# report

September 27, 2019

## 1 CMSC426, Project 1, Report, Shiyuan Duan

### 1.1 Introduction

When processing images, programers always convert an image into a matrix. Usually the image matrix will be very large and hard to handle. In this project, we will show how to use PCA to reduce an 150*130 dimentional image into a lower dimention while keeping most of it's feature.

### 1.2 Eigen faces

Eigen faces of a face image dataset shows the vector in which way the data set has the largest variance. Eigen faces are the new basis we will be projecting our trainning image.

#### 1.2.1 Step1: Reading image files

```
In [1]: from matplotlib import pyplot as plt
        import matplotlib.image as img
        import numpy as np
        import os

        # number of images
        M = 3772

        # image dimension
        N1 = 150
        N2 = 130

        # Readin images and store it in A
        img_dirs = os.listdir('./Train1')
        A = []
        for img_dir in img_dirs:
            image = img.imread('./Train1/'+img_dir)
            image = np.reshape(image,(1,150*130))

            A.append(image[0])
        A = np.asarray(A)
        A = np.transpose(A)
```

Now images are read into a 15900*3772 matrix. We can then further process all the images in the training set.

### 1.2.2 Step 2: Obtain mean face

In the second step we have to obtain the mean value of the entire matrix and subtract mean from each element in the image matrix.

```
In [2]: mean_face = np.mean(A)
        A = A - mean_face
```

### 1.2.3 Step 3: Obtain eigen values and eigen vectors.

Next step is to obtain eigen vectors to get an idea of in which directions the dataset has to max variation. We start from getting the covariance matrix. Since $AA^T$ is too large to manipulate, we will be looking for eigen vectors and eigen values of $A^TA$ instead. Note that $AA^T$ and $A^TA$ share same eigen values and eigen vector of $AA^T$ is just that of $A^TA$ dot with A

```
In [3]: AT = np.transpose(A)
        cov1 = np.dot(AT,A)
```

Obtain eigen values as w and eigen vectors as v

```
In [4]: w, v = np.linalg.eig(cov1)
```

Obtain eigen vectors of $AA^T$, u are the eigen vectors representing directions with max variation.

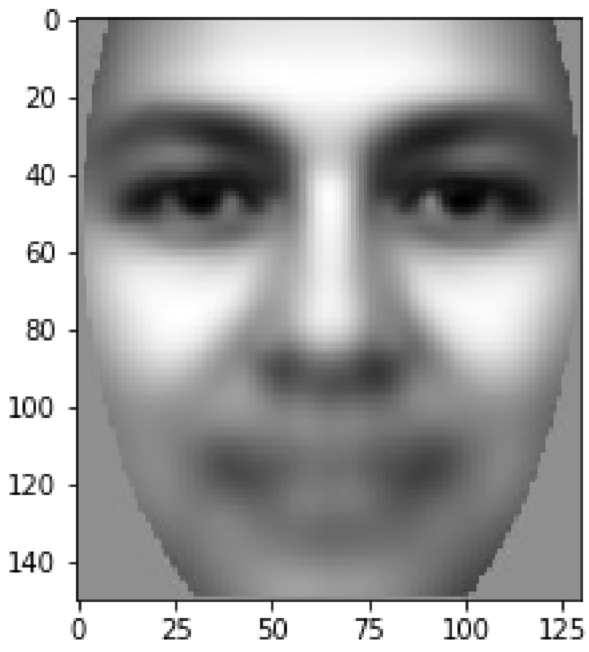
### 1.2.4 Step 4: Visualize eigen faces

```
In [5]: u = np.dot(A, v)
```

Now we can take a look at the eigen faces. Note that they are not normalized at this point but this does not matter since we are just visualizing it.

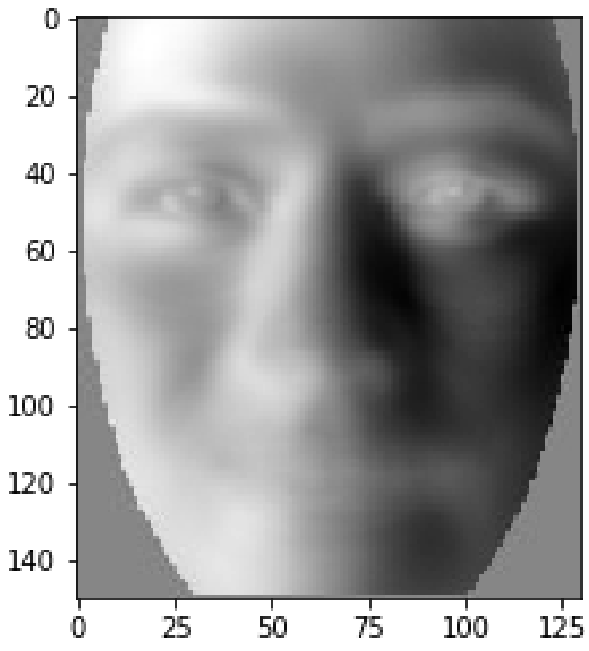
**Eigen face 1:**

```
In [6]: ef1 = np.reshape(u[:,0],(150,130))
        plt.imshow(ef1, interpolation='nearest') #display the image
        plt.gray()  #grayscale conversion
        plt.show()
```



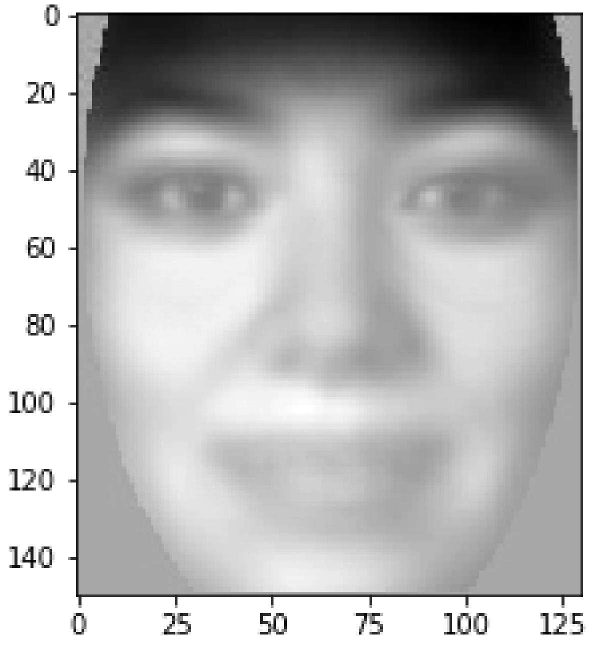
**Eigen face 2:**

```
In [7]: ef2 = np.reshape(u[:,1],(150,130))
        plt.imshow(ef2, interpolation='nearest') #display the image
        plt.gray()  #grayscale conversion
        plt.show()
```



**Eigen face 3:**

```
In [8]: ef3 = np.reshape(u[:,2],(150,130))
        plt.imshow(ef3, interpolation='nearest') #display the image
        plt.gray()  #grayscale conversion
        plt.show()
```



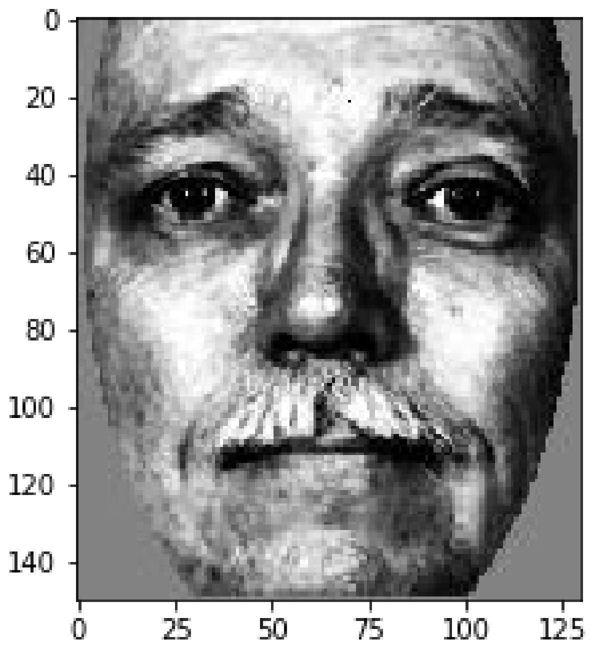
### 1.2.5 Step 5: Representing an image in eigen basis

In theory, we can project an image onto an eigen basis. For example if we project the original image onto the first K eigen vectors we are reducing the original image into a K dimensional space. Technically the larger the K is the better but the concept of PCA is to reduce a image from a high dimensional space to a low dimenstional space. Therefore we will be looking at a K that keeps most of it's features but is low enough so we can manipulate.

Basically we are trying to see how are the original image and the new image with only K dimention complexity compare.

**Original image:**

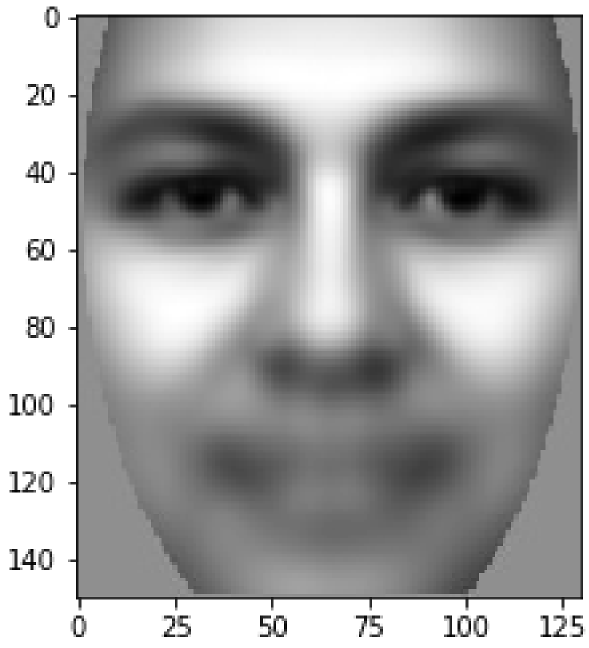
```
In [9]: f1 = np.reshape(A[:,0],(150,130))
        plt.imshow(f1, interpolation='nearest') #display the image
        plt.gray()  #grayscale conversion
        plt.show()
```



**K=1** When K=1 we are just taking 1 dimensional into account and let's see how will this compare to the original image

```
In [11]: K = 1
         testing_face = 0

         w_truncated = w[0:K]
         u_truncated = u[:,0:K]

         weights = []
         for i in range(0,K):
             u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
             weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,testing_face],(150*130
             weights.append(weight[0][0])



         combined_face = np.zeros([150*130,1],dtype='uint8')

         for j in range(0,K):
             eigen_v = np.reshape(u[:,j],(150*130,1))
             eigen_v = eigen_v / np.linalg.norm(eigen_v)
             w_j = weights[j]
             combined_face = combined_face + w_j*eigen_v
```

```
        combined_face = np.reshape(combined_face,(150,130))
        plt.imshow(combined_face, interpolation='nearest') #display the image
        plt.gray()  #grayscale conversion
        plt.show()
```

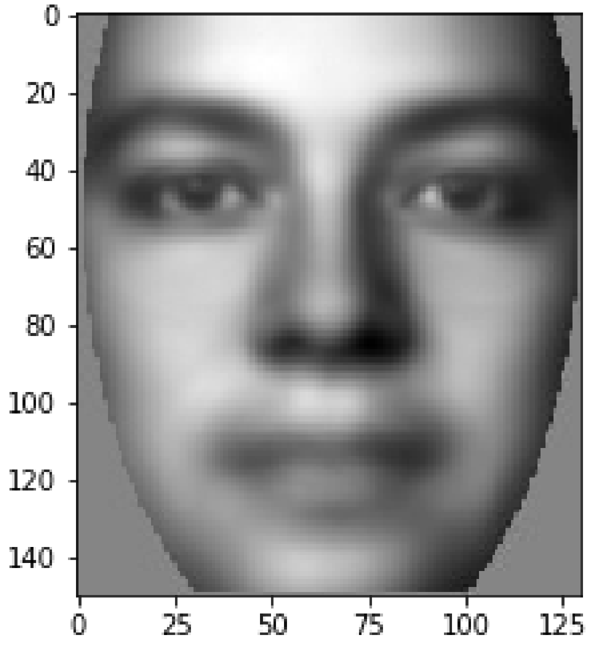


We can see that this is just the first eigen face. Now, let's increase dimenstions of feature we take into consideration.

**K=10**

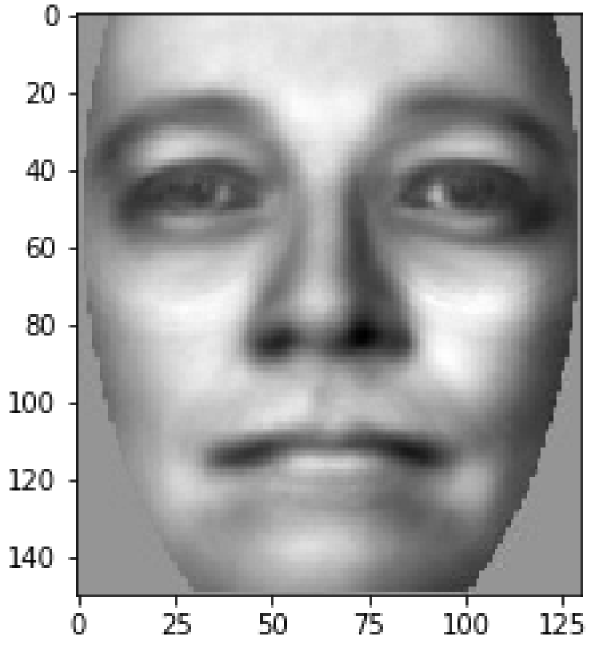
```
In [12]: K = 10
        testing_face = 0

        w_truncated = w[0:K]
        u_truncated = u[:,0:K]

        weights = []
        for i in range(0,K):
            u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
            weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,testing_face],(150*130
            weights.append(weight[0][0])
```

```
combined_face = np.zeros([150*130,1],dtype='uint8')

for j in range(0,K):
    eigen_v = np.reshape(u[:,j],(150*130,1))
    eigen_v = eigen_v / np.linalg.norm(eigen_v)
    w_j = weights[j]
    combined_face = combined_face + w_j*eigen_v



combined_face = np.reshape(combined_face,(150,130))
plt.imshow(combined_face, interpolation='nearest') #display the image
plt.gray()  #grayscale conversion
plt.show()
```



**K = 100**

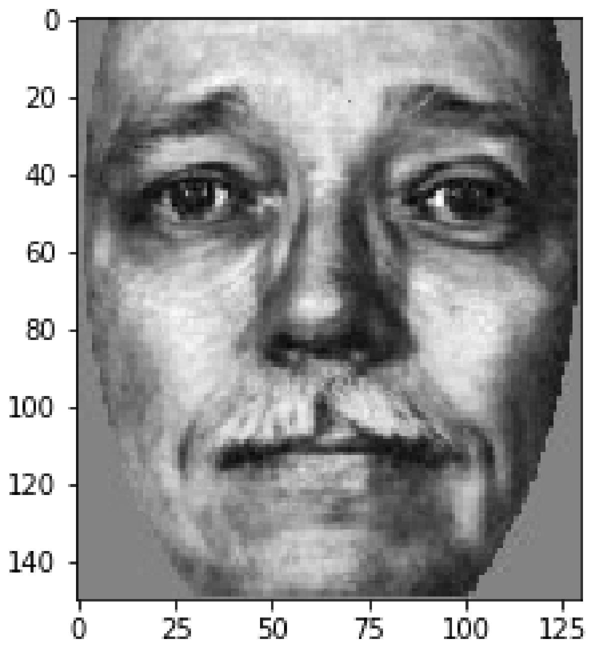
In [13]:
```
K = 100
testing_face = 0

w_truncated = w[0:K]
u_truncated = u[:,0:K]

weights = []
for i in range(0,K):
    u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
    weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,testing_face],(150*130
    weights.append(weight[0][0])



combined_face = np.zeros([150*130,1],dtype='uint8')

for j in range(0,K):
    eigen_v = np.reshape(u[:,j],(150*130,1))
    eigen_v = eigen_v / np.linalg.norm(eigen_v)
    w_j = weights[j]
    combined_face = combined_face + w_j*eigen_v



combined_face = np.reshape(combined_face,(150,130))
plt.imshow(combined_face, interpolation='nearest') #display the image
plt.gray()  #grayscale conversion
plt.show()
```



**K=1000**

```
In [15]: K = 1000
         testing_face = 0
```

```
w_truncated = w[0:K]
u_truncated = u[:,0:K]

weights = []
for i in range(0,K):
    u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
    weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,testing_face],(150*130
    weights.append(weight[0][0])



combined_face = np.zeros([150*130,1],dtype='uint8')

for j in range(0,K):
    eigen_v = np.reshape(u[:,j],(150*130,1))
    eigen_v = eigen_v / np.linalg.norm(eigen_v)
    w_j = weights[j]
    combined_face = combined_face + w_j*eigen_v



combined_face = np.reshape(combined_face,(150,130))
plt.imshow(combined_face, interpolation='nearest') #display the image
plt.gray()  #grayscale conversion
plt.show()
```



We can see that when we are adding up the first 1000 eigen faces with specific weights, we are almost getting the exact same picture. The original image has a dimension if 19500 while the one above only has dimension of 1000 and it is much easier to manipulate. This is the idea of PCA: Reduce the dimenstion while keeping most of it's features.

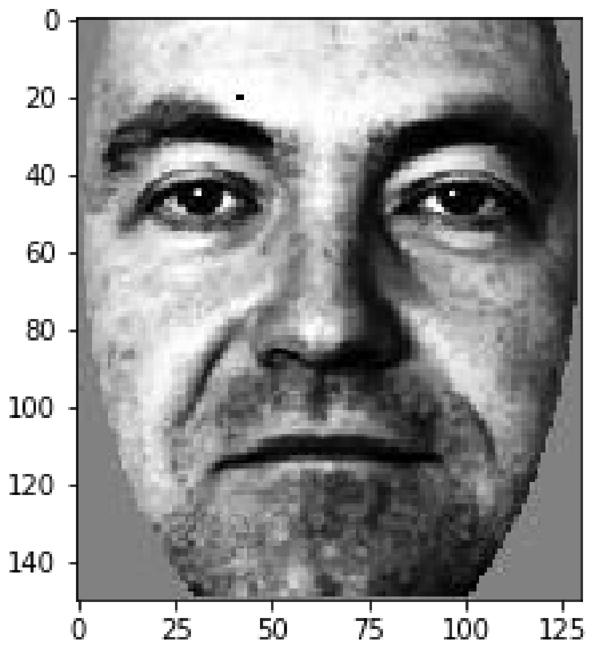
## 1.3 Face recognition.

After reducing the image dimension we can then do face recognition. For example if we are given an image we can project it onto our eigen basis obtained from training set and obtain weights coresponding to each eigen vector. We then can compare the weigts of the test image with each image in the training set. The closest one will be a match.

### 1.3.1 Test image

```
In [17]: ## Reading in test set
         N1 = 150
         N2 = 130

         # Readin images and store it in A
         img_dirs = os.listdir('./Test1')
         B = []
         for img_dir in img_dirs:
             image = img.imread('./Test1/'+img_dir)
             image = np.reshape(image,(1,150*130))

             B.append(image[0])
         B = np.asarray(B)
         B = np.transpose(B)
```

```
In [19]: ## Visualizing test image
         image_testing = 10
         rand_p = B[:,image_testing]
         rand_p = np.reshape(rand_p,(150,130))
         plt.imshow(rand_p, interpolation='nearest') #display the image
         plt.gray()  #grayscale conversion
         plt.show()
```

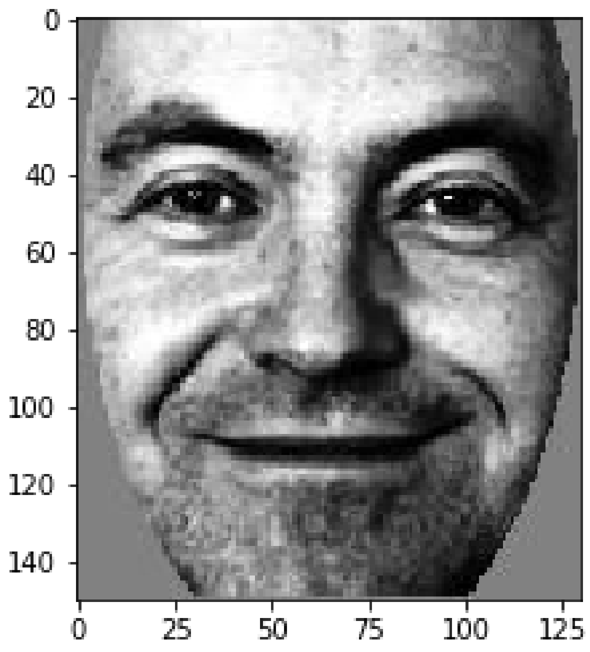


```
In [24]: # Projecting test image onto eigen basis
         K = 100
         test_weights = []
         for i in range(0,K):
             u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
             weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(B[:,10],(150*130,1)))
             test_weights.append(weight[0][0])

In [25]: # Calculate weights and compare it with the training set, return the match image's in
         import math
         min_val = math.inf
         min_indx = -1
         for j in range(3772):
             weights = []
             for i in range(K):
                 u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
                 weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,j],(150*130,1)))
                 weights.append(weight[0][0])

             diff = np.linalg.norm(np.array(weights) - np.array(test_weights))
             if diff < min_val:
                 min_val = diff
                 min_indx = j

         print(min_indx)
```

```
2642
```

after we have obtained the match image we can visualize it to test if it's a good match

```
In [26]: match_img = A[:,min_indx]
         match_img = np.reshape(match_img,(150,130))
         plt.imshow(match_img, interpolation='nearest') #display the image
         plt.gray()  #grayscale conversion
         plt.show()
```

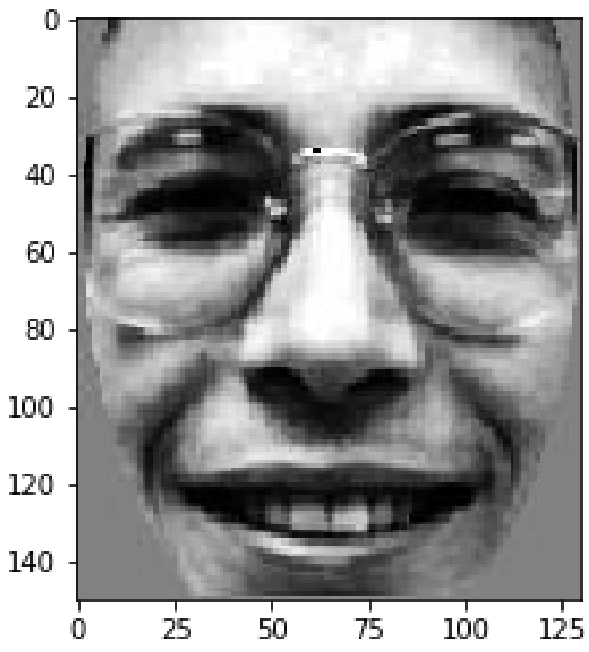


This image is the same person as the test image. We can see that after the pca processes we can still find a perfect match. This is because pca keeps most of the original features.

### 1.3.2 Accuarcy

With a K value of 100, this method will not always accuarte. When K = 100, our method will only give us a match that they look similar but things like winkles are ignored. For example if we are looking for a match for the following image

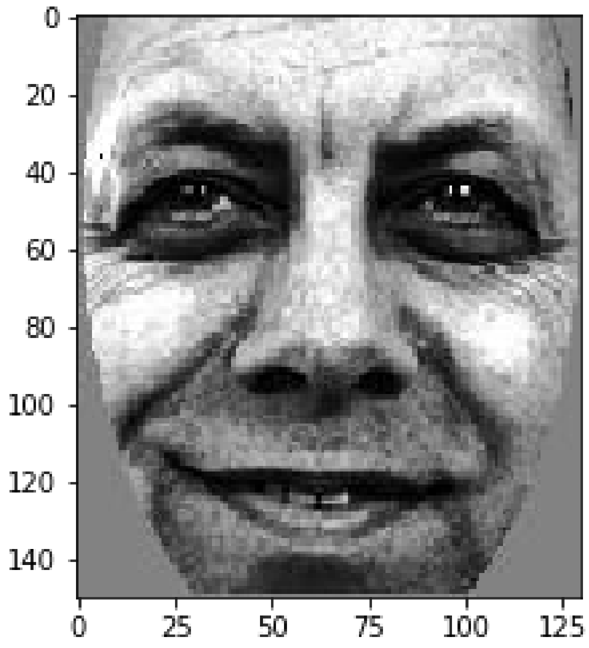
```
In [27]: image_testing = 0
         rand_p = B[:,image_testing]
         rand_p = np.reshape(rand_p,(150,130))
         plt.imshow(rand_p, interpolation='nearest') #display the image
         plt.gray()  #grayscale conversion
         plt.show()
```



Our method gives us a match image shown below

```
In [28]: K = 100
         test_weights = []
         for i in range(0,K):
             u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
             weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(B[:,image_testing],(150*13
             test_weights.append(weight[0][0])

         import math
         min_val = math.inf
         min_indx = -1
         for j in range(3772):
             weights = []
             for i in range(K):
                 u_i = u_truncated[:,i]/np.linalg.norm(u_truncated[:,i])
                 weight = np.dot(np.reshape(u_i,(1,150*130)),np.reshape(A[:,j],(150*130,1)))
                 weights.append(weight[0][0])

             diff = np.linalg.norm(np.array(weights) - np.array(test_weights))
             if diff < min_val:
                 min_val = diff
                 min_indx = j

         match_img = A[:,min_indx]
         match_img = np.reshape(match_img,(150,130))
```

```
plt.imshow(match_img, interpolation='nearest') #display the image
plt.gray()  #grayscale conversion
plt.show()
```



To increase the accuarcy of this algrithm, the best way to do is to increase the eigen vectors we are using. In the previous problem we are using k=100 to save time. We can increase it to 1000 and leave it there for half an hour we probably will find a better match. Another way is to add more images to the training set. Maybe we are not finding a perfect match is because there's not enought training image in the training set

## 1.4 Conclusion

From this project I learned how to reduce dimension of a dataset with PCA algrithm while keeping most of it's features. This project is a perfect demonstrating of PCA's properties. After PCA we reduced the complexicty significantly and still be able to manipulate it as if it were the original image because PCA keeps most of it's features.

```
In [ ]:
```