Eigenfaces

November 30, 2016

0.1 Eigenfaces

Eigenfaces is an excercise from the book: "Coding the Matrix (Linear Algebra through Applications to Computer Science)" By Philip N. Klein - Brown University
Author of Script: Giordi Azonos

```
In [1]: #eigenFacesFucntions.py is the script that contains all the necessary funct
        import eigenFacesFunctions as myfunc
        import numpy as np
        import numpy.linalg as la
        import matplotlib.pyplot as pt
        %matplotlib inline
```

```
In [2]: height = 189
        width = 166
```

```
In [3]: def displayFace(pixels_array,as_vector=True, Title='Face'):
            """
            Displays a grayscale matplotlib plot of the pixels 1-D vector,
            with the given 'Tile'. The pixels vector can also be given as a
            pixels array by flagging the 'as_vector' variable as False.
            """
            if as_vector == True:
                face = pixels_array.reshape((height,width))
            else:
                face = pixels_array
            new_fig = pt.figure()
            new_fig.suptitle(Title, fontsize=14, fontweight='bold')
            new_plot = new_fig.add_subplot(111)
            new_plot.imshow(face, cmap='gray')
```

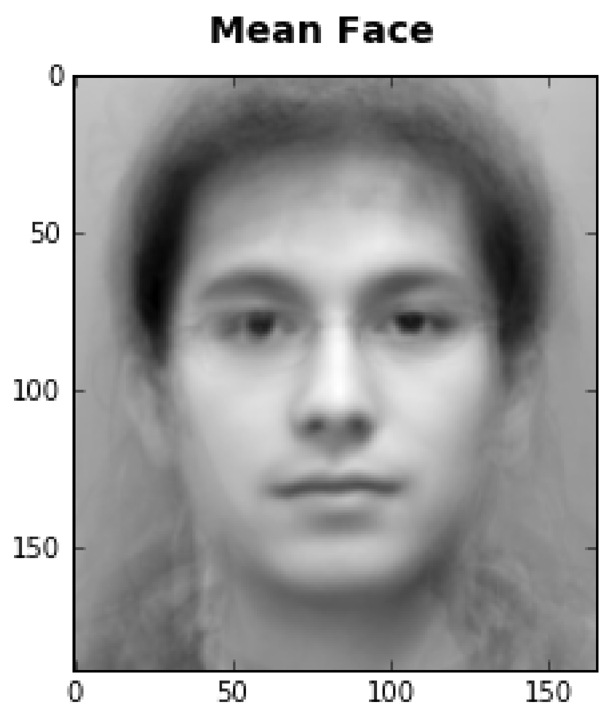
```
In [4]: def reconstructFaceStepbyStep(originalface_vec, weights_vec, e_faces):
            """
            Receives:
                .A vector of the face you are trying to re-construct named 'origina
                .A weights vector that corresponds to the image you want to re-cons
                .An e_faces matrix that corresponds to the singular values subspace
```

```
        Displays the re-construction of the given face singular vector by singu
        that is, it first uses just one singular vector, prints the ratio of th
        reconstruction vs the original face, and plots the re-constructed image
        the original image. And so on for the 20 singular vectors.
        """
        for n in range(1,21):
            # approximation of image using n eigenvectors
            approximation = np.dot(weights_vec[0:n], e_faces[:, 0:n].T)
            # Compute the distance from the approximation to the subspace of ei
            distance = np.sum(approximation.dot(e_faces)**2)
            ratio = distance/la.norm(originalface_vec)**2
            print ('    Ratio using '+str(n)+' singular vectors = ' +str(ratio)

            if(n==1 or n%5 == 0):
                #I dont want to plot all faces, just a few examples.
                fig = pt.figure(figsize=(20,10))
                fig.suptitle('Original vs Approximation', fontsize=14, fontweig

                plot1 = fig.add_subplot(121)
                plot1.set_title('Mean Adjusted Original Face')
                plot1.imshow(originalface_vec.reshape((height,width)), cmap='gr

                plot2 = fig.add_subplot(122)
                plot2.set_title('Reconstructed with '+str(n)+' singular vectors
                plot2.imshow(approximation.reshape((height,width)), cmap='gray'
```

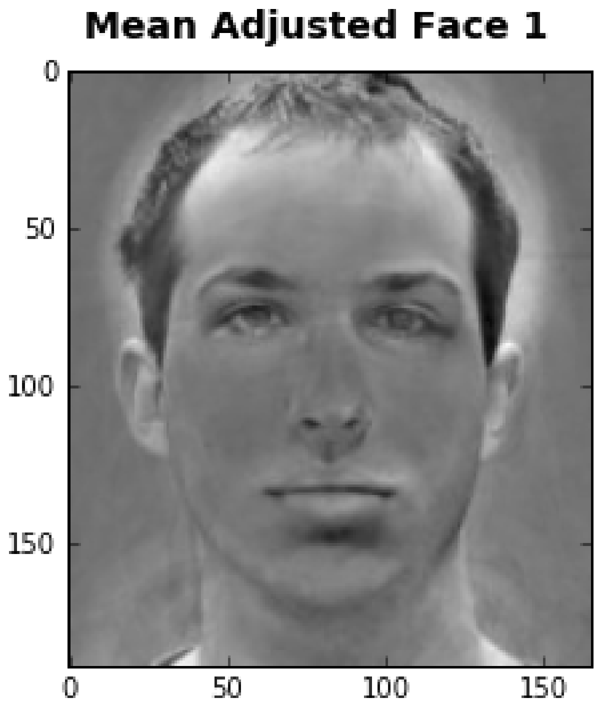
In [5]:
```
# Path where the face images are stored.
faces_path = '/Users/user/Documents/MyProjects/Lab Eigenfaces/faces'
unclassif_faces_path = '/Users/user/Documents/MyProjects/Lab Eigenfaces/unc
```

In [6]:
```
# 1. Load Images to Python. Images are loaded as 1-D vectors of pixels unle
# stated otherwise in the 'as_vector' variable.
# The resulting array has each row as a vector of pixels for each image.
face_images = myfunc.loadImages(faces_path, as_vector=True)
maybe_faces = myfunc.loadImages(unclassif_faces_path, as_vector=True)
```

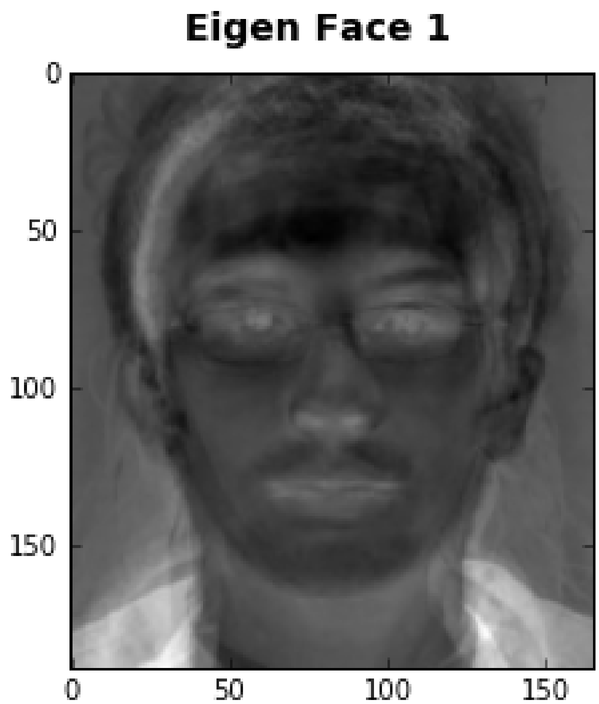
In [7]:
```
# "Average" face
mu = np.mean(face_images, 0)
displayFace(mu, Title='Mean Face')
```

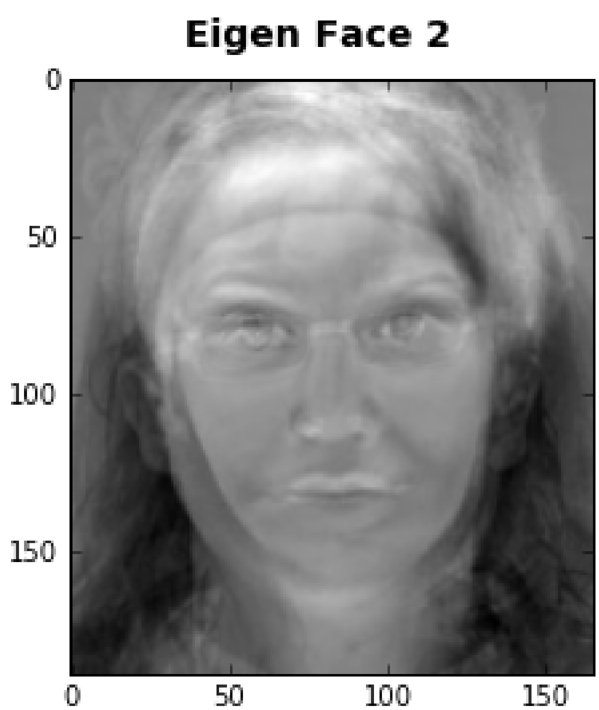


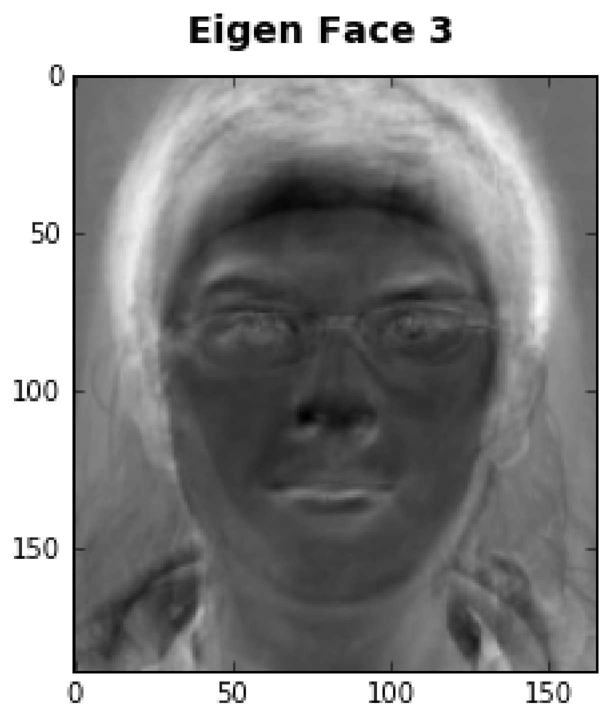

```
In [8]: # Mean adjusted Data.
        ma_data = face_images - mu
        ma_maybe_faces = maybe_faces - mu
        displayFace(ma_data[0], Title='Mean Adjusted Face 1')
```




```
In [9]: # Perform the SVD decomposition over the mean adjusted array of data.
        # The columns of U form an orthonormal basis for the eigenspace of the cova
        # matrix. Each col of U is a singular vector corresponding to an eigenface.
        U, S, V = la.svd(ma_data.transpose(), full_matrices=False)
        e_faces = U
        all_weights = np.dot(ma_data, e_faces) # Each row corresponds to the weight

In [10]: # Plot the First Three eigenfaces
         # First three eigen faces are the ones that explain most part of the varia
         displayFace(U[:,0], Title='Eigen Face 1')
         displayFace(U[:,1], Title='Eigen Face 2')
         displayFace(U[:,2], Title='Eigen Face 3')
```


Eigen Face 1
0
50
100
150
0
50
100
150


Eigen Face 2
0
50
100
150
0
50
100
150




```
In [11]: # The columns of U are in decreasing importance, with the first column bei
         # eigen faces with the most importance, that is, the one that explains the
         # in the space of faces, and so on. Hence we can work with just the first
         n=10
         n_e_faces = U[:,0:n]
```

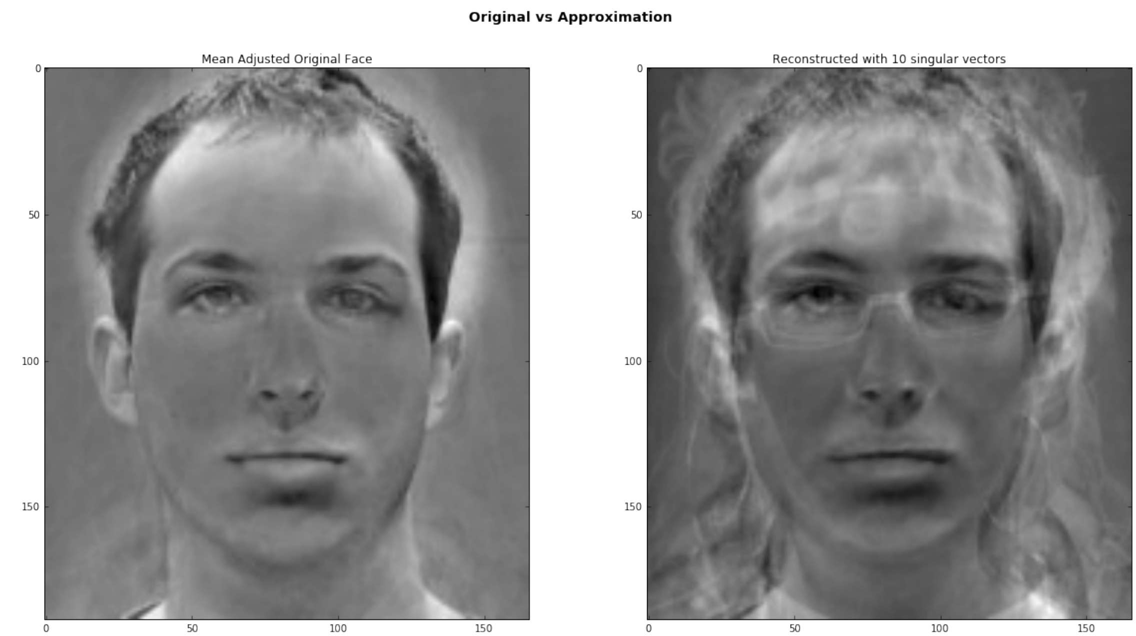
```
In [12]: # Reconstruct eigenvalue-by-eigenvalue face at img_ix
         img_ix=0
         reconstructFaceStepbyStep(ma_data[img_ix], all_weights[img_ix], e_faces)
```

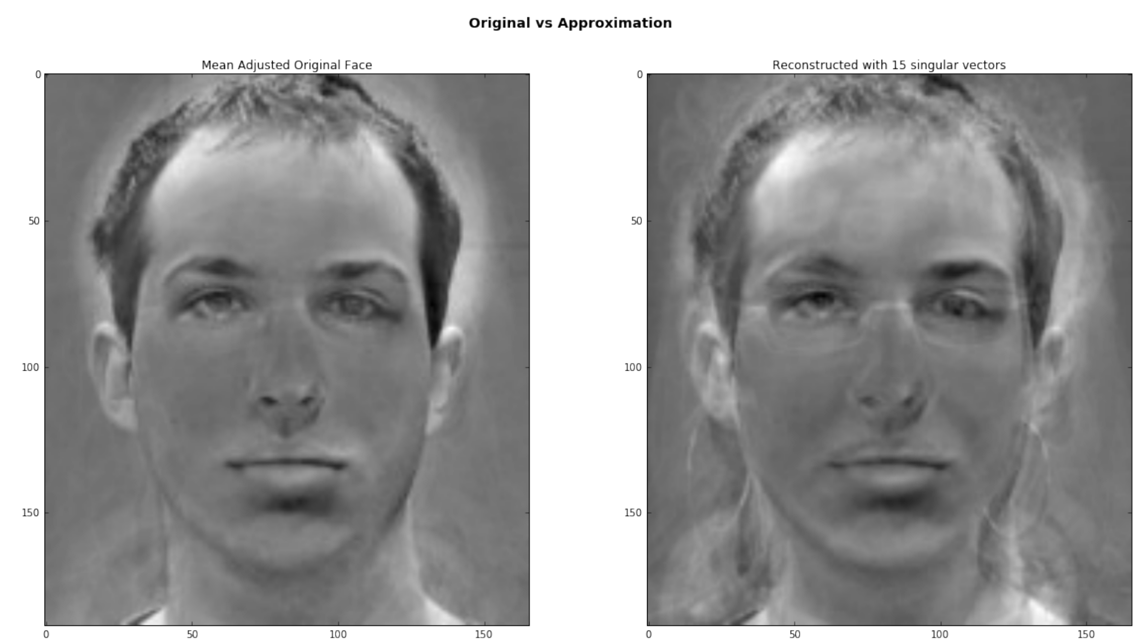
```
Ratio using 1 singular vectors = 0.508289702842
Ratio using 2 singular vectors = 0.532456460904
Ratio using 3 singular vectors = 0.537769003797
Ratio using 4 singular vectors = 0.5857712944
Ratio using 5 singular vectors = 0.728656767571
Ratio using 6 singular vectors = 0.769996706577
Ratio using 7 singular vectors = 0.784609062864
Ratio using 8 singular vectors = 0.8285391835
Ratio using 9 singular vectors = 0.828577520097
Ratio using 10 singular vectors = 0.833730983477
Ratio using 11 singular vectors = 0.836635193049
Ratio using 12 singular vectors = 0.838510090524
Ratio using 13 singular vectors = 0.856664869185
Ratio using 14 singular vectors = 0.895774089674
```

```
Ratio using 15 singular vectors = 0.899489511991
Ratio using 16 singular vectors = 0.931360151677
Ratio using 17 singular vectors = 0.933233530906
Ratio using 18 singular vectors = 0.999334889607
Ratio using 19 singular vectors = 1.0
Ratio using 20 singular vectors = 1.0
```

Original vs Approximation

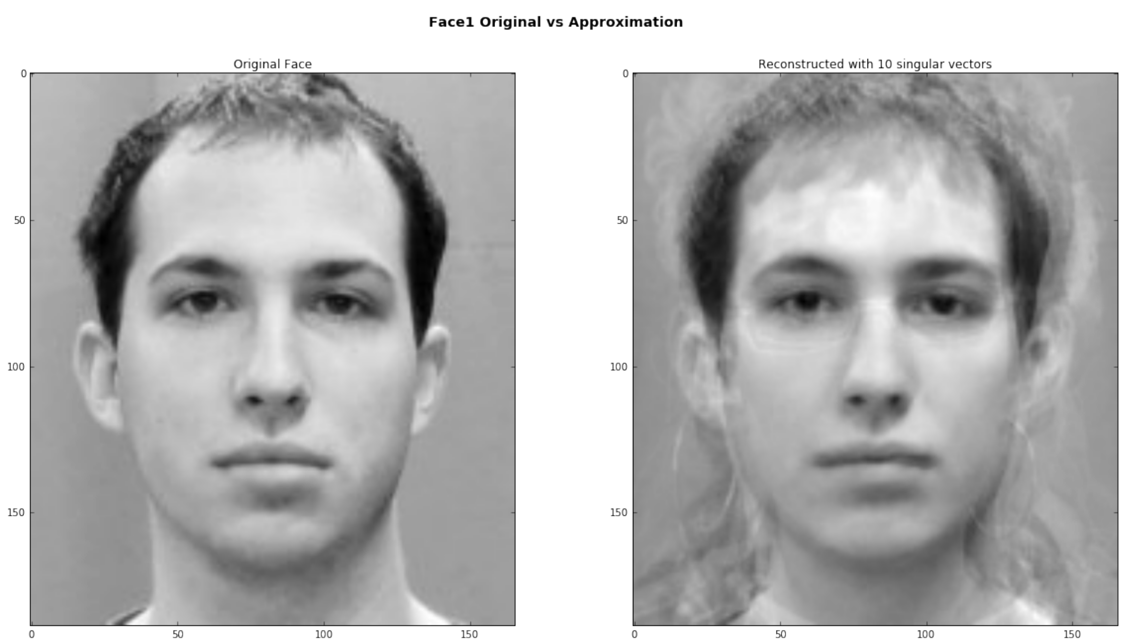
Original vs Approximation


Original vs Approximation
Mean Adjusted Original Face
Reconstructed with 10 singular vectors
0
50
100
150


Original vs Approximation
Mean Adjusted Original Face
Reconstructed with 15 singular vectors
0
50
100
150


```
In [13]: #Reconstruction of Faces
         # Attempt to reconstruct a face using the first 10 most significant eigen
         # This is important for image compression. You can just save the most impo
         # eigen vectors, and then reconstruct the image from this eigen vectors, h
         # the SVD decomposition is an expensive algorithm.
         img_ix = 0
         average_distance = 0
         display_images = True
         for image in ma_data:
             almost_face = myfunc.approximate(image, n_e_faces)

             dist = myfunc.distanceToEigenspace(image, n_e_faces.T)/1000000
             average_distance += dist
             print('"Distance" of image'+str(img_ix+1)+' to '+str(n)+' e_faces = '+

             if (display_images and img_ix % 5 == 0):
                 # img%5==0 because I dont want to plot all faces, just a few examp
                 fig = pt.figure(figsize=(20,10))
                 fig.suptitle('Face'+str(img_ix+1)+' Original vs Approximation', fo

                 plot1 = fig.add_subplot(121)
                 plot1.set_title('Original Face')
                 plot1.imshow((image+mu).reshape((height,width)), cmap='gray')

                 plot2 = fig.add_subplot(122)
                 plot2.set_title('Reconstructed with '+str(n)+' singular vectors')
                 plot2.imshow((almost_face+mu).reshape((height,width)), cmap='gray'
```

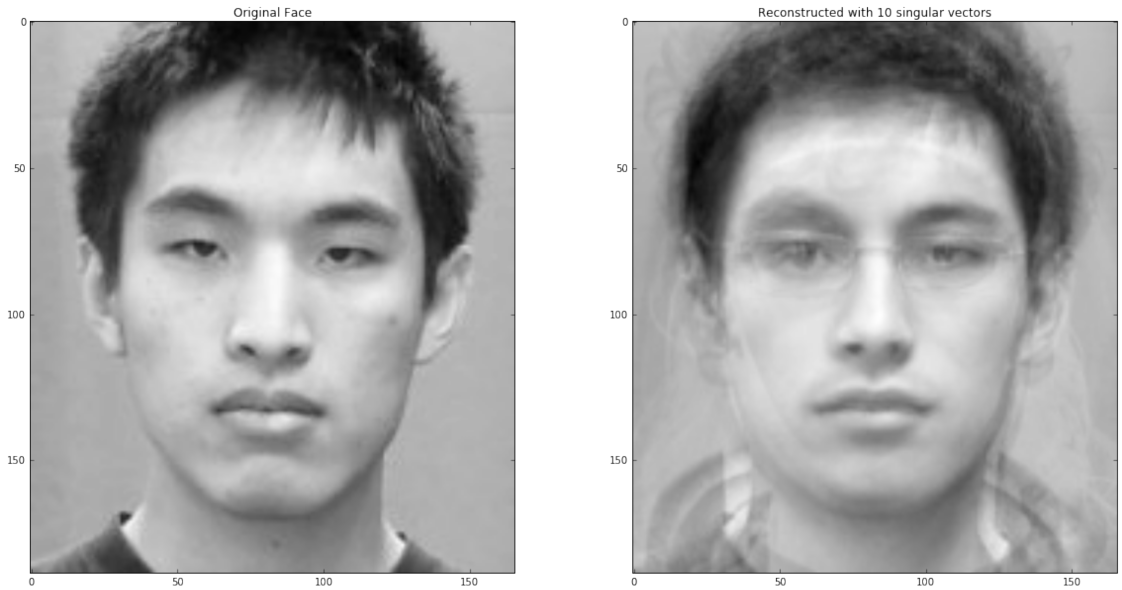
```
            img_ix += 1
        average_distance /= ma_data.shape[0]
        print ('Average Distance ='+str(format(round(average_distance), ',f')))

"Distance" of image1 to 10 e_faces = 8.000000
"Distance" of image2 to 10 e_faces = 16.000000
"Distance" of image3 to 10 e_faces = 3.000000
"Distance" of image4 to 10 e_faces = 5.000000
"Distance" of image5 to 10 e_faces = 2.000000
"Distance" of image6 to 10 e_faces = 2.000000
"Distance" of image7 to 10 e_faces = 2.000000
"Distance" of image8 to 10 e_faces = 4.000000
"Distance" of image9 to 10 e_faces = 9.000000
"Distance" of image10 to 10 e_faces = 10.000000
"Distance" of image11 to 10 e_faces = 6.000000
"Distance" of image12 to 10 e_faces = 3.000000
"Distance" of image13 to 10 e_faces = 13.000000
"Distance" of image14 to 10 e_faces = 6.000000
"Distance" of image15 to 10 e_faces = 7.000000
"Distance" of image16 to 10 e_faces = 10.000000
"Distance" of image17 to 10 e_faces = 9.000000
"Distance" of image18 to 10 e_faces = 6.000000
"Distance" of image19 to 10 e_faces = 12.000000
"Distance" of image20 to 10 e_faces = 7.000000
Average Distance =7.000000
```



Face6 Original vs Approximation

Face11 Original vs Approximation

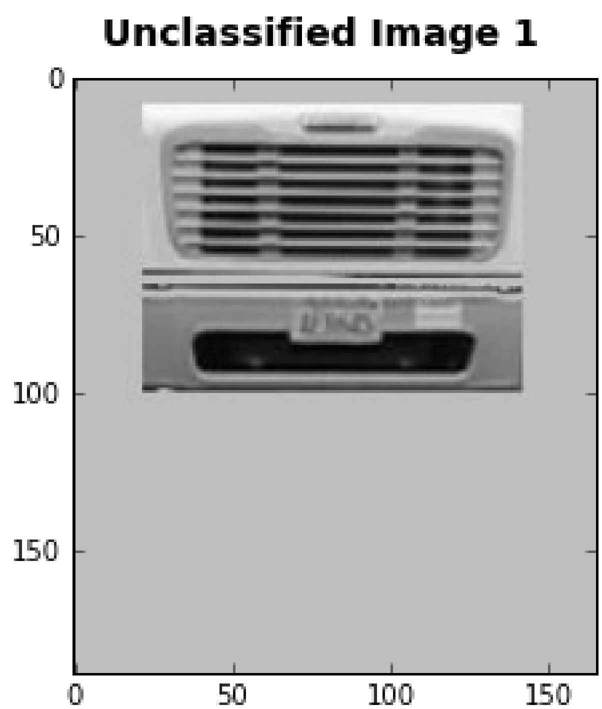
Face16 Original vs Approximation



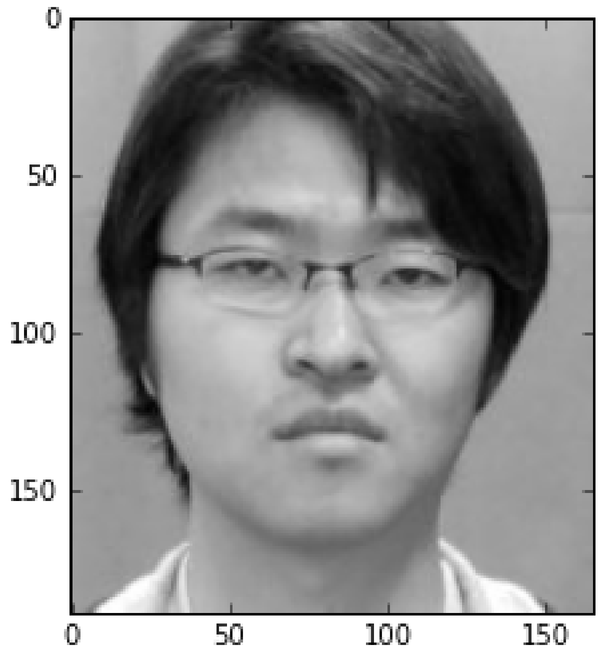

```
In [14]: # Classification of Maybe Faces:
         # Now we will work with some images that may be faces, or may not be faces
         # as faces or not faces.
         n=10
         # After analizing the maybe faces data set, I reached the conclusion that
         # a distance of the image to the eigen faces subspace greater than 40,000,
         # implies that the image is not a face. From our data set, every image tha
         # is a face has a distance to the eigen faces subspace less than 40,000,0(
         threshold = 40
         display_images = True
         img_ix = 0
         for image in ma_maybe_faces:
             if display_images: displayFace(image+mu, Title='Unclassified Image '+s

             dist = myfunc.distanceToEigenspace(image, n_e_faces.T)/1000000
             print('"Distance" of image'+str(img_ix+1)+' to '+str(n)+' e_faces = '+

             if dist > threshold:
                 print ('=> Image '+str(img_ix+1)+' is not a face.')
             else:
                 print ('=> Image '+str(img_ix+1)+' is a face.')

             img_ix += 1
```

```
"Distance" of image1 to 10 e_faces = 58.000000
=> Image 1 is not a face.
"Distance" of image2 to 10 e_faces = 39.000000
```

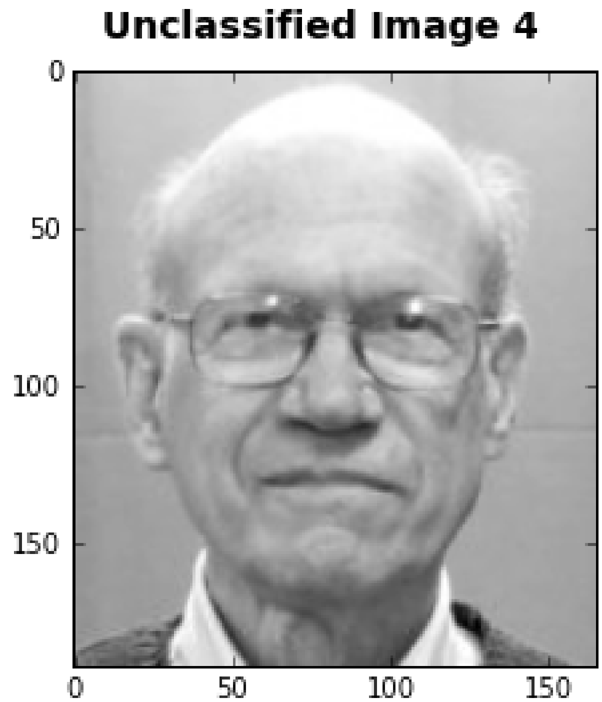
```
=> Image 2 is a face.
"Distance" of image3 to 10 e_faces = 25.000000
=> Image 3 is a face.
"Distance" of image4 to 10 e_faces = 32.000000
=> Image 4 is a face.
"Distance" of image5 to 10 e_faces = 24.000000
=> Image 5 is a face.
"Distance" of image6 to 10 e_faces = 38.000000
=> Image 6 is a face.
"Distance" of image7 to 10 e_faces = 107.000000
=> Image 7 is not a face.
"Distance" of image8 to 10 e_faces = 43.000000
=> Image 8 is not a face.
"Distance" of image9 to 10 e_faces = 102.000000
=> Image 9 is not a face.
"Distance" of image10 to 10 e_faces = 90.000000
=> Image 10 is not a face.
"Distance" of image11 to 10 e_faces = 334.000000
=> Image 11 is not a face.
```



Unclassified Image 2

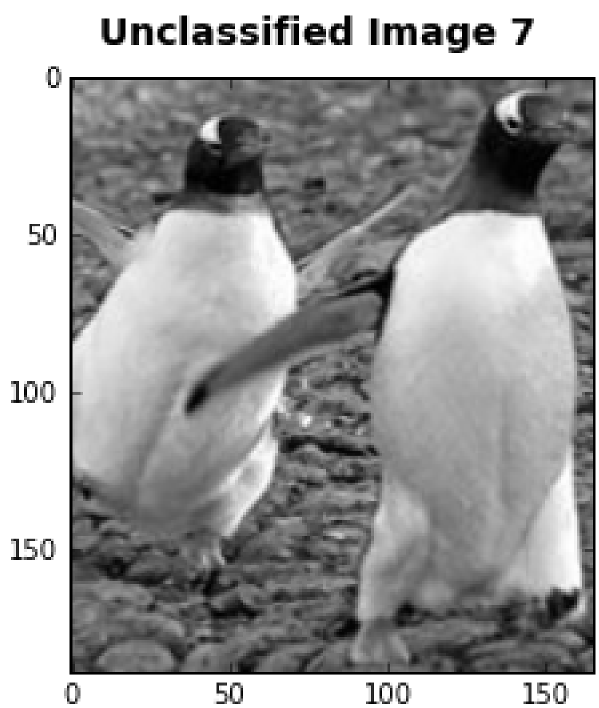


Unclassified Image 3

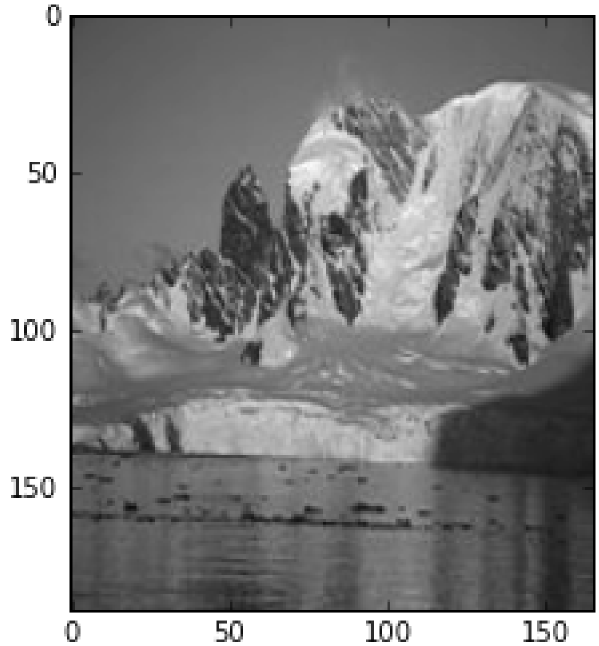

Unclassified Image 4
0
50
100
150
0
50
100
150


Unclassified Image 5
0
50
100
150
0
50
100
150

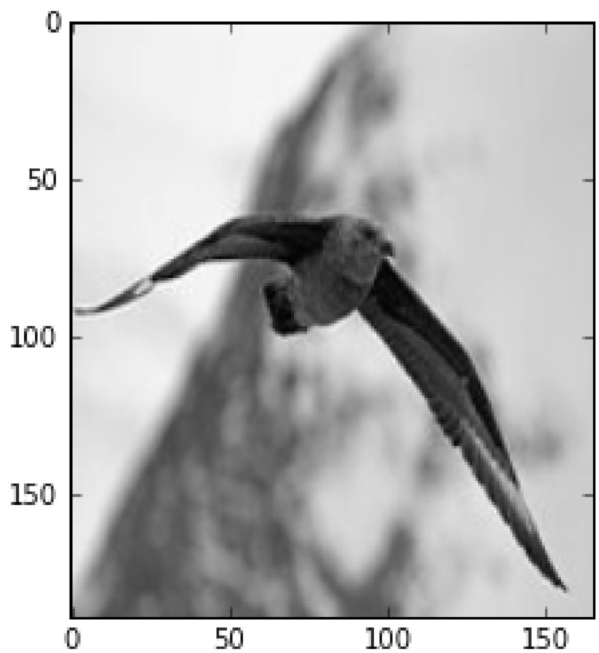

Unclassified Image 6
0
50
100
150
0
50
100
150


Unclassified Image 7
0
50
100
150
0
50
100
150

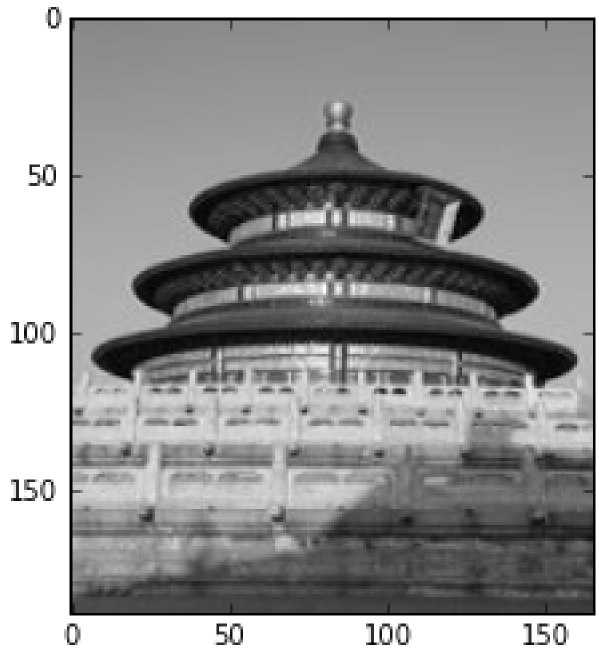
Unclassified Image 8



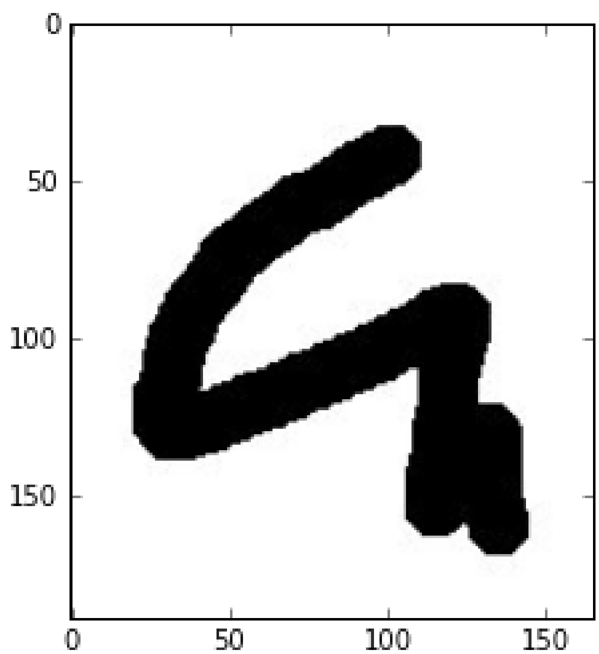
Unclassified Image 9



Unclassified Image 10



Unclassified Image 11




```
In [ ]:
```