition rate
   In my calculations, I found that there were 2 false positives among 30 tests and no non face image was recognized as a face. So there was a 93% accuracy with my tests. However, this also depends on the test images I use. With normal faces, I achieved 95% accuracy.

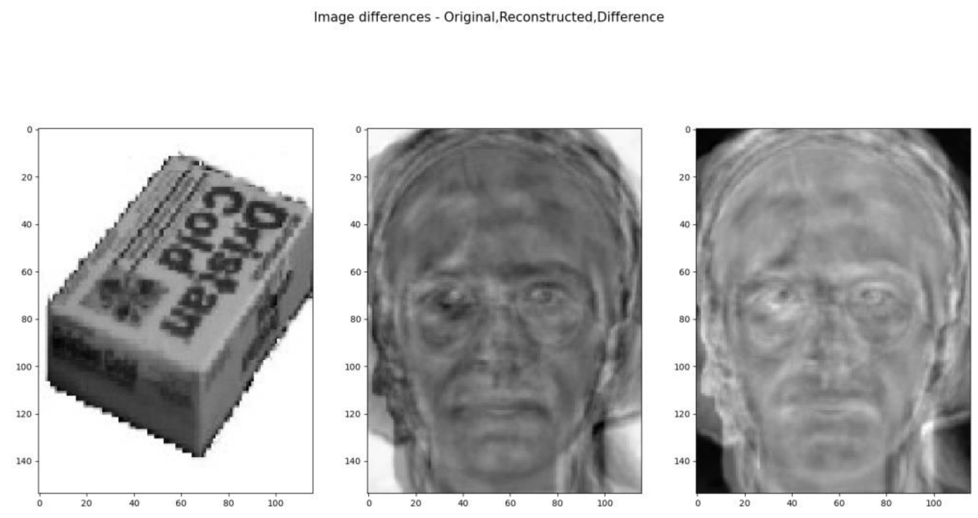
c) What happens when you do not include glasses in the experiment?
When I didn't include the images of glasses in the experiment, I did not see much of a difference. However, I believe that not including images with glasses will improve the rate of recognition since glasses hinder identifying the essential features of a face such as the eyes. Glasses may also contribute towards glares and other light dispersions causing noise in the image.

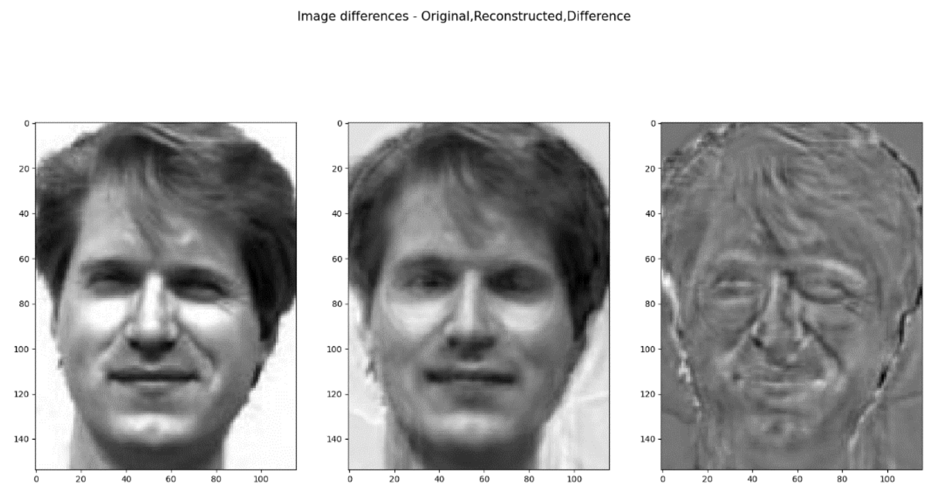
d and e) Find difference images

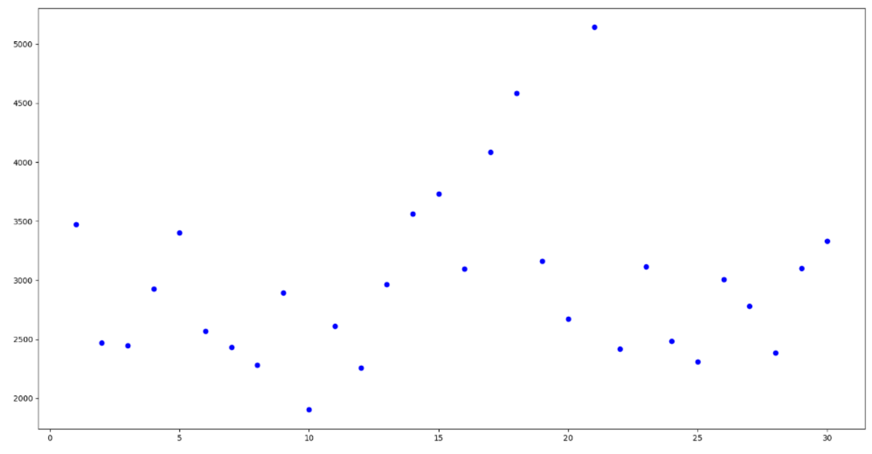
At the last of the test cases, I've added tests to show the difference between the original image and the reconstructed image
The code is as shown below

```
#Shows difference between Original and Reconstructed images
    def show_diff(self):
        diff_img = self.img - self.reconstructed_img
        fig = plt.figure()
        fig.suptitle("Image differences - Original,Reconstructed,Difference",f
ontsize=15)
        plt.subplot(1,3,1)
        plt.imshow(self.img, cmap='gray')
        plt.subplot(1,3,2)
        plt.imshow(self.reconstructed_img, cmap='gray')
        plt.subplot(1,3,3)
        plt.imshow(diff_img, cmap='gray')
        plt.show()
        frobenius_diff = LA.norm(diff_img)
        global frobenius_norm
        frobenius_norm.append(frobenius_diff)
```

Some examples are







Frobenius Norm