Pattern Recognition using Principal Component Analysis

By Nilangshu Bidyanta, March 21, 2010
http://sites.google.com/site/binarydigits10/
(Version 1.8 – August 31, 2010)

The problem:

Imagine we have an N–Dimensional data set. We want to represent this data with lesser dimensions (i.e. reduce its size) thus making it easier to store / analyze / handle the data. But at the same time, we want to preserve most of the information that this data represents (since a reduction in the number of dimensions *will* result in a loss of information). Or, from the pattern recognition point of view, let's say we want to classify a set of patterns, represented by images, and then use this set as a lookup table to identify similar patterns. Storing all images in the memory, may not meet the memory constraints. One way out is to retain only prominent features which make up the patterns.

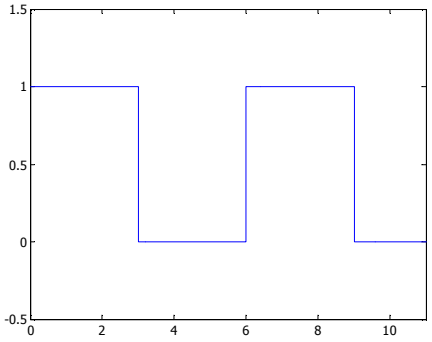
One of the solutions:

Use a multivariate technique (it's math-jargon wherein observations depend on multiple variables) to project the data onto orthogonal vectors and take only a handful of those components -- the ones which are large enough and then represent the original data in terms of these components. This method is aptly termed as Principal Component Analysis or PCA for short. This method is also popularly known as Eigenface because of its application in face detection / recognition systems.

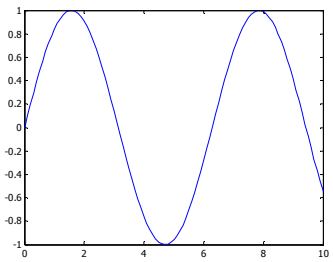
Introduction:

So the above didn't make much sense! But you'll be able to understand most of it by the time you finish this article, so be patient! As for now, analogous to PCA is Fourier series expansion of signals, where a periodic signal is broken up into basic sinusoids having frequencies that are multiples of the fundamental frequency.

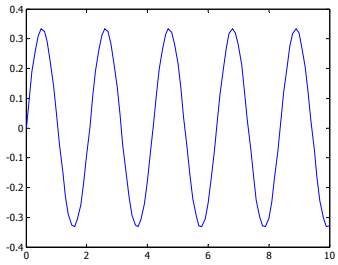
It has been reproduced here for the reader's convenience. Assume that there is a need to decompose the following square wave into a sum of harmonically related sinusoids. The mathematics of obtaining the Fourier series representation isn't shown here because intuitive understanding is more important.



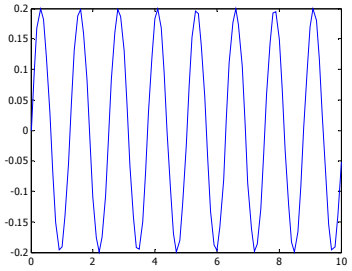
The math gives us an infinite number of sinusoids and assuming the ideal case in which all of them can be stored in memory and added, it would result in the above square wave (subject to Gibbs' phenomenon, of course). In this example, only the first six (which are also the six most prominent components) are taken and added together.



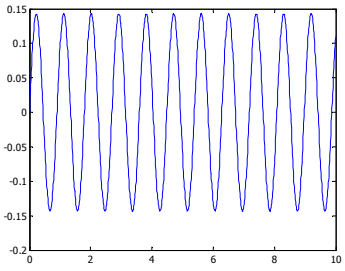
$\sin(t)$



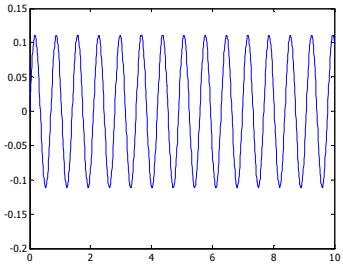
$\frac{\sin(3t)}{3}$



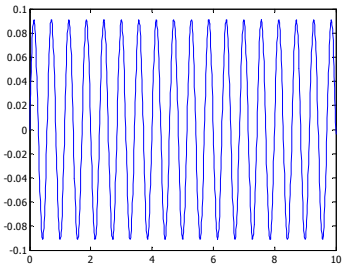
$\frac{\sin(5t)}{5}$



$\frac{\sin(7t)}{7}$

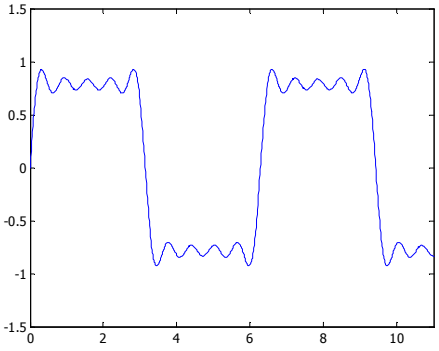


$\frac{\sin(9t)}{9}$

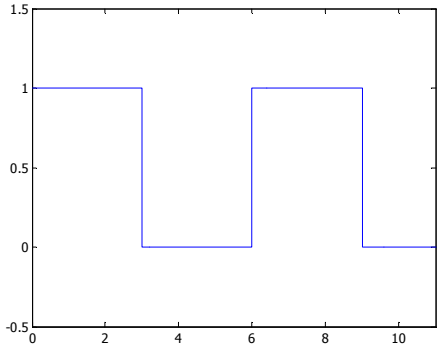


$\frac{\sin(11t)}{11}$

Summing the above sinusoids the following approximation of the square wave is obtained:



Approximation



Actual Square wave

Getting back to the analogy, the “image” is the square wave; the sinusoids correspond to the “projection” of the “image” on the orthogonal vectors that were mentioned before. We took only “a handful of components” (i.e. only the most prominent ones) since we haven’t summed the sinusoids all the way to infinity. The approximation of the square wave corresponds to the original image being re-constructed using its components.

If every image can be broken into such components, they are bound to share some components. If that is the case, a relatively large number of images can be represented by a handful of components just like signals other than the square wave can be decomposed into a sum of sinusoids too. This eases the task of classification of images.

Preliminaries:

Now that you have a basic idea of what the problem is and how it can be solved, we get into the mathematics. Be sure to read **Appendix A** if you do not have an intuitive understanding of multi-dimensional vector space. **Appendix B** has a few theorems related to matrices (particularly of eigenvectors) for those who are mathematically inclined. The reader is assumed to have gone through a basic course in matrices, statistics, computer (MATLAB) programming and of course, common sense!

Introduction to PCA:

The basic idea in pattern recognition is to compare the input pattern with ones that are already stored in the memory and check for a match. Obviously one could simply store the images in memory and do a pixel-to-pixel comparison to check if the patterns match. So why don't we do it?! It's because of noise originating from the image capturing device and also due to slight variations resulting from placement, orientation and source of the patterns. PCA helps to overcome these problems to a certain extent. It does so by reducing some of the information content of the image while preserving information about certain prominent features. Generally "feature" refers to characteristics of the image such as eyes, nose etc. as in case of a human face. But it's difficult for a computer to understand features in a way we humans do.

Images are generally represented as 2D matrices in computers. But to apply PCA, it is convenient to represent images as vectors. How the 2D → 1D conversion is actually done is discussed a little later. Once converted to a vector, each pixel in the image is treated as a dimension in a multi-dimensional space (incase of doubt, refer **Appendix A**). In this entire article, images and patterns have been used interchangeably.

To make computers recognize features, the basis of the image vector is rotated to orient it with the features present in the image. The new basis is also orthogonal, but projection of the image on the new basis gives the amount or weight of a particular feature that the image is made up of. Each orthogonal vector of the basis is regarded as a feature vector. The projections on the feature vectors are obtained by taking the dot product of the position vector of the image with each orthogonal component. Note that "features" here does not refer to a human's interpretation of the image. These are simply vectors which do not help in intuitively interpreting the image but point to mathematical patterns hidden within the image data. Technically, the directions along which the image data varies (i.e. the directions of the basis vectors) in multi-dimensional space are called Principal axes.

For example, the pattern can be made up of 20% of Feature1, 7% of Feature2, -80% of Feature3 (there is a minus before 80%) etc. where the percentage values can refer to normalized magnitudes of the image along the feature vectors. Also note that the combination of feature vectors must be **linear**. Each image put through PCA is decomposed **linearly** in terms of these features. Thus, instead of storing all reference images in the memory, only the weights of the features need to be stored – it's light on the memory and processing is faster! If the weights of the feature of an input image matches or are nearby to ones stored in the memory, the pattern is recognized. The set of reference images (i.e. the ones used to form the feature vectors) is also called training set.

We can further reduce memory utilization and speed up processing by reducing the number of features, taking only the most prominent features. Also, a reduction in features is called for since the less prominent ones will be more affected by noise. Redundancy might be prevalent in the less prominent features. How many features do we remove? That is best found out by experimenting with the training set keeping in mind that relevant information should not be lost. After finding out which dimensions should be retained, the image vector is expressed in terms of components lying along the k orthogonal unit vectors (the selected normalized feature vectors), where k is the number of dimensions that have been retained and n is the original number of dimensions. Obviously, here k is less than n.

Mathematically, the last few paragraphs mean that having an image data matrix X, there exists an orthogonal matrix V (containing the basis vectors), which acts as a transformation matrix for X, to give Y. The image data matrix is nothing but a matrix in which the columns are individual images in vector form. The final matrix Y is the original data expressed in terms of components along the new basis vectors.

$$Y = VX$$

The above equation is obvious since a matrix multiplication is nothing but rotation and scaling of the vectors contained in the image data matrix X. If we closely inspect the above multiplication, it is seen that:

$$Y = \begin{bmatrix} \vec{v}_1 \\ \vdots \\ \vdots \\ \vec{v}_n \end{bmatrix} \begin{bmatrix} \vec{x}_1 & \cdots & \cdots & \vec{x}_n \end{bmatrix}$$

where elements of the first matrix are the basis vectors (arranged in rows) and those of the second matrix are column vectors of the original data matrix. On expanding the above matrix:

$$Y = \begin{bmatrix} \vec{v}_1 \cdot \vec{x}_1 & \cdots & \cdots & \vec{v}_1 \cdot \vec{x}_n \\ \vec{v}_2 \cdot \vec{x}_1 & \cdots & \cdots & \vec{v}_2 \cdot \vec{x}_n \\ \vdots & \vdots & \vdots & \vdots \\ \vec{v}_n \cdot \vec{x}_1 & \cdots & \cdots & \vec{v}_n \cdot \vec{x}_n \end{bmatrix}$$

Each column gives the components of the corresponding image data column vector in terms of the new basis. Of these, only those basis vectors along which the data shows large variability are selected. Summarizing, we can say that:

<Training set + noise> $\xrightarrow{\text{PCA}}$ <Feature set prominent in the given group of images>
+
<Training images expressed in terms of the features>

<Input image + noise> $\xrightarrow{\text{PCA}}$ <Reconstructed image in terms of elements of feature set>

So basically, we need to find the orthogonal matrix V from the information given in the data matrix X such that it satisfies the matrix equation and conditions imposed on Y. In the context of pattern recognition, V is found by analyzing training images which are representatives of patterns we are trying to recognize. For example, if we are designing a handwriting recognition system, the training images would be different characters written multiple times by different people. For face recognition, a set of images of faces is needed.

The training set:

The training set contains normalized images of the patterns to be recognized. By normalized, it is meant that the image features (as perceived by humans) should be aligned across all images; say, in face recognition, the position and orientation of eyes, nose, chin etc. should be same for all images. Also the images should be of the same size and share the same color space – grayscale (helps save computation time since it is assumed that color information does not contain any feature vectors).

I'm assuming that we already have the training images, properly oriented and scaled. Let a set of these images (assuming there are m elements in this set) be represented as (the k^{th} one is shown below):

$$I_k = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nn} \end{bmatrix}_{n \times n} \quad 0 \le a_{ij} \le 255,\ a_{ij} \in I$$

It is assumed that the images are grayscale, square and are of dimension $n \times n$.

A set of 25 training images for face recognition
Image: Computer vision Eigenface tutorial, Drexel University

The next step is to convert every image matrix into a column vector by placing each row after the previous one and then re-arranging the resultant row matrix in a column.

$$I_k = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nn} \end{bmatrix}_{n \times n} \quad \Rightarrow \quad I_k = \begin{bmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1n} \\ a_{21} \\ \vdots \\ a_{2n} \\ \vdots \\ a_{n1} \\ \vdots \\ a_{nn} \end{bmatrix}_{n^2 \times 1}$$

Note the dimensions of the column vector – this should be obvious.

From this point on, there is more than one way to intuitively explain the solution. I've tried to provide a summary of all the explanations that I've gone through.

Feature Set Extraction:

After the training set is obtained, we need to extract the features characteristic to this set of patterns. For getting a better understanding of the concepts involved, explanation will be provided assuming the dimensions are limited to 2 and at max. 3. These can be extended to higher dimensions later – use your imagination!

A part of this explanation is from a paper by Jonathon Shlens of the Center for Neural Science, New York University.

To obtain the feature set, the common features present in all the images are removed. While applying PCA we are interested only in the unique features of each image, so we remove a part of the redundant data from the input patterns. This is done by subtracting the "average pattern", defined for the image vector as below, from each of the training patterns. The set of image vectors we obtain after performing this operation is called the mean centered training set.

$$\bar{I} = \frac{1}{m} \sum_{k=1}^{m} I_k$$

$$I_{ck} = I_k - \bar{I}$$

In the second equation, I_{ck} is the k^{th} image centered at its mean.

Before we move on, let's go through some basic concepts of statistics and how they can be applied to multi-dimensional data.

Variance & Covariance:

Variance is given by the following equation:

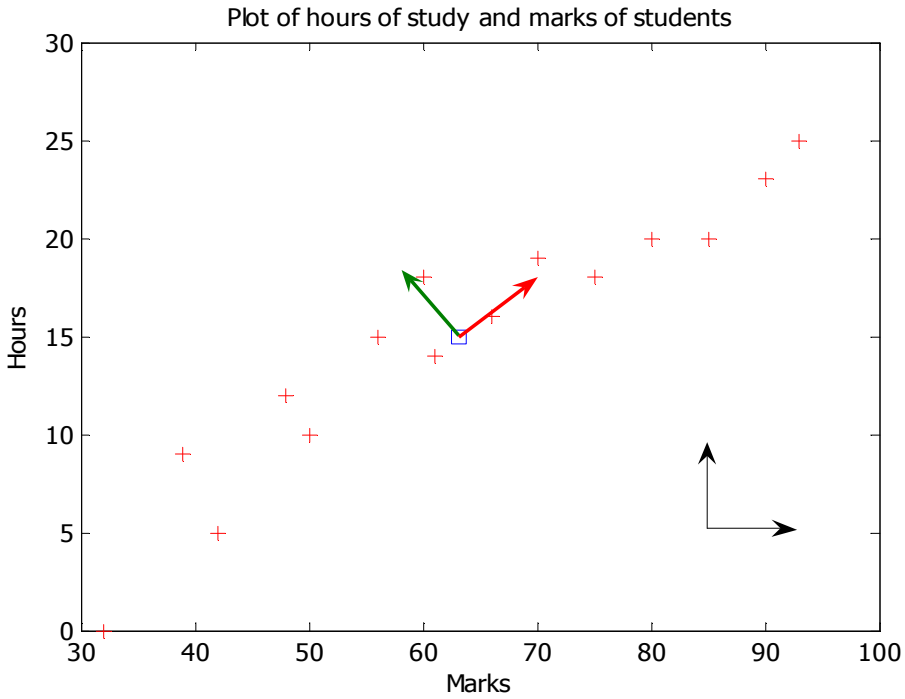
$$\sigma^2 = \text{var}(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where n is the number of samples and rest of the symbols have standard meanings. What this basically denotes is the square of deviation of the data from its mean. A large variance denotes that the data has a lot of variability in it. Data with smaller variance might indicate less interesting dynamics or might even imply noise. Just think about it – how can a measurement be unique if it remains constant across all observations? A variance of zero implies that the data remains constant.

A more generalized form of variance is the covariance which is defined as below. Covariance is defined as the degree of linear relationship that the two variables possess.

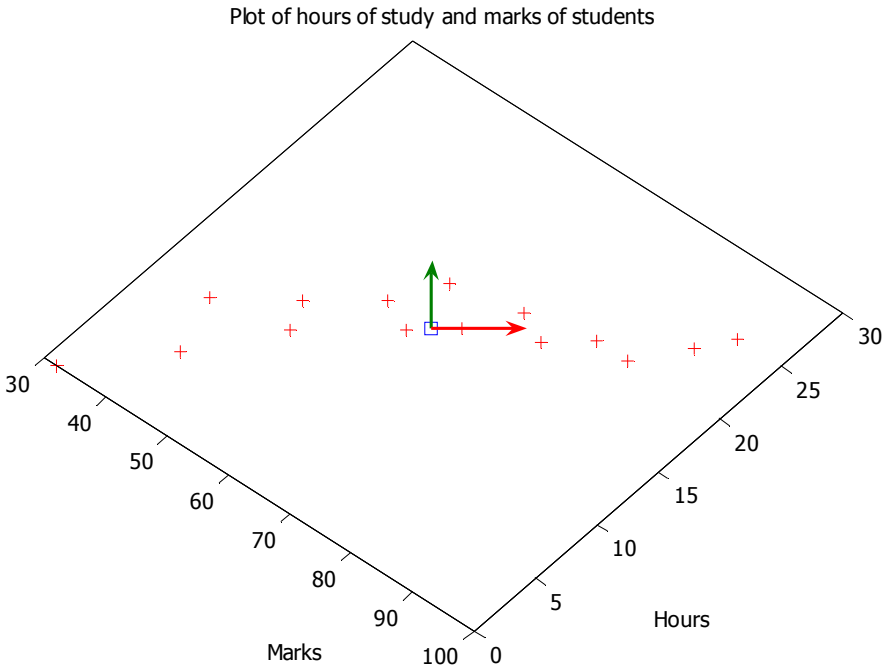
$$\text{cov}(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

Clearly, *cov(x, x)* = variance of *x*. The magnitude of covariance of a pair of data shows how closely they are related. The sign shows if they vary proportionally (positive) or inversely (negative). If the covariance is zero then the two data are completely unrelated. Let's take an example to understand the above concepts. A survey was conducted to see how hours of study affect the marks of students. This is what the study showed:



The covariance of the above set of parameters is 121.4381 and the variance of *hours* is 46.0667 and the variance of *marks* is 359.8381. The blue square in the above plot has the coordinates of the mean of *hours* and *marks*. Currently the axis of reference is the one in the bottom right corner (in black). But let's say we had the option of rotating the axis to the one centered at the mean.

This is exactly what PCA does for multi-dimensional data. One of the arms of the new axis lies along the direction of maximum variance of the data set. Remember variance is the variability in measurement with reference to the expected value or mean. The maximum deviation of data is seen along the direction of the red arrow. The green arrow points towards the direction of minimum variance. For higher dimensional data, green points in the direction of the next largest variance. So now the question arises, why pick this orientation for the new axis?!

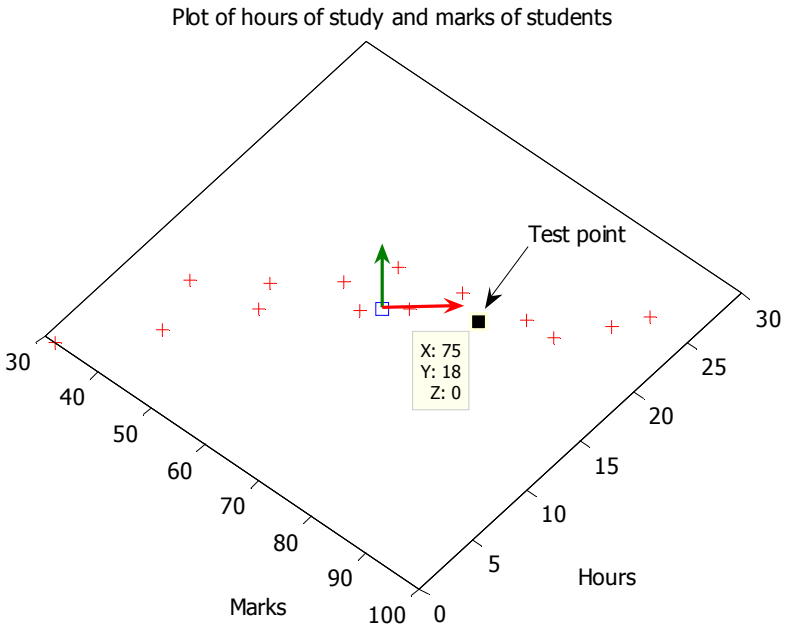


Rotated axis of reference is shown by the green and red arrows.
Direction of maximum variance is given by the red arrow.
The direction of the next largest variance is given by the green arrow.

Well, one reason is that a smaller variance indicates an almost constant value which is evident from the formula. These don't characterize the data that well. Only those measured variables which show a large variance contribute to a data point's uniqueness. Since the data is 2D, we only have 2 axes. But for multi-dimensional data, the first axis is in the direction of largest variance, the next axis is in the direction of the 2^{nd} largest variance and so on.

Another reason for changing the axes or more technically, the basis, is to better represent the data since the re-expressed data will be in terms of feature vectors because they lie along the directions of decreasing variability. Thus the feature vectors are arranged according to their importance.

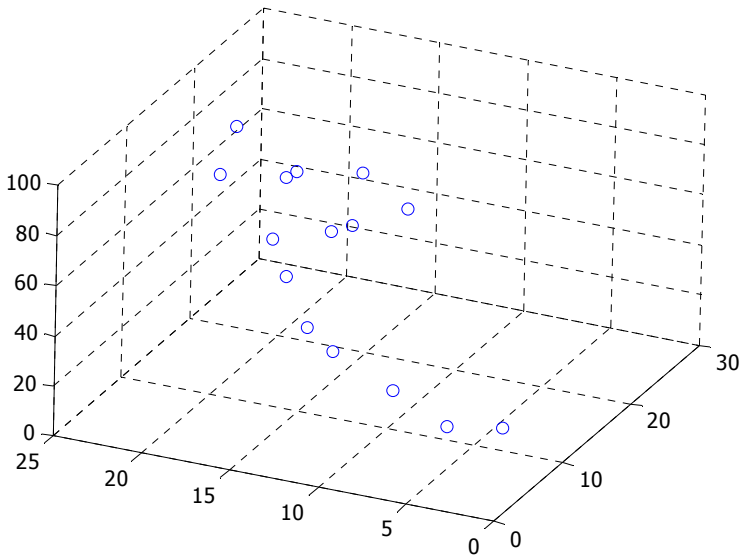
The paragraph is explained with an example.



How is the data read after change of basis? This can be explained by taking the case of the test point shown in the figure above. It is represented by (75 marks, 18 hours) in the old coordinate system. Let's give arbitrary names to the new axes – *red* and *green*. Variations along these axes could be called being *redder* or *greener*. Since each point in the plot corresponds to a student, the following statement is true – the *test student* is around 17 units redder and 3 units less green than the average student. It can be clearly seen that the red and green axes are nothing but characteristics of the student, with red being a more important characteristic than *marks* **or** *hours*. This is because the variability along the red axis is more than the variability of *marks* and *hours*. Risking over-simplification in the 2D case, the above set of points can be *completely* represented in terms of the projections along the *red* axis.

Okay, so we've got a multi-dimensional data set which in our case is an n^2 dimensional image vector. The new axes have been oriented to match the direction of maximum variability but there's another condition which needs to be satisfied – no two dimensions should convey similar information. Repetition of information is called redundancy. Thus we want the new basis to exhibit minimum redundancy.

So how does one represent this condition mathematically? With covariance! Consider the following 3D plot. Here we have data from 3 sources which **should not** have any linear relationship between them.



The above plot was obtained by using 3 vectors, plotted as *(X,Y,Z) = (g,h,m)*

$$g = [10 \quad 15 \quad 24 \quad 15 \quad 11 \quad 17 \quad 1 \quad 21 \quad 6 \quad 18 \quad 17 \quad 19 \quad 21 \quad 11 \quad 19]$$
$$h = [9 \quad 15 \quad 25 \quad 14 \quad 10 \quad 18 \quad 0 \quad 20 \quad 5 \quad 19 \quad 16 \quad 20 \quad 23 \quad 12 \quad 18]$$
$$m = [89 \quad 12 \quad 48 \quad 64 \quad 13 \quad 75 \quad 35 \quad 21 \quad 17 \quad 69 \quad 54 \quad 41 \quad 77 \quad 97 \quad 8]$$

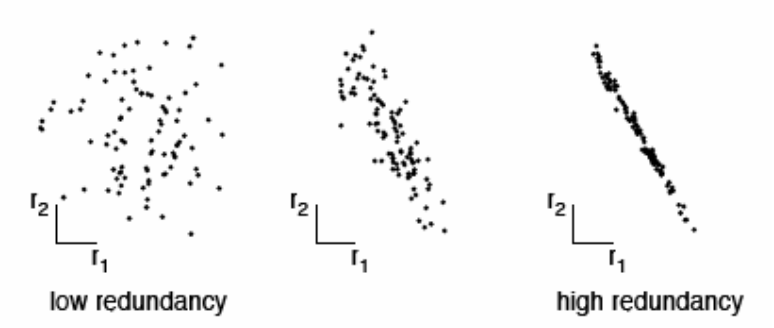
Now let's take a look at the covariance of the dimensions – g, h and m

$$\text{cov}(g,h) = 41.57$$
$$\text{cov}(g,m) = 10.92$$
$$\text{cov}(h,m) = 27.42$$

Remember, the magnitude of covariance gives how closely the two dimensions are related linearly. Since all of them are positive we can say that the dimensions vary proportionally. But looking at the magnitude of covariance between g and h indicates that these two vary almost exactly in the same way, which means that their ratio remains more or less a constant. Thus knowing one, the other can be approximated easily. Hence one of them can be scrapped without much loss in information conveyed by the data as a whole. But which one should we remove? The answer can be found by looking at the covariance of h and m which is higher than that between g and m.

Hence dimension h can be safely removed without much loss of relevant information. The data can be more concisely represented without dimension h and this is the central idea behind dimensionality reduction in PCA. The previous analysis implies that **not all** the sources were unique as they were thought to be. Hence covariance can be used to find out which dimensions of the new basis are redundant.

The following figure tries to give an intuitive understanding of the relation between redundancy and covariance (taken from the paper by Jonathan Shlens)



High redundancy corresponds with
high magnitude of covariance (and vice-versa)

In the diagram above, r_1 and r_2 assumed to be independent dimensions of the data set. The figure on the left shows really low covariance and low redundancy. The one on the right has a high redundancy since its covariance is high.

By now we've introduced most of the concepts needed to solve the problem. Incase you feel over-overwhelmed, here's a short summary of everything so far:

1. The objective was to take the training patterns, calculate their features (which made them different from each other) and then store the *prominent* features for recognizing unknown patterns. Later, these features were used to re-express the patterns. Thus with dimensionality reduction we can easily identify unknown patterns with the knowledge of the magnitude of their projections on the feature vectors.

2. Calculating the feature vectors involved rotating the basis so that it corresponds to the direction of maximum variability in the multi-dimensional data space. Hence variance was introduced.

3. It was made sure that the new basis did not convey redundant features; hence the concept of covariance was introduced.

4. Also, the 2D image was converted to a vector and the mean was subtracted from all training vectors to remove *some* of the redundant features present in them.

The last set of equations we defined were the ones which centered the training vectors on their mean. It's reproduced here for the reader's convenience:

$$\bar{I} = \frac{1}{m}\sum_{k=1}^{m} I_k$$

$$I_{ck} = I_k - \bar{I}$$

Next, we define a matrix X as under; the mean centered image vectors form the columns of the matrix X.

$$X = \begin{bmatrix} I_{c1} & I_{c2} & \cdots & I_{cm} \end{bmatrix}_{n^2 \times m}$$

Let us define C_X as follows

$$C_X = \frac{1}{n} XX^T$$

On expanding the above expression (not shown here but can be carried out in a symbolic data handler like MATLAB) we get the following matrix:

$$C_X = \begin{bmatrix} \text{cov}(a_{11}, a_{11}) & \text{cov}(a_{11}, a_{12}) & \cdots & \text{cov}(a_{11}, a_{nn}) \\ \text{cov}(a_{12}, a_{11}) & \text{cov}(a_{12}, a_{12}) & \cdots & \text{cov}(a_{12}, a_{nn}) \\ \vdots & \vdots & \ddots & \vdots \\ \text{cov}(a_{nn}, a_{11}) & \text{cov}(a_{nn}, a_{12}) & \cdots & \text{cov}(a_{nn}, a_{nn}) \end{bmatrix}_{n^2 \times n^2}$$

C_X is called the covariance matrix. A closer look at this matrix shows that the main diagonal elements are the variance of each dimension and the non-diagonal elements are the covariance of the dimensions. Here, a single dimension corresponds to a single pixel of the image and its color value in grayscale gives a sense of magnitude along that dimension. The matrix is a **symmetric** matrix meaning that its transpose is equal to the matrix itself.

Getting back to the very first equation of this article

$$Y = VX$$

The matrix *X* represents the set of training vectors arranged in columns and centered at their mean. As mentioned before centering at the mean results in removal of some redundant features. So basically the matrix *X* represents a partial feature matrix of the training patterns conveying the information "how different each training image is from the average image". It forms the **partial** feature matrix since it *does not* correspond to the prominent features that were being discussed under variance. We're now in a position to find a matrix *V* that can transform X into *Y*. To do this, some conditions are set on *Y* in terms of its covariance matrix - C_Y

Recall that as far as variance and covariance are concerned, variance needs to be maximized and covariance needs to be minimized since:

1. A large variance indicates interesting dynamics in the training patterns; near zero or small variance indicates non-changing data.

2. A large magnitude of covariance indicates redundancy in data conveyed by the training images.

So the target covariance matrix C_Y needs to be a diagonal matrix since it needs to have non-zero variances and ideally zero covariance of its dimensions:

$$C_Y = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & \vdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}$$

Simplifying the basic equation for the target covariance matrix C_Y :

$$C_Y = \frac{1}{n} YY^T = \frac{1}{n}(VX)(VX)^T = V\left(\frac{1}{n} XX^T\right)V^T = VC_X V^T$$

The covariance matrix C_X is a symmetric matrix and hence eigen-decomposition (diagonalization) can be applied to it as follows:

$$C_X = PDP^{-1} = PDP^T$$

In the above equation, *P* is the matrix with eigenvectors of C_X as its columns. Also, since C_X is a symmetric matrix, the matrix *P* turns out to be an orthogonal matrix i.e. the transpose and inverse of *P* are equal. For proofs related to eigenvectors, refer to **Appendix B**.

Putting the decomposed value of C_X in the equation for C_Y :

$$C_Y = VPDP^T V^T = VPDP^{-1}V^T$$

Till now we haven't put any conditions on matrix *V*. There's a trick involved in solving the problem from this point on. We **assume** *V* to be equal to the transpose of *P* which makes the following equation true due to *P* being orthogonal:

$$V = P^T = P^{-1}$$

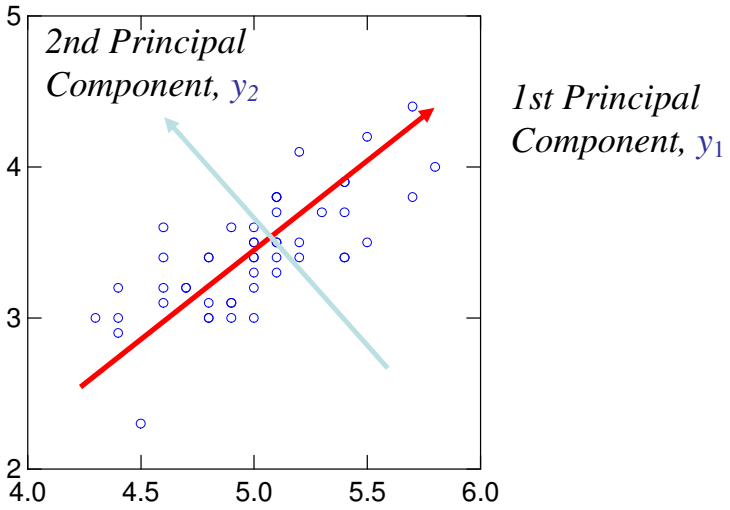
Substituting the value of *V* from the above equation into the equation for C_Y we obtain:

$$C_Y = P^{-1}PDP^{-1}\left(P^{-1}\right)^{-1} = IDP^{-1}P = IDI = D$$

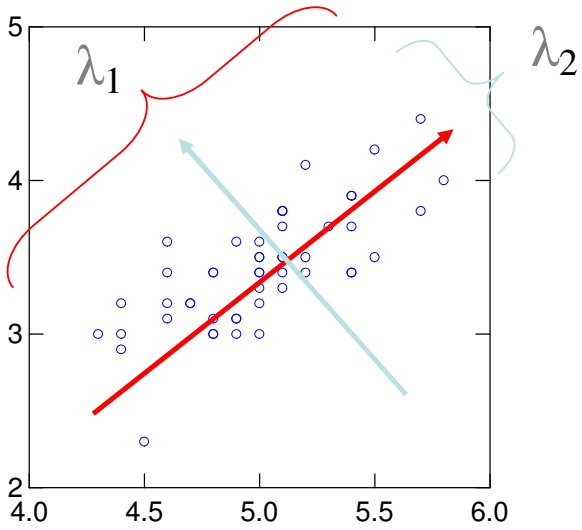
Thus we get C_Y equal to the diagonal matrix D (it is diagonal from the definition of eigen - decomposition). This is exactly what we wanted – the diagonal elements should be non-zero and non-diagonal elements should be zero.

Therefore the *V* is a matrix formed by arranging the eigenvectors of the covariance matrix of *X* in rows. The transformed matrix *Y* then turns out to be the projections of the members of matrix *X* on the eigenvectors. If you've gone through **Appendix B** you'll find that eigenvectors of symmetric matrices form an orthogonal basis vectors in R^n. Also, since the diagonal elements of *D* are nothing but the eigenvalues of C_X, the largest variance corresponds to the largest eigenvalue. Since eigenvalues and eigenvectors form a pair, an eigenvector with a large eigenvalue is a prominent feature vector.

Thus the direction of the eigenvector with the largest eigenvalue is the direction along which the data shows maximum variability.



Example of PCA in 2 dimensions



Eigenvalues and variance

There are two issues that need to be dealt with at this stage. One of them is that there are as many eigenvectors as the matrix's rank. If the matrix is a full rank matrix, **all** dimensions can be represented by eigenvectors. But if the matrix has a rank less than the dimension of the matrix then we need to "fill" *V* with orthogonal vectors to facilitate multiplication of *V* and *X*. Of course this doesn't change the final solution since the variance (eigenvalues) of these added vectors is zero.

The second problem lies in the calculation of the eigenvectors of the covariance matrix. The dimension of the covariance matrix is $n^2 \times n^2$ which is a **HUGE** number! If we take images that are 256 pixels in height and width, we'd end up with a covariance matrix of size 65536 x 65536. Calculating the eigenvalues and eigenvectors of such a large matrix is a pain in the *arse*!

To avoid this problem, the following manipulation is carried out:

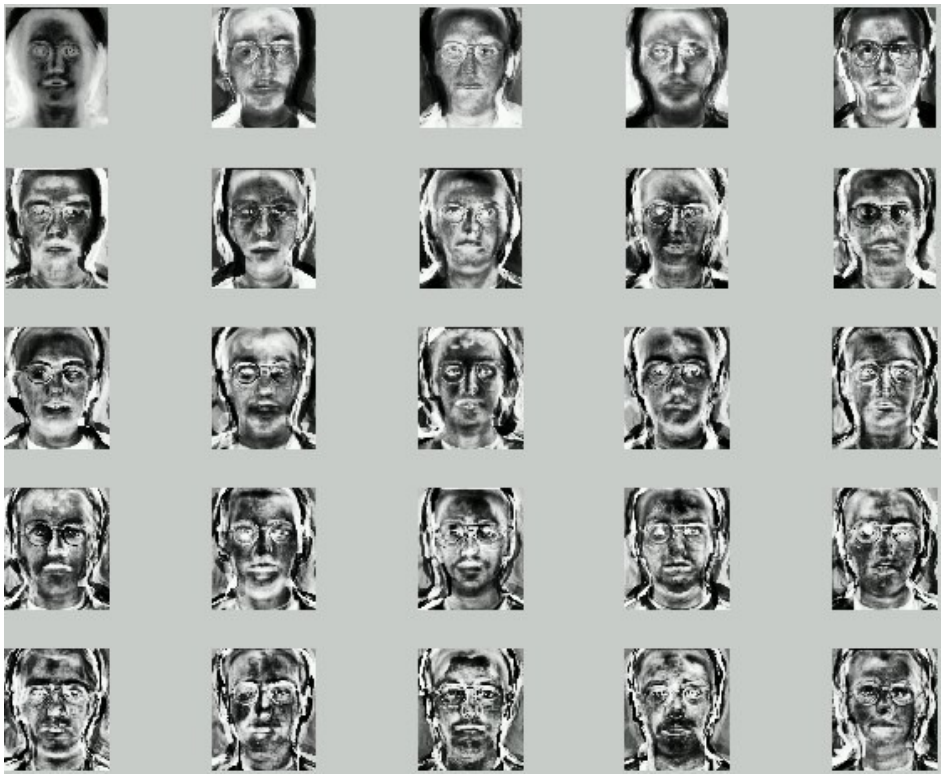
$$L = X^T X$$
$$Lv = \lambda v$$
$$\Rightarrow XLv = X\lambda v$$
$$\Rightarrow XX^T Xv = \lambda Xv \Rightarrow C_X Xv = n\lambda Xv$$
$$\Rightarrow C_X (Xv) = (n\lambda)(Xv) \Rightarrow C_X Z = \mu Z$$

Symbols have their usual meanings. The eigenvectors P, of the covariance matrix are found from the product of the matrix X and eigenvectors of matrix L. The dimension of L is m x m where the value of m generally varies from twenty to a few hundreds, proportionate to the number of training images. This reduces memory utilization.

Finally, to complete the training session, we need to project the training images on these eigenvectors and store the projections (dot product magnitude) on them as a vector of weights. But we do not take **all** eigenvectors for projecting the training images since only the ones with large eigenvalues (and hence large variances) form feature vectors which are significant to the data set. Let these weights be stored in a vector named Δ_i where i takes values over the range of training patterns.

$$\Delta_i = V \cdot (I - \bar{I}) = P^T (I - \bar{I}) = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_R \end{bmatrix}_{R \times n^2} \begin{bmatrix} I_{11} \\ I_{12} \\ \vdots \\ I_{nn} \end{bmatrix}_{n^2 \times 1} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_R \end{bmatrix}_{R \times 1}$$

where P_i are the first R eigenvectors (in decreasing order of eigenvalues) of the covariance matrix as column vectors and I is the pattern to be recognized and other symbols hold regular meanings.



Example of what the eigenvectors in PCA analysis look like once converted back to 2D matrices and viewed as images.

Pattern detection and recognition:

For pattern detection and recognition we first calculate the weights of the input pattern using the last equation. It is denoted by Δ. Then the Euclidean distance between the weight vectors of the input image and training images, taken one at a time, are calculated. If this distance falls below a particular threshold value, set heuristically, an input pattern similar to the class of patterns being analyzed is said to have been **detected**. For completeness, the formula to find the Euclidean distance in a multi-dimensional space is given below:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\cdots+(p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}.$$

where *p* and *q* are the N dimensional vectors (weight vectors Δ and Δ_i in our case). For pattern **recognition** the Euclidean distance is calculated and then the minimum distance is found. The input image is then said to represent the training image which gives this minimum distance.

$$\forall\, i \in I^{+} \Rightarrow d(\Delta, \Delta_i) \le \Theta_{\text{det}} \qquad \text{Pattern Detection}$$

$$\forall\, i \in I^{+} \Rightarrow \min\left\| d(\Delta, \Delta_i) \right\| \le \Theta_{rec} \qquad \text{Pattern Recognition}$$

Summarizing the ENTIRE process

$$(image - mean) = \delta_1 v_1 + \delta_2 v_2 + \cdots + \delta_k v_k \quad where\ k << n$$

where *k* is the new number of dimension and *n* the original number of dimensions, δ_i is the component of the pattern along the eigenvector (feature vector) v_i.

The above equation can also be represented pictorially as shown below:

The top four images are the feature vectors or eigenvectors (v_i in the above equation) viewed as images. At the bottom left is the mean centered face and the numbers are the values of δ_i. The "image equation" at the bottom shows how PCA works visually.

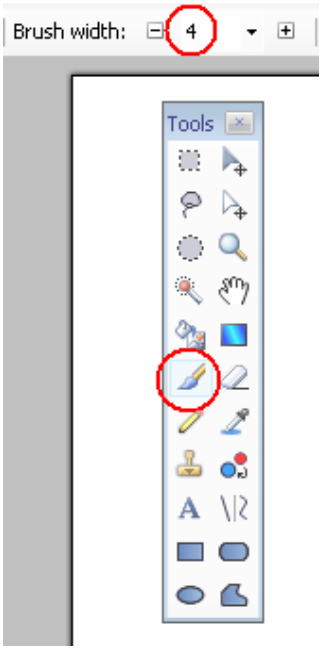
Algorithm Implementation

What good is theory without putting it to practice? I've written a MATLAB program to demonstrate PCA in identifying if a letter written by the user is an "a" or "b" or "c" or "d". Since this is just a demonstration, the program has no optimizations nor is it user friendly. You can download the training set images and the program from my site. Given below are the code and a few design notes.

Design Issues:

I've taken an easy way out by drawing the training data in the computer rather than scanning my own handwriting. By doing this, I did not have to design/apply noise removal algorithms to the set which itself can be a very interesting topic to discuss. The training data accompanying the code was drawn in MS Paint (Paint dotNET can also be used) using the brush tool, with the medium round cross section, through a touchpad. Also, the program expects the "handwritten" character to be drawn to a file named "char1.png" that resides in the same directory as the main program. Each character is expected to be 64 pixels in width and height, black in color

Tool and settings for MS Paint



Tool and settings for Paint dotNET

Code for the main module:

```
%% For compiling into an EXE, comment if running within MATLAB
%%function result = PCompAn(dummy)

%% Main module for using Principal Component Analysis (PCA) for recognizing
%% handwritten user input. Designed to identify either "a" or "b" or "c" or
%% "d".

%% Clear the workspace and screen, close all open figure windows
clc; clear all; close all;
```

```
%% Pre-process the images so that their dimensions are 64 x 64 while
%% ensuring that the images touch the border of the canvas. Each character
%% has eight training images associated with it.
dataChar = [];
sum = 0;
[dataChar sum] = preprocess('Training Images\charA\charA', 8, dataChar, sum);
[dataChar sum] = preprocess('Training Images\charB\charB', 8, dataChar, sum);
[dataChar sum] = preprocess('Training Images\charC\charC', 8, dataChar, sum);
[dataChar sum] = preprocess('Training Images\charD\charD', 8, dataChar, sum);

%% Get the average image
Iavg = sum / (8*4);

%% Train the module to interpret the input data
tr_vectors = trainMe(dataChar, Iavg);

%% Attempt at identifying the input character
result = recognize('char1', tr_vectors(:,1:32), dataChar, Iavg,...
                   tr_vectors(:,33:end));

if result(2) == 1
    disp('Recognized character is "a"')
elseif result(2) == 2
    disp('Recognized character is "b"')
elseif result(2) == 3
    disp('Recognized character is "c"')
elseif result(2) == 4
    disp('Recognized character is "d"')
end
```

Code for the preprocessing module:

```
function [S sum] = preprocess(group_name, max_img, S, sum)
%% Pre-processing Module
I = zeros(64);      %% Images are assumed to be 64 x 64 pixels in size
for i = 1:max_img
    %% Load the image
    fileName = strcat(group_name, int2str(i), '.png');

    %% Read each image
    I = double(imread(fileName));

    %% Convert to a black and white image
    I_bw = bwlabel(im2bw(I, graythresh(I)));

    %% Inverse the image to get the character in white over a black canvas
    I_bw = invbw(I_bw);

    %% Find the bounding box
    bbox = regionprops(I_bw, 'BoundingBox');

    %% Now resize the image in the bounding box to 64 x 64
    I = res_bbox(I, bbox);

    %% Get the sum of the images
    sum = sum + I(:);

    %% Append the column vectors to the matrix S
    S = [S I(:)];
end
```

Code for the black and white colour inversion module:

```
function inv = invbw(I)
%% Invert a black and white image of size 64 x 64 pixels
inv = zeros(64);
for i = 1:64
    for j = 1:64
        inv(i,j) = not(I(i,j));
    end
end
```

Code for resizing the image to the bounding box:

```
function img = res_bbox(I, bbox)
%% Resize the image such that the white character inside the image touches
%% the canvas borders

%% Get the coordinates of the bounding box from the structure
numcol = floor(bbox.BoundingBox(3));
numrow = floor(bbox.BoundingBox(4));

irow = ceil(bbox.BoundingBox(2));
icol = ceil(bbox.BoundingBox(1));

%% This will store the portion of the image to be resized, that is, the
%% part of the image contained by the bounding box
tmp = zeros(numrow, numcol);

%% Get the portion of the image to be resized
for i = irow:irow+numrow-1
    for j = icol:icol+numcol-1
        tmp(i-irow+1, j-icol+1) = I(i,j);
    end
end

%% Resize the part of the image contained in the bounding box to a 64 x 64
%% pixels image
img = imresize(tmp, [64 64],'bicubic');
```

Code for producing feature vectors weights of training vectors:

```
function tr_vectors = trainMe(dataChar, I_Avg)
%% Obtain the training vectors

%% Subtract the average image from each column of the matrix 'dataChar'
len = size(dataChar);
for i = 1:len(2)
    dataChar(:, i) = dataChar(:, i) - I_Avg;
end

%% Find the eigenvectors of the covariance matrix using the manipulation as
%% discussed in the theory. Get them in the descending order. To do this we
%% use eigs() instead of eig() and get the 28 most largest eigenvalues
%% (magnitude).
L = dataChar'*dataChar;
[v d] = eigs(L,28);
V = (dataChar*v)';
D = 64 * d;          %% Since we have 64 pixels on each side, n = 64
```

```
%% Get the weight vectors of the training sets
tr_vectors = [V*dataChar V];        %% 28 x 32 matrix
```

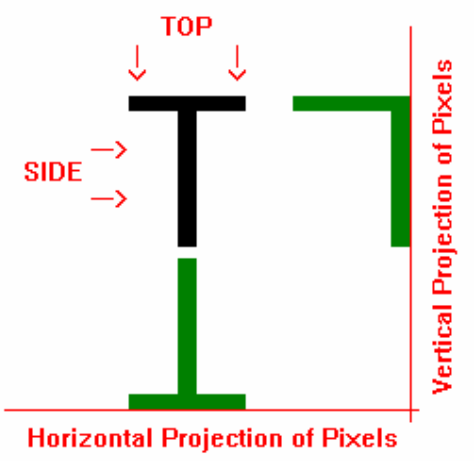
Code for recognizing characters:

```
function result = recognize(inpImage, wt_vectors, dataChar, I_Avg, eigenV)
%% Preprocess the input image, there's only one image to preprocess,
%% unlike a set of training images
%% Load the image
fileName = strcat(inpImage, '.png');

%% Read each image
img = double(imread(fileName));

%% Convert to a black and white image
I_bw = bwlabel(im2bw(img, graythresh(img)));

%% Inverse the image to get the character in white over a black canvas
I_bw = invbw(I_bw);

%% Find the bounding box
bbox = regionprops(I_bw, 'BoundingBox');

%% Now resize the image in the bounding box to 64 x 64
img = res_bbox(img, bbox);

%% Convert the input image into a vector and normalize it about the mean
%% which is equivalent to removing redundant data common to all images
img = img(:) - I_Avg;       %% 'img' is a 4096 x 1 vector

%% Get the projections of the input pattern on the training vectors
W = eigenV * img;           %% W is a 28 x 1 vector

%% Find the 'distance' between the 'input pattern weights' and the
%% 'training weights'
dist = [];
for i = 1:32
    dist = [dist pdist([W'; wt_vectors(:,i)'])];
end

[mDist ID] = min(dist);
ID = floor(ID/8)+1;
result = [mDist ID];
```

Miscellaneous Notes:

The above code isn't a very robust implementation of the algorithm, particularly because it employs minimal error-detection features and has training data only for my handwriting. To make it work better, supplement or replace the training set with your own. Also, noise reducing algorithms need to be added.

A few optimizations can be employed to reduce the chances of erroneous detections. For example, one could check the average number of pixels in each character or horizontal and vertical projections of characters or the approximate number of pixels in each bin below the character as shown below.



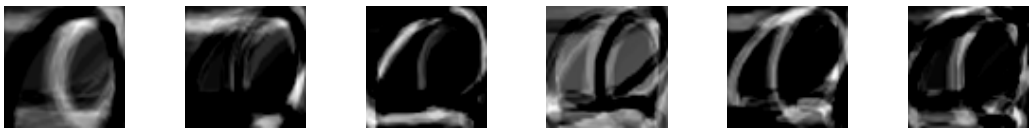
The pixel projections of the letter 'T'

These "projections" are nothing but a representation of the total number of pixels when the pattern is viewed from the "TOP" and the "SIDE" labeled in the figure above. These projections are more or less characteristic of a pattern, hence can be used to double check the result from the PCA algorithm.

A better alternative to Euclidean distance would be the Mahalanobis distance since it is scale invariant.

In the implementation, the number of eigenvectors taken up as feature vectors is equal to '*m*' or the number of training images (32 in this case), minus the number of 'unique' members in the training set which is 4 in the current case. Thus we end up with 28 feature vectors – a drastic reduction from 65536 eigenvectors!

Below are the 6 most prominent eigenvectors (feature vectors) represented as images:



And the reconstruction of the letter 'a' using the 28 feature vectors (minus the mean):

Handwritten characters are recognized according to the shape of the pattern. Hence the grayscale information was removed from the image. That's why turning the image into black and white from grayscale will not change the output of the program.

In case the program crashes, especially in the function used to resize the image, make sure that the image is not empty.

Appendix A

An N-dimensional matrix:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nn} \end{bmatrix}_{n\times n}$$

An N-dimensional vector:

$$\begin{bmatrix} a_1 & a_2 & \cdots & \cdots & a_n \end{bmatrix}_{1\times n}$$

Dimensionality can roughly be thought of as degrees of freedom. In statistics, it is the number of variables which are simultaneously affected by an observation. For example, when the government takes the census of the country every ten years, the data is 1 dimensional since the only variable here is population. A digital image can be thought of as a 2 dimensional data – since every point in the image matrix can be represented as $f(x, y)$ (to be read as a function of variables 'x' and 'y').

Of course, the dimensions may or may not be logically related. This means that a vector whose first element gives the height of an individual, the second one the time he spends surfing the internet and the third one being the age of the individual, is a valid 3 dimensional vector. So basically, dimensions (for our purpose) are nothing but the elements of a vector. Thus for image data, its dimensions are its pixels.

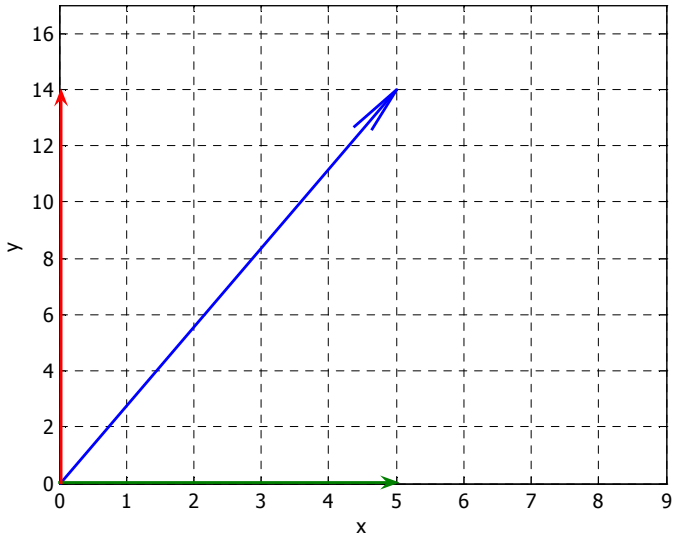
2D and 3D data can easily be visualized, but when we move into higher dimensions, this becomes a difficult task. Try imagining a 5 dimensional space and a point or line in such a space! Therefore, most examples will be visualized only for those lower dimensions. Keep in mind, even though we can't "see" multi-dimensional data, they are there simply because mathematics does not put any restrictions on their existence (pretty abstract, huh?!).

Projection of vectors:

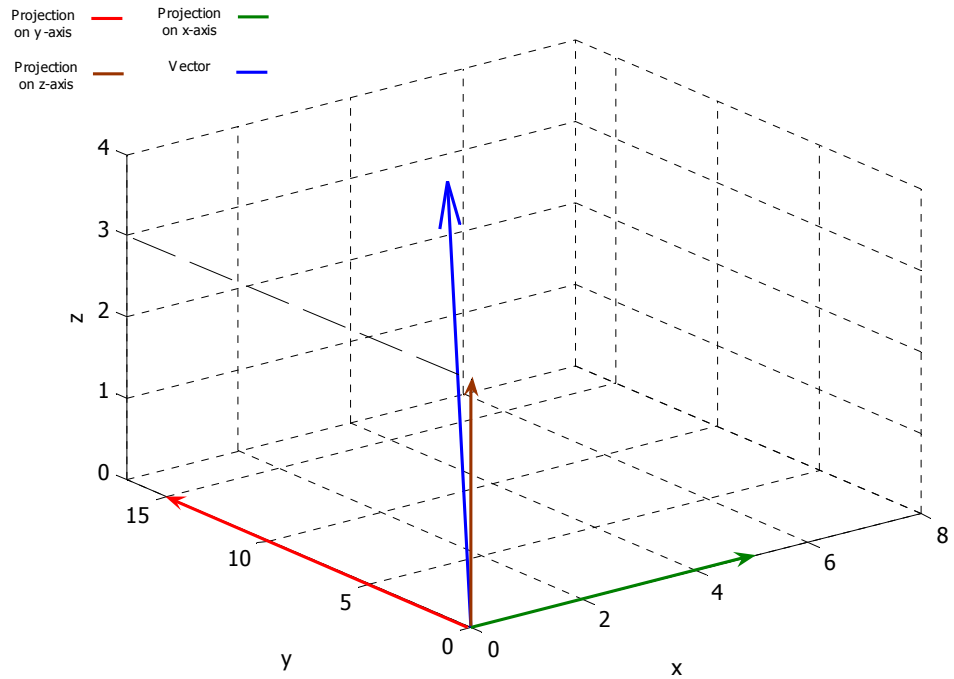
A vector, as we all know, consists of a magnitude and a direction. The magnitude gives the "strength" of the vector and the direction gives the orientation w.r.t. an origin. For 2nd and 3rd dimensions, vectors can be represented using a combination of angles and magnitudes.

The vector below can be represented as:

$$\sqrt{5^2+14^2}\angle\tan^{-1}\left(\frac{14}{5}\right)=14.86\angle 70.34^\circ$$



A 3 dimensional vector, like the one shown below, can also be represented by its magnitude and direction. The direction will be in terms of angles made by it with the X, Y and Z axes. But what happens when we want to represent a higher dimension vector? Visualizing the "angle" it makes with the axes is difficult! That's where projection of vectors comes in.



Projecting a vector on another vector can be thought of as obtaining the "shadow" of the given vector in the direction of the desired vector. With reference to the 2D vector two images above, the green line on the X-axis is the projection of the vector on this axis. Similarly, the red line represents the projection on the Y-axis.

Mathematically, it is the component of the given vector that lies in the direction of the desired vector and is expressed as a dot product of the two vectors. The following expression gives the magnitude of the projection of vector $\vec{a}$ on vector $\vec{b}$ and vice-versa:

$$\vec{a} \cdot \vec{b} = \left|\vec{a}\right|\left|\vec{b}\right| \cos\theta$$

In the figure above, a 3D vector (shown in blue) is shown along with its projections on the X, Y and Z axes. The components can be identified with the key that has been included in the figure (valid for both 2D & 3D plots). The X-axis component measures 5 units, Y-axis component, 15 units and the Z-axis component measures 3 units. To represent this vector in terms of its components, we need to know another term called basis.

Basis:

Simply put, the basis is a set of vectors which can linearly represent **any** vector whose dimension is same as theirs but they themselves cannot be represented by the other vectors from that set. Generally, these vectors are scaled to have unit length. In linear algebra and statistics, a basis is chosen so that its vectors are mutually perpendicular. Such a basis is known as an orthogonal basis. An example of this type of basis is the X, Y, Z axes of the Euclidean 3 dimensional space for which the basis can be written in the matrix form as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

It can also be written as individual vectors:

$$\begin{aligned} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} &= \hat{i} && \text{X-axis} \\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} &= \hat{j} && \text{Y-axis} \\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} &= \hat{k} && \text{Z-axis} \end{aligned}$$

To denote a vector in the Euclidean 3 dimensional space, we simply need to multiply the projections of the vector on each axis with the corresponding basis vector for that axis, i.e. for the previous 3D example, the final vector is given by the following expression:

$$\vec{v} = 5\hat{i} + 15\hat{j} + 3\hat{k}$$

From the above, we also conclude that “basis” and “axes” can be used interchangeably.

So, in order to represent higher dimension vectors (say, 5), one of the possible basis would be:

$$\begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix} = \hat{i} \quad \text{X-axis}$$
$$\begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix} = \hat{j} \quad \text{Y-axis}$$
$$\begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix} = \hat{k} \quad \text{Z-axis}$$
$$\begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix} = \hat{l} \quad \text{A-axis}$$
$$\begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix} = \hat{m} \quad \text{B-axis}$$

And the new vector equation is:

$$\vec{v} = 5\hat{i} + 15\hat{j} + 3\hat{k} + 6\hat{l} + 1\hat{m}$$

Note that the above equation represents the position vector of a point in 5 dimensional space. This concept can be extended to any dimension. In PCA, every training set member is represented as a point in a multi-dimensional space. Also note that bases **need not** form a unitary matrix when they are shown in the matrix form. For example, taking the 2 dimensional case, standard basis is:

$$\begin{bmatrix}1 & 0\end{bmatrix} = \hat{i} \quad \text{X-axis}$$
$$\begin{bmatrix}0 & 1\end{bmatrix} = \hat{j} \quad \text{Y-axis}$$

But the following is an equally valid basis for 2 dimensions (also represented as $\Re^2$):

$$\begin{bmatrix}1 & 1\end{bmatrix} = \hat{i} \quad \text{X`-axis}$$
$$\begin{bmatrix}1 & -1\end{bmatrix} = \hat{j} \quad \text{Y`-axis}$$

Ramblings:

The image vectors dealt with in PCA are generally grayscale images. If each image is 256 x 256 pixels in size, then the resultant image vector will have $256^2 = 65536$ dimensions. Here, each dimension is a pixel and its grayscale value is the value along the dimension. All images with a total of 65536 pixels are nothing but points in the 65536 dimension space. Finding the "distance" between these points is equivalent to finding the difference of their position vectors.

By the way, the total number of points in 65536 dimension space, taking the case of grayscale images, is 256^{65536} = <gives infinity when calculated on my computer> !!

Appendix B

1. **(Linear independence of eigenvectors)**
   *Let $\lambda_1, \lambda_2, \lambda_3 \cdots, \lambda_k$ be **distinct** eigenvalues of a $n \times n$ matrix. Then the corresponding eigenvectors $x_1, x_2, x_3 \cdots, x_k$ form a linearly independent set.*

   The following proof is from Kreyszig Erwin. Advanced Engineering Mathematics. John Wiley & Sons, Eighth Edition.

   Suppose that the conclusion is false. Let *r* be the largest integer such that $\{x_1, \cdots, x_r\}$ is a linearly independent set. Then *r* < *k* and the set $\{x_1, \cdots, x_{r+1}\}$ is linearly dependent. Thus there are scalars $\{c_1, \cdots, c_{r+1}\}$, not all zero, such that:

   $$c_1 x_1 + \cdots + c_{r+1} x_{r+1} = 0 \qquad (1)$$

   Multiplying both sides by **A** and using $Ax_j = \lambda_j x_j$, we obtain

   $$c_1 \lambda_1 x_1 + \cdots + c_{r+1} \lambda_{r+1} x_{r+1} = 0 \qquad (2)$$

   To get rid of the last term, we subtract λ_{r+1} times (1) from this, obtaining

   $$c_1(\lambda_1 - \lambda_{r+1})x_1 + \cdots + c_r(\lambda_r - \lambda_{r+1})x_r = 0$$

   Here $c_1(\lambda_1 - \lambda_{r+1}) = 0, \cdots, c_r(\lambda_r - \lambda_{r+1}) = 0$ since $\{x_1, \cdots, x_r\}$ is linearly independent.
   Hence $c_1 = \cdots = c_r = 0$, since all the eigenvalues are distinct. But with this, (1) reduces to $c_{r+1} x_{r+1} = 0$, hence $c_{r+1} = 0$, since $x_{r+1} \neq 0$ (an eigenvector!). This contradicts the fact that not all scalars in (1) are zero. Hence the conclusion of the theorem must hold.
   This theorem immediately implies the following.

2. **(Basis of eigenvectors)**
   *If a $n \times n$ matrix **A** has n **distinct** eigenvalues, then **A** has a basis of eigenvectors for C^n (or R^n)*

3. **(Orthogonal eigenvectors)**
   *The eigenvectors of a **symmetric** matrix with **distinct** eigenvalues are orthogonal.*

   Let any two eigenvectors of a symmetric matrix **A** be x_1 and x_2 with eigenvalues λ_1 and λ_2 respectively. From the eigenvalue equation:

   $$Ax_1 = \lambda_1 x_1 \text{ and } Ax_2 = \lambda_2 x_2$$

   Also, $A^T = A$ since the matrix is assumed to be symmetric

Using the above equations, we simplify $\lambda_1(x_1 \cdot x_2)$ as follows (that's a dot product **not** a multiplication!):

$$\begin{aligned}
&\lambda_1(x_1 \cdot x_2) \\
&\Rightarrow (\lambda_1 x_1) \cdot x_2 = (Ax_1) \cdot x_2 \\
&\Rightarrow (Ax_1)^T x_2 = {x_1}^T A^T x_2 = {x_1}^T A x_2 \\
&\Rightarrow {x_1}^T \lambda_2 x_2 = \lambda_2 {x_1}^T x_2 \\
&\Rightarrow \lambda_2(x_1 \cdot x_2)
\end{aligned}$$

The above equality can also be re-written as:

$(\lambda_1 - \lambda_2)(x_1 \cdot x_2) = 0$
$\because \lambda_1$ and λ_2 are distinct eigenvalues $\therefore (\lambda_1 - \lambda_2) \neq 0$
$\Rightarrow (x_1 \cdot x_2) = 0$

Obviously, the last line (dot product of the two eigenvectors being zero) implies that they are orthogonal to each other. Whenever finding out eigenvectors, make sure that their magnitude is scaled to unity, it is a widely used notation.

References:

1. http://www.pages.drexel.edu/~sis26/Eigenface%20Tutorial.htm
2. http://en.wikipedia.org/wiki/Eigenface
3. www.snl.salk.edu/~shlens/pca.pdf
4. http://cnx.org/content/m12531/latest/
5. http://blog.zabarauskas.com/eigenfaces-tutorial/
6. Kreyszig Erwin. Advanced Engineering Mathematics. John Wiley & Sons, Eighth Edition.