**The Problem**

The goal of this assignment was implement the eigenfaces algorithm for face recognition. In order to accurately evaluate the results of this assignment later in this report, we were given a series of images to test our code on. There are too many images to display the entire data set here. To show a sample of the data, the following images are from the dataset



The data set is a mixture of face images and non-face images. The face images vary in expressions and facial features so that the algorithm may learn these differences and account for them when classifying new test images.

**The Process**

The first thing we have to do is read in the images and split them into training and testing data sets. The training data sets exclusively contained face images while the testing data set contained all the non-face images as well as the face images. I split like this so that the algorithm does not learn non-face images. Additionally, I designated nine of the eleven expressions per individual as training data and the remaining two expressions as testing data. I split the data this way in an attempt to follow the 80-20 rule as closely as possible.

The next task is to perform PCA on the training data. In order to perform PCA, we must first calculate the average face of the data set. We do this by summing up each face and dividing this summation by the number of faces in the training set. Next, we have to subtract the average face from each of the faces in the training set, in order to isolate each individual’s facial variations. Let’s name this new matrix *A*. Once we have these, there are three ways we may compute PCA. The first way is to multiply the transpose of *A* by *A*, then use MATLAB’s eig function on the matrix product of these two. Using eig will give us the eigen vectors and values. Conversely, we may use MATLAB’s svd function on , where n is the number of face images in *A*. The last way to perform PCA on *A* is to us MATLAB’s eig function on the matrix product of *A* transpose and *A*. Once we have the eigen vectors, we perform matrix multiplication with these vectors and the matrix *A*. The difference between this approach and the first approach is that the computing resources required to carry out the computations of the third approach are significantly less. This is due to the fact that *A* transpose by *A* is smaller dimensions than *A* by *A* transpose.

Multiplying *A* by the eigen vectors gives us a new matrix called the *eigenfaces*. However, since eigen values can take on very large values, the first thing we must do is normalize the *eigenfaces*. In order to normalize the faces, we iterate over each column vector, normalizing each vector individually of the others. We normalize each vector by dividing its elements by its magnitude.

Now that the *eigenfaces* are normalized, the next step is to isolate the N most significant faces, where N is a positive, real, number. In order to isolate the N most significant vectors, we must first concatenate the *eigenfaces* matrix with the egien values. Since the eigen values are stored in a diagonal matrix, where only the main diagonal holds non-zero numbers, we use MATLAB’s diag function to retrieve these values into a column vector. Once we have this vector, we may append it to the beginning of the *eigenfaces* matrix. With this we matrix, we can use MATLAB’s sortrows function to sort by the eigen values, then extract only the top N eigen faces from the sorted matrix.

Once we have the N most significant eigen faces, let’s name this matrix *Nfaces*. We may begin processing the test data. We begin by subtracting the average face (that we computed for the training set) from each image in the test set, let’s call this matrix *gamma*. Now, we perform matrix multiplication on *eigenfaces* and *gamma*. Likewise, we also want the *testingWeights* for the training set, for reasons we will discuss later. Therefore, we perform matrix multiplication on *eigenfaces* and *A*. Now that we have the weights for each image in the testing set, we want to try to reconstruct our images. Therefore, we perform matrix multiplication on the transpose of the *eigenfaces* and the *testingWeights*, then we add on the average face that we originally subtracted. This give us the algorithm’s rendition of the faces. In order to perform classification on the test set, we subtract the weights of each face in the test set from the weights of each image in the training set and find the training image that gives us the smallest squared Euclidian distance for a given test image. The weights with the smallest distance to the test weights is who the algorithm classified the test image to be. There is, however, an additional step for non-face images. We must add a threshold for what the maximum allowable distance is. If the distance exceeds this threshold, then the test image must be a non-face image.

The Results

In the following sequence of images, the image on the left is the matched training image, the middle image is the test image, and the image on the right is the reconstructed image. For this test, I will use a set of 20 test face images with N being the 10 most significant eigenfaces.















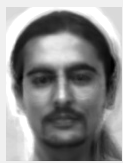














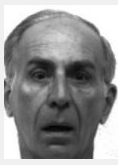






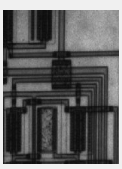




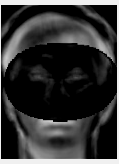


As you can see from these images, the minimum N values that yields recognizable reconstructions is 10. I determined this empirically since, when N is lower than 10, the reconstructed faces are extremely blurry and are too similar (i.e. no distinguishing features). Naturally, as N increases, the reconstructed faces look more and more like their original images. Also, as you can see from the reconstruction, test, and matched training images, that the recognition rate for these 20 images is 100%. If I remove the images containing the glasses attribute from the training data set, then the recognition rate decreases by 5%, misclassifying one image in the testing set. This could be due to the fact that the algorithm did not learn the glasses feature. In other words, the algorithm did not adjust for the glasses feature in the training process. Therefore, its predictive capabilities may be slightly hindered.

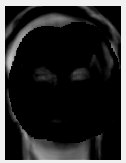
The series of images below represents the non-face images (left), reconstructed images (middle), and the difference between the non-face images and reconstructed images (right).

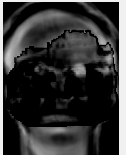






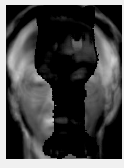
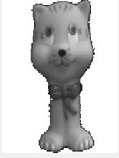


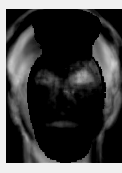


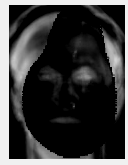
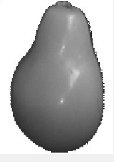


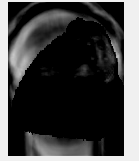


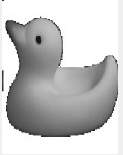


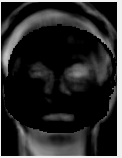


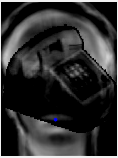


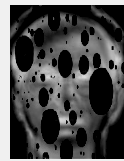


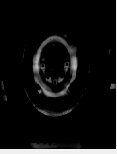






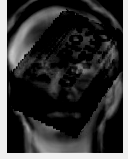




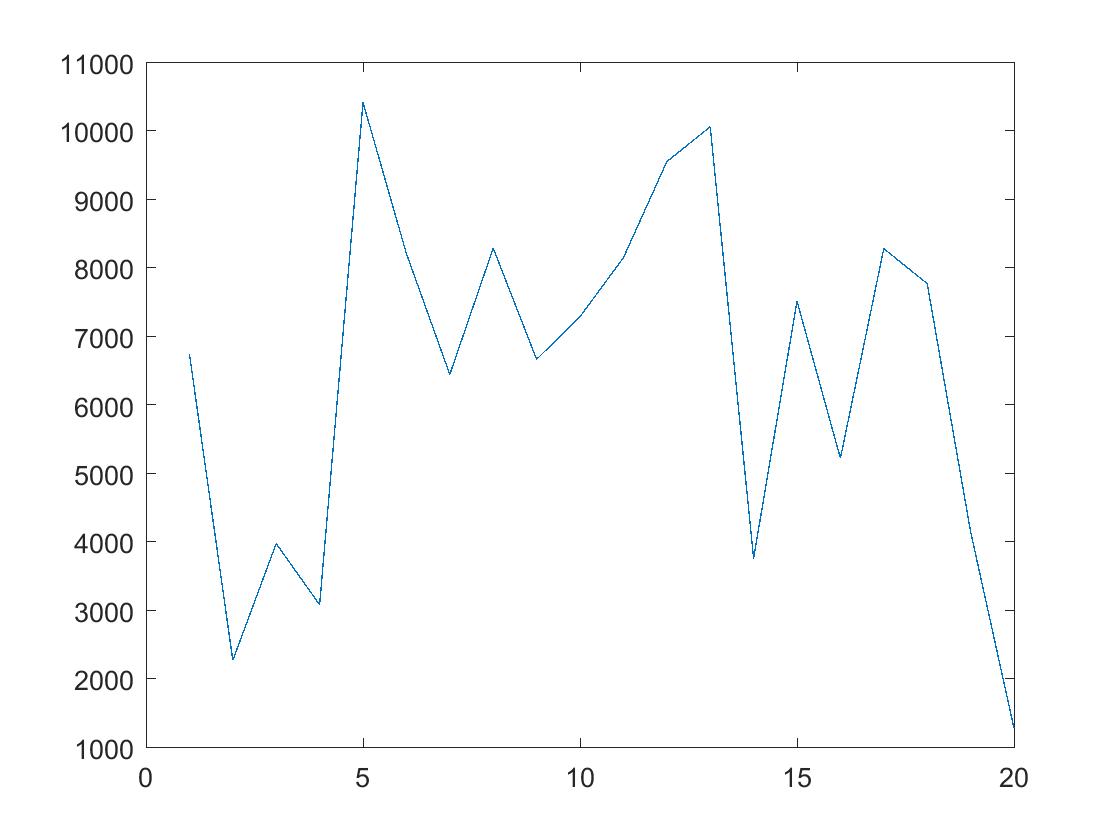




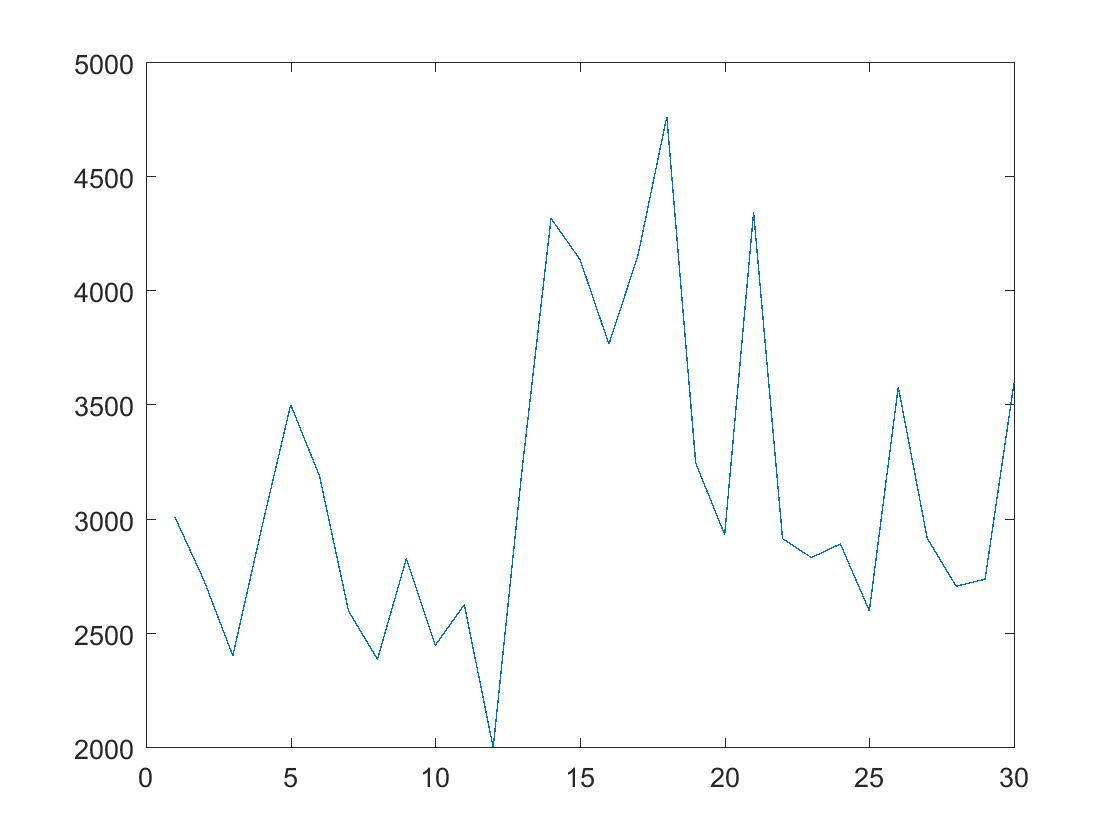




The graph below depicts the Frobenius norm of each difference image from the set of non-face images.



The graph below depicts the Frobenius norm of each difference image from the set of face images. The difference face image, as in the non-face image, is the original minus the reconstructed image.



The major difference between the two plots of the Frobenius norms for both the non-face images and face images is that there is much more variation in the non-face image Frobenius norm values than in the face image Frobenius norm values. The values for the non-face images range from 1000 to 11000 while the values for the face images range from 2000 to 5000. This means that the reconstructed face images are much closer to their originals than the reconstructed non-face images. This makes sense when you take into account the fact that this algorithm was trained exclusively on face images.