# Facial Recognition Using Eigenfaces

Grant Timmerman and Danny Vance
Math 136

June 7, 2013

**Abstract**

This paper examines the algorithm of using eigenfaces in order to recognize faces in static images. By performing a mathematical process called principal component analysis (PCA) on a large set of images depicting different faces, we can discover the core "ingredients" - the eigenfaces, derived from statistical analysis of many pictures of faces. The set of eigenfaces combined with the average face allows us to approximate any human face. Remarkably, it does not take many eigenfaces combined together to achieve a fair approximation of most faces. This algorithm can also be applied in lossy image data compression because each person's face can be recorded with a set of coefficients that correspond each eigenface. The original face can then be reconstructed using these coefficients and a computed average face. This paper aims to determine the effectiveness of this technique with regards to accuracy, computational speed, and data compression as well as to consider its drawbacks that can make the technique ineffective in recognizing faces.

**Key Words:** Eigenfaces, Eigenvectors, Principal Component Analysis (PCA), Eigenface Subspace, Covariance Matrix, Rank, Nullity, Inner Products, Projection, Orthonormal Basis

# 1 Introduction

Facial recognition is a difficult problem in computer vision. Human faces are complex, natural objects that tend to not have easily identifiable edges and features. Because of this, it is difficult to develop a mathematical model of the face that can be used as prior knowledge when analyzing a particular facial image. [1]

Although facial recognition is a high-level visual problem, there is quite a bit of structured information that we can take advantage of. We can represent any image as a list of pixel brightness values, which we can manipulate just as we would any column vector. By finding the most significant pieces of information in a set of faces, we can

characterize each face by the degree to which they each possess these important features. We will find that these principal features can be represented by eigenvectors, which we call “eigenfaces”, of a particular matrix. Together, the eigenfaces form an orthonormal basis for all the variation between the images. The process of recognizing a face is achieved through projecting the face’s image into the subspace spanned by the eigenfaces (“face space”) and then classifying the face by comparing its position in the face space with the positions of known individuals.

# 2 Procedure

## 2.1 Eigenface Initialization

### 2.1.1 Setup images

Acquire an initial set of face images (the “training set”) in grayscale, cropped to only contain the face, and then rescaled to fixed dimensions. In our case we scaled images in our training set to 80 pixels x 80 pixels. It is crucial that photos are centered and of the same size. [3]

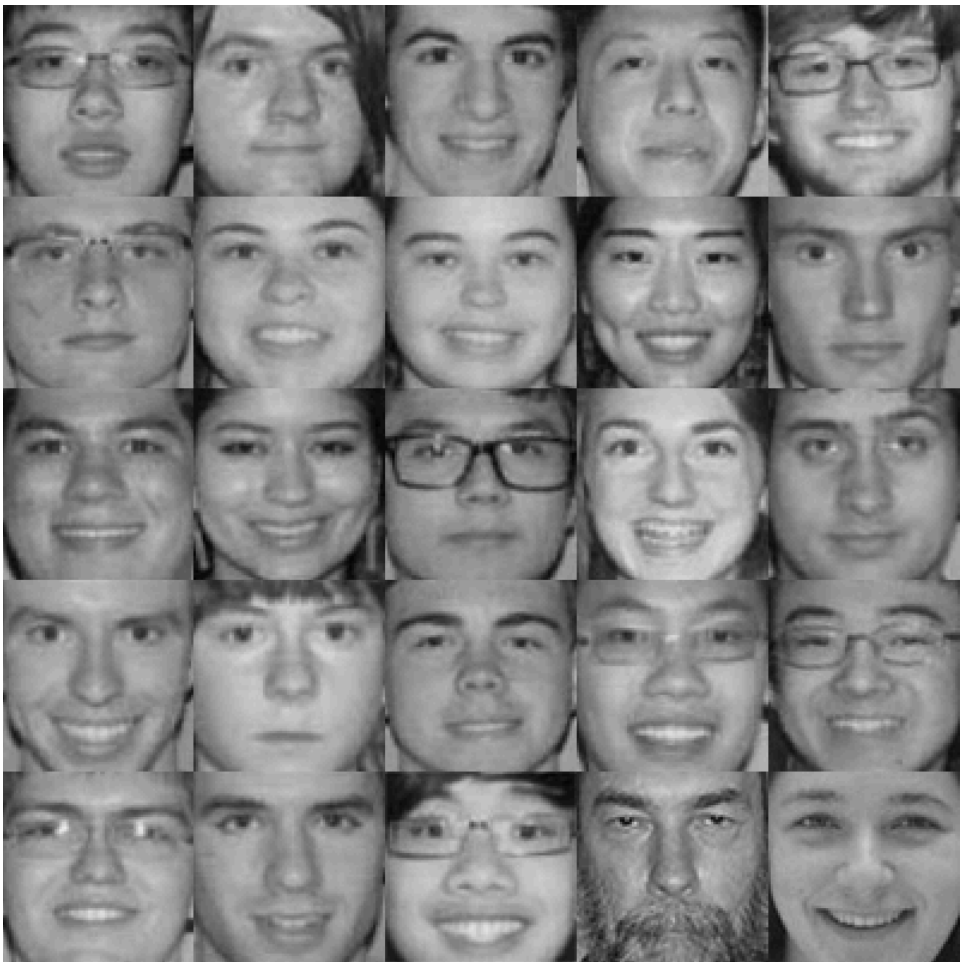


**Figure 1:** The face images we used as our training set.

### 2.1.2 Compute the Average Face

If the photos in the training set are each $m$ by $n$, we can represent each photo as a flattened $mn$ by 1 matrix (we'll call it $v$) that stores values for the brightness of each pixel in one very large list. The average face $W$ is simply the arithmetic average of all of our $M$ image vectors[2]:

$$W = \frac{1}{M}\sum_{i=1}^{M} v_i$$

Here is the result from our training set:

**Figure 2:** The average face of the training set.

This average face encapsulates the vast majority of information contained in each picture. We can subtract this redundant data and focus on the variance between pictures.

### 2.1.3 Find Difference Vectors

Subtract the average face vector $W$ from each photo vector $v_i$ to get a difference vector $D_i$, and arrange these vectors in a matrix $A = \{D_1, D_2, D_3, \ldots D_M\}$.[2] Note that $A$ is $mn$ by $M$. Since each image vector has brightness values on a scale from 0 (black) to 1 (white) in Mathematica, the difference vectors have values that range from -1 to 1.

**Figure 3:** The difference faces of the training set

As you can see, the luminosity of the pictures severely affects the difference vectors. In order to visualize the difference vectors we translated and scaled the values from a -1 to 1 scale to a 0 to 1 scale. This step is necessary because this is the scale that Mathematica uses for brightness. As a result, a value of 0.5 (gray as shown on the sidebars) represents a difference 0 from the average face.

### 2.1.4 The Covariance Matrix

We can compute the covariance matrix $C$ for our data as follows[2]:

$$C = \frac{1}{M}\sum_{n=1}^{M} D_i {D_i}^T = AA^T$$

This $mn$ by $mn$ matrix represents the correspondence between every two pixel variables. The entry in the $i$th row and $j$th column represents the covariance between the $i$th and $j$th pixel brightness variables. A high positive covariance value means that a high positive change in one variable will result in a high positive change in the other variable. A high

negative covariance means the same change in the first variable will result in a high negative change in the second. Covariance values of zero signify that two variables are completely independent. It's clear that these covariance values are the same, despite the order of the two variables being compared, so $c_{ij} = c_{ji}$ for all $0 < j, i \leq mn$ and therefore the covariance matrix is symmetrical.

### 2.1.5 Finding the Right Basis

Once we have our covariance matrix, we want to describe the variance in as few dimensions as possible. If we could find the right basis for the variation between our training faces, we'd know everything about the distribution of the set. The basis we seek would describe everything in terms of the directions of greatest variation, essentially reducing the number of variables from $mn$ to however many elements we include in our basis. The elements of this basis our known as the *principal components* of the data, and the process of dimension reduction from which we derive them is known as *Principal Component Analysis*[1].

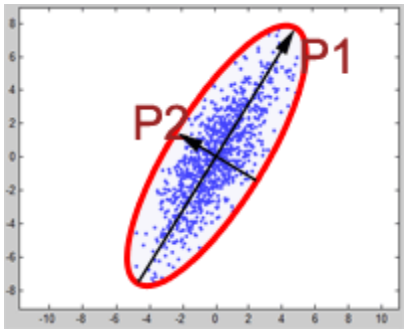



**Figure 4:** The principal axes of a given set of data.

If we consider this basis as a set of unit length vectors, then we can form a set of "axes" with which to interpret variation. If we consider our training set as unit measurements along those axes, then the variables in our basis should have zero covariance. So in this basis, every covariance value corresponding to two different variables (when $i \neq j$) is 0 and every value corresponding to the covariance of a variable with itself (when $i = j$) is nonzero. This means the covariance matrix of our training set in this basis is diagonal! Recall that if a matrix has a diagonal representation, then the basis for that representation is a basis of eigenvectors. So we need only find a basis of eigenvectors for the covariance matrix in order to describe all of the variation in our training set. [1] In particular, we are interested in eigenvectors corresponding to nonzero eigenvalues. We're in luck, because every symmetric real-valued matrix, and hence every covariance matrix, has an *orthonormal basis* of eigenvectors. So the basis we seek does, in fact, exist. We just need to find it.

### 2.1.6 A Computational Shortcut

We still have a slight problem. The covariance matrix for any given set of training faces is absolutely huge. For our set, the covariance matrix is 6400 by 6400 and finding the eigenvectors of something that large isn't exactly easy. Any attempt to do so in Mathematica will surely fail, using up all available memory and crashing the kernel. What we need is a shortcut.

Consider the matrix $A^TA$. Like our covariance matrix, $A^TA$ is symmetric, which can easily be shown:

$$(A^TA)^T = A^T(A^T)^T = A^TA$$

Unlike our covariance matrix, $A^TA$ is rather small. In fact, it is only $M$ by $M$, or in the case of our training set, 25 by 25. It turns out that the eigenvectors of $A^TA$ and of the covariance matrix are also closely related. Consider the following propositions:

*Let A be any m by n matrix with m > n.*

**Proposition 1:** *$A^TA$ and A have the same nullspace.*

*Proof.* It is obvious that the $\ker A \subseteq \ker A^TA$. If $A\mathbf{x} = \mathbf{0}$ for some $\mathbf{x}$, then:

$$A^TA\mathbf{x} = A^T \cdot \mathbf{0} = \mathbf{0}.$$

To prove that $\ker A^TA \subseteq \ker A$, suppose $\mathbf{x} \in \ker A^TA$. Then $A^TA\mathbf{x} = \mathbf{0}$. Therefore $\langle A^TA\mathbf{x}, \mathbf{x}\rangle = 0$. But we also know that:

$$\langle A^TA\mathbf{x}, \mathbf{x}\rangle = \mathbf{x}^TA^TA\mathbf{x} = (A\mathbf{x})^TA\mathbf{x} = \langle A\mathbf{x}, A\mathbf{x}\rangle = \mathbf{0} \Longrightarrow A\mathbf{x} = \mathbf{0}$$

So we know $\ker A^TA \subseteq \ker A$ and $\ker A \subseteq \ker A^TA$. Therefore $\ker A^TA = \ker A$. □

**Proposition 2:** *$A^TA$ and $AA^T$ have the same rank.*

*Proof.* By the rank-nullity theorem, we know the following:

1) $\text{rank}A + \text{nullity}A = n$
2) $\text{rank}A^T + \text{nullity}A^T = m$
3) $\text{rank}A^TA + \text{nullity}A^TA = n$
4) $\text{rank}AA^T + \text{nullity}AA^T = m$

By proposition 1, we know that $\text{nullity}A = \text{nullity}A^TA$, so by this fact, and 1) and 3), we know that $\text{rank}A = \text{rank}A^TA$. If we let $B = A^T$, then $AA^T = B^TB$. And by what we've just shown, we know that $\text{rank}B = \text{rank}B^TB$, so $\text{rank}A^T = \text{rank}AA^T$. The ranks of both $A$ and $A^T$ are equal to the number of pivots in the row-reduced echelon form of $A$, so we know that $\text{rank}A = \text{rank}A^T$. Putting this all together we have:

$$\text{rank}A^TA = \text{rank}A = \text{rank}A^T = \text{rank}AA^T$$

So $A^TA$ and $AA^T$ have the same rank. □

**Proposition 3:** *If $v$ is an eigenvector of $A^TA$ corresponding to a nonzero eigenvalue $\lambda$, then $Av$ is an eigenvector of $AA^T$ associated with the same eigenvalue.*

*Proof.* Since $\lambda \neq 0$, we have $A^TAv = \lambda v \neq 0$, so we know $Av \neq 0$. Furthermore:

$$AA^T(Av) = A(A^TAv) = A(\lambda v) = \lambda(Av)$$

Therefore $Av$ is an eigenvector of $AA^T$ corresponding to $\lambda$.[1] □

**Proposition 4**: *$AA^T$ and $A^TA$ have the same nonzero eigenvalues, counting multiplicity.*[1]

*Proof.* $A^TA$ is an $n$ by $n$ matrix, so it has $n$ eigenvalues, counting multiplicity.
$A^TA$ is an $m$ by $m$ matrix, so it has $m$ eigenvalues, counting multiplicity.
We know by proposition 3 that $AA^T$ has all $n$ eigenvalues of $A^TA$. We need only show that all the additional $m - n$ eigenvalues of $AA^T$ are equal to zero.

Let $p = \text{rank}A^TA = \text{rank } AA^T$. Recall from proposition 2 that we know:

1) $p + \text{nullity}A^TA = n$
2) $p + \text{nullity}AA^T = m$

Then we know that $\text{nullity}A^TA = n - p$ and $\text{nullity}AA^T = m - p$.
It follows that $\text{nullity}AA^T - \text{nullity}A^TA = m - n$.
Therefore every eigenvalue of $AA^T$ that is not an eigenvalue of $A^TA$ is equal to zero.
$AA^T$ and $A^TA$ have all the same nonzero eigenvalues, counting multiplicity. □

---

What this all means for finding our basis is that we don't actually have to directly compute the eigenvectors of our covariance matrix. We can instead compute the eigenvectors of $A^TA$. This matrix is much smaller than the covariance matrix, so Mathematica can do this for us in no time. To get the eigenvectors of the covariance matrix we need, namely the vectors associated with nonzero eigenvalues, we need only take the eigenvectors of $A^TA$ and multiply each in the front by $A$.

So if $\{w_1, w_2, \ldots, w_M\}$ is the set of all eigenvectors of $A^TA$, then the set of eigenvectors of our covariance matrix is $\{Aw_1, Aw_2, \ldots, Aw_M\}$. This set of vectors acts as a basis for all the variation in our training set. Future computations are made easier if we normalize the entire set right away. Like our original image vectors, each eigenvector is $mn$ by 1, just as we'd expect. These vectors are called *eigenfaces*, and we can represent each one as a picture.

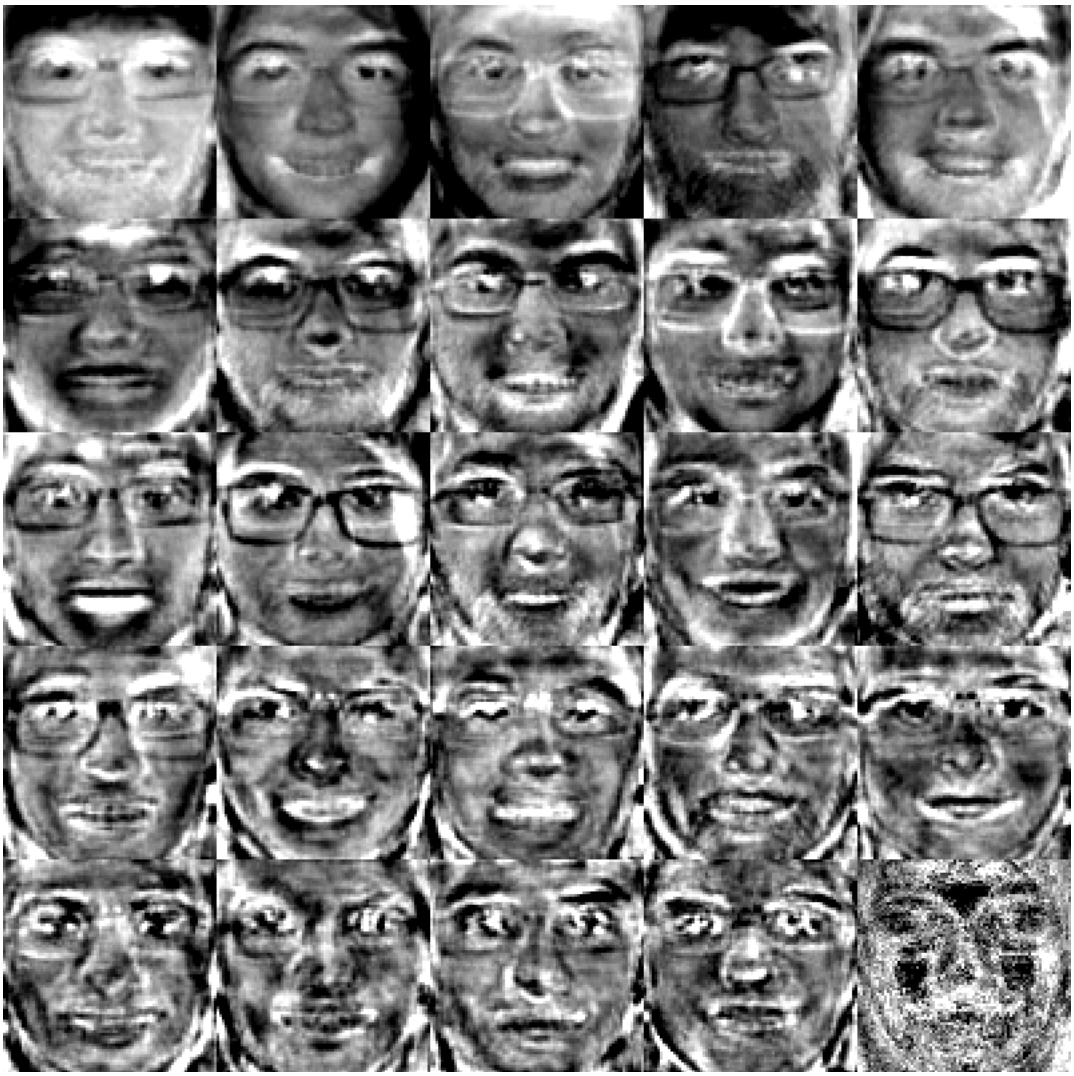


**Figure 5:** All 25 eigenfaces of the training set. The most important facial component is on the top left and the least important facial component is on the bottom right.

### 2.1.7 Interpreting the Eigenfaces

The eigenfaces have a natural order of importance. If we are to visualize them as axes once again, the axes with the greatest significance, the ones which tend to have the highest unit measurements from picture to picture, are the eigenfaces associated with the largest eigenvalues.[1] Mathematica automatically gives us the eigenvectors in descending order of importance, so this requires no additional work to figure out.

From here on out, we'll denote our set of normalized eigenfaces as $\{u_1, \ldots, u_P\}$, where we choose to keep our $P$ most significant eigenfaces. We call the set spanned by these eigenfaces the *eigenface subspace* or the "face space".

## 2.2 Projecting and Reconstructing Faces

### 2.2.1 Project a Photo to the Eigen Space

To project a photo vector $v$ onto the eigenface subspace, we need to find the scalar coefficients that make up a reconstructed face. To do this we can simply take the inner product of the photo's difference vector, $D = v - W$, with each eigenfaces to get each coefficient. So for each coefficient $k_n$, we have:

$$k_n = \langle D, u_n \rangle$$

Once we have the coefficients for a given image vector, we can get a projected image $x$ by adding the average face to the sum of the eigenface vectors ($u_n$) multiplied by their corresponding coefficients ($k_n$)[1]:

$$x = W + \sum_{n=1}^{P} k_n u_n$$

Where $P$ is the number of eigenvectors we choose to use.

### 2.2.2 Projecting Training Set Photos to the Eigen Space

Here are some examples of projected faces:

**Figure 6:** Random faces from the training set constructed using a random number of eigenfaces.

Here is every possible reconstruction combination for the training set images:

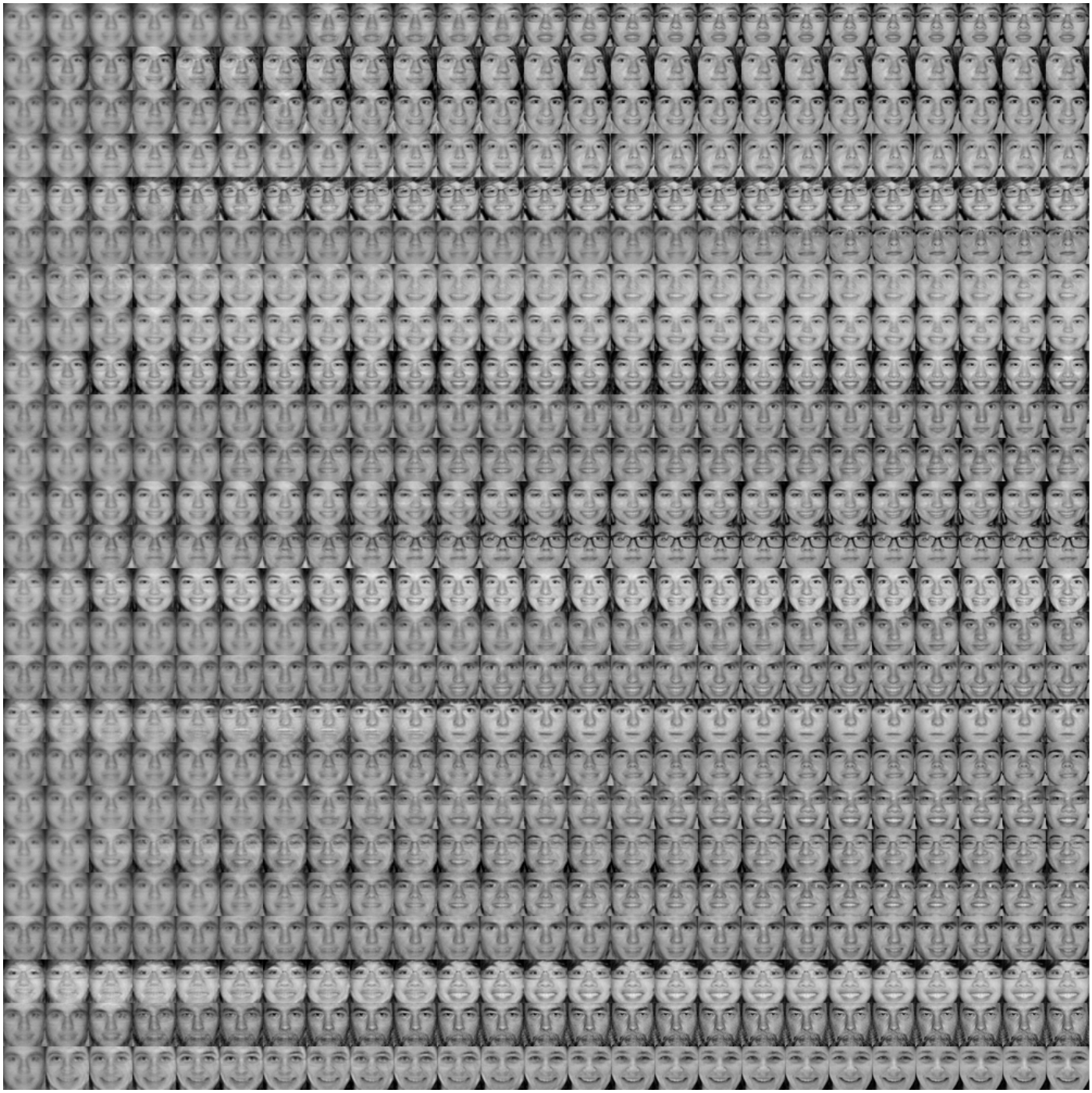


**Figure 7:** The reconstructed faces where each row represents a person and the $ith$ column is a reconstructed face using $i$ eigenfaces (i.e. the first column represents the faces reconstructed with 1 eigenface and the last column represents the faces reconstructed using 25 eigenfaces.)

As you can see, the training photos only require a few eigenfaces to become distinct from the rest. The first five iterations have a larger impact on altering the face than the next ten iterations. Also note that the last iteration, the iteration where the number of eigenfaces applied is equal to the number of faces in the training set recreates the original image with 0% error. (We will prove this Facial Reconstruction Accuracy section.)

### 2.2.3 Projecting Other Photos to the Eigen Space

While projecting our training set photos is interesting and can be applicable to data compression, the real application of eigenfaces is in facial recognition. Thus, it is more useful to project photos not in our training set ("foreign photos") to our face space, so we can try to recognize them. Here we have chosen a few different photos of subjects in the training set as well as random pictures of a baby orangutan, a doll, and a flower:

**Figure 8:** Photos of our foreign set

**Figure 9:** The reconstructed faces of our foreign set where each row represents a foreign photo and the $ith$ column represents a reconstructed face using $i$ eigenfaces.

**Figure 10:** Our original foreign set alongside the reconstructed set using all eigenfaces.

Although the reconstructed photos appear disfigured and not smooth, they do somewhat resemble the people they are supposed to represent (those which apply of course). We can see that factors such as lighting, angle of the head, smiles, and image cropping all severely affect the resulting reconstructed image.

## 2.3 Measuring Reconstruction Accuracy

### 2.3.1 Accuracy Formula

To measure how lossy our image reconstruction is, we define a formula to get the percent difference between two image vectors $v_1$ and $v_2$ as such (assuming $\dim(v_1) = \dim(v_2)$):

$$diff(v_1, v_2) := \frac{100}{\dim(v_1)} \sum_{n=1}^{\dim(v_1)} |v_{1,n} - v_{2,n}|$$

As long as our image vector $v_1$ and $v_2$ range from 0 to 1, we will always get the correct percent difference $diff(v_1, v_2)$ ranges from 0% (identical vectors) to 100% ($diff$ of a vector of 0's and a vector of 1's). This formula is used to compare reconstructed faces to the original images in the training set.

### 2.3.2 Accuracy of Reconstructed Training Faces

Recall the grid of reconstructed training faces. By using our accuracy formula, we can calculate exactly how accurately our faces were reconstructed and plot the percent error for each face on a separate line:

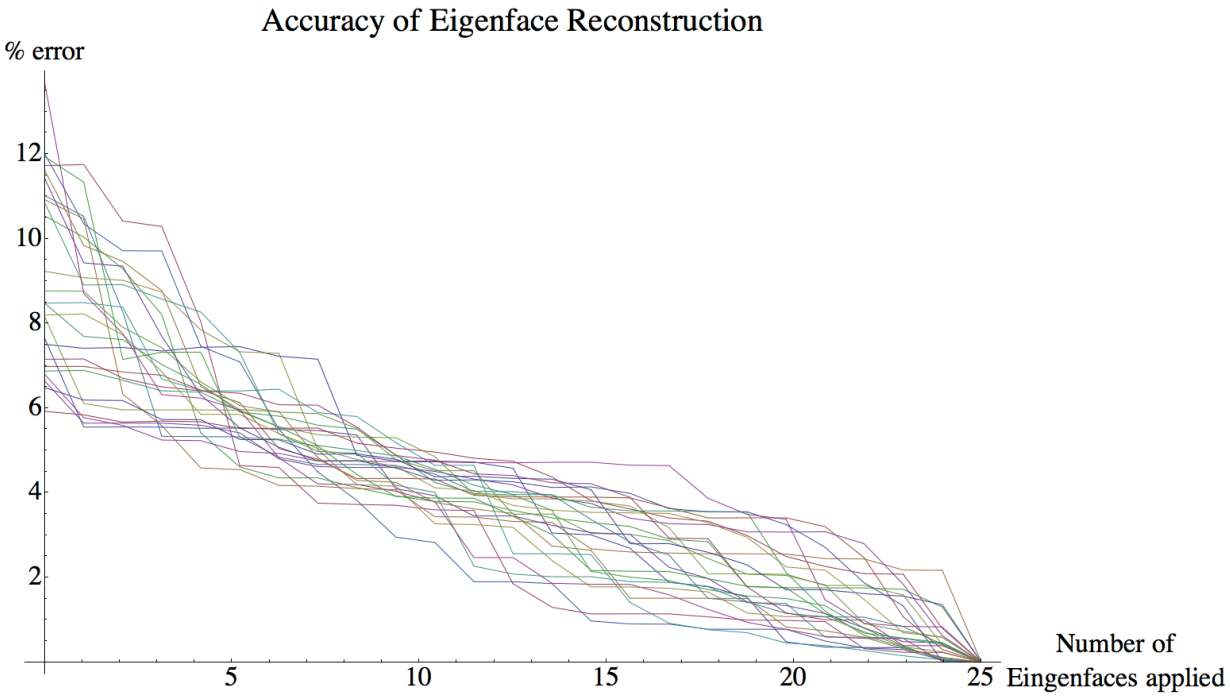



**Figure 11:** A graph showing the percent error of each reconstructed face over the number of eigenfaces applied to create that face.

Remarkably, the average difference between a reconstructed face with 0 applied eigenfaces and the original face is roughly 9%. This means that, on average, the training faces vary from the mean face by 9%.

As we apply more eigenfaces to the reconstructed image, we monotonically approach the perfect reconstruction of the face, at 25 eigenfaces.

## 2.4 Facial Recognition

### 2.4.1 Technique

To "recognize" a reconstructed face $x$ of a subject in our training set, we seek to find a training set vector $v_0$ such that $diff(x, v_0) \leq diff(x, v)$ for all $v$ in the training set. In other words, we seek to find the argmin of the difference function.

### 2.4.2 Recognizing Training Faces

Below are some examples of facial recognition using the technique above. Our results are organized in order of closest match by the following grid structure:

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

**Figure 12:** The organization of our results from 1 being the closest match to 25 being the furthest match.

#### 2.4.2.1 Example 1

**Figure 13.1:** A reconstructed training face using 5 eigenfaces.

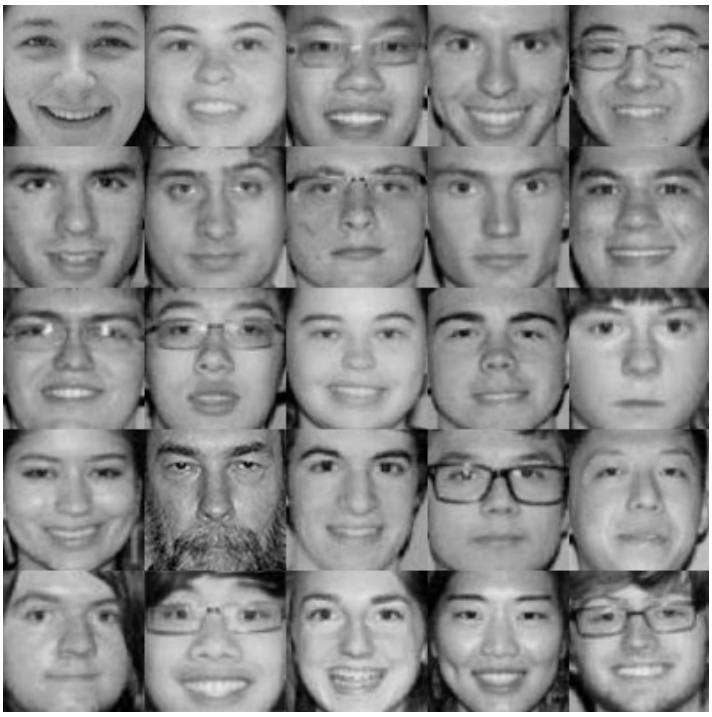


**Figure 13.2:** A list of closest matches from the top left to bottom right.

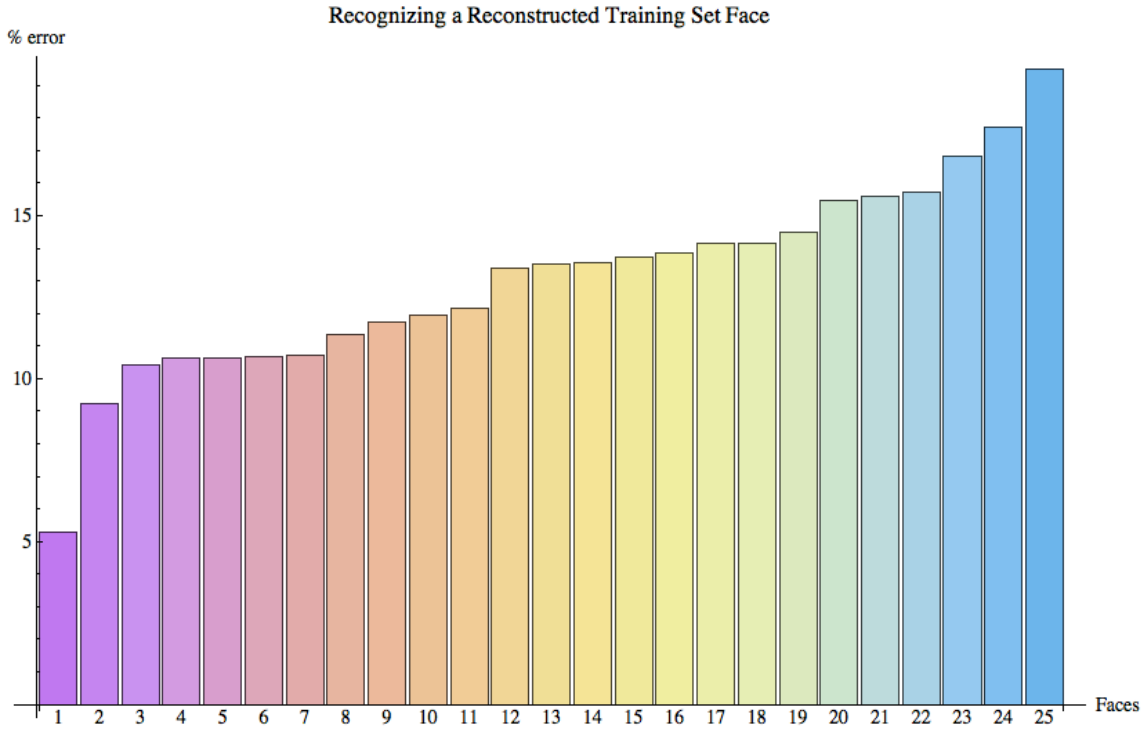



**Figure 13.3:** The percent error between the reconstructed face and the training set faces in ascending order.

In this example five eigenfaces was enough (in fact this was the threshold) to distinguish this reconstructed face from other possible matches in the training set.

#### 2.4.2.2 Example 2

**Figure 14.1:** A reconstructed training face using 25 eigenfaces.

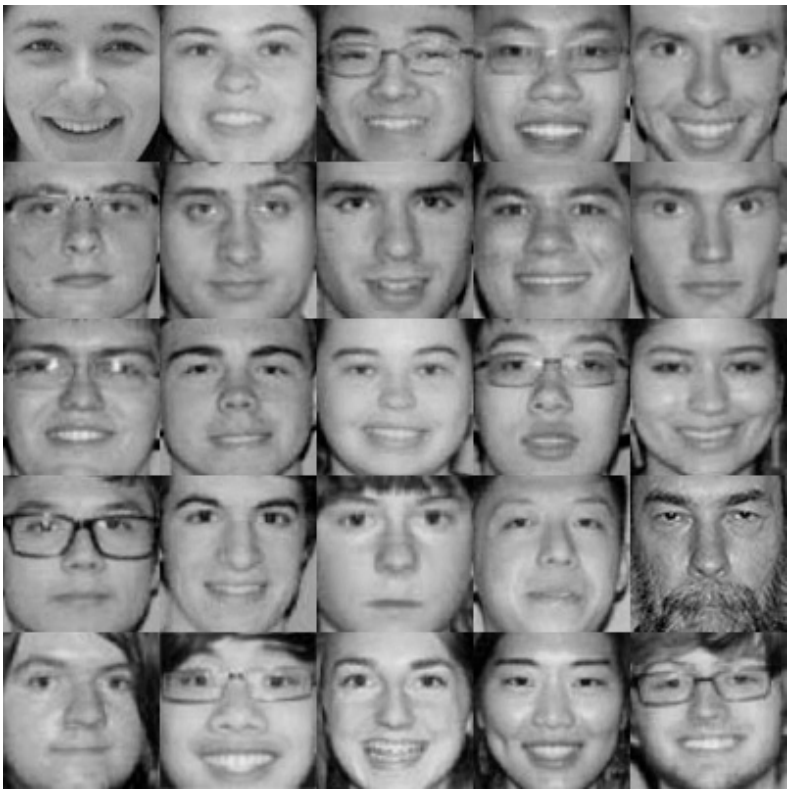


**Figure 14.2:** A list of closest matches from the top left to bottom right.

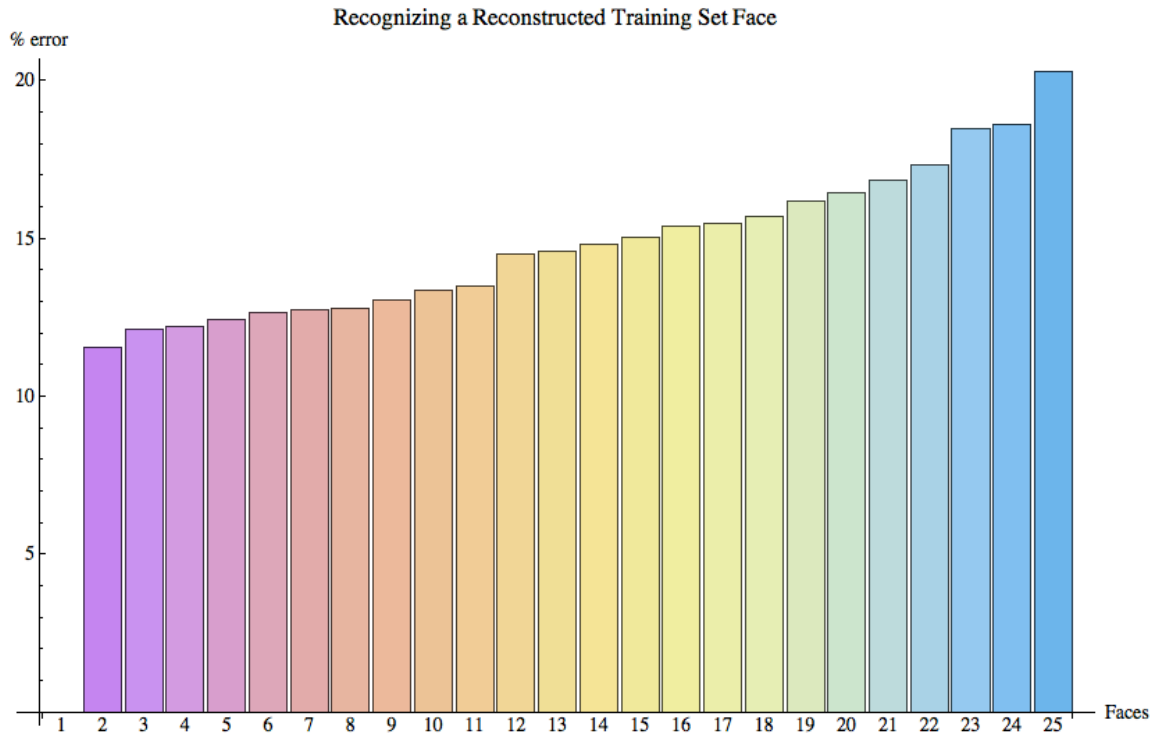



**Figure 14.3:** The percent error between the reconstructed face and the training set faces in ascending order.

In this example the full set of eigenfaces gives a perfect match to the original training face, which is why there is 0% error for face 1 in the graph. Also, as we increase the number of applied eigenfaces, the gap between the other potential matches and the 1st match increases dramatically.

### 2.4.2.3 Example 3

**Figure 15.1:** A reconstructed training face using 8 eigenfaces.

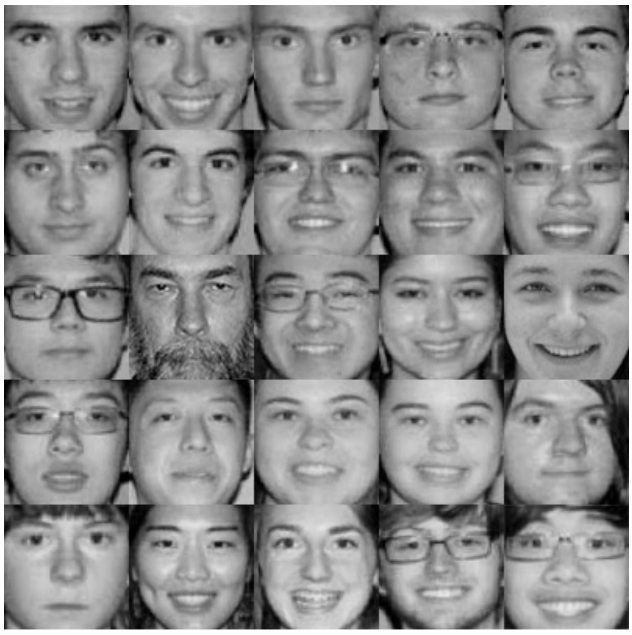


**Figure 15.2:** A list of closest matches from the top left to bottom right.

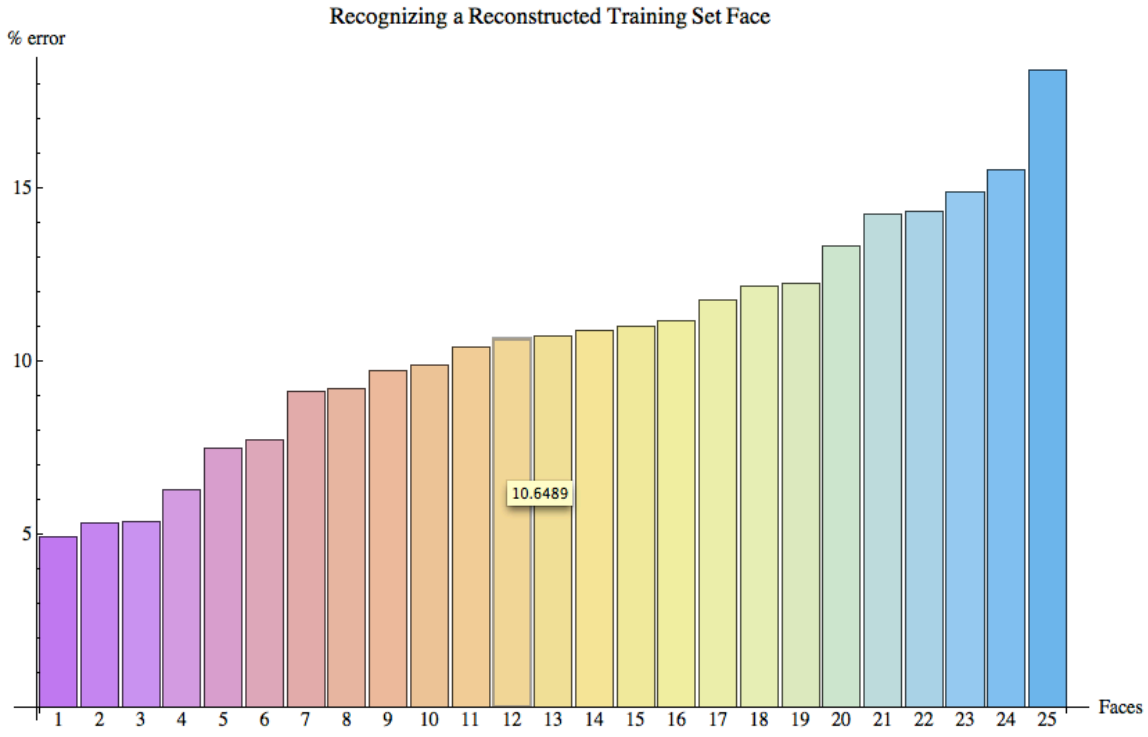



**Figure 15.3:** The percent error between the reconstructed face and the training set faces in ascending order.

Although 8 eigenfaces were applied, the algorithm suggested the correct match second. This is due to the fact that many of the training faces (as you can see in the first row of *Figure 15.2*) have very similar features and all have relatively low error percentages.

#### 2.4.2.3 Summary

It turns out that on average, four eigenfaces are required to recognize the correct face as the closest match. In one case, it takes zero eigenfaces to correctly match the reconstructed face. This is because this face is closest to the average face. Below are diagrams showing the number of eigenfaces necessary to successfully match the reconstructed training face:

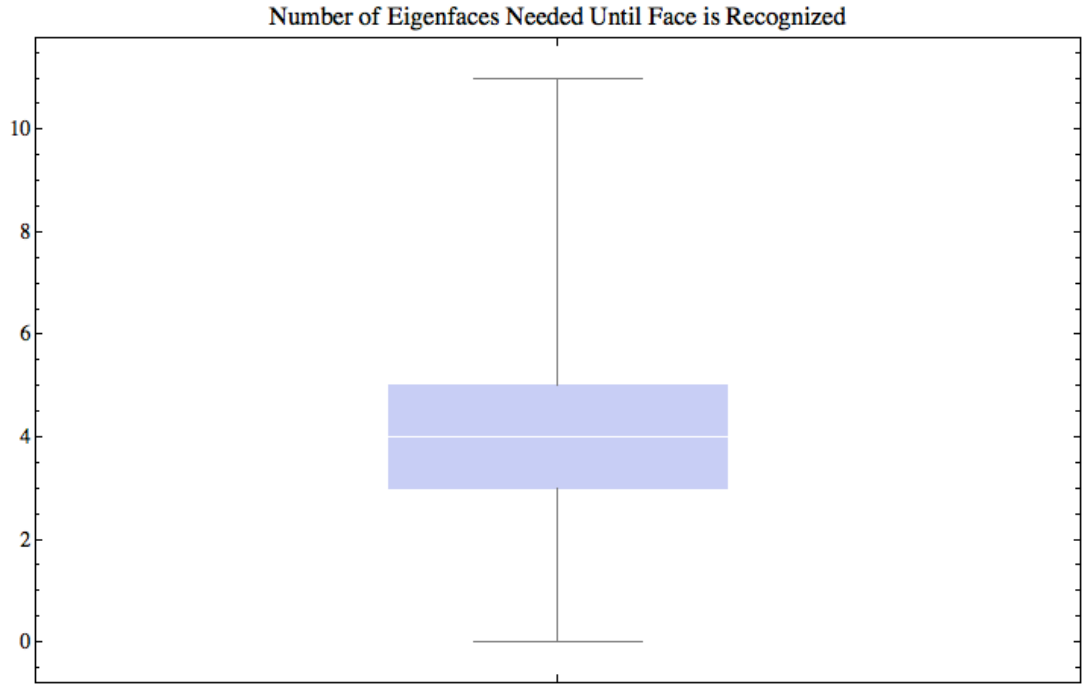



**Figure 16:** A box-and-whisker showing the number of eigenfaces needed to recognize a training face.

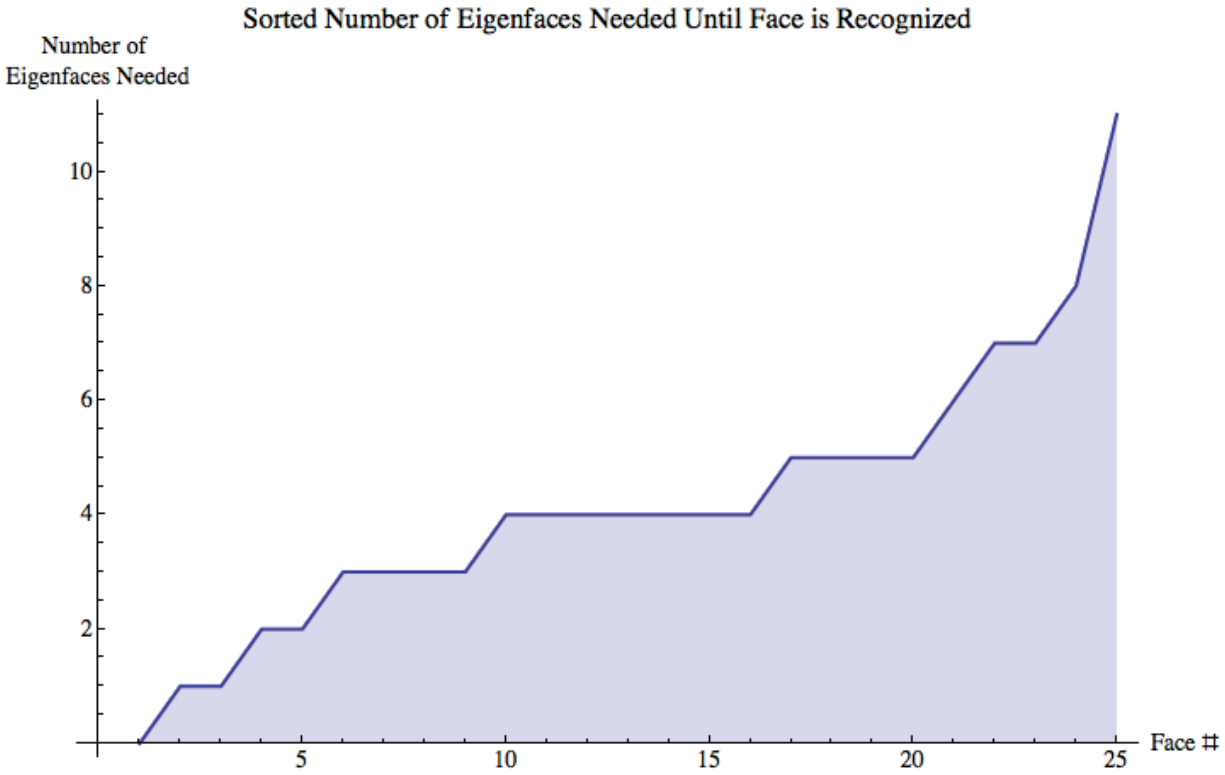



**Figure 17:** A sorted connected list plot showing the number of eigenfaces needed to recognize a training face.

### 2.4.3 Recognizing Foreign Faces

Recognizing foreign faces is the same as recognizing training set faces. However, the projected face is much less likely to resemble the original photo since the foreign image is not spanned by the set of our eigenfaces (most likely). Here are the results of recognizing our foreign set:


**Figure 18:** A grid showing the original foreign image (column 1), the reconstructed/projected image using all eigenfaces (column 2), and the top 5 matches.

The results are pretty good. The results for the first six foreign faces (from top to bottom) are the following: 2nd, 2nd, 5th, 20th, N/A, 2nd. The tilt and scale of subject #4's head affected the results. She is in fact 20/25 on the list. Subject #3 is most likely mismatched because of the top of the hair seen in the foreign image and lack of hair in the training set image. The facial expression and angle of the photo are also very different from that of our training set photo. However, in practice we would never be able to use all of our eigenfaces. We would also have a much larger training set, which would increase the span of our face space, resulting in higher accuracy.

# 3 Conclusion

The eigenface approach provides a practical solution that is well-fit to facial recognition and image data compression without much processing power required. It is fast, fairly simple, and has shown to work well in a somewhat constrained environment. However, the eigenface algorithm is not without its drawbacks. In particular, different luminosity levels, varying angles, and issues with cropping and scaling can significantly distort the results. Likewise, different facial expressions, hair and other features can lead to poor recognition. Any of these varying conditions within our training set can lead PCA to extract components that describe variation in the conditions, rather than in the faces. We used a very small training set, compared to what is often used in practice. If we used a larger set, the results would be more accurate and generalizable to a larger range of faces. Nonetheless, our findings demonstrate the potential power of this facial recognition approach.

# References

## Books

[1] Turk, Matthew & Pentland, Alex. (1991). Eigenfaces for Recognition. *Journal of Cognitive Neuroscience*, 3(1), 1-4.

[2] Alsamman, A., & Alam, M. S. (January 01, 2005). Comparative study of face recognition techniques that use joint transform correlation and principal component analysis. Applied Optics, 44, 5, 688-92.

[3] Hancock, P. J. (January 01, 2000). Evolving faces from principal components. Behavior Research Methods, Instruments, & Computers : a Journal of the Psychonomic Society, Inc, 32, 2, 327-33.

## People We've Consulted

- Thomas Duchamp
- Katia Nepom

# Mathematica Workbook

# Facial Recognition using Eigenfaces

By Grant Timmerman and Danny Vance

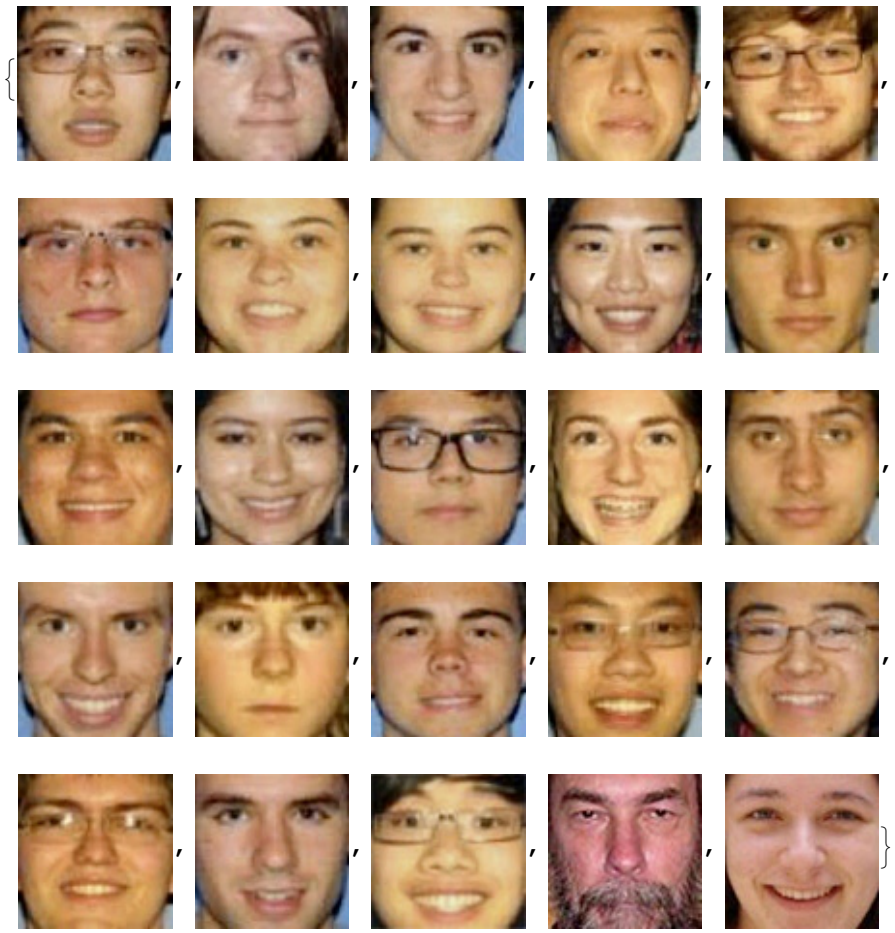
---

## 1 - Setting Up a Training Set

### First import the images we want to use

```
SetDirectory[StringJoin[NotebookDirectory[], "math_images_resized"]];

photos = Map[Import, {"f1.jpg", "f2.jpg", "f3.jpg", "f4.jpg", "f5.jpg", "f6.jpg",
    "f7.jpg", "f8.jpg", "f9.jpg", "f10.jpg", "f11.jpg", "f12.jpg", "f13.jpg",
    "f14.jpg", "f15.jpg", "f16.jpg", "f17.jpg", "f18.jpg", "f19.jpg",
    "f20.jpg", "f21.jpg", "f22.jpg", "f23.jpg", "f24.jpg", "f25.jpg"}]
numPhotos = Length[photos];
```



## Next convert all the photos to grayscale

```
grayPhotos = Map[ColorConvert[#, "Grayscale"] &, photos]
```

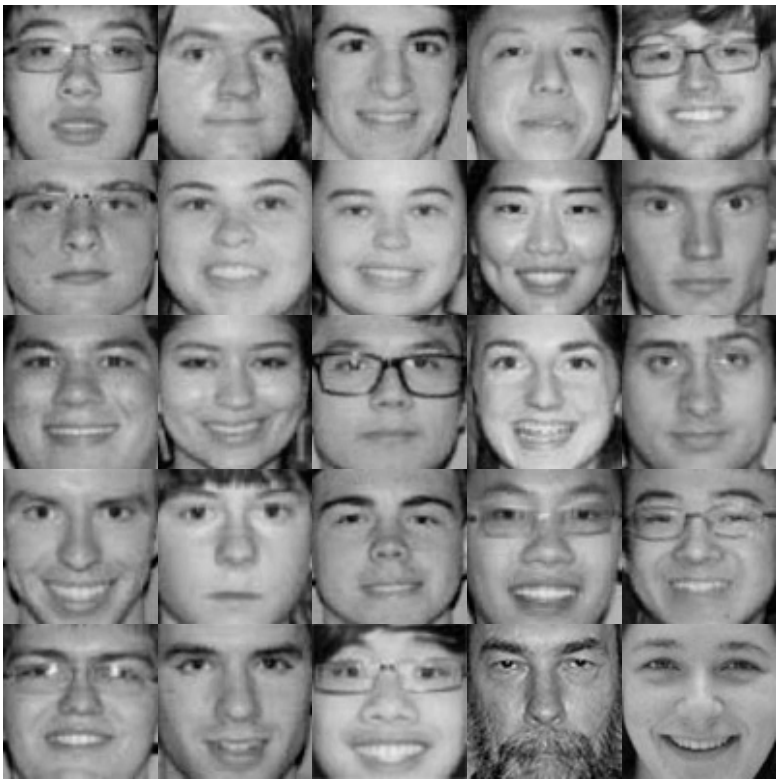
## Create useful methods for converting an image to a vector vice versa

```
imageToVector[img_] := Flatten[ImageData[img]];
imageSize = Sqrt[Dimensions[imageToVector[grayPhotos[[1]]]]];
(* Assumes the photo is square *)
vectorToImage[vec_] := Image[Partition[vec, Sqrt[Dimensions[vec][[1]]]]];
grayPhotosVector = Map[imageToVector, grayPhotos];
```

## (Optional) Display the images in a grid as a reference

```
grayPhotosMatrix = Transpose[Map[imageToVector, grayPhotos]];
```

```
grayPhotosGrid = Partition[grayPhotos, 5];
Grid[grayPhotosGrid, Spacings → {0, 0}]
```



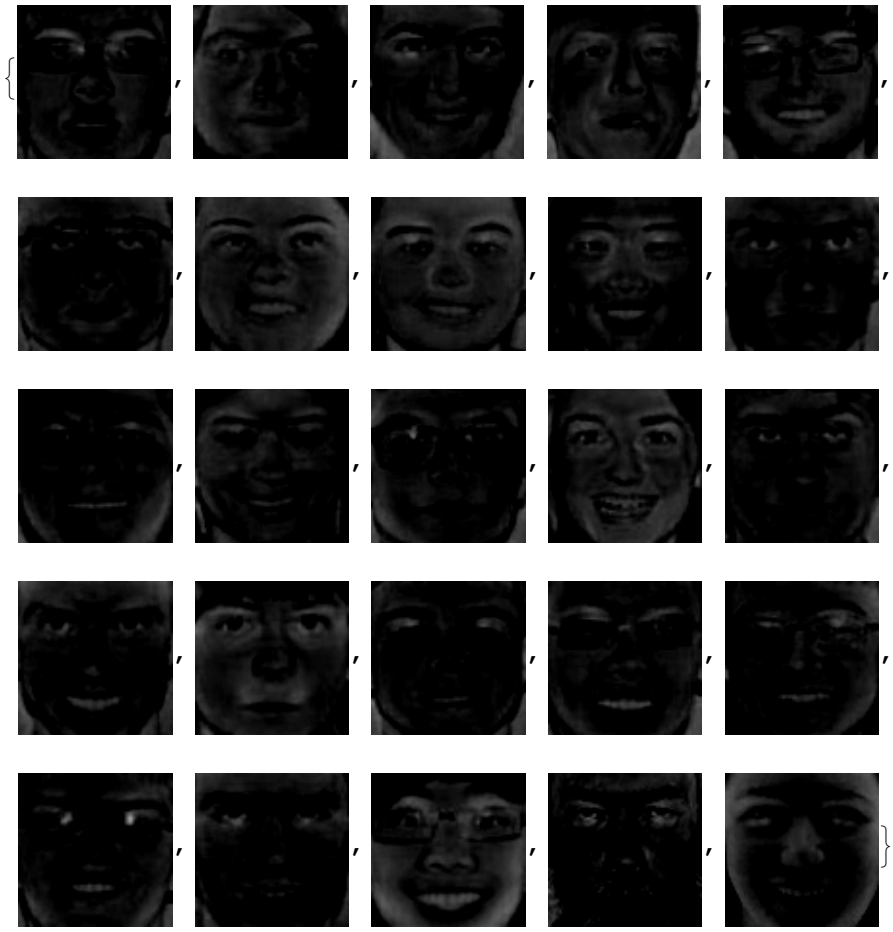
# 2 - Calculate the Mean and Difference Faces

## Calculating the mean face

```
meanFaceMatrix = Apply[Plus, Map[ImageData, grayPhotos]] / numPhotos;
meanFaceVector = Flatten[meanFaceMatrix];
meanFaceImage = Image[meanFaceMatrix]
```

## Calculating the difference face and difference vectors

```
differenceFaces = Map[ImageSubtract[#, meanFaceImage] &, grayPhotos]
differenceVectors = Map[imageToVector, differenceFaces];
```



## (Optional) Display the images in a grid as a reference

```
(*displayedDifferencesFaces = 0;*)
getDisplayVector[vec_] := Module[
    {},
    Map[.5 + .5 # &, vec]
  ];
displayDifferenceVectors = Map[getDisplayVector[#] &, differenceVectors];
displayDifferenceFaces = Map[vectorToImage, displayDifferenceVectors];
differenceFacesGrid = Partition[displayDifferenceFaces, 5];
Grid[differenceFacesGrid, Spacings → {0, 0}]
```

## The difference vector of the mean face:

```
vectorToImage[getDisplayVector[meanFaceVector - meanFaceVector]]
```

# 3 - Finding Eigenfaces

## Find the small matrix

```
babyMatrix = differenceVectors.Transpose[differenceVectors];
```

## Choose a fixed number of eigenfaces and diagonalize the eigenvectors of the normalized small matrix

```
numEigenFaces = numPhotos;
```

```
eigenfaceSystem = Eigensystem[babyMatrix];
eigenfaces =
  Map[Normalize[Transpose[differenceVectors].#] &, eigenfaceSystem[[2]]];
diagonalCovarianceMatrix = DiagonalMatrix[eigenfaceSystem[[1]]];
```

### (Optional) Display the images in a grid as a reference

```
getEigenfaceDisplayVector[vec_] := Module[{},
    Map[.5 + 25 # &, vec]
  ];
displayEigenfaces =
  Map[vectorToImage[getEigenfaceDisplayVector[#]] &, eigenfaces];
Grid[Partition[displayEigenfaces, 5], Spacings → {0, 0}]
```

## 4 - Project Images into the Eigenspace

### Define a function that projects an image to the eigenspace

```
projectImageToFaceSpace[imageVector_, meanFaceVector_, eigenfaces_] :=
  Module[{differenceVector},
   differenceVector = imageVector - meanFaceVector;
   Map[differenceVector.# &, eigenfaces]
  ];
```

### Calculate the eigenface coefficients for each face you wish to project

```
(* A set of eigenface coefficients for each face *)
eigenfaceCoefficients =
  Map[projectImageToFaceSpace[#, meanFaceVector, eigenfaces] &, grayPhotosVector];
```

## 5 - Recontruct Images from Eigenspace

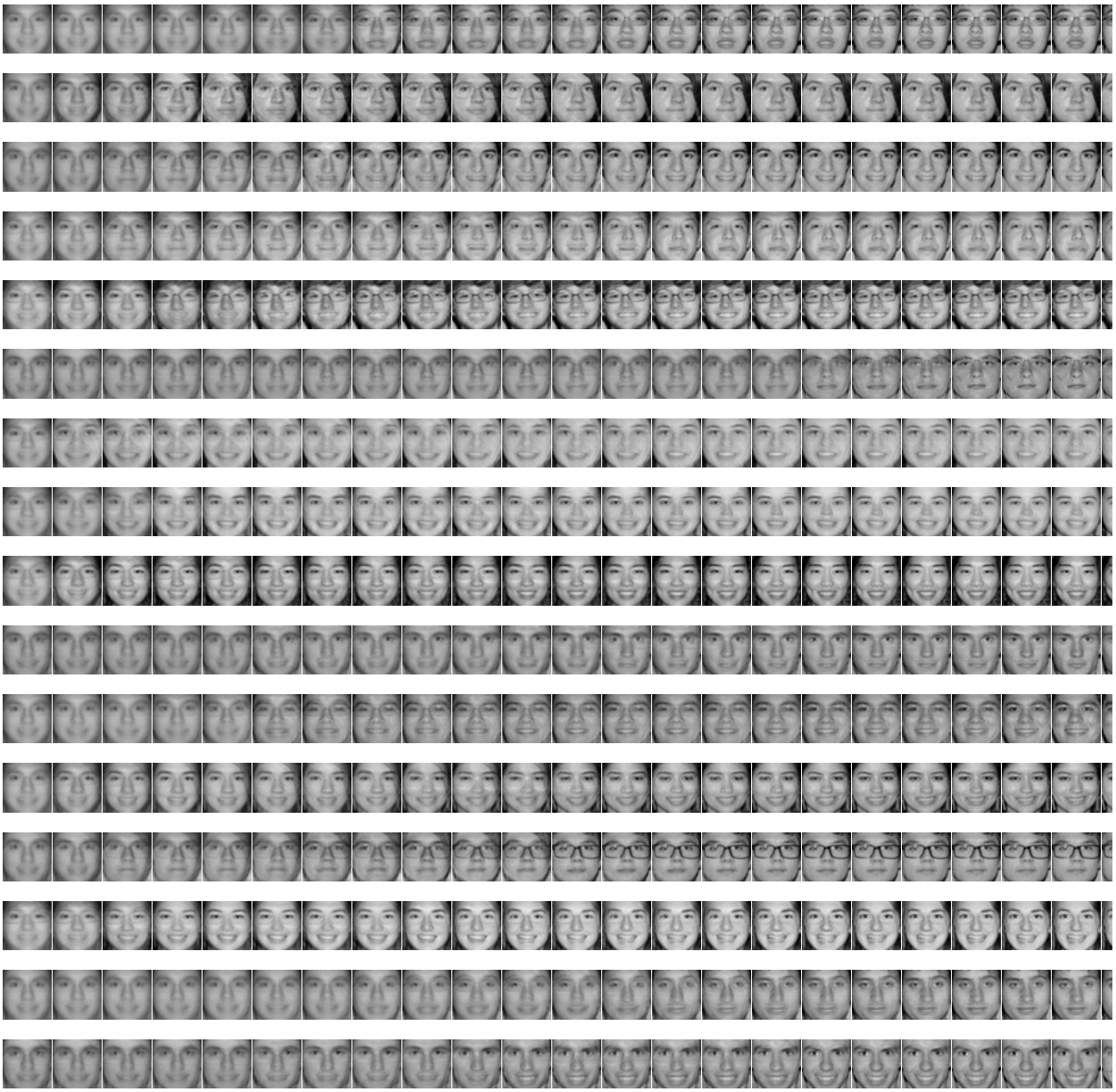
### Define a function that reconstructs a face vector from eigenface coefficients

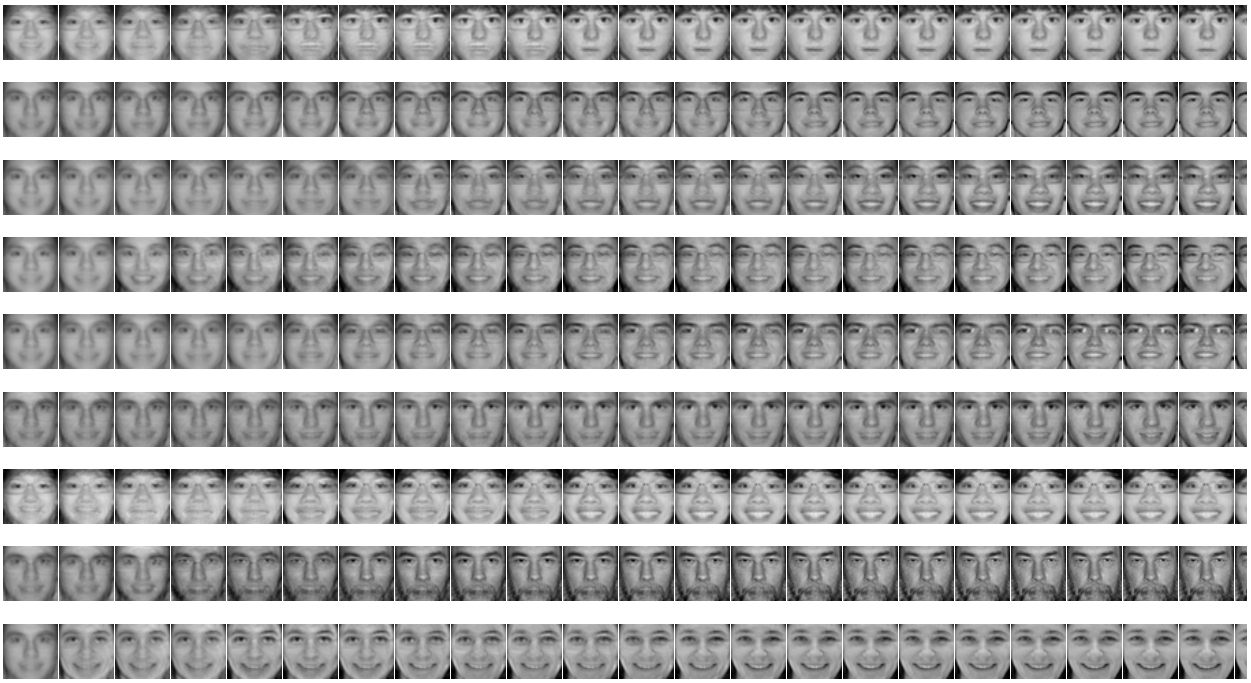
```
rebuildVectorFromEigenfaces[coefficients_,
    meanFaceVector_, eigenfaces_, numEigenfaces_] :=
  meanFaceVector + Apply[Plus, (coefficients * eigenfaces)[[ ;; numEigenfaces]]];
```

### Reconstruct some training set faces

```
randomFaces =
  Map[vectorToImage, {rebuildVectorFromEigenfaces[eigenfaceCoefficients[[5]],
      meanFaceVector, eigenfaces, 3], rebuildVectorFromEigenfaces[
      eigenfaceCoefficients[[23]], meanFaceVector, eigenfaces, 24],
     rebuildVectorFromEigenfaces[eigenfaceCoefficients[[3]],
      meanFaceVector, eigenfaces, 13], rebuildVectorFromEigenfaces[
      eigenfaceCoefficients[[12]], meanFaceVector, eigenfaces, 6],
     rebuildVectorFromEigenfaces[eigenfaceCoefficients[[24]],
      meanFaceVector, eigenfaces, 1], rebuildVectorFromEigenfaces[
      eigenfaceCoefficients[[2]], meanFaceVector, eigenfaces, 22]}];
Grid[Partition[randomFaces, Length[randomFaces] / 2], Spacings → {0, 0}]
```

```
reconstructedTrainingFaces = Table[
   Table[
    ImageResize[
     vectorToImage[
      rebuildVectorFromEigenfaces[eigenfaceCoefficients[[photoNum]],
       meanFaceVector, eigenfaces, numEigenfaces]
     ], {40}],
    {numEigenfaces, numPhotos}],
   {photoNum, numPhotos}
  ];
Grid[reconstructedTrainingFaces, Spacings → {0, 0}]
```

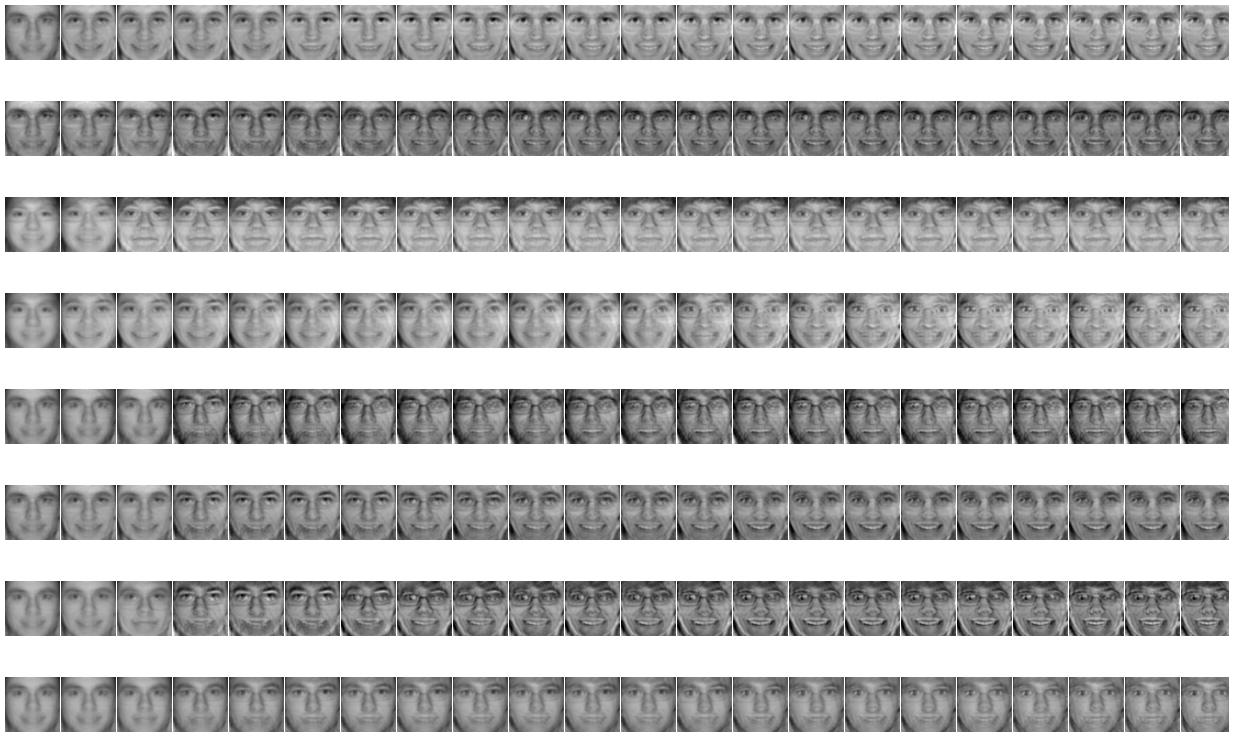




## Reconstruct some faces not in training set

```
otherPhotos = Map[Import, {"fb1.jpg", "fb2.jpg", "fb3.jpg", "fb4.jpg",
    "orangutan.jpg", "steven.jpg", "kirby.jpg", "flower.jpg"}];
numOtherPhotos = Length[otherPhotos];
otherGrayPhotos = Map[ColorConvert[#, "Grayscale"] &, otherPhotos];
Grid[Partition[otherGrayPhotos, Length[otherPhotos]], Spacings → {0, 0}]
otherGrayPhotosVector = Map[imageToVector, otherGrayPhotos];
```

```
otherEigenfaceCoefficients =
  Map[projectImageToFaceSpace[#, meanFaceVector, eigenfaces] &,
   otherGrayPhotosVector];
imageSize = 50;
otherReconstructedFaces =
  Table[
   Table[
    ImageResize[
     vectorToImage[
      rebuildVectorFromEigenfaces[otherEigenfaceCoefficients[[photoNum]],
       meanFaceVector, eigenfaces, numEigenfaces]
     ], {imageSize}],
    {numEigenfaces, numPhotos}],
   {photoNum, numOtherPhotos}
  ];
{Grid[otherReconstructedFaces, Spacings → {0, 0}],
 Grid[Partition[Map[ImageResize[#, {imageSize}] &, otherGrayPhotos], 1],
  Spacings → {0, 0}]}
```



```
Grid[{otherGrayPhotos,
  Table[vectorToImage[rebuildVectorFromEigenfaces[otherEigenfaceCoefficients[[
      photoNum]], meanFaceVector, eigenfaces, numPhotos]],
   {photoNum, numOtherPhotos}]}, Spacings → {0, 0}]
```

# 6 - Measure Reconstruction Accuracy

## Define a function that gets the difference between two image vectors

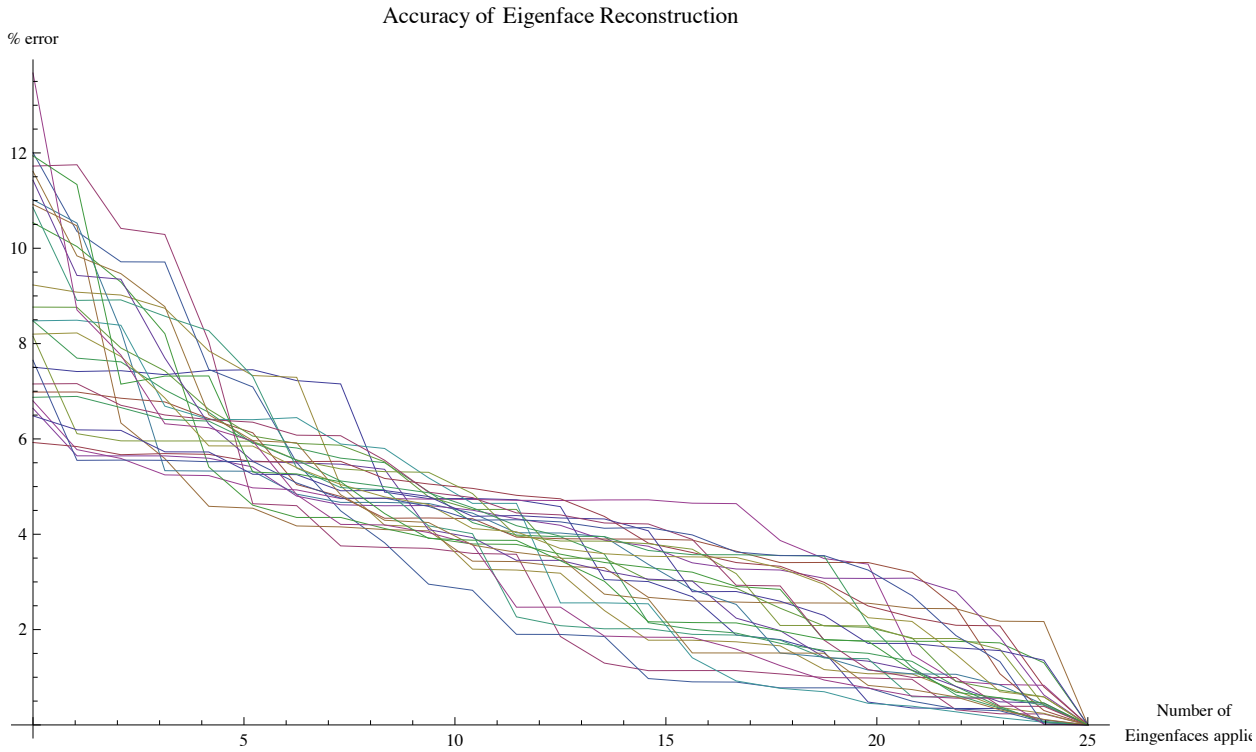
```
getImageVectorDifference[imageVector1_, imageVector2_] :=
  Module[{differenceVector},
   differenceVector = imageVector1 - imageVector2;
   Total@Abs@differenceVector / Times @@ Dimensions@imageVector1 * 100
  ];
```

## Calculate the differences between the normal images and the reconstructed images

```
imgNum = 21;
imgSize = 50;
accuracyTable = {
   Table[ImageResize[grayPhotos[[imgNum]], imgSize], {i, 0, 25}],
   Table[ImageResize[vectorToImage[rebuildVectorFromEigenfaces[
       eigenfaceCoefficients[[imgNum]], meanFaceVector, eigenfaces,
       numEigenfaces]], imgSize], {numEigenfaces, 0, 25}],
   Table[StringJoin[
     "\n",
     ToString[
      NumberForm[getImageVectorDifference[
        grayPhotosVector[[imgNum]], rebuildVectorFromEigenfaces[
         eigenfaceCoefficients[[imgNum]], meanFaceVector, eigenfaces, i]], 2]
     ],
     "%"], {i, 0, 25}]
  };
Grid[accuracyTable, Spacings → {0, 0}];
getAccuracyTable[imageVectors1_, imageVectors2_] := Module[
   {},
   {
    Map[vectorToImage, imageVectors1],
    Map[vectorToImage, imageVectors2],
    Table[
     StringJoin[
      "\n",
      ToString[
       NumberForm[
        getImageVectorDifference[imageVectors1[[1]], imageVectors2[[i]]], 2]
      ],
      "%"], {i, 1, 25}
    ]
   }
  ];
```

## Graph the accuracy over number of applied eigenfaces

```
diffs = Table[Table[getImageVectorDifference[grayPhotosVector[[faceNum]],
      rebuildVectorFromEigenfaces[eigenfaceCoefficients[[faceNum]],
       meanFaceVector, eigenfaces, numEigenfaces]],
     {numEigenfaces, 0, 24}], {faceNum, 1, 25}];
faceNum = 2;
ListPlot[diffs, Joined → True, InterpolationOrder → 1,
 DataRange → {0, 25}, PlotLabel → "Accuracy of Eigenface Reconstruction",
 AxesLabel → {"Number of\nEingenfaces applied", "% error"}]
```



# 7 - Facial Recognition

## To recognize a face in the training set, we seek the training set image with the minimum variance with the reconstructed image

```
getPercentMatches[imgNum_, numEigenfaces_] := Module[{},
   inputVector = rebuildVectorFromEigenfaces[eigenfaceCoefficients[[imgNum]],
     meanFaceVector, eigenfaces, numEigenfaces];
   differencePercentagesBetweenFaces = Table[
     getImageVectorDifference[grayPhotosVector[[i]], inputVector], {i, 25}];
   sortedOrder = Sort[differencePercentagesBetweenFaces]
  ];
getMatchedPhotos[matchOrder_] := Module[{},
   Table[grayPhotos[[Position[
       differencePercentagesBetweenFaces, sortedOrder[[i]]][[1, 1]]]], {i, 25}]
  ];
getPercentMatchesOther[imgNum_, numEigenfaces_] := Module[{},
   inputVector = rebuildVectorFromEigenfaces[otherEigenfaceCoefficients[[
      imgNum]], meanFaceVector, eigenfaces, numEigenfaces];
   differencePercentagesBetweenFaces = Table[
     getImageVectorDifference[grayPhotosVector[[i]], inputVector], {i, 25}];
   sortedOrder = Sort[differencePercentagesBetweenFaces]
  ];
```

## (Optional) Create a grid that shows the order of the face matches

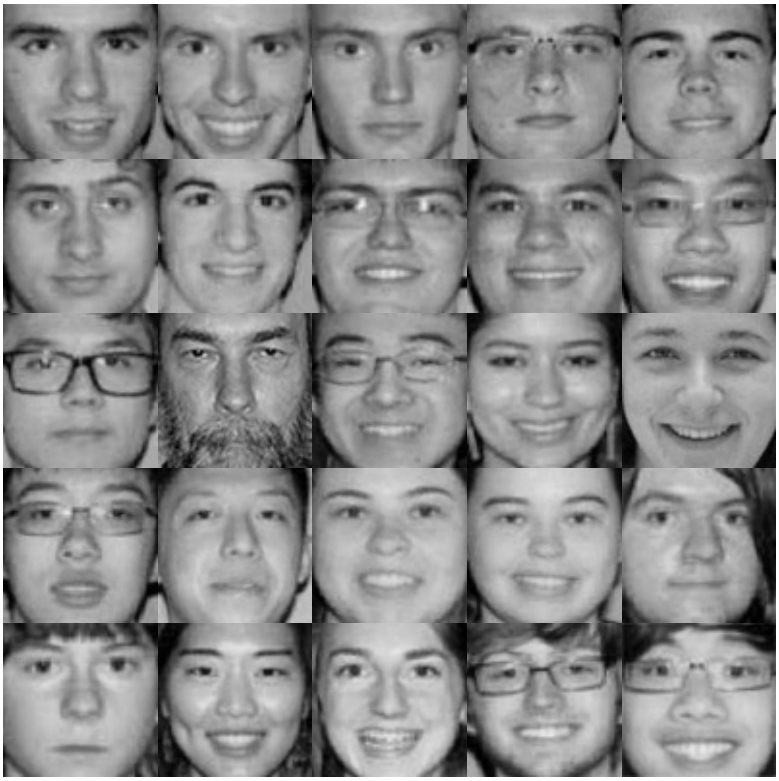
```
Style[Grid[Partition[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
    17, 18, 19, 20, 21, 22, 23, 24, 25}, 5], Frame → All], FontSize → 60]
```

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

## Recognize Training Faces

### Choose a face number (faceNum) from the training faces set and number of eigenfaces to apply (numEigen) to recreate the face

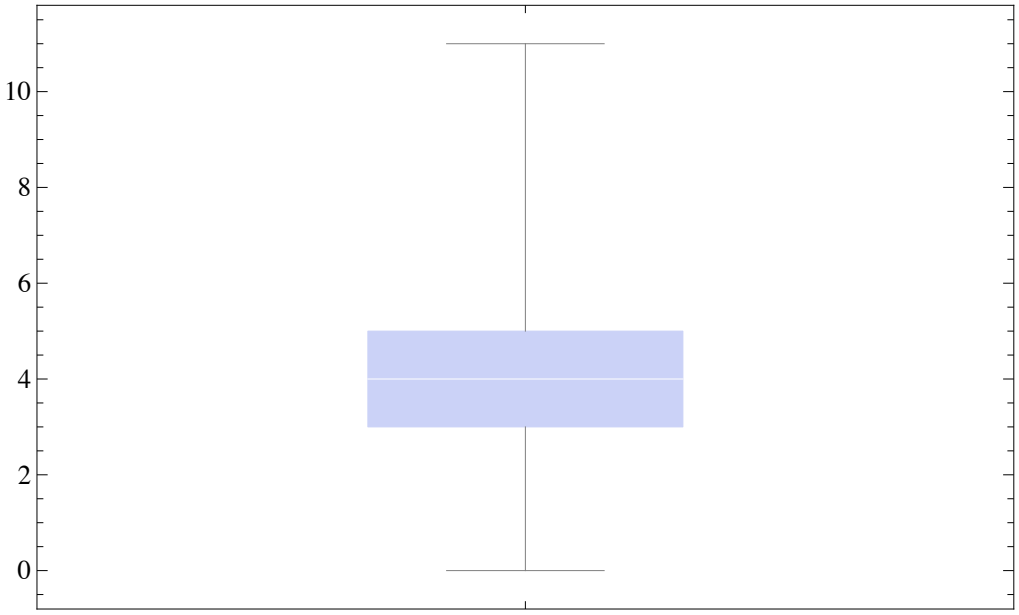
```
faceNum = 16;
numEigen = 8;
recreatedPhoto = vectorToImage[rebuildVectorFromEigenfaces[
    eigenfaceCoefficients[[faceNum]], meanFaceVector, eigenfaces, numEigen]];
Grid[{{grayPhotos[[faceNum]], recreatedPhoto}}, Spacings → {0, 0}]
order = getPercentMatches[faceNum, numEigen];
Grid[Partition[getMatchedPhotos[order], 5], Spacings → {0, 0}]
SetOptions[BarChart, BaseStyle → {FontFamily → "Times", FontSize → 14}];
BarChart[order, ChartStyle → "Pastel", AxesLabel → {"Faces", "% error"},
 PlotLabel → "Recognizing a Reconstructed Training Set Face",
 ChartLabels -> {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12",
   "13", "14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24", "25"}]
```



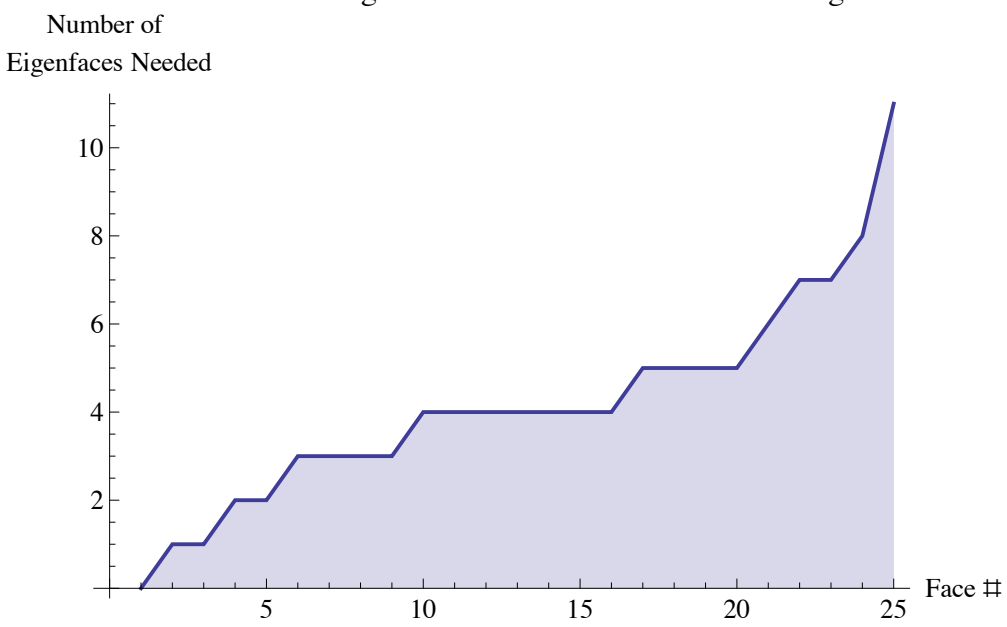

## Results of Recognizing Training Faces

```
numEigenToRecognize =
  {8, 5, 7, 3, 4, 3, 2, 4, 2, 5, 6, 4, 4, 4, 0, 11, 4, 3, 5, 3, 7, 1, 1, 4, 5};
SetOptions[ListPlot, BaseStyle → {FontFamily → "Times", FontSize → 14}];
SetOptions[BoxWhiskerChart, BaseStyle → {FontFamily → "Times", FontSize → 14}];
GraphicsGrid[{{BoxWhiskerChart[numEigenToRecognize,
    PlotLabel → "Number of Eigenfaces Needed Until Face is Recognized"]},
  {ListPlot[Sort[numEigenToRecognize], Filling → Axis,
    PlotStyle → {Thick, PointSize[Large]}, Joined → True,
    PlotLabel → "Sorted Number of Eigenfaces Needed Until Face is Recognized",
    AxesLabel → {"Face #", "Number of \nEigenfaces Needed"}]}}]
```

Number of Eigenfaces Needed Until Face is Recognized




Sorted Number of Eigenfaces Needed Until Face is Recognized

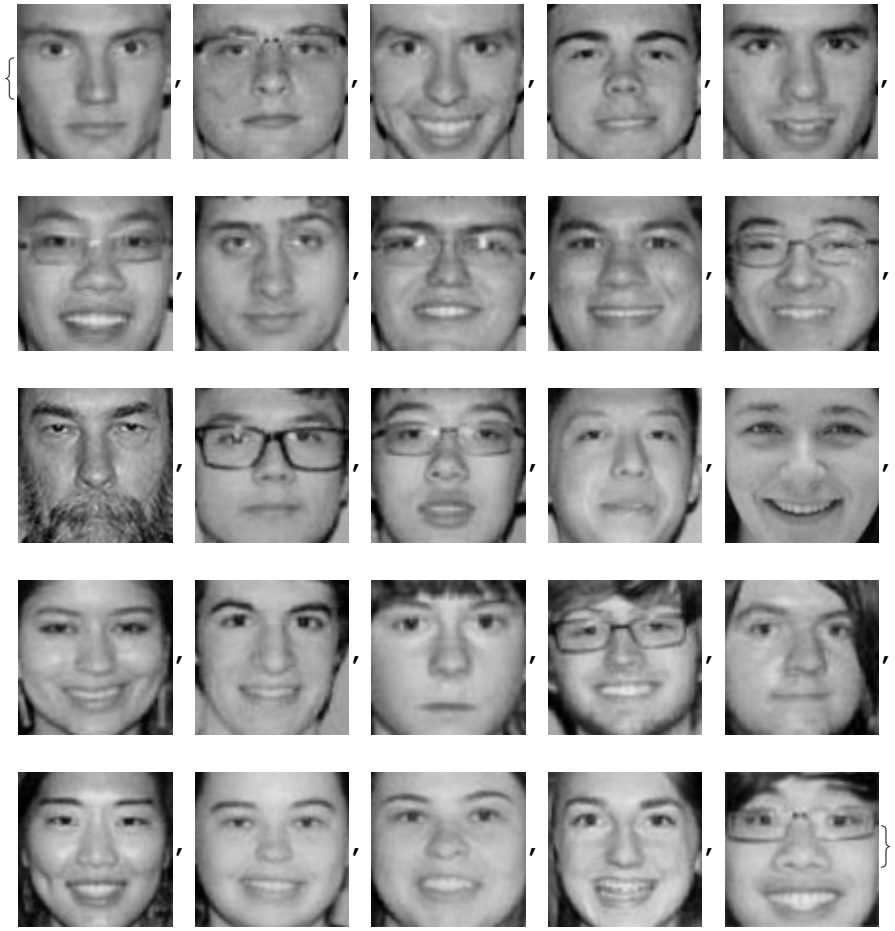


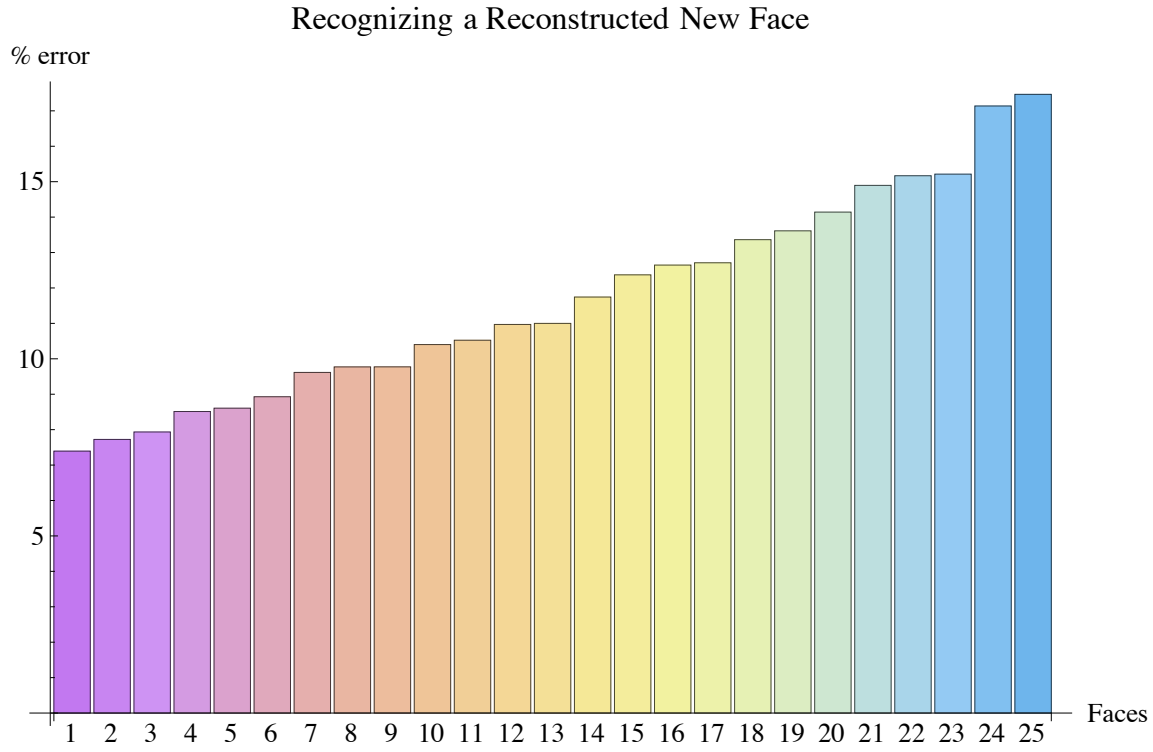
## Recognizing Foreign Faces

### Choose a face number (faceNum) from the foreign faces set and number of eigenfaces to apply (numEigen) to recreate the face

```
faceNum = 8;
numEigen = 24;
recreatedPhotos = Table[
   vectorToImage[rebuildVectorFromEigenfaces[otherEigenfaceCoefficients[[i]],
     meanFaceVector, eigenfaces, numEigen]], {i, numOtherPhotos}];
{otherGrayPhotos[[faceNum]], recreatedPhotos[[faceNum]]};
order = getPercentMatchesOther[faceNum, numEigen];
getMatchedPhotos[order]

BarChart[order, ChartStyle → "Pastel", AxesLabel → {"Faces", "% error"},
 PlotLabel → "Recognizing a Reconstructed New Face",
 ChartLabels -> {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12",
   "13", "14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24", "25"}]
```




Recognizing a Reconstructed New Face
% error
15
10
5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Faces

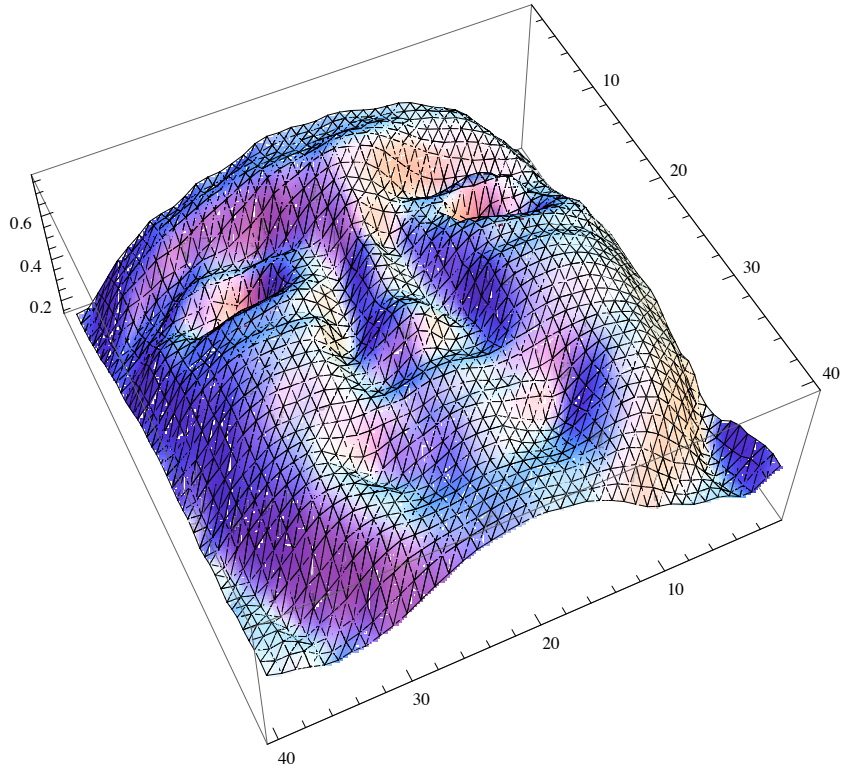
## Results of Recognizing Foreign Faces

```
t = Table[{Grid[{{otherGrayPhotos[[i]], recreatedPhotos[[i]]}}, Spacings → {0, 0}],
    Grid[{getMatchedPhotos[getPercentMatchesOther[i, 24]][[1 ;; 5]]},
     Spacings → {0, 0}]}, {i, 8}];
Grid[t, Spacings → {2, 0}]
```

# (Optional) Face Visualizations

## Mean Face Luminosity Visualization

```
imgSize = 40;
ListPlot3D[Partition[imageToVector[
    ImageResize[meanFaceImage, {imgSize, imgSize}]], imgSize], Mesh → All]
```



## Face Luminosity Visualization

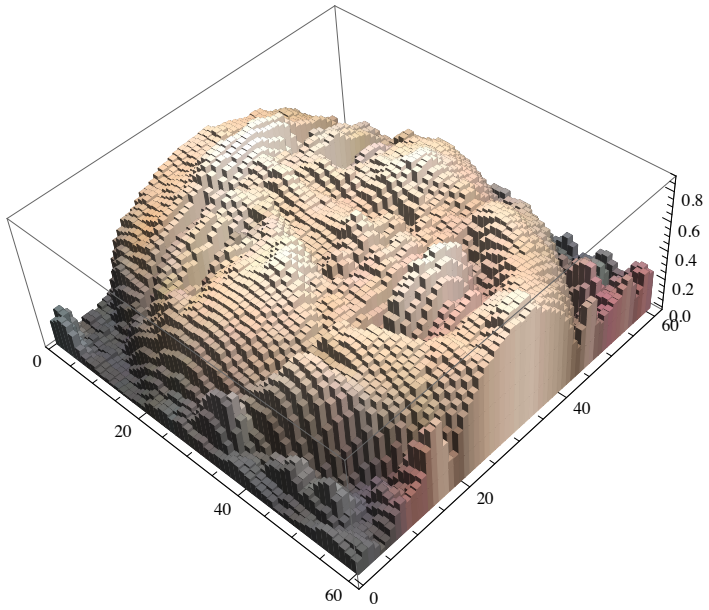
```
density = 60;
picNum = 9;
alpha = 1;
blurValue = 0;
colorData =
  ImageData[Blur[ImageResize[photos[[picNum]], {density, density}], blurValue]];

getColor[x0_, y0_] := Module[
   {x = x0, y = y0},
   y *= density;
   x *= density;
   If[y < 1, y = 1,];
   If[x < 1, x = 1,];
   If[y > density, RGBColor[0, 0, 0, alpha],
    If[x > density , RGBColor[0, 0, 0, alpha], pictureColor[Round[x], Round[y]]]]
  ];

pictureColor[x_, y_] := RGBColor[colorData[[x]][[y]][[1]],
   colorData[[x]][[y]][[2]], colorData[[x]][[y]][[3]], alpha];
resizeImage[i_] :=  Blur[ImageResize[grayPhotos[[i]], {density, density}],
   blurValue];
getImageVector[i_] := Transpose[
   Partition[imageToVector[resizeImage[i]], density]];

v1[i_] := ListPlot3D[{getImageVector[i]}, Mesh → None,
   InterpolationOrder → 1, ColorFunction → Function[{x, y, z}, getColor[x, y]]];
v2[i_] := ListPointPlot3D[{getImageVector[i]},
   ColorFunction → Function[{x, y, z}, getColor[x, y]],
   BoxRatios → {1, 1, .5}, PlotStyle → PointSize[Large]];
v3[i_] := ListPlot3D[{getImageVector[i]}, InterpolationOrder → 0, ColorFunction →
    Function[{x, y, z}, getColor[x, y]], Mesh → None, BoundaryStyle → None];
v4[i_] := DiscretePlot3D[getImageVector[i][[k]][[j]], {j, 1, density},
  {k, 1, density}, ColorFunction → Function[{x, y, z}, getColor[x, y]],
  ExtentSize → Full, PlotStyle → EdgeForm[], ExtentElementFunction → "Cube"]
```

```
v4[picNum]
```




## Animations

```
(*Export["/animation.gif",Table[rebuildFromEigenfaces[
    coefficients, meanImage, eigenfaces, k], {k, 1, 25}],"GIF"];*)
```