

Image Recognition Using the Eigenpicture Technique

(with Specific Applications in Face Recognition and Optical Character
Recognition)

Neil Muller

University of Cape Town

Department of Mathematics and Applied Mathematics

Submitted for M. Sc. (Applied Mathematics)

Submitted: October 1997

The University of Cape Town has been given
the right to reproduce this thesis in whole
or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Image Recognition Using the Eigenpicture Technique

(with Specific Applications in Face Recognition and Optical Character
Recognition)

Neil Muller

University of Cape Town

Department of Mathematics and Applied Mathematics

Submitted for M. Sc. (Applied Mathematics)

Submitted: October 1997

The University of Cape Town has been given
the right to reproduce this thesis in whole
or in part. Copyright is held by the author.

Contents

1	Introduction	1
1.1	Background	1
1.2	Outline	3
2	Face Recognition	5
2.1	The Basic Principle of Eigenfaces.	5
2.2	Basic Concepts	6
2.2.1	Notation	6
2.2.2	Example Images	7
2.3	Calculating the Eigenfaces.	8
2.3.1	Basic Derivation	8
2.3.2	Reducing the Computational Cost	11
2.3.3	Some Conclusions from the Eigenface Calculation	13
2.3.4	An Algorithm for Calculating the Eigenfaces	15
2.3.5	Images of the Eigenfaces	15
2.4	Representing Faces using Eigenfaces	16
2.4.1	Basic Reconstruction	16
2.4.2	Results	18
2.4.3	Reconstruction Error	22
2.5	Recognizing Faces using Eigenfaces	26
2.5.1	Description	26

2.5.2	An Algorithm for Recognition	27
2.5.3	Results	27
2.6	Image Segmentation using Eigenfaces	29
2.6.1	Problem Background	29
2.6.2	A Maximum Likelihood Statistic for Face Location . .	31
2.6.3	An Algorithm for Locating the Face	36
2.6.4	Results	37
2.7	Updating the Eigenfaces	38
2.7.1	Problem Description	38
2.7.2	Perturbation Analysis	40
2.7.3	Reduction	40
2.7.4	Finding φ	42
2.7.5	Efficient Recalculation of the Eigenfaces	47
2.7.6	An Algorithm for Calculating the New Eigenfaces . .	48
2.7.7	Results using the New Eigenfaces	49
2.7.8	Converting the Database	51
2.7.9	Results of Converting the Database	52
2.8	Testing the System	54
2.8.1	Test Description and Results	54
2.8.2	Comments	56
3	Optical Character Recognition	59
3.1	Introduction	59
3.2	Subdividing the Training Set	60
3.2.1	Problem Description	60
3.2.2	Classification Background	61
3.2.3	Leader Principle with K-means Refinement	61
3.2.4	Pairwise Nearest Neighbor	66

3.2.5	Eigenpicture Refinement	68
3.3	Basic Classification Implementation	68
3.3.1	Algorithms for Classification	68
3.3.1.1	Leader Principle (Heuristic classification)	68
3.3.1.2	K-means Refinement	69
3.3.1.3	PNN Classification	70
3.3.1.4	Eigenpicture Refinement	72
3.3.2	Classification Results	72
3.3.2.1	Heuristic and K-means Classification	73
3.3.2.2	PNN Classification	75
3.3.2.3	Comparative Reconstruction Performance	76
3.4	A Simple OCR System	78
3.4.1	Implementation Details	78
3.4.2	Algorithm for OCR System	80
3.4.3	Results	82
3.4.3.1	Recognising a Font in the Training Set.	83
3.4.3.2	Recognising a Font not in the Training Set.	85
3.4.3.3	Comments	87
3.4.3.4	K-means with Improved Segmentation Algorithm.	88
3.5	Improving the Recognition Rule	91
3.5.1	Description	91
3.5.2	Modified Algorithm	93
3.5.3	Results	94
3.6	Recognising Imperfectly Scanned Images	96
3.7	Using the Update Method	99
3.7.1	Problem Description	99
3.7.2	Algorithm	99

3.7.3	Results	100
3.7.3.1	Reconstruction Performance	100
3.7.3.2	Recognition Performance	100
4	Conclusions	107
A	Code for Algorithms	110
A.1	Face Recognition	110
A.1.1	Calculating Eigenfaces	110
A.1.2	Representing and Reconstructing Images	111
A.1.3	Recognising an Image	112
A.1.4	Locating the Face	113
A.1.5	Updating the Eigenfaces	114
A.2	OCR	115
A.2.1	Classification	115
A.2.1.1	Leader Principle	115
A.2.1.2	K-means Refinement	117
A.2.1.3	PNN Based Classifier	119
A.2.1.4	Eigenpicture Refinement	126
A.2.2	OCR Systems	129
A.2.2.1	Improved Recognition Rule	129
A.2.2.2	Updating the Eigenletters	138
B	OCR Training Sets	143
B.1	The Original 10 Fonts in the Training Set	143
B.2	The 3 New Fonts	144
C	Full Listing of All the Classes	145
D	Bibliography	149

List of Figures

2.1	Two images from the training set	7
2.2	Average Face	7
2.3	“Caricatures” of two images in the training set	8
2.4	1st, 3rd, 20th and 40th eigenfaces	15
2.5	Eigenvalues of the covariance matrix.	17
2.6	Image in training set	19
2.7	Reconstruction of figure 2.6 using both 10 and 40 values . . .	19
2.8	Reconstruction error from using 40 value in fig. 2.7.	20
2.9	Graph of the coefficients used in fig. 2.7.	20
2.10	Image of rose.	21
2.11	Reconstruction of fig. 2.10.	21
2.12	Image not in training set	21
2.13	Reconstruction of fig. 2.12 using 10 and 40 values.	22
2.14	Error from using 40 value reconstruction in fig. 2.13	22
2.15	Graph of the elements of fig. 2.13’s reconstruction	23
2.16	Individual not represented in the training set.	23
2.17	Reconstruction of fig. 2.16	23
2.18	Bar graph of reconstruction errors.	25
2.19	Image outside of database	28
2.20	Best match from database	28

2.21	Graph comparing representations of image (2.19) and best match (2.20)	29
2.22	Graph of representation of 2.19 against that of best match . .	29
2.23	Non-face perfectly described by the eigenfaces	30
2.24	Image before location algorithm	37
2.25	Image after location	37
2.26	Outline comparison between fig. 2.24 and fig. 2.25	38
2.27	Original image, poorly described by training set	39
2.28	Reconstruction of image poorly represented by training set .	39
2.29	Error between fig. 2.27 and 2.28	39
2.30	Representation of fig. 2.27 using fully recalculated eigenfaces.	49
2.31	Error of fig. 2.30	50
2.32	Reconstruction of fig. 2.27, using updated eigenfaces.	50
2.33	Error from fig. 2.32	50
2.34	Graph of the angles between the updated and recalculated eigenfaces	51
2.35	Reconstruction of figure 2.6 using correct representation . . .	53
2.36	Reconstruction of figure 2.6 using converted representation .	53
2.37	Error between figure 2.35 and figure 2.36	53
2.38	Difference between representations used in figures 2.35 and 2.36	54
2.39	Images from the manchester database	55
2.40	Error margins for a correct match	56
2.41	Error margins for incorrect match	57
3.1	Diagram of a kd-tree	67
3.2	Table of class averages	73
3.3	Elements of class 8.	73
3.4	Elements of class 27.	74
3.5	Elements of class 35.	74

3.6	Elements of k-means class 1	75
3.7	Elements of PNN class 35	76
3.8	Reconstruction results using the different methods	77
3.9	Filters for classes 27 and 35.	79
3.10	Before and after filtering	79
3.11	Radon transform of 3.12 at 90°	80
3.12	Font in training set	83
3.13	Results using eigenletters from full alphabet on fig. 3.12	84
3.14	Results using eigenletters from "natural" classification on fig. 3.12	84
3.15	Results using eigenletters from k-means classification on fig. 3.12	85
3.16	Results using eigenletters from PNN classification on fig. 3.12	85
3.17	Font not in training set.	86
3.18	Results using eigenletters from full alphabet on figure 3.17.	86
3.19	Results using eigenletters from "natural" classification on fig. 3.17.	87
3.20	Results using eigenletters from k-means classification on fig. 3.17.	87
3.21	Results using eigenletters from PNN classification on fig. 3.17	87
3.22	Results using k-means eigenletters with improved segmenta- tion algorithm on fig. 3.12.	89
3.23	Results using k-means eigenletters with improved segmenta- tion algorithm on fig. 3.17.	89
3.24	Recognition performance for fonts in the training set.	90
3.25	Recognition performance for fonts not in the training set.	91
3.26	Results of using modified recognition rule on fig. 3.12.	94
3.27	Results of using modified recognition rule on fig. 3.17.	94
3.28	Recognition performance for fonts in the training set	95
3.29	performance for fonts not in training set.	96

3.30	Original image which will be scanned	96
3.31	Figure 3.30 scanned at 100 × 100 dpi.	97
3.32	Results of recognizing fig. 3.31.	97
3.33	Figure 3.30 scanned at 200 × 200 dpi.	98
3.34	Results of using OCR system on fig. 3.33	98
3.35	Reconstruction using full set of eigenletters and the updated eigenletters.	103
3.36	Alphabet in original training set.	104
3.37	Alphabet newly included into training set.	104
3.38	Results of using full set of eigenpictures on fig. 3.36.	105
3.39	Results of using updated eigenpictures on fig. 3.36.	105
3.40	Results of using full set of eigenpictures on fig. 3.37	105
3.41	Results on using updated eigenpictures on fig. 3.37	105
3.42	Comparative performance of updated eigenletters against full recalculation.	106
C.1	Letter g against letter S	148

List of Tables

3.1	Results of using four different classification methods on fig. 3.12.	84
3.2	Results of using 4 classifications methods on fig. 3.17.	86
3.3	Results of using improved segmentation algorithm.	88
3.4	Results of using the modified recognition rule.	94
3.5	Results using actual scanned data.	97
3.6	Results using updated and fully recalculated eigenletters. . .	101

Chapter 1

Introduction

1.1 Background

Almost all personal identification systems currently in use have serious shortcomings and should soon become obsolete. The main problem is that, since these systems are based on abstract 'patterns' such as personal identification numbers (PIN) or even handwritten signatures on credit cards and checks, they verify the accuracy of the pattern and not the individual presenting the 'pattern'. For example, the forger only has to imitate the image of the signature without needing any of the characteristics that uniquely define the individual. To strengthen the point that a handwritten signature does not contain any personal characteristics we remind the reader that many companies use stamped signatures. The ease with which these systems can be beaten naturally lead to widespread fraud. Although banks are understandably reluctant to release any details, the amounts are staggering.

It is no surprise that alternative identification systems are being investigated. Some of the more promising systems are collectively known as Biometric Personal Identification Systems. These systems are all somehow based on unique biometric features of each individual. Thus, they no longer rely, at least not exclusively, on an abstract pattern but are based on the physical/biometric characteristics of each individual. Many of the ideas are familiar—fingerprints have been used for a long time by law enforcement agencies. The novel aspect is the availability of technology, both hardware and software, that allow the development of automated systems.

Currently many systems are under development and some of these have even found their way into commercial applications. There is no doubt that identification systems of the future will rely heavily on their biometric components, which include iris and retina patterns, vein patterns on the hand,

speech, finger geometry and even body odour, a system aptly known as The Bloodhound!

The biometric system we investigate in this dissertation uses facial images for identification purposes. As will become clear it shares the difficulties of general pattern recognition and biometric identification systems, and then it introduces some particular difficulties of its own, not the least one of size.

In general two different types of problems occur: Firstly there is the *recognition* problem where a given image is matched against a data base and the best match from the database has to be found. A successful match is recorded only if the best match is 'close enough'. Secondly there is the *verification* problem where a given image is compared with a specific image in the database and the goal is to decide whether the given image matches the one in the database. In both cases one ultimately has to compare two different images and decide whether they are of the same object or not.

The main problem in many biometric identification systems is that the object itself is a biological entity and therefore subject to variation. For example, if we are thinking of facial images, then the image depends on the facial expression at the time of recording, aging over time, or other arbitrary changes in appearance including, changing hairstyles, addition/removal of glasses, etc. Obviously, if anyone wishes to disguise him/herself, the system is bound to fail. Thus the major difficulty with most biometric systems is to distinguish between natural variations between images of the same object and the (hopefully) different type of variation indicating an image of a different object. Unfortunately, in general it is impossible to completely separate the images belonging to a single object from images belonging to another object. Thus, it becomes impossible to design a totally reliable system. A major goal of any biometric system is to find the best separation between the images from different objects. Often this entails a transformation to different coordinates. For example we shall describe our facial images in terms of the so-called eigenfaces which become the basis of comparison.

Another major problem shared by most systems is one of measurement—perfect data to work with happens only too rarely. In practice many systems spend a considerable amount of time massaging the data. In the case of facial images one finds that the lighting varies between different images of the same object, especially if the images are obtained from locations separated in time and space, using different equipment. Since we actually deal only with lighting variations, as reflected from different parts of the face, lighting normalization is crucial to a successful system. In addition, the images may appear on different scales (some may be smaller or larger than others) at different locations in the image, scale- and location normalization is also essential.

The size of facial images provide a major computational challenge. Even images at the modest resolution of 100×100 pixels contain a large amount of data. At this resolution one has to calculate the eigenvalues and eigenvectors of a $10\,000 \times 10\,000$ matrix which is large by most standards. Thus a major effort goes into designing computationally efficient algorithms.

1.2 Outline

In the first part of this dissertation, we present a detailed description of the eigenface technique first proposed by Sirovich and Kirby [1] and subsequently developed by several groups, most notably the Media Lab at MIT (see [3-10]). Other significant contributions have been made by Rockefeller University [11], whose ideas have culminated in a commercial system known as FaceIt. For a different techniques (i.e. not eigenfaces) and a detailed comparison of some other techniques, the reader is referred to [5]. Although we followed ideas in the open literature (we believe there that there is a large body of advanced proprietary knowledge, which remains inaccessible), the implementation is our own. In addition, we believe that the method for updating the eigenfaces to deal with badly represented images presented in section 2.7 is our own.

The next stage in this section would be to develop an experimental system that can be extensively tested. At this point however, another, nonscientific difficulty arises, that of developing an adequately large data base. The basic problem is that one needs a training set representative of all faces to be encountered in future. Note that this does not mean that one can only deal with faces in the database, the whole idea is to be able to work with any facial image. However, a data base is only representative if it contains images similar to anything that can be encountered in future. For this reason a representative database may be very large and is not easy to build. In addition for testing purposes one needs multiple images of a large number of people, acquired over a period of time under different physical conditions representing the typical variations encountered in practice. Obviously this is a very slow process.

Potentially the variation between the faces in the database can be large suggesting that the representation of all these different images in terms of eigenfaces may not be particularly efficient. One idea is to separate all the facial images into different, more or less homogeneous classes. Again this can only be done with access to a sufficiently large database, probably consisting of several thousand faces.

While continuing with the time consuming process of developing a suffi-

ciently large database of facial images, we decided to test the technique in a situation where data is more freely available. It is against this background that we decided to look into the possibility of developing an Optical Character Recognition (OCR) system based on an 'eigenpicture' technique. Not only is it easier to find different representations of the same character, as expressed through different fonts, but it also provides an ideal testing ground for our idea of separating the dataset into different classes. It is this system which forms the second part of this dissertation. We believe that this application of the eigenpicture technique to the OCR problem is original.

Returning to the classification idea, we note that there is a 'natural' classification of the letters, namely alphabetical order (see section 3.2.1). This classification is of course based on the phonetic qualities of the different letters, rather than their appearance, i.e. the alphabet is arranged on how the letters sound, rather than on how they look. Thus one finds remarkable differences between the same letter in different fonts. For the eigenletter technique, we classify the letters according to their appearance, rather than their sound. The efficiency of the resulting system is compared with the 'natural', phonetic, classification as well as the case where no classification whatsoever, is used. These systems are then compared with commercial OCR software, with illuminating results.

As expected, the OCR implementation of the eigenpicture technique introduces problems peculiar to itself. An OCR system is a three-stage process. During the first stage a potential letter/character is located in the image. The radon transform combined with a location algorithm similar to one used for faces proves to be fairly efficient (not much effort was spent to optimize the procedure). Once a character has been located, the second stage consists of identifying the character. This stage has seen the most extensive development and undergone the most thorough testing, and several improvements are discussed. The performance of the end result is comparable or only slightly worse than that of commercial systems on the same test sets. During the third stage the character is mapped back to paper in such a way that the overall appearance matches that of the original document. Although an essential step in any commercial application, it holds little interest for us mainly because we do not believe that we are able to improve on existing commercial systems.

The ideas mentioned above were all implemented in R-lab version 2.0.6 (a Matlab-like programming environment) on a 100 MHz Pentium running Linux 2.0.30. The actual programs for both sections of the dissertation are included in Appendix A.

Chapter 2

Face Recognition

For the sake of simplicity, we will consider the eigenpicture technique under the assumption that we are dealing with facial images only. In this context, the technique is usually referred to as the eigenface technique.

2.1 The Basic Principle of Eigenfaces.

For this study, we choose to use the simple case of 8-bit greyscale images with a resolution of 112×112 pixels, mainly because the majority of the images in our original data set were taken at that resolution. However, the discussion which follows is independent of the resolution.

Each image is thus described by 12 544 elements, where each element's value is an integer in the range 0(black) to 255(pure white). So we can view each image as a vector with 12 544 elements. Alternatively, we can say that the facial images are represented by vectors in a 12 544 dimensional linear vector space. However, note that this vector space can describe every possible greyscale image with the same resolution.

Since facial images are similar to each other, these images are actually represented by a much lower dimensional subspace of this high dimensional vector space. The idea of eigenfaces is to describe this subspace, using the following idea:

From an initial set of facial images, find a low-dimensional representation of the faces which preserves the significant information.

Clearly, if such a representation can be found, we only need to describe the

images in terms of this representation, which should significantly simplify the recognition problem.

2.2 Basic Concepts

2.2.1 Notation

In the rest of this chapter, we will be using the following definitions:

Training set	Our initial (representative) set of faces.
M	The number of images in training set. (In our case, about 100)
N	The number of elements in each image. (In our case 12 544)
\mathbf{I}_n	The n 'th face in training set. We define \mathbf{I}_n to be a vector. The usual matrix representation of an image is transformed into a vector by "stacking" the columns of the matrix. Borrowing a notational convention from Statistics, if we define B_n as the image's matrix representation, then $\mathbf{I}_n = \text{vec } B_n$.
\mathbf{J}	A facial image not in the training set.
\mathbf{A}	The average face. By definition

$$\mathbf{A} = \frac{1}{M} \sum_{n=1}^M \mathbf{I}_n. \quad (2.1)$$

\mathbf{X}_n	The normalised deviation of the n 'th face from the average. By definition $\mathbf{X}_n = \frac{\mathbf{I}_n - \mathbf{A}}{\ \mathbf{I}_n - \mathbf{A}\ }$.
\mathbf{u}_l	The l 'th eigenface.
Ω	The N dimensional vector space representing all the faces. Recall that $N = 12\,544$ in our case.

We shall introduce more notational conventions as they become necessary.

Figure 2.1: Two images from the training set



2.2.2 Example Images

Figure 2.1 shows two images from the training set used in this dissertation.

Note that the position of the faces in all the images is standardised, so as to reduce the amount of unnecessary variation in the database. This can be done by hand for small applications, but for any serious application, some automatic method is required. This issue will be discussed in section 2.6.

Figure 2.2: Average Face



Figure 2.2 shows our average face, calculated from (2.1). We subtract it from all our images to remove the most common elements. This produces the so-called “caricatures”, which are defined as

$$C_n = I_n - A.$$

Figure 2.3 shows the caricatures of the two images shown in figure 2.1. Since the caricatures contain both positive and negative values, we translate the images so that zero maps onto the middle of the color range. Thus grey represents small values, while large positive values are white and large negative values are black. We then scale these caricatures so that they have a L_2 -norm of 1 to produce the X_n 's,

Figure 2.3: “Caricatures” of two images in the training set



$$\mathbf{X}_n = \frac{\mathbf{C}_n}{\|\mathbf{C}_n\|}.$$

2.3 Calculating the Eigenfaces.

2.3.1 Basic Derivation

Our goal is to obtain an efficient representation of the images. To do this, we need to find those features common to all the faces. These features will become the basic building blocks with which we can represent any individual face, rather like a mathematical identikit.

The technique we use here is variously known as the Hotelling transform (image processing), the Karhunen-Loeve transform, or as Principal Component Analysis or PCA (Statistics). The idea is to find a set of orthonormal basis vectors for a low dimensional sub-space which will adequately describe the facial images. These basis vectors are the eigenfaces.

To obtain the eigenfaces, we require that they satisfy the following standard conditions (see Jolliffe [13]) :

- $\lambda_1 = \max_{\|\mathbf{u}_1\|=1} \frac{1}{M} \sum_{n=1}^M (\mathbf{u}_1^T \mathbf{X}_n)^2$ for all $\mathbf{u}_1 \in \Omega$.
- $\lambda_l = \max_{\|\mathbf{u}_l\|=1} \frac{1}{M} \sum_{n=1}^M (\mathbf{u}_l^T \mathbf{X}_n)^2$ for all $\mathbf{u}_l \in \Omega$, subject to $\mathbf{u}_l^T \mathbf{u}_p = 0$ for all $p < l$.

These conditions imply that each successive eigenface points in the “direction” of the maximum variation in the images, subject to the constraints of

orthonormality. Since we are maximizing subject to constraints, the problem is conveniently solved by the use of Lagrange multipliers. (The following proof is based on one presented by Jolliffe [13], but we have reworked it somewhat to highlight aspects of particular interest to the rest of this dissertation.)

We view each vector \mathbf{X}_k as a multivariate random variable

$$\mathbf{X}_k = \begin{bmatrix} X_{1k} \\ \vdots \\ X_{Nk} \end{bmatrix},$$

so we define the covariance matrix of the X_{jk} 's as

$$C = \frac{1}{M} \sum_{k=1}^M \mathbf{X}_k \mathbf{X}_k^T \quad (2.2)$$

where C is clearly a symmetric matrix, and since

$$\begin{aligned} \mathbf{a}^T C \mathbf{a} &= \frac{1}{M} \mathbf{a}^T \left(\sum_{k=1}^M \mathbf{X}_k \mathbf{X}_k^T \right) \mathbf{a} \\ &= \frac{1}{M} \sum_{k=1}^M \left(\mathbf{a}^T \mathbf{X}_k \right) \left(\mathbf{X}_k^T \mathbf{a} \right) \\ &= \frac{1}{M} \sum_{k=1}^M \left(\mathbf{a}^T \mathbf{X}_k \right)^2 \\ &\geq 0 \end{aligned}$$

it is also positive semi-definite.

So our conditions are:

- $\lambda_1 = \max_{\|\mathbf{u}_1\|=1} \mathbf{u}_1^T C \mathbf{u}_1$, for all $\mathbf{u}_1 \in \Omega$.
- $\lambda_l = \max_{\|\mathbf{u}_l\|=1} \mathbf{u}_l^T C \mathbf{u}_l$, for all $\mathbf{u}_l \in \Omega$ and subject to $\mathbf{u}_l^T \mathbf{u}_p = 0$ for all $p < l$.

From the theory of Lagrange multipliers, it follows that a function $f(\mathbf{x})$ subject to a constraint $g(\mathbf{x}) = k$, is maximised provided that

$$\nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}).$$

In the case of \mathbf{u}_1 , we choose $f(\mathbf{u}_1) = \mathbf{u}_1^T C \mathbf{u}_1$ and $g(\mathbf{u}_1) = \mathbf{u}_1^T \mathbf{u}_1 = 1$.

Since $\nabla f = 2C\mathbf{u}_1$ (since C is symmetric) and $\nabla g = 2\lambda\mathbf{u}_1$ it follows immediately that

$$C\mathbf{u}_1 = \lambda_1\mathbf{u}_1.$$

Thus \mathbf{u}_1 is an eigenvector of C with length 1 and our conditions imply that \mathbf{u}_1 is the eigenvector with the largest eigenvalue.

Our second requirement introduces the additional constraint of orthogonality. Since C is a symmetric matrix, its eigenvectors are orthogonal. Consequently, for \mathbf{u}_2 to be orthogonal to \mathbf{u}_1 , \mathbf{u}_2 must be some linear combination of the other eigenvectors of C .

Let \mathbf{w}_i be the normalised eigenvector of C with eigenvalue, γ_i , where $\gamma_i \geq \gamma_{i+1}$. Then $\mathbf{u}_1 = \mathbf{w}_1$ and we can express \mathbf{u}_2 as

$$\mathbf{u}_2 = \sum_{i=2}^N \alpha_i \mathbf{w}_i \quad (2.3)$$

for some set of α 's.

Since we require $\mathbf{u}_2^T \mathbf{u}_2 = 1$, it follows that

$$\sum_{i=2}^N \alpha_i^2 = 1. \quad (2.4)$$

From $\lambda_2 = \mathbf{u}_2^T C \mathbf{u}_2$ and (2.3), it follows that

$$\lambda_2 = \sum_{i=2}^N \alpha_i^2 \gamma_i.$$

This has to be maximised subject to the constraint (2.4). If we assume that the eigenvalues of C are distinct, then using Lagrange multipliers, it follows readily that $\alpha_2 = 1$ and $\alpha_3 = \dots = \alpha_N = 0$. Therefore $\mathbf{u}_2 = \mathbf{w}_2$. Thus the second eigenface is the eigenvector of C with the second largest eigenvalue. A similar argument applies to the rest of the eigenfaces..

Therefore, our original problem of finding a good basis for the \mathbf{X} 's is equivalent to finding the eigenvectors and eigenvalues of C .

This also allows us to derive a second meaning for the λ 's. If we define the matrix U to consist of all the eigenvectors of C , then C is diagonalised by

$$\Lambda = U^T C U$$

where Λ is diagonal with the eigenvalues λ_i on the diagonal.

Transforming the \mathbf{X}_k 's to a new set of variables

$$\mathbf{y}_k = U^T \mathbf{X}_k \quad (2.5)$$

the covariance matrix of the y_{lk} 's is

$$\begin{aligned} \frac{1}{M} \sum_{k=1}^M \mathbf{y}_k \mathbf{y}_k^T &= \frac{1}{M} \sum_{k=1}^M U^T \mathbf{X}_k \mathbf{X}_k^T U \\ &= \Lambda. \end{aligned}$$

Since this is diagonal, the elements of the \mathbf{y}_k are independent and the diagonal elements, which are the eigenvalues, are the corresponding variances. Therefore

$$\text{Var}(y_l) = \lambda_l.$$

Since the \mathbf{X}_k 's are facial images, the individual pixels are correlated and not independent. However, if we are dealing with a representative sample, we assume that the images are independent of each other. The transformation (2.5) creates a new set of variables, \mathbf{y}_k where the individual elements are also independent.

2.3.2 Reducing the Computational Cost

Conceptually, the situation is simple. The eigenfaces are the eigenvectors of the covariance matrix and the eigenvalues are the variances in the "direction" of the associated eigenfaces. Computationally, though, the problem is complex. In our case, the covariance matrix has $12\,544 \times 12\,544$ elements, which is too large for conventional techniques. We need to simplify the problem. This can be done in different ways, such as by using the SVD. We have, however, chosen to use the method proposed by Sirovich and Kirby [1].

We assume that each eigenvector is a linear combination of the \mathbf{X} 's, i.e.

$$\mathbf{u}_l = \sum_{k=1}^M a_{lk} \mathbf{X}_k. \quad (2.6)$$

Since they describe the same sub-space, this is a perfectly reasonable assumption.

Since the \mathbf{u}_l 's are eigenvectors of C , they satisfy $\lambda_l \mathbf{u}_l = C \mathbf{u}_l$ and (2.6) becomes

$$\lambda_l \sum_{k=1}^M a_{lk} \mathbf{X}_k = C \sum_{k=1}^M a_{lk} \mathbf{X}_k.$$

Using (2.2), it follows that

$$\begin{aligned} \lambda_l \sum_{k=1}^M a_{lk} \mathbf{X}_k &= \frac{1}{M} \sum_{n=1}^M \mathbf{X}_n \mathbf{X}_n^T \sum_{k=1}^M a_{lk} \mathbf{X}_k \\ &= \frac{1}{M} \sum_{k=1}^M a_{lk} \sum_{n=1}^M \mathbf{X}_n \mathbf{X}_n^T \mathbf{X}_k. \end{aligned} \quad (2.7)$$

Defining the scalar quantities

$$K_{nk} = \mathbf{X}_n^T \mathbf{X}_k$$

then (2.7) becomes

$$\lambda_l \sum_{k=1}^M a_{lk} \mathbf{X}_k = \frac{1}{M} \sum_{n=1}^M \mathbf{X}_n \sum_{k=1}^M a_{lk} K_{nk}$$

and equating the coefficients of the \mathbf{X}_k 's (assuming the \mathbf{X}_k 's are independent), we arrive at

$$\lambda_l a_{lk} = \frac{1}{M} \sum_{n=1}^M a_{ln} K_{kn} \quad \forall k \in [1 \dots M]. \quad (2.8)$$

Finally, if we define $\mathbf{a}_l = [a_{l1} \ a_{l2} \ \dots \ a_{lM}]^T$ and $\mathbf{K}_k = [K_{1M} \ \dots \ K_{kM}]^T$, then

$$\sum_{n=1}^M a_{ln} K_{nk} = \mathbf{K}_k^T \mathbf{a}_l$$

and (2.8) becomes

$$M \lambda_l \mathbf{a}_l = \mathbf{K} \mathbf{a}_l \quad (2.9)$$

where

$$K = \begin{bmatrix} \mathbf{K}_1^T \\ \mathbf{K}_2^T \\ \vdots \\ \mathbf{K}_M^T \end{bmatrix}.$$

Thus \mathbf{a}_l is an eigenvector of the $M \times M$ matrix K with eigenvalue $M\lambda_l$. Note that if the \mathbf{X}_k 's are not normalised, we can get extremely large values in K , which can lead to numerical instabilities while calculating the eigenvalues and eigenvectors.

From (2.9), it follows that the problem of finding the eigenvectors of C is reduced to one of finding the eigenvectors of K . Since K is $M \times M$ and M is usually much smaller than N , the magnitude of the problem has been reduced to manageable proportions.

Since this method can only produce at most M non-zero eigenvalues, we will only obtain a maximum of M basis vectors for our subspace. This is quite reasonable since there are only M images describing the subspace we are interested in. As we shall see shortly, most of these make no significant contribution and will also be discarded.

2.3.3 Some Conclusions from the Eigenface Calculation

It is informative to work backwards from the definition of the \mathbf{a}_k 's, since in practice we will calculate the \mathbf{a}_k 's and then construct the \mathbf{u}_l 's.

We start by defining \mathbf{a}_k as an eigenvector of the matrix K , with eigenvalue $M\lambda_k$, where $K_{nk} = \mathbf{X}_n^T \mathbf{X}_k$ and define $\mathbf{u}_l = \sum_{k=1}^M a_{lk} \mathbf{X}_k$, i. e.

$$\sum_{n=1}^M a_{ln} \mathbf{X}_k^T \mathbf{X}_n = M\lambda_l a_{lk} \quad (2.10)$$

Then

$$\begin{aligned} C\mathbf{u}_l &= \frac{1}{M} \sum_{k=1}^M \mathbf{X}_k \mathbf{X}_k^T \sum_{n=1}^M a_{ln} \mathbf{X}_n \\ &= \frac{1}{M} \sum_{k=1}^M \sum_{n=1}^M a_{ln} \mathbf{X}_k \mathbf{X}_k^T \mathbf{X}_n \end{aligned} \quad (2.11)$$

By substituting (2.10) into (2.11), it follows that

$$\begin{aligned}
 C\mathbf{u}_l &= \sum_{k=1}^M \lambda_l a_{lk} \mathbf{X}_k \\
 &= \lambda_l \sum_{k=1}^M a_{lk} \mathbf{X}_k \\
 &= \lambda_l \mathbf{u}_l
 \end{aligned}$$

showing that \mathbf{u}_l is an eigenvector of C , as expected.

The orthogonality of the \mathbf{u}_l 's follows immediately from the fact that they are the eigenvectors of C , where C is symmetric. Since we require $\|\mathbf{u}_l\| = 1$, it is necessary to find how $\|\mathbf{u}_l\|$ depends on $\|\mathbf{a}_l\|$.

So we consider

$$\begin{aligned}
 \mathbf{u}_j^T \mathbf{u}_j &= \left(\sum_{i=1}^M a_{ji} \mathbf{X}_i \right)^T \left(\sum_{k=1}^M a_{lk} \mathbf{X}_k \right) \\
 &= \sum_{i=1}^M \sum_{k=1}^M a_{ji} \mathbf{X}_i^T \mathbf{X}_k a_{lk} \\
 &= \sum_{i=1}^M a_{ji} \sum_{k=1}^M K_{ik} a_{lk}.
 \end{aligned}$$

From (2.9), it follows that

$$\begin{aligned}
 \mathbf{u}_j^T \mathbf{u}_l &= \sum_{i=1}^M a_{ji} M \lambda_l a_{li} \\
 &= M \lambda_l \mathbf{a}_j^T \mathbf{a}_l.
 \end{aligned} \tag{2.12}$$

Since K is symmetric, the \mathbf{a}_k 's are orthogonal, so from (2.12)

$$\mathbf{u}_j^T \mathbf{u}_l = \begin{cases} M \lambda_l \|\mathbf{a}_l\|^2 & j = l \\ 0 & j \neq l \end{cases}$$

Therefore, for the \mathbf{u}_l 's to be orthonormal, we need to scale the \mathbf{a}_k 's so that

$$\|\mathbf{a}_k\|^2 = \frac{1}{M \lambda_k}.$$

2.3.4 An Algorithm for Calculating the Eigenfaces

To implement this we used this algorithm.

The actual code is listed as *eigface.m* in appendix A.1.1.

```

Begin
  Read  $M$ 
   $I_n$  = image  $n$  of training set for  $n = 1, \dots, M$ 
   $A$  = average of  $I_n$ 
  Let  $X_n = \frac{I_n - A}{\|I_n - A\|}$  for  $n = 1, \dots, M$ 
  Let  $K = X^T X$ 
  Calculate  $Kv_n = \lambda_n v_n$  for  $n = 1, \dots, M$ 
  Sort  $v$ 's according to  $\lambda$ 's
  Calculate eigenfaces:  $U_n = v_n^T \times X$ 
  Rescale :  $U_n = \frac{U_n}{\|U_n\|}$ 
End

```

2.3.5 Images of the Eigenfaces

Figure 2.4: 1st, 3rd, 20th and 40th eigenfaces

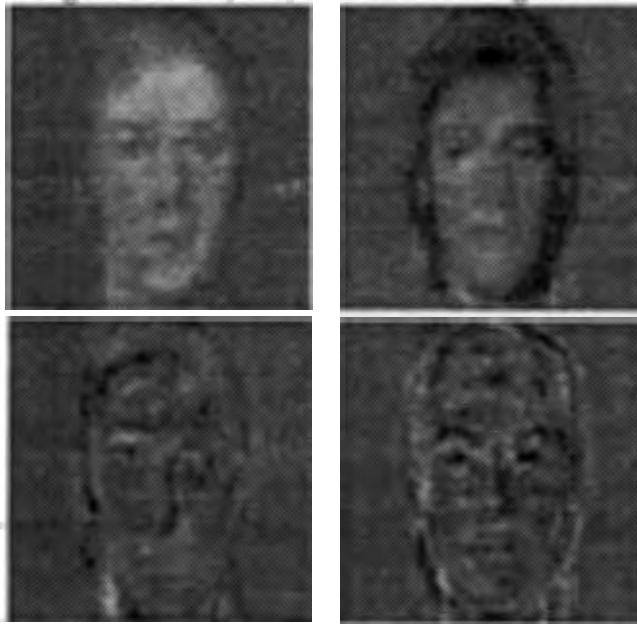


Figure 2.4 shows the first, third, twentieth and fortieth eigenfaces (going from left to right and top to bottom). If we compare them, we note that the first two have a very smooth appearance. This indicates that they only

describe general features about the shape of the face. The 20th eigenface has rather more specific features, which indicates that it starts providing information useful to uniquely describe an individual face. The 40th eigenface resembles a particular individual, thus the information it describes will not be applicable to many of the images. In general, this is not a particularly desirable characteristic, but is unavoidable in this case due to the small size of the training set.

In these images, as in figure 2.3, we have scaled the images so that zero corresponds to grey. With a L_2 -norm of 1, the values in the actual eigenface are quite small. So we transformed the image elements to make full use of the available greyscale range.

2.4 Representing Faces using Eigenfaces

2.4.1 Basic Reconstruction

We have calculated a set of basis vectors \mathbf{u}_l (our eigenfaces) which span the subspace described by the faces in the training set. Any image in this training set can be described as a linear combination of these eigenfaces. If the training set is truly representative, one can reasonably expect that images which not actually in the training set will also be well represented by our eigenfaces. We will discuss how to represent these images in terms of the eigenfaces.

Any image in the training set can be written as a linear combination of the eigenfaces. This implies

$$\begin{aligned}\mathbf{X}_k &= y_{1k}\mathbf{u}_1 + y_{2k}\mathbf{u}_2 + \cdots + y_{Mk}\mathbf{u}_M \\ &= U\mathbf{y}_k\end{aligned}$$

where $U = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_M \end{bmatrix}$.

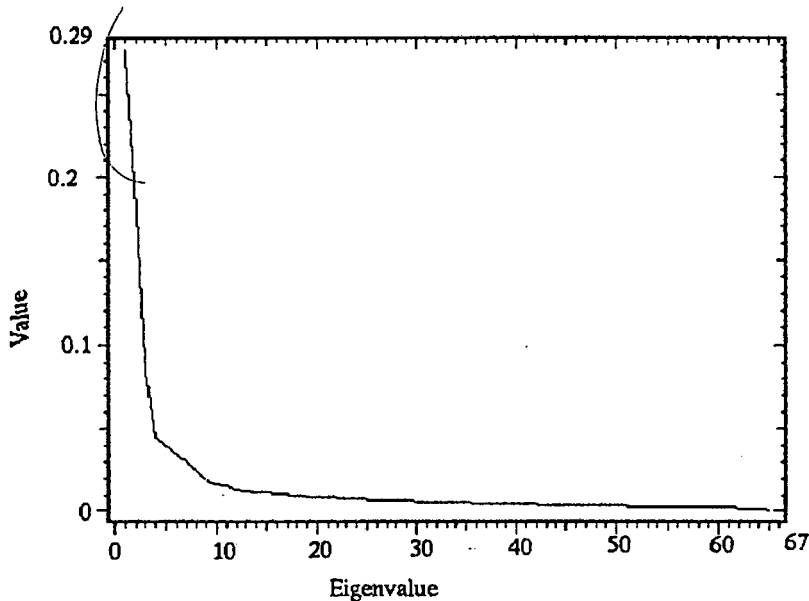
Since the \mathbf{u}_l 's are orthonormal, $U^{-1} = U^T$, so for any \mathbf{X}_k in the vector space spanned by the eigenfaces,

$$\mathbf{y}_k = U^T\mathbf{X}_k.$$

Using the transformation in this form requires storing M coefficients. Though in general $M \ll N$, M may still be large in many applications (of the order of several hundred.) Recall that we define the eigenfaces so that each

eigenface describes as much of the variation between the images as possible, subject to the appropriate constraints. Thus the eigenfaces with the smallest associated eigenvalues describe the least significant variations, which should neither be particularly noticeable if lost in reconstruction, nor be particularly important in distinguishing between 2 different individuals. Thus we should be able to discard some of the less important eigenfaces without adversely impacting on the representation of the image.

Figure 2.5: Eigenvalues of the covariance matrix.



If we look at a graph of the eigenvalues (figure 2.5), we can see that the magnitude of the eigenvalues drops rapidly. Thus the vast majority of the eigenfaces describe only minor variations, which can be treated as negligible.

In most of our experiments, we rather arbitrarily choose to keep 40 coefficients. Similar figures are quoted in the literature (for example see Sirovich and Kirby [1] or Turk and Pentland [10]). In our case, where we use a small database, this figure gives very good results. With our database, even 30 coefficients give satisfactory results, but we chose to err on the conservative side by using 40. Very little information is available for large databases, so it is unclear how many eigenfaces we would keep in practice.¹

Since we are retaining only 40 coefficients, we only need the first 40 eigenfaces. So our transformation equations are

¹ Kodak developed a system, presumably based on ideas similar to eigenfaces, where 50 bytes is sufficient to reconstruct a face.

$$\mathbf{X}_k \approx U \mathbf{y}_k$$

where

$$\mathbf{y}_k = U^T \mathbf{X}_k$$

and U is now defined as $U = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_{40} \end{bmatrix}$. It is important to note that our reconstruction is no longer exact, even for images in the training set.

To actually restore the original picture, we still need to reverse the normalisation step on the \mathbf{X}_k 's. Thus we have that

$$\mathbf{I}_k \approx \|\mathbf{I}_k - \mathbf{A}\| U \mathbf{y}_k + \mathbf{A}.$$

Any new image can of course be treated in a similar way. However, as we can see, dividing by the norm introduces an extra factor which is required to reconstruct the original image. But the normalisation is just a scaling on the coefficients. This is important in calculating the eigenfaces since we want to avoid overly large values in the eigenvector calculation. However, for these calculations, there is no pressing need to scale these, so we choose to ignore the normalisation step.

Then, given a image \mathbf{J} which is not in the training set, its eigenface coefficients are

$$\mathbf{y}_J = U^T (\mathbf{J} - \mathbf{A}). \quad (2.13)$$

Similarly, the reconstruction of the image is

$$\tilde{\mathbf{J}} = U \mathbf{y}_J + \mathbf{A}. \quad (2.14)$$

If \mathbf{J} is sufficiently well described by the images in the training set, then $\mathbf{J} \approx \tilde{\mathbf{J}}$.

The implementation of these equations is extremely simple. For the details, see *process.m* in appendix A.1.2.

2.4.2 Results

Figure 2.6 shows one of the the images in our training set. Consequently, it is inside the space described by the eigenfaces and a full reconstruction

Figure 2.6: Image in training set



is possible if the full set of eigenfaces is used. Figure 2.7 shows the partial reconstruction using a small subset of the eigenfaces. More specifically, figure 2.7 shows the reconstruction using the first 10 and the first 40 eigenfaces respectively.

The first reconstruction only describes the basic shape of the face and little else and has only a slight resemblance to the original. However the reconstruction using 40 values is a considerable improvement and this image shows only minor differences from the original.

Figure 2.7: Reconstruction of figure 2.6 using both 10 and 40 values



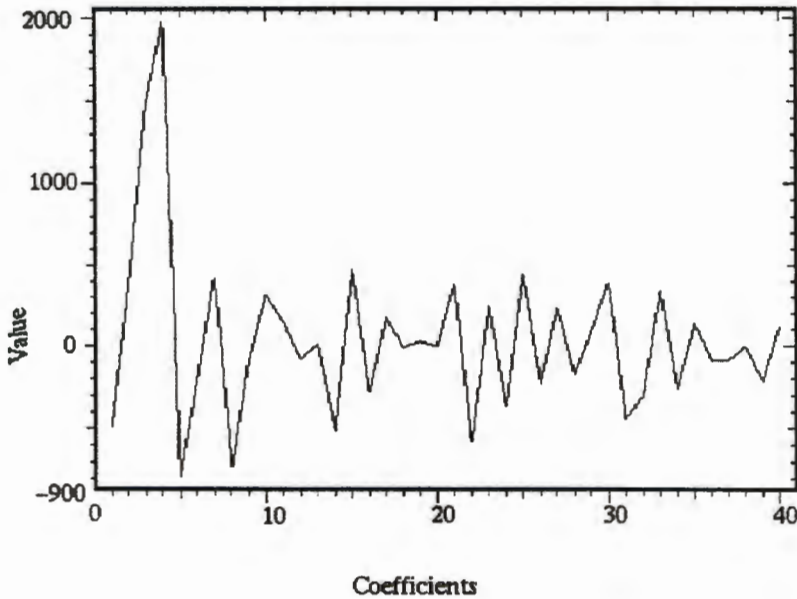
In figure 2.8, we display the absolute error between the image and its 40 coefficient reconstruction. We notice that the error is concentrated near edges, such as the nose and mouth. This is a desirable feature since the eye is not as sensitive to some loss of detail in these regions, a fact exploited by many image compression algorithms.

The reconstruction coefficients are shown in figure 2.9. Note that most of the information is contained in the first few coefficients as these elements have the greatest magnitude. However, from the reconstructions shown in figure 2.7, it is obvious that these first few coefficients are not able to identify individual features - they only describe general features about the shape of the face.

Figure 2.8: Reconstruction error from using 40 value in fig. 2.7.



Figure 2.9: Graph of the coefficients used in fig. 2.7.



At this point, let us briefly point out the key difference between this scheme and standard image compression algorithms, such as JPEG or wavelet based methods. Generally, in image compression we have no knowledge about the image. It can lie anywhere with the 12 544 dimensional image space. Compression algorithms such as JPEG try to exploit the correlation between different regions within a single image. However, if we restrict ourselves to only facial images, it is possible to exploit the correlation between the different images. Eigenfaces use this idea to eliminate the redundant information between the different faces. It is quite clearly not suitable for general images.

To illustrate this point, we will attempt to use our eigenfaces to reconstruct the rose shown in figure 2.10. Since it has no face-like features, we expect a reconstruction using eigenfaces to be rather poor.

Figure 2.10: Image of rose.



In figure 2.11, we show the results of reconstructing the rose (using 40 values). Not unexpectedly, the reconstructed image is face-like, and does not look like the original image at all.

Figure 2.11: Reconstruction of fig. 2.10.



Figure 2.12: Image not in training set



Images in the training set are reconstructed very effectively. In practice, however, we want to be able to represent any arbitrary facial image we might encounter. In figure 2.12 we have an image that is not in our training set. However, another image of the same person is in our training set, so this image is similar to some images in the training set.

As before, we reconstruct figure 2.12 using both 10 and 40 coefficients in figure 2.13. The reconstruction with 10 coefficients again bears little resem-

Figure 2.13: Reconstruction of fig. 2.12 using 10 and 40 values.



blance to the original.

If we consider the reconstruction with 40 values, there is again little difference between the original and the reconstruction.

Figure 2.14: Error from using 40 value reconstruction in fig. 2.13



The error in this second reconstruction is shown in figure 2.14. Again, the error is concentrated around the edges.

We plot the coefficients in figure 2.15. Again, the first few the coefficients are the largest. However, they serve mainly to describe the shape of the face and not any particularly distinguishing features.

The next question is whether we are able to represent images which are not of individuals in the training set. Figure 2.16 shows such a face and its reconstruction is shown in figure 2.17. Although the reconstruction bears some resemblance to the original, it is not particularly good. This illustrates the importance of having a representative training set.

2.4.3 Reconstruction Error

The visual comparison of the images in the previous section indicates that the reconstruction error is small for images in the training set and that the

Figure 2.15: Graph of the elements of fig. 2.13's reconstruction

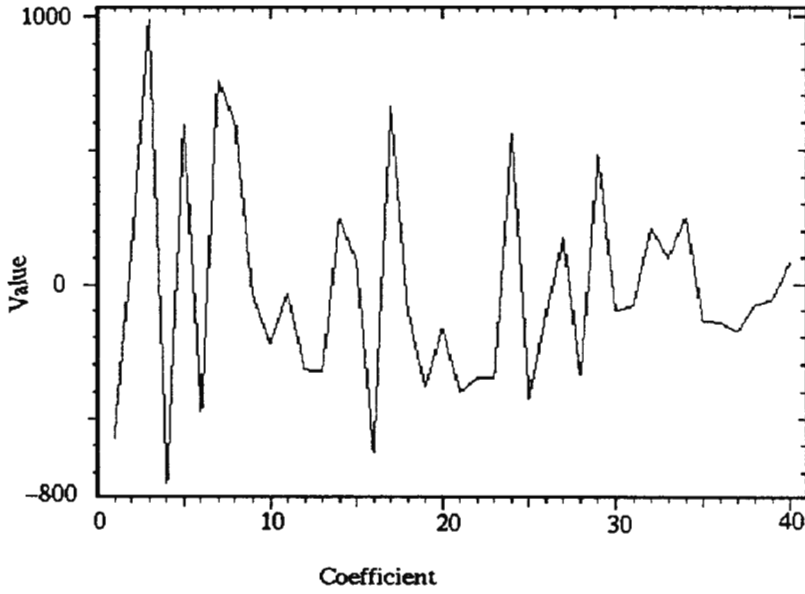


Figure 2.16: Individual not represented in the training set.



Figure 2.17: Reconstruction of fig. 2.16



quality of the reconstruction degrades for images outside the training set. In order to be more precise, we need to quantify the error.

Equations (2.13) and (2.14) can be viewed as a projection operation where the image $\mathbf{J} - \mathbf{A}$ is projected onto the subspace described by the eigenfaces. Thus a natural measure is the projection error, which is the distance between the reconstruction and the original image

$$\epsilon(\mathbf{J}) = \left\| (\mathbf{J} - \mathbf{A}) - (\tilde{\mathbf{J}} - \mathbf{A}) \right\|. \quad (2.15)$$

We define $\mathbf{C}_{\mathbf{J}} = \mathbf{J} - \mathbf{A}$ and $\widetilde{\mathbf{C}}_{\mathbf{J}} = \tilde{\mathbf{J}} - \mathbf{A}$. Then (2.15) becomes $\epsilon(\mathbf{J}) = \left\| \mathbf{C}_{\mathbf{J}} - \widetilde{\mathbf{C}}_{\mathbf{J}} \right\|$. Since this measure requires calculating $\widetilde{\mathbf{C}}_{\mathbf{J}}$, it is computationally expensive. Calculating $\|\mathbf{C}_{\mathbf{J}}\|$ is easy since we obtain $\mathbf{C}_{\mathbf{J}}$ in calculating the representation, and calculating the length involves only N multiplications. To calculate $\|\widetilde{\mathbf{C}}_{\mathbf{J}}\|$ requires that we multiply by the matrix UU^T , which is a much more expensive operation. However, since $\epsilon(\mathbf{J})$ is obtained from an orthogonal projection, $\mathbf{C}_{\mathbf{J}} - \widetilde{\mathbf{C}}_{\mathbf{J}}$ is orthogonal to $\widetilde{\mathbf{C}}_{\mathbf{J}}$. Thus by Pythagoras' theorem, we can write

$$\epsilon^2(\mathbf{J}) = \|\mathbf{C}_{\mathbf{J}}\|^2 - \|\widetilde{\mathbf{C}}_{\mathbf{J}}\|^2.$$

However, from our definitions, note that

$$\begin{aligned} \|\widetilde{\mathbf{C}}_{\mathbf{J}}\|^2 &= \|\tilde{\mathbf{J}} - \mathbf{A}\|^2 \\ &= \|U\mathbf{y}_j\|^2 \\ &= \mathbf{y}_j^T U^T U \mathbf{y}_j \end{aligned}$$

and, since the U is orthogonal, it follows immediately that

$$\|\widetilde{\mathbf{C}}_{\mathbf{J}}\|^2 = \|\mathbf{y}_j\|^2.$$

Thus our error measure is simply

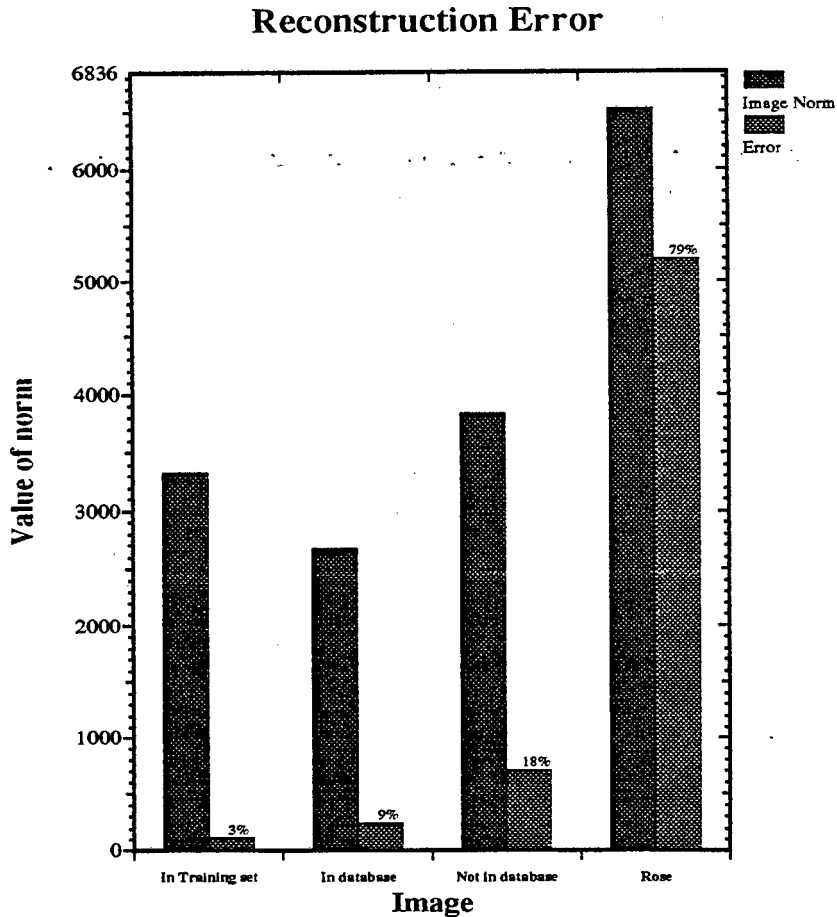
$$\epsilon^2(\mathbf{J}) = \|\mathbf{C}_{\mathbf{J}}\|^2 - \|\mathbf{y}_j\|^2 \quad (2.16)$$

which is much easier to evaluate.

$\epsilon(\mathbf{J})$ is commonly known as the “distance from face space” or DFFS measure, which sometimes scaled by $\|\mathbf{C}_{\mathbf{J}}\|$, to convert it to the relative error, $r = \frac{\epsilon(\mathbf{J})}{\|\mathbf{C}_{\mathbf{J}}\|}$. This is a appropriate measure for deciding how face-like an image is.

We can now calculate the errors for the images used in the previous section. The graph in figure 2.18 shows $\|\mathbf{C}_{\mathbf{J}}\|$ as well as r_{40} for all these images.

Figure 2.18: Bar graph of reconstruction errors.



This illustrates how good the reconstruction of faces in the training set is. It also shows the poor reconstruction of the rose.

If we calculate the reconstruction error using only 10 coefficients we find that the majority of the information about the image (in terms of the norm) is contained in the first 10 eigenfaces. However, as we saw, it is the later eigenfaces which provide most of the detail required to recognise an individual. Typically r_{10} is only 5-10% larger than r_{40} .

This measure can also be used to decide how many eigenfaces we should use for our representations. An obvious choice would be to keep enough eigenfaces so that $\epsilon(X_k)$ is below some threshold for each image in the

database.

2.5 Recognizing Faces using Eigenfaces

2.5.1 Description

The results of the previous section suggest that it is possible to represent facial images using about 40 eigenfaces, at least for relatively small datasets without too much variation between the faces. Moreover, each eigenface coefficient has a very specific meaning – it describes how much of the information about the image is represented by the corresponding eigenface. This allows a simple recognition rule.

In a face recognition problem, we usually have a set of faces for which we know the identity of the people involved and the eigenface representation for all these faces. We assume that for all these images $\epsilon(\mathbf{J})$ is small. We call this set the database of known faces. We acquire one additional image. This image could either be of somebody in the database, or of somebody not in the database (an unknown face), or it may not be the image of a face at all. Our problem is to determine which of the three possibilities applies, and if the image is a face in the database, to correctly identify the individual.

The case of a image of something other than a face is easily dealt with. Since the eigenfaces describe a subspace defined by the facial images in the training set, only images which are sufficiently similar to those in the training set will be well described, i.e. have a small value for $\epsilon(\mathbf{J})$. A new image will only be considered a face if $\epsilon(\mathbf{J})$ is below a certain threshold. (If we consider our example of the rose, we see that this has a very large value for the reconstruction error and will thus be quickly rejected as not being the image of a face.)

For the actual identification, we simply compare the eigenface coefficients, given them all equal weights. Of course, it is possible that the recognition performance can be improved by using different weights. We will describe this possibility in more depth in connection with the OCR problem in section 3.5.

Following Turk and Pentland [5], we choose to use the L_2 -norm of the difference between two representations as our measure of similarity. Thus, if we have an new image \mathbf{P} , where $\epsilon^2(\mathbf{P})$ is small, we will decide that \mathbf{P} matches image \mathbf{Q} in the database if $\|\mathbf{y}_P - \mathbf{y}_Q\|$ is smaller than a specified threshold and a minimum over all the faces in the database.

2.5.2 An Algorithm for Recognition

This is a simple algorithm which we used when searching the database for a match.

The actual code is listed as *findmatch.m* in appendix A.1.3.

```

// We assume that  $U$  = eigenvectors,  $A$  = the average
// and  $D$  stores the coefficients of our database
// We have 2 threshold values:
// ImThreshold = threshold to accept as image
// ImMatch = threshold to accept closest match
// as a match
Begin
  I = image to find match for
  Let  $y = U^T(I - A)$ 
  Let  $\epsilon = \|I - A\|^2 - \|y\|^2$  // get error
  if ( $\epsilon < \text{ImThreshold}$ ) // accept as image of face
    Let  $\text{err}_n = \|y - D_n\|$  for all  $n$ 
    match = value of  $n$  for which  $\text{err}_n$  is a minimum
    if ( $\text{err}_{\text{match}} < \text{ImMatch}$ )
      Successful match is match
    else
      No matching image
    end if
  else
    Image is not a face
  end if
End

```

Although this algorithm is by no means particularly efficient, it does indicate how simply we can implement a image recognition system using this technique.

2.5.3 Results

To demonstrate the technique, we take our earlier example of a face not in the database, see figure 2.19, and try to find the best match in our database of 100 images.

Figure 2.20 shows the best match². Although this is a different image, it is clearly the same person.

²This image is reconstructed from the representation in the database.

Figure 2.19: Image outside of database

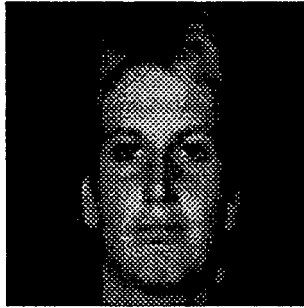
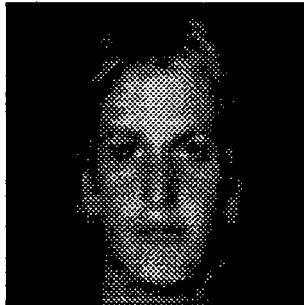


Figure 2.20: Best match from database



It is informative to compare the representations of the two images. If we plot the two on the same set of axes, as in figure 2.21, we see that the two graphs are very similar indeed. The most significant differences occur in the later coefficients, while the earlier ones are extremely close to each other.

Another view is given in figure 2.22 where we plot the two representations against each other. The graph clearly has a distinctly linear trend.

Since we are now plotting the magnitude of the coefficients, it is important to remember that the coefficients with the highest magnitudes correspond to the first few eigenfaces, since they have the largest variance. We can see that the graph is very close to a straight line at these high magnitudes, indicating that the first few coefficients are very similar.

Also, from this graph, we can see that most of the variation between the two representations is concentrated near those values which are close to zero. Since these correspond to the later eigenfaces, this agrees with the behaviour observed in figure 2.21.

This example of course is not a test of the recognition performance of the technique. We will discuss the results of testing this system in section 2.8.

Figure 2.21: Graph comparing representations of image (2.19) and best match (2.20)

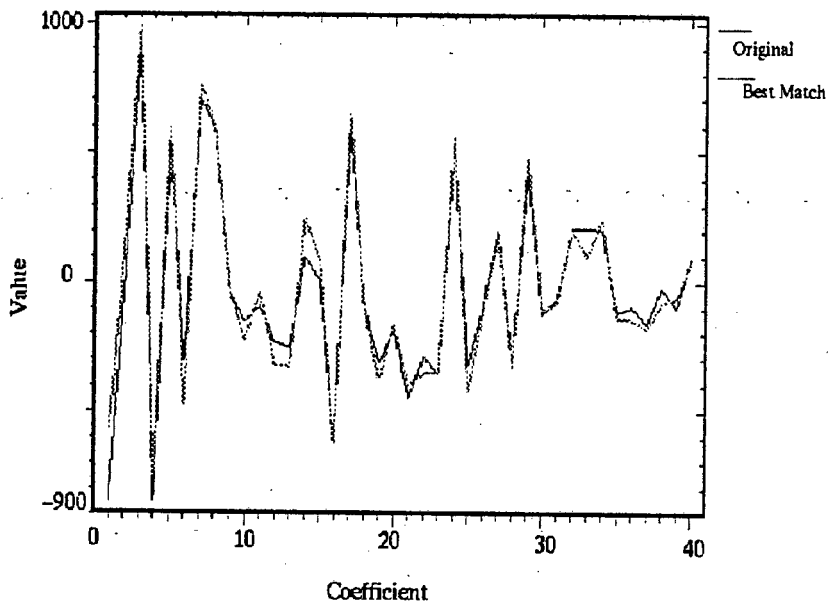
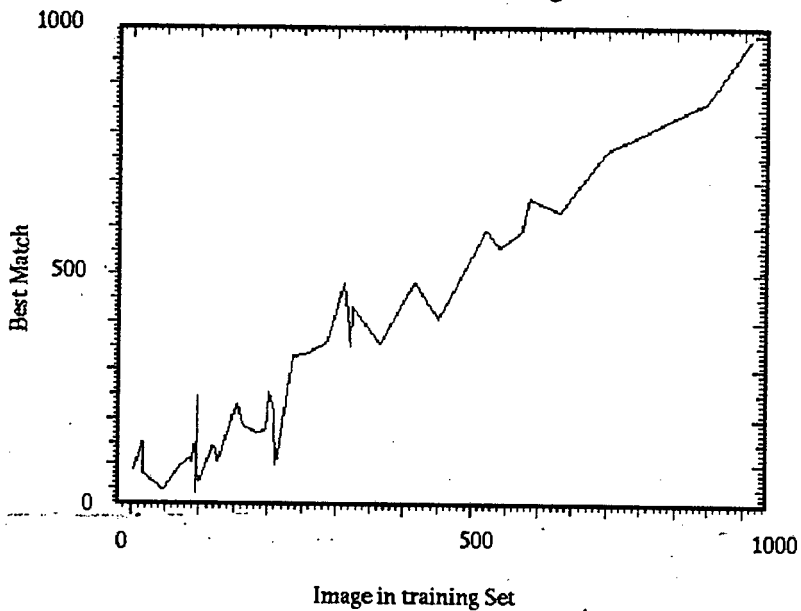


Figure 2.22: Graph of representation of 2.19 against that of best match



2.6 Image Segmentation using Eigenfaces

2.6.1 Problem Background

In section 2.2, when calculating the average face, we highlighted the need to centre the face in an image. In small scale experiments, this can be done

by hand, but in practice, we would like some automatic method to locate and centre the face. Simply put, we wish to find the most face-like section of any given image, then normalise its location.

The usual method of locating an object in an image is to use the correlation between the image and a reference image of the object of interest. The problem is that we have no sufficiently general image of a face to use as a reference image. For example, the average face is such a general image that it is also correlated with many images which are not faces.

Earlier, we introduced the distance from face space $\epsilon(\mathbf{J})$ (equation (2.16)), which we use to measure the reconstruction error. We explained how this measure can be used in determining whether a certain image is indeed a face or not. A fairly obvious method of locating a face in an image would be to search for that section of an image which minimises $\epsilon(\mathbf{J})$ (see Pentland, Moghadden and Starner [4]).

However, $\epsilon(\mathbf{J})$ can be small for non-facelike images. For instance, by using some random coefficients, we are able to create the image in figure 2.23. This is clearly not a face, even though it has some facelike characteristics. More precisely, we created this image by taking a random set of coefficients to reconstruct an image from the eigenfaces. We then scaled this image to fit into our greyscale range and then projected back onto the space spanned by the eigenfaces. This particular image has very large coefficients for the 30th and 31st eigenfaces, which means that the arrangement of the coefficients differs quite markedly from those shown earlier.

Thus this image is perfectly represented by the eigenfaces, which means that $\epsilon(\mathbf{J}) = 0$. If we use the DFFS to locate a face, this image would be preferred over an actual face. Although this is not a problem in the recognition phase since it should not be close enough to any of the images in the database to register as a match, it is not desirable when locating a face. Thus the distance from face space is an inadequate measure of how facelike an image is.

Figure 2.23: Non-face perfectly described by the eigenfaces



2.6.2 A Maximum Likelihood Statistic for Face Location

We now turn to a statistical estimate (first proposed by Moghadden and Pentland [6]) of how facelike an image is to find the location of a face in a general image. We assume that the facial images have a multivariate Gaussian (normal) distribution in the total image space Ω . Since the matrix C is defined as the covariance matrix of the \mathbf{X}_n 's, it follows from the definition of the multivariate Gaussian distribution that

$$P(\mathbf{X}|\Omega) = \frac{e^{-\frac{1}{2}(\mathbf{X}^T C^{-1} \mathbf{X})}}{(2\pi)^{\frac{N}{2}} |C|^{\frac{1}{2}}} \quad (2.17)$$

where N is the dimension of our image space (in our case $N \approx 100^2$) and $\mathbf{X} = \mathbf{I} - \mathbf{A}$ for some image \mathbf{I} .³

A sufficient statistic for this probability is

$$d(\mathbf{X}) = \mathbf{X}^T C^{-1} \mathbf{X}.$$

Furthermore, we know that C can be written as

$$C = U^T \Lambda U$$

where U is now defined as containing *all* the eigenvectors. Then

$$\begin{aligned} d(\mathbf{X}) &= \mathbf{X}^T C^{-1} \mathbf{X} = \mathbf{X}^T U^T \Lambda^{-1} U \mathbf{X} \\ &= \mathbf{y}^T \Lambda^{-1} \mathbf{y}. \end{aligned}$$

However, Λ is diagonal, so Λ^{-1} is also diagonal. Thus the expression can be simplified to

$$\begin{aligned} d(\mathbf{X}) &= \sum_{i=1}^N \frac{y_i^2}{\lambda_i} \\ &= \sum_{i=1}^P \frac{y_i^2}{\lambda_i} + \sum_{i=P+1}^N \frac{y_i^2}{\lambda_i} \end{aligned}$$

³We use the notation \mathbf{X} conditional on Ω ($P(\mathbf{X}|\Omega)$) to indicate that we are dealing with the images in relation to the entire image space, and not the subspace described by the eigenfaces.

where P is the number of eigenvectors we keep for reconstruction purposes.

Note that this allows us to write

$$P(\mathbf{X}|\Omega) = P_F(\mathbf{X}|\Omega) P_{\tilde{F}}(\mathbf{X}|\Omega) \quad (2.18)$$

where

$$P_F(\mathbf{X}|\Omega) = \frac{e^{-\frac{1}{2} \left(\sum_{i=1}^P \frac{y_i^2}{\lambda_i} \right)}}{(2\pi)^{\frac{P}{2}} \prod_{i=1}^P \lambda_i^{\frac{1}{2}}}$$

and

$$P_{\tilde{F}}(\mathbf{X}|\Omega) = \frac{e^{-\frac{1}{2} \left(\sum_{i=P+1}^N \frac{y_i^2}{\lambda_i} \right)}}{(2\pi)^{\frac{N-P}{2}} \prod_{i=P+1}^N \lambda_i^{\frac{1}{2}}}.$$

Here $P_F(\mathbf{X}|\Omega)$ is the true distribution in “face-space”, while $P_{\tilde{F}}(\mathbf{X}|\Omega)$ is the distribution in the space orthogonal to “face-space”. (i.e. $F \perp \tilde{F}$).

This measure $d(\mathbf{X})$ requires that we store all the eigenvectors and eigenvalues. Looking for an approximation $\hat{d}(\mathbf{X})$, which uses only the available eigenfaces, we write

$$\hat{d}(\mathbf{X}) = \sum_{i=1}^P \frac{y_i^2}{\lambda_i} + \frac{1}{\rho} \left(\sum_{j=P+1}^N y_j^2 \right)$$

where ρ is chosen to minimise the error.

Assuming that \mathbf{y} represents the image exactly, it follows from $\mathbf{y} = U\mathbf{X}$ and the orthonormality of U that

$$\|\mathbf{X}\| = \|\mathbf{y}\|.$$

Therefore

$$\begin{aligned} \epsilon^2(\mathbf{X}) &= \|\mathbf{y}\|^2 - \sum_{i=1}^P y_i^2 \\ &= \sum_{i=1}^N y_i^2 - \sum_{i=1}^P y_i^2 \\ &= \sum_{i=P+1}^N y_i^2 \end{aligned}$$

and it follows that

$$\hat{d}(\mathbf{X}) = \sum_{i=1}^P \frac{y_i^2}{\lambda_i} + \frac{1}{\rho} (\epsilon^2(\mathbf{X}))$$

We can now estimate the $P(\mathbf{X}|\Omega)$ as the product of two independent Gaussian distributions. By substituting $\hat{d}(\mathbf{X}|\Omega)$ into equation 2.17,

$$\begin{aligned} \hat{P}(\mathbf{X}|\Omega) &= \left[\frac{e^{-\frac{1}{2} \left(\sum_{j=1}^P \frac{y_j^2}{\lambda_j} \right)}}{(2\pi)^{\frac{P}{2}} \left(\prod_{i=1}^P \lambda_i^{\frac{1}{2}} \right)} \right] \times \left[\frac{e^{-\frac{\epsilon^2(\mathbf{X})}{2\rho}}}{(2\pi\rho)^{\frac{N-P}{2}}} \right] \\ &= P_F(\mathbf{X}|\Omega) \hat{P}_{\tilde{F}}(\mathbf{X}|\Omega). \end{aligned} \quad (2.19)$$

Here $\hat{P}_{\tilde{F}}(\mathbf{X}|\Omega)$ is the estimate for $P_{\tilde{F}}(\mathbf{X}|\Omega)$.

We choose ρ so that this estimate for $P(\mathbf{X}|\Omega)$ is as accurate as possible. We followed Moghadden and Pentland [6] who calculated the optimal value of ρ by minimising

$$J(\rho) = \mathbf{E} \left[\log \frac{P(\mathbf{X}|\Omega)}{\hat{P}(\mathbf{X}|\Omega)} \right].$$

This measure comes from information theory and is known as the Kullman-Leibler Divergence, also referred to as the informational divergence or information for discrimination. It provides a measure of the error made by assuming that the true distribution is $\hat{P}(\mathbf{X}|\Omega)$ when it is actually $P(\mathbf{X}|\Omega)$. It can be shown that $J(\rho)$ is always nonnegative and that the larger $J(\rho)$, the less the correspondence between the distributions, and thus the less accurate the results obtained by using $\hat{P}(\mathbf{X}|\Omega)$ instead of $P(\mathbf{X}|\Omega)$ (See Cover and Thomas [17] for more details). Thus minimising $J(\rho)$ optimises our approximate distribution $\hat{P}(\mathbf{X}|\Omega)$.

Substituting (2.18) and (2.19) into the definition of $J(\rho)$, we see that

$$\begin{aligned} J(\rho) &= \mathbf{E} \left[\log \frac{P_{\tilde{F}}(\mathbf{X}|\Omega)}{\hat{P}_{\tilde{F}}(\mathbf{X}|\Omega)} \right] \\ &= \mathbf{E} \left[\log \left(\frac{\exp \left(-\frac{1}{2} \sum_{i=P+1}^N \frac{y_i^2}{\lambda_i} \right)}{(2\pi)^{\frac{N-P}{2}} \prod_{i=P+1}^N \sqrt{\lambda_i}} \times \frac{(2\pi\rho)^{\frac{N-P}{2}}}{\exp \left(-\frac{1}{2} \sum_{i=P+1}^N y_i^2 \right)} \right) \right] \end{aligned}$$

$$\begin{aligned}
&= \mathbf{E} \left[\log \left(\frac{\rho^{\frac{N-P}{2}} \exp \left(-\frac{1}{2} \sum_{i=P+1}^N \frac{y_i^2}{\lambda_i} \right)}{\exp \left(-\frac{1}{2} \sum_{i=P+1}^N y_i^2 \right) \prod_{i=P+1}^N \lambda_i^{\frac{1}{2}} \right)} \right] \\
&= \mathbf{E} \left[\log \left(\left(\prod_{i=P+1}^N \frac{\rho}{\lambda_i} \right)^{\frac{1}{2}} \left(e^{-\frac{1}{2} \left(\sum_{i=P+1}^N \left(\frac{y_i^2}{\lambda_i} - \frac{y_i^2}{\rho} \right)} \right)} \right) \right) \right] \\
&= \mathbf{E} \left[\frac{1}{2} \left(\sum_{i=P+1}^N \log \frac{\rho}{\lambda_i} \right) - \frac{1}{2} \left(\sum_{i=P+1}^N \left(\frac{y_i^2}{\lambda_i} - \frac{y_i^2}{\rho} \right) \right) \right] \\
&= \frac{1}{2} \left(\sum_{i=P+1}^N \left(\log \frac{\rho}{\lambda_i} - \left(\frac{\mathbf{E}[y_i^2]}{\lambda_i} - \frac{\mathbf{E}[y_i^2]}{\rho} \right) \right) \right). \tag{2.20}
\end{aligned}$$

However, in section (2.3.1), we showed that

$$\begin{aligned}
\mathbf{E}[y_i^2] &= \mathbf{Var}(y_i) \\
&= \lambda_i.
\end{aligned}$$

Substituting this into (2.20),

$$\begin{aligned}
J(\rho) &= \frac{1}{2} \sum_{i=P+1}^N \left(\log \frac{\rho}{\lambda_i} - \left(\frac{\lambda_i}{\lambda_i} - \frac{\lambda_i}{\rho} \right) \right) \\
&= \frac{1}{2} \sum_{i=P+1}^N \left(\frac{\lambda_i}{\rho} - 1 + \log \frac{\rho}{\lambda_i} \right)
\end{aligned}$$

and

$$\frac{\delta J(\rho)}{\delta \rho} = \frac{1}{2} \sum_{i=P+1}^N \left(-\frac{\lambda_i}{\rho^2} + \frac{1}{\rho} \right).$$

Then, to minimise $J(\rho)$, we require

$$\frac{1}{2\rho} \sum_{i=P+1}^N \left(1 - \frac{\lambda_i}{\rho} \right) = 0$$

which amounts to

$$\rho = \frac{1}{N-P} \sum_{i=P+1}^N \lambda_i.$$

Thus ρ is simply the average of the eigenvalues we ignore - clearly a reasonable choice. Since we only calculate M non-zero eigenvalues, we assume for simplicity that $\lambda_i = 0 \forall i > M$.

So final form of $\hat{d}(\mathbf{X})$ is then

$$\begin{aligned}\hat{d}(\mathbf{X}) &= \sum_{i=1}^P \frac{y_i^2}{\lambda_i} + \frac{N-P}{\sum_{i=P+1}^N \lambda_i} \left(\sum_{i=P+1}^N y_i^2 \right) \\ &= \sum_{i=1}^P \frac{y_i^2}{\lambda_i} + \frac{N-P}{\sum_{i=P+1}^N \lambda_i} \epsilon^2(\mathbf{X}).\end{aligned}$$

We now have enough information to calculate the estimated probability $\hat{P}(\mathbf{X}|\Omega)$. To convert this to the maximum likelihood estimator, we define x_{ij} as a section of the image centred at (i, j) . This can be rescaled and treated as a full size image. Our likelihood estimate is $S_{ij} = \hat{P}(x_{ij}|\Omega)$. The maximum likelihood then represents the region which we identify as containing the face. We can handle a multi-scale search by varying the size of x_{ij} .

Since $\hat{d}(\mathbf{X})$ is a sufficient statistic for $\hat{P}(\mathbf{X}|\Omega)$, we need only calculate $\hat{d}(\mathbf{X})$, rather than the entire probability expression (see Rice [25]). Since we are looking for that region which has the highest probability of being a face, we want to maximise $\hat{P}(\mathbf{X}|\Omega)$. However, we note that

$$\hat{P}(\mathbf{X}|\Omega) = \frac{e^{-\frac{\hat{d}(\mathbf{X})}{2}}}{(2\pi)^{\frac{N}{2}} \left(\prod_{i=1}^P \lambda_i^{\frac{1}{2}} \right) \left(\rho^{\frac{N-P}{2}} \right)}. \quad (2.21)$$

This is obviously a maximum if $\hat{d}(\mathbf{X}|\Omega)$ is a minimum. So to maximise $\hat{P}(\mathbf{X}|\Omega)$, we need to minimise $\hat{d}(\mathbf{X})$.

Since we are only dealing with the most basic face-like characteristics of an image, we do not need the same degree of accuracy here as for the reconstruction of the faces. Since the first 10 eigenfaces describe the basic shape of the face, we can use $P = 10$ without adversely affecting performance.

Although the image in figure 2.23, is constructed from the eigenfaces, it does not satisfy the normality condition. Thus it will not be mistaken for a face when searching an image.

2.6.3 An Algorithm for Locating the Face

This is a simplistic implementation of the process described above and although it should be possible to improve the implementation, it performs adequately for our purposes.

We search the image for that region which minimises $d(\mathbf{X})$. We loop over all possible scales, resizing the image to the appropriate scale at each stage. We then crop successive regions from the image and calculate $d(\mathbf{X})$ for each region. We eventually return that region with the minimum value.

The actual code is listed in appendix A.1.4.

```

// We assume = eigenfaces, Ave = the average
//  $\lambda$  = the eigenvalues
// We assume 10 eigenfaces are being used
Begin
  Let  $\rho = \frac{\sum_{i=1}^M \lambda[i]}{N-10}$  // calculate  $\rho^4$ 
  I = image to process
  loop through all scales
    NewIm = Resize I to suitable scale.
    loop across NewIm
      Test = section of interest from NewIm
       $\mathbf{y} = U^T(\text{Test} - \mathbf{A})$ 
       $\text{ProbF} = \frac{\sum_{n=1}^M y_n^2}{\lambda_n}$ 
       $\text{ProbO} = \frac{\|\text{Test}\|^2 - \|\mathbf{y}\|^2}{\rho}$ 
       $d = \text{ProbO} + \text{ProbF}$ 
      // Actual probability =  $\frac{\exp(-\frac{d}{2})}{\text{constant}}$ 
      // Thus, for maximum, we need d
      // to be minimum
      if d is minimum then bestface = Test
    end loop across NewIm
  end scale loop
  Best possible face now in bestface
End

```

*Since we only have M non-zero eigenvalues, we assumed that $\lambda_i = 0 \forall i > M$. We could also try some sort of extrapolation technique to estimate the remaining eigenvalues, but since we expect these to be very small in any case, it is unlikely that this will gain us any major benefits in performance.

2.6.4 Results

Consider the image shown in figure 2.24. The face is clearly off centre and it is necessary to shift it to a more central position.

The result of using the algorithm described above is shown in fig. 2.25. This image is better centred, and due to the fact that we do a multiscale search, it has also been enlarged to fit the frame.

Figure 2.24: Image before location algorithm



Figure 2.25: Image after location



A better illustration of the difference between figures 2.24 and 2.25 is obtained by overlaying the outlines of the two images, which are found by applying a standard edge detection technique. Thus in figure 2.26 we see clearly how the face has shifted.

This technique is not adversely affected by variations in the background since the background makes a reasonably uniform contribution to the error. And even if we do not find the exact centre of the face due to the effects of the background, we should still be sufficiently close for the error to be minor (i.e. only 1 or 2 pixels out). Thus we can successfully locate faces even in images where the background is not controlled. Having centred the face, we can then filter out the background.

A final point is that this algorithm finds that region of an image that best corresponds to the training set. If the training set contains images which

Figure 2.26: Outline comparison between fig. 2.24 and fig. 2.25



vary widely in position and scale, this algorithm will perform poorly. However, if the training set is well normalised, the algorithm performs well. Thus we need some well normalised training set before we can normalise our images. However, since we need only to describe the basic characteristics of the face, we if have a small, well normalised set of faces, we can use the eigenfaces of these to normalise a much larger database, which would then serve as our training set for the eigenfaces used represent faces.

In implementing this algorithm, we have not encountered any false detections. However one should keep in mind the limited nature of our training set. On the other hand, extensive tests performed by Moghadden and Pentland [6] indicate that, , under similar experimental conditions, the maximum likelihood estimator is at least 10% better than the DFFS measure and 20-30% better than the correlation based technique.

2.7 Updating the Eigenfaces

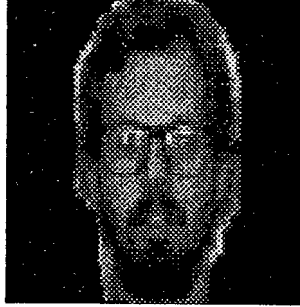
2.7.1 Problem Description

The range of faces which can be described by our eigenfaces is determined by the training set. Only if our training set is sufficiently representative will we be able to represent any face well. If a particular image is poorly represented, we will have to modify our training set and thus our eigenfaces to describe this new image. Naturally, we would like to do this as efficiently as possible.

In figure 2.27, we show the original image of a face that is poorly represented by our set of eigenfaces.

The reconstruction is shown in figure 2.28. This is clearly unacceptable and we need to modify our set of eigenfaces to describe this new face. Figure 2.29 shows the error in using this representation. Note that the information

Figure 2.27: Original image, poorly described by training set



needed to recognise the individual is lost in this reconstruction.

Figure 2.28: Reconstruction of image poorly represented by training set

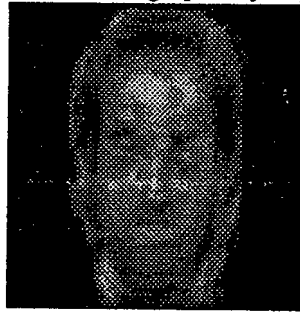
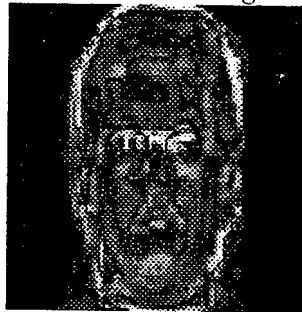


Figure 2.29: Error between fig. 2.27 and 2.28



We obviously need to recalculate the eigenfaces. This could easily be done by simply adding the new face to our training set and redoing the whole calculation, but that would be inefficient. Instead, we now describe a method whereby we update the eigenfaces, rather than performing the much more expensive full recalculation.

2.7.2 Perturbation Analysis

We want to modify the eigenfaces so that the new image is also well represented, but without affecting the description of the faces already in the training set. This implies that the majority of the eigenfaces will not change significantly, a situation that can be described by perturbation theory.

Consider the perturbed eigenvalue problem

$$(C + \delta C)(\mathbf{u}_l + \delta \mathbf{u}_l) = (\lambda_l + \delta \lambda_l)(\mathbf{u}_l + \delta \mathbf{u}_l) \quad (2.22)$$

where δC is the known change in C , $\delta \lambda_l$ the unknown change in λ_l and $\delta \mathbf{u}_l$ the unknown change in \mathbf{u}_l .

Expanding (2.22) and keeping the first order terms leads to

$$(C - \lambda_l I) \delta \mathbf{u}_l = (\delta C + \delta \lambda_l I) \mathbf{u}_l. \quad (2.23)$$

For a solution to exist, the right hand side must be an element of the column space of $C - \lambda_l I$. Thus, by the Fredholm alternative, it is orthogonal to the null-space of $C - \lambda_l I$. But the null-space of $C - \lambda_l I$ is the eigenvector \mathbf{u}_l . Since the eigenvectors are orthogonal, it follows that

$$\mathbf{u}_l^T (-\delta C + \delta \lambda_l I) \mathbf{u}_l = 0$$

or

$$\delta \lambda_l = \mathbf{u}_l^T \delta C \mathbf{u}_l. \quad (2.24)$$

This is an expression for the change in the eigenvalue. Substituting back into (2.23), one can use the pseudoinverse to calculate the change in the eigenvectors. The only problem is that C is a $N \times N$ matrix, where N is very large.

2.7.3 Reduction

We have side-stepped a similar problem earlier, when dealing with the the original derivation of eigenfaces in section 2.3.2. A similar line of reasoning can be followed here.

First, note that our assumption that P eigenfaces describe every face in the original training set accurately is equivalent to assuming that

$$C \approx \lambda_1 \mathbf{u}_1 \mathbf{u}_1^T + \lambda_2 \mathbf{u}_2 \mathbf{u}_2^T + \cdots + \lambda_P \mathbf{u}_P \mathbf{u}_P^T. \quad (2.25)$$

Additionally, we assume that

$$\delta C \approx \varphi \varphi^T \quad (2.26)$$

for some unknown φ which depends partly on the new face.

Assuming that the new eigenvectors are some combination of the old eigenvectors and of φ , we write

$$\tilde{\mathbf{u}}_l = \sum_{j=1}^P v_{jl} \sqrt{\lambda_l} \mathbf{u}_j + v_{(P+1)l} \varphi$$

where $\tilde{\mathbf{u}}_l = \mathbf{u}_l + \delta \mathbf{u}_l$ is the l 'th eigenvector of the new system.

We substitute this into our perturbed eigenvector equation (2.22) and use (2.25) and (2.26) to obtain

$$\begin{aligned} & (\lambda_l \mathbf{u}_l \mathbf{u}_l^T + \cdots + \lambda_P \mathbf{u}_P \mathbf{u}_P^T + \varphi \varphi^T) (v_{1l} \sqrt{\lambda_l} \mathbf{u}_1 + \cdots + v_{(P+1)l} \varphi) \\ & = (\lambda_l + \delta \lambda_l) (v_{1l} \sqrt{\lambda_l} \mathbf{u}_1 + \cdots + v_{(P+1)l} \varphi). \end{aligned}$$

Using a similar argument to section 2.3.2, it follows that $\mathbf{v}_l = [v_{1l} \cdots v_{(P+1)l}]$ must satisfy the eigenvector equation

$$(\lambda_l + \delta \lambda_l) \mathbf{v}_l = K^{new} \mathbf{v}_l$$

where

$$K^{new} = \begin{bmatrix} \lambda_1 & \cdots & 0 & \sqrt{\lambda_1} \mathbf{u}_1^T \varphi \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \lambda_P & \sqrt{\lambda_P} \mathbf{u}_P^T \varphi \\ \sqrt{\lambda_1} \mathbf{u}_1^T \varphi & \cdots & \sqrt{\lambda_P} \mathbf{u}_P^T \varphi & \varphi^T \varphi \end{bmatrix}. \quad (2.27)$$

This matrix is only $(P+1) \times (P+1)$, which represents a considerable reduction in the scale of the problem. Also it has an extremely sparse structure, which means that the new eigenvectors and eigenvalues can be calculated very efficiently. But first, we discuss how we find an expression for ϕ .

2.7.4 Finding φ

To find φ , we start by considering a new training set which consists of our original training set as well the new, badly represented face, a total of $M+1$ members.

Let us define the new average face as

$$\tilde{\mathbf{A}} = \sum_{n=1}^{M+1} \mathbf{I}_n \quad (2.28)$$

and the new normalised deviation of the n 'th image from the average as

$$\tilde{\mathbf{X}}_n = \frac{\mathbf{I}_n - \tilde{\mathbf{A}}}{\|\mathbf{I}_n - \tilde{\mathbf{A}}\|}. \quad (2.29)$$

The eigenfaces of this new training set are the eigenvectors of the new covariance matrix

$$C + \delta C = \frac{1}{M+1} \sum_{n=1}^M \tilde{\mathbf{X}}_n \tilde{\mathbf{X}}_n^T + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T.$$

Since

$$\begin{aligned} \tilde{\mathbf{X}}_n &= \frac{\mathbf{I}_n - \tilde{\mathbf{A}}}{\|\mathbf{I}_n - \tilde{\mathbf{A}}\|} \\ &= \frac{\mathbf{X}_n \|\mathbf{I}_n - \mathbf{A}\| + \mathbf{A} - \tilde{\mathbf{A}}}{\|\mathbf{I}_n - \mathbf{A}\|} \end{aligned}$$

it follows that

$$\begin{aligned} C + \delta C &= \frac{M}{M+1} \left[\frac{1}{M} \sum_{n=1}^M \left(\frac{\mathbf{X}_n \|\mathbf{I}_n - \mathbf{A}\| + \mathbf{A} - \tilde{\mathbf{A}}}{\|\mathbf{I}_n - \tilde{\mathbf{A}}\|} \right) (\dots)^T \right] \\ &\quad + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T. \end{aligned} \quad (2.30)$$

Using (2.1) and (2.28), it is obvious that

$$\mathbf{A} = \tilde{\mathbf{A}} - \frac{1}{M} (\mathbf{I}_{M+1} - \tilde{\mathbf{A}})$$

and

$$\begin{aligned} \|\mathbf{I}_n - \mathbf{A}\| &= \left\| \mathbf{I}_n - \tilde{\mathbf{A}} + \frac{1}{M} (\mathbf{I}_{M+1} - \tilde{\mathbf{A}}) \right\| \\ &= \left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\| \end{aligned}$$

where $\mathbf{Q}_s = \mathbf{I}_s - \tilde{\mathbf{A}}$.

Substituting this into (2.30), it follows that

$$\begin{aligned} C + \delta C &= \frac{1}{M+1} \sum_{n=1}^M \left(\frac{\mathbf{X}_n \left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\| - \frac{1}{M} \mathbf{Q}_{M+1}}{\|\mathbf{Q}_n\|} \right) (\dots)^T \\ &\quad + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T. \end{aligned}$$

Multiplying out the various factors, we can rewrite this as

$$\begin{aligned} C + \delta C &= \frac{1}{M+1} \sum_{n=1}^M \frac{\left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\|^2 \mathbf{X}_n \mathbf{X}_n^T}{\|\mathbf{Q}_n\|^2} \\ &\quad - \frac{1}{M+1} \sum_{n=1}^M \frac{\left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\| (\mathbf{Q}_{M+1} \mathbf{X}_n^T + \mathbf{X}_n \mathbf{Q}_{M+1}^T)}{M \|\mathbf{Q}_n\|^2} \\ &\quad + \frac{1}{M+1} \sum_{n=1}^M \frac{\mathbf{Q}_{M+1} \mathbf{Q}_{M+1}^T}{M^2 \|\mathbf{Q}_n\|^2} \\ &\quad + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T \\ &= \frac{1}{M+1} \sum_{n=1}^M \frac{1}{\|\mathbf{Q}_n\|^2} \left(\left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\|^2 \right) \mathbf{X}_n \mathbf{X}_n^T \\ &\quad - \frac{1}{M+1} \sum_{n=1}^M (\mathbf{X}_n \tilde{\mathbf{X}}_{M+1}^T + \tilde{\mathbf{X}}_{M+1} \mathbf{X}_n^T) \left(\frac{\|\mathbf{Q}_{M+1}\| \left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\|}{M \|\mathbf{Q}_n\|^2} \right) \\ &\quad + \frac{1}{M+1} \sum_{n=1}^M (\tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T) \left(1 + \frac{\|\mathbf{Q}_{M+1}\|^2}{M^2 \|\mathbf{Q}_n\|^2} \right) \end{aligned} \quad (2.31)$$

where we use $\mathbf{Q}_{M+1} = \|\mathbf{Q}_{M+1}\| \tilde{\mathbf{X}}_{M+1}$.

To simplify this expression, let us consider $\|\mathbf{Q}_n\|$ in more depth.

We know $\|\mathbf{Q}_n\|^2 \approx \|\mathbf{y}\|^2 = \sum_{i=1}^N y_i^2$. In section 2.6, we assumed that the y_i 's were normally distributed with a mean of 0 and variance λ_i . Furthermore, we know from the definition of the eigenfaces that the y_i 's are independent (see section 2.3.1).

Since $y_i \sim N(0, \lambda_i)$,⁵ it follows that $\mathbf{E}(y_i^2) = \mathbf{Var}(y_i) = \lambda_i$ and it can be shown that $\mathbf{Var}(y_i^2) = 2\lambda_i^2$ (See for example Rice [25]).

So it follows that

$$\mathbf{E}(\|\mathbf{Q}_n\|^2) = \mathbf{E}\left(\sum_{i=1}^N y_i^2\right) = \sum_{i=1}^N \lambda_i$$

and since the y_i 's are independent, we know that

$$\mathbf{Var}(\|\mathbf{Q}_n\|^2) = \mathbf{Var}\left(\sum_{i=1}^N y_i^2\right) = 2 \sum_{i=1}^N \lambda_i^2.$$

The standard deviation is given as

$$\sigma_{Q^2} = \sqrt{\mathbf{Var}(\|\mathbf{Q}_n\|^2)} = \sqrt{2} \left(\sqrt{\sum_{i=1}^N \lambda_i^2} \right).$$

Chebyshev's theorem states that if we have a random variable U with mean μ_X and a standard deviation of σ_X , then the probability that $\mu_X - k\sigma_X < U < \mu_X + k\sigma_X$ is at least $(1 - \frac{1}{k^2})$.⁶ So we can write

$$\|\mathbf{Q}_n\|^2 = \sum_{i=1}^N \lambda_i + k_n \sigma_{Q^2} \quad (2.32)$$

with $|k|$ small.

In this case, we can assume that

$$\sqrt{\sum_{i=1}^N \lambda_i^2} \ll \sum_{i=1}^N \lambda_i$$

⁵We use $N(\mu, \sigma^2)$ to indicate a normal distribution with mean μ and variance σ^2 .

⁶This implies that at least 75% of the time, a random variable must lie within 2 standard deviations of the mean. This is also a lower bound on the spread of the data. In many cases, the actual values are much higher.

since $(\sum_{i=1}^N \lambda_i)^2 - \sum_{i=1}^N \lambda_i^2 = \sum_{i=1}^N \sum_{j \neq i} \lambda_i \lambda_j$ and several eigenvalues are significant. We typically find that $(\sum_{i=1}^N \lambda_i)^2 \approx 10 \sum_{i=1}^N \lambda_i^2$. Consequently, $\sigma_{Q^2} \ll \sum_{i=1}^N \lambda_i$.

Then, if we write

$$\begin{aligned} \frac{\|\mathbf{Q}_n\|^2}{\|\mathbf{Q}_m\|^2} &= \frac{\sum_{i=1}^N \lambda_i + k_m \sigma_{Q^2}}{\sum_{i=1}^N \lambda_i + k_n \sigma_{Q^2}} \\ &= 1 + \frac{(k_m - k_n) \sigma_{Q^2}}{\sum_{i=1}^N \lambda_i + k_n \sigma_{Q^2}} \end{aligned}$$

it follows that $\frac{\|\mathbf{Q}_n\|^2}{\|\mathbf{Q}_m\|^2} \approx 1$ for any n and m and in particular

$$\frac{\|\mathbf{Q}_{M+1}\|}{\|\mathbf{Q}_n\|} \approx 1. \quad (2.33)$$

Furthermore, we know that

$$\begin{aligned} \left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\|^2 &= \mathbf{Q}_n^T \mathbf{Q}_n + \frac{2}{M} \mathbf{Q}_n^T \mathbf{Q}_{M+1} + \frac{1}{M^2} \mathbf{Q}_{M+1}^T \mathbf{Q}_{M+1} \\ &= \|\mathbf{Q}_n\|^2 + \frac{2}{M} \|\mathbf{Q}_n\| \|\mathbf{Q}_{M+1}\| \cos \theta_n + \frac{1}{M^2} \|\mathbf{Q}_{M+1}\|^2 \\ &\quad + \frac{1}{M^2} \|\mathbf{Q}_{M+1}\|^2 \end{aligned} \quad (2.34)$$

where θ_n is the angle between \mathbf{Q}_n and \mathbf{Q}_{M+1} , so

$$\begin{aligned} \left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\| &= \sqrt{\|\mathbf{Q}_n\|^2 \left(1 + \frac{2 \|\mathbf{Q}_n\| \|\mathbf{Q}_{M+1}\| \cos \theta_n}{M \|\mathbf{Q}_n\|^2} + \frac{\|\mathbf{Q}_{M+1}\|^2}{M^2 \|\mathbf{Q}_n\|^2} \right)} \\ &\approx \|\mathbf{Q}_n\| \sqrt{1 + \frac{2 \cos \theta_n}{M} + \frac{1}{M^2}} \end{aligned}$$

using (2.33).

Since $\left(\frac{2 \cos \theta_n}{M} + \frac{1}{M^2} \right) < 1$, we can use a Taylor series expansion to see that

$$\frac{\left\| \mathbf{Q}_n + \frac{1}{M} \mathbf{Q}_{M+1} \right\|}{M} = \frac{\|\mathbf{Q}_n\|}{M} \left(1 + O\left(\frac{1}{M}\right) \right). \quad (2.35)$$

We are only interested in a first order approximation to δC . So using (2.34) and (2.35) and ignoring the higher order terms, we can rewrite (2.31) as

$$\begin{aligned}
C + \delta C &\approx \frac{1}{M+1} \sum_{n=1}^M \mathbf{X}_n \mathbf{X}_n^T \left(1 - \frac{2\mathbf{Q}_n^T \mathbf{Q}_{M+1}}{M \|\mathbf{Q}_n\|^2} \right) \\
&\quad - \frac{1}{M+1} \sum_{n=1}^M \frac{\|\mathbf{Q}_{M+1}\|}{M \|\mathbf{Q}_n\|} (\mathbf{X}_n \tilde{\mathbf{X}}_{M+1}^T + \tilde{\mathbf{X}}_{M+1} \mathbf{X}_n^T) \\
&\quad + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T \\
&= \frac{M}{M+1} C - \frac{1}{M(M+1)} \sum_{n=1}^M \mathbf{X}_n \mathbf{X}_n^T \left(\frac{2\|\mathbf{Q}_n\| \|\mathbf{Q}_{M+1}\| \cos \theta_n}{\|\mathbf{Q}_n\|^2} \right) \\
&\quad - \frac{\|\mathbf{Q}_{M+1}\|}{M(M+1)} \left[\left(\sum_{n=1}^M \frac{\mathbf{X}_n}{\|\mathbf{Q}_n\|} \right) \tilde{\mathbf{X}}_{M+1}^T + \tilde{\mathbf{X}}_{M+1} \left(\sum_{n=1}^M \frac{\mathbf{X}_n}{\|\mathbf{Q}_n\|} \right)^T \right] \\
&\quad + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T \tag{2.36}
\end{aligned}$$

where θ_n is the angle between \mathbf{Q}_n and \mathbf{Q}_{M+1} .

Since $\cos \theta_n \leq 1$ and $\frac{\|\mathbf{Q}_{M+1}\|}{\|\mathbf{Q}_n\|} \approx 1$, the term involving $\cos \theta_n$ is clearly a second order term and can be ignored. We remember that $\sum_{n=1}^M \mathbf{X}_n = 0$ and from (2.32) it follows that $\|\mathbf{Q}_n\|$ is almost constant over n . So $\sum_{n=1}^M \frac{\mathbf{X}_n}{\|\mathbf{Q}_n\|}$ is small and can be ignored. So we can rewrite (2.36) as

$$\begin{aligned}
C + \delta C &= \frac{M}{M+1} C + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T \\
&= C - \frac{1}{M+1} + \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T.
\end{aligned}$$

However, the second term just introduces a scaling on C . Numerical experiments indicate that this has very little effect on the eigenvectors, so we ignore this term and use

$$\delta C \approx \frac{1}{M+1} \tilde{\mathbf{X}}_{M+1} \tilde{\mathbf{X}}_{M+1}^T$$

so

$$\varphi = \frac{1}{\sqrt{M+1}} \tilde{\mathbf{X}}_{M+1}. \tag{2.37}$$

We can then use this to calculate the new eigenvectors and eigenvalues.

2.7.5 Efficient Recalculation of the Eigenfaces

The sparse structure allows us to solve the new eigenvector problem very efficiently. We will briefly discuss some of the details here.

Equation (2.37) allows us to approximate δC . Equation (2.24) relates the change in the eigenvalues to δC . So

$$\delta\lambda_j \approx u_j^T (\delta C) u_j = \frac{1}{M+1} (u_j^T \tilde{\mathbf{X}}_{M+1})^2.$$

This gives us a good starting approximation to the new eigenvalues for $1 \leq j \leq P$. So if we use the standard inverse power method, as described in Golub and Van Loan [22] or Trefethen and Bau [23], we will start close to the correct solution, ensuring rapid convergence.

We want to calculate the eigenvectors of K^{new} (see equation 2.27). Since it is almost diagonal, its eigenvectors have similar directions to the elementary unit vectors. Thus we have good initial approximations for both the eigenvalues and eigenvectors.

Moreover, because of its special structure, it is easy to show that the LU decomposition of K^{new} is

$$L = \begin{bmatrix} 1 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 \\ \frac{1}{\lambda_1} \sqrt{\frac{\lambda_1}{M+1}} \mathbf{u}_1^T \tilde{\mathbf{X}}_{M+1} & \cdots & \frac{1}{\lambda_P} \sqrt{\frac{\lambda_P}{M+1}} \mathbf{u}_P^T \tilde{\mathbf{X}}_{M+1} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} \lambda_1 & \cdots & 0 & \sqrt{\frac{\lambda_1}{M+1}} \mathbf{u}_1^T \tilde{\mathbf{X}}_{M+1} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & \lambda_P & \sqrt{\frac{\lambda_P}{M+1}} \mathbf{u}_P^T \tilde{\mathbf{X}}_{M+1} \\ 0 & \cdots & 0 & Z(U) \end{bmatrix}$$

where

$$Z(U) = \frac{1}{M+1} \left(1 - (\mathbf{u}_1^T \tilde{\mathbf{X}}_{M+1})^2 - \cdots - (\mathbf{u}_P^T \tilde{\mathbf{X}}_{M+1})^2 \right).$$

Both these matrixes are sparse and operations involving these can be done efficiently. If we were to use the inverse power method, we would need the LU

decomposition of the matrix $K^{new} - \hat{\lambda}_l I$, where $\hat{\lambda}_l$ is our approximation to the l 'th eigenvalue. These matrixes are equally sparse and we can calculate their LU decompositions just as easily.

One potential problem occurs if $\lambda_k + \delta\lambda_k \approx \lambda_{k+1} + \delta\lambda_{k+1}$. If this is the case, both starting points could converge to the same eigenvalue. However, we know that the eigenvectors have to be orthogonal. So, if we suspect that the two eigenvalues are too close to each other, we can calculate one eigenvalue and its associated eigenvector. Then, using Gram-Schmidt orthogonalisation, we can produce a starting vector orthogonal to this eigenvector. We are guaranteed to converge to a different eigenvalue.⁷

However, the above approximation of the changes in the eigenvalues only gives us P eigenvalues and eigenvectors. We need to find all $P + 1$ eigenvectors.

To find the last eigenvalue, we note that, from our definition of K^{new} , the eigenvalues of $C + \delta C$ and K^{new} are the same. Thus

$$\text{Tr}(C + \delta C) = \text{Tr} K^{new}$$

which implies that

$$(\lambda_1 + \delta\lambda_1) + \dots + (\lambda_P + \delta\lambda_P) + \lambda_{P+1} = \lambda_1 + \dots + \lambda_P + \frac{1}{M+1}$$

so

$$\frac{1}{M+1} - \delta\lambda_1 - \dots - \delta\lambda_P = \lambda_{P+1}.$$

This allows us to calculate the last eigenvalue, from which we can easily find the last eigenvector, which is the final eigenface.

2.7.6 An Algorithm for Calculating the New Eigenfaces

This implements the basic recalculation. We do not however, implement the details of calculating the eigenvectors of K^{new} as such algorithms are well documented elsewhere.

The actual code is listed as *eigupdate.m* in appendix A.1.5.

```
// Assume U = old eigenfaces
```

⁷Unless we actually have a repeated eigenvalue. In this case, we will converge to the same eigenvalue, but get the second associated eigenvector.

```

// A = the average face
// λ = the eigenvalues
// M = number of images in the
//     original training set.
// We keep 40 eigenfaces.
// We assume that we have already loaded an image
// into I and it is sufficiently badly represented
// to require the recalculation of the eigenfaces

```

```
Begin
```

$$A_{\text{new}} = \frac{(MA+I)}{M+1}$$

$$\varphi = \frac{I - A_{\text{new}}}{\sqrt{M+1}} \quad // \delta C \approx \varphi \varphi^T$$

$$L_{ii} = \lambda_i \text{ for } i = 1, \dots, 40.$$

$$Q_i = \frac{U_i}{\sqrt{\lambda_i}} \text{ for } i = 1, \dots, 40$$

$$L_{41i} = L_{i41} = \varphi^T Q_i \text{ for } i = 1, \dots, 40$$

$$L_{4141} = \varphi^T \varphi$$

Calculate $Lv_n = \gamma_n v_n$ for all n

New eigenfaces : $U_{\text{new}_i} = v_n^T Q$

Normalise: $U_{\text{new}_i} = \frac{U_{\text{new}_i}}{\|U_{\text{new}_i}\|}$

```
End
```

2.7.7 Results using the New Eigenfaces

We wish to compare the updated eigenfaces with those obtained by the alternative method of adding the new face to the training set and recalculating the eigenfaces. Since it was poorly represented before, we return to the image shown in figure 2.27.

The reconstruction of our new face using the fully recalculated set of eigenfaces is shown in figure 2.30. As expected, the representation is now acceptable and the error in the representation (figure 2.31) is now much more random.

Figure 2.30: Representation of fig. 2.27 using fully recalculated eigenfaces.

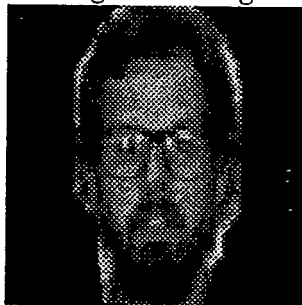
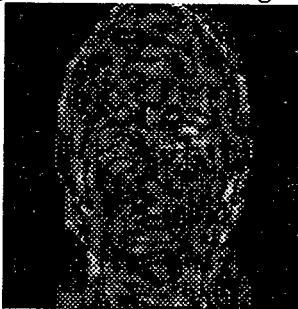


Figure 2.31: Error of fig. 2.30



We compare this with the reconstruction using the updated eigenfaces shown in figure 2.32.

Figure 2.32: Reconstruction of fig. 2.27, using updated eigenfaces.

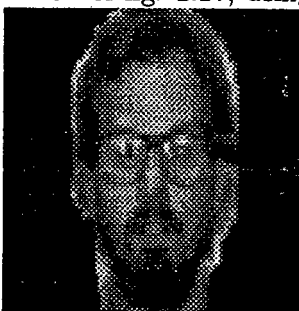
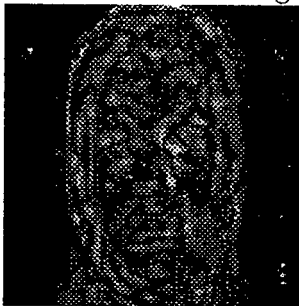


Figure 2.33: Error from fig. 2.32

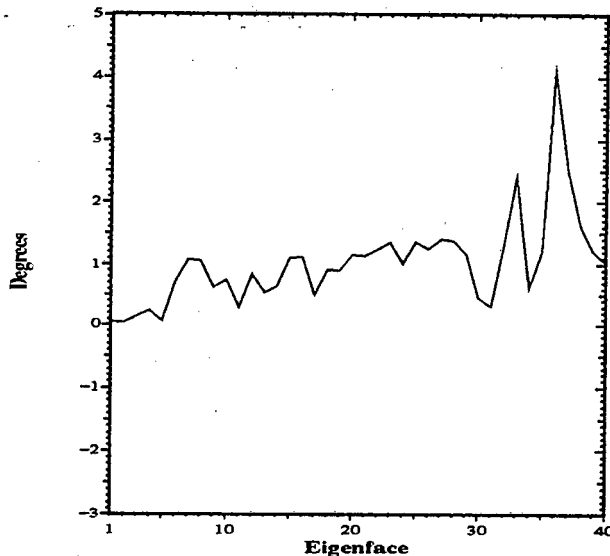


Again, we achieve a very good representation of the image. Also, the error in our reconstruction is now equally random (figure 2.33), indicating that the updated eigenfaces gives a good representation of the new face.

To compare how closely the two set of eigenfaces correspond, we use the angle between each eigenface from the recalculated set and the corresponding eigenface from the updated set. In this case, we found that the average angle between two corresponding eigenfaces was about 1° , which indicates that the

two sets of eigenfaces are very similar. A graph of all the angles is shown in figure 2.34.

Figure 2.34: Graph of the angles between the updated and recalculated eigenfaces



We will consider how the new set of eigenfaces describe those faces already in the database in the next section.

2.7.8 Converting the Database

Once we have calculated the new eigenfaces (\tilde{u}_i), then we need to convert images represented in terms of the old eigenfaces to the new eigenfaces. For reasons of efficiency, we would prefer to update our old representations, rather than a complete recalculation of the coefficients.

Recall, from our earlier comments on reconstruction in section 2.4, that the representation of a new image in terms of our new basis will be

$$\tilde{y}_j = \tilde{U}^T (\mathbf{J} - \tilde{\mathbf{A}})$$

and that our reconstruction of an image in terms of the old database is

$$\mathbf{J} \approx \mathbf{U} \mathbf{y}_J + \mathbf{A}.$$

By combining these two definitions, then, given the old representation \mathbf{y}_n , our new representation $\tilde{\mathbf{y}}_n$ will be

$$\tilde{\mathbf{y}}_n = \tilde{U}^T(U\mathbf{y}_n + \mathbf{A} - \tilde{\mathbf{A}}).$$

Since \mathbf{A} and $\tilde{\mathbf{A}}$ are constant, we can write

$$\tilde{\mathbf{y}}_n = \tilde{U}^T U \mathbf{y}_n + \mathbf{B} \quad (2.38)$$

where

$$\mathbf{B} = \tilde{U}^T(\mathbf{A} - \tilde{\mathbf{A}}).$$

Since $\tilde{U}^T U$ is a $P \times P$ matrix, where P is again the number of eigenfaces we retain, this calculation can be done very efficiently.

Since this is easy to implement, we refer the reader to *convert.m* in appendix A.1.5 for the details.

Finally, note that the conversion (2.38) is not used for the new face, since the new face is not well represented by the original eigenfaces.

An important point is that since this method can only work well for images well described by the original set of eigenfaces, we would not use this for the new face (since it was badly represented) or any new image which we would subsequently encounter.

2.7.9 Results of Converting the Database

To illustrate this conversion, we consider figure 2.6, which was well described by our original eigenfaces. In figure 2.35, we show the representation using the full recalculation. This neatly illustrates that our updated eigenfaces are indeed capable of describing the faces in the original database well.

For comparison, the result of using equation (2.38) to convert the original representation to the new set of eigenfaces is shown in figure 2.36.

These two images are very similar, indicating that the conversion gives a good approximation to the true representation. It is of particular interest to look at the difference between these two images, which is shown in figure 2.37. Somewhat surprisingly, this resembles the new face we added to the training set. However, the difference is small, with a maximum value of approximately 10^{-3} , which means it has no effect on the appearance.⁸

⁸Since we are using a fixed greyscale range range of 0 to 256, a difference of 10^{-3} can at most cause a difference of 1 greyscale level (due to the round-off error). However, this hardly constitutes a major difference between the images.

Figure 2.35: Reconstruction of figure 2.6 using correct representation

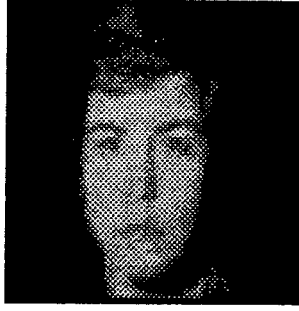
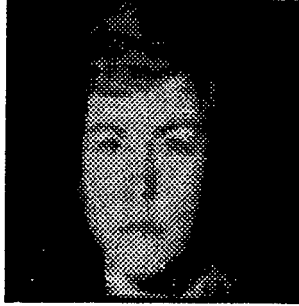


Figure 2.36: Reconstruction of figure 2.6 using converted representation



The resemblance to the new face can be explained by remembering that our original set of eigenfaces had little information in the “direction” of the new face. By calculating the new representation from the original image, we will now keep any the contribution of this new direction, information which was lost in the projection from our old eigenfaces to the new set. But since this new direction was unimportant in the images in our original database, the error is insignificant.

Figure 2.37: Error between figure 2.35 and figure 2.36

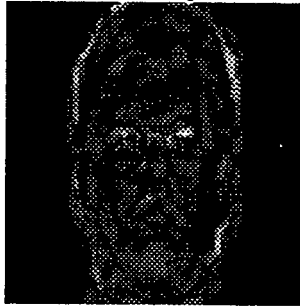
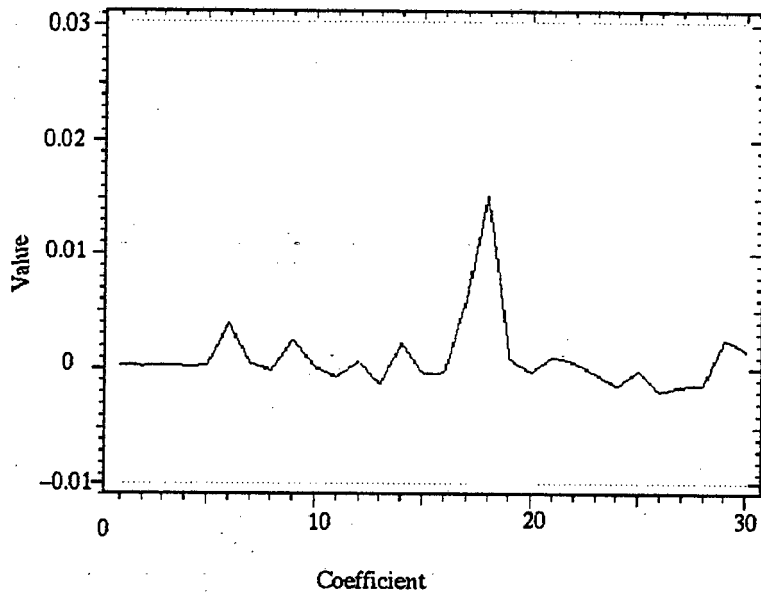


Figure 2.38 plots the relative differences between coefficients of the two representations. As we can see, the largest error is of the order of 1%. This is very small, which indicates again the the error is insignificant.

Figure 2.38: Difference between representations used in figures 2.35 and 2.36



2.8 Testing the System

2.8.1 Test Description and Results

We are interested in this technique for image recognition. Although we discussed a simple recognition system in section 2.5, we did not test it.

To test the system, we used a database of 100 images as our training set. Our training set consisted of images from two databases, one from MIT and another from the Olivetti Research Laboratory. We then considered two test cases. The first contained 60 new images of people in the training set, while the second contained 10 images of people not in the training set.

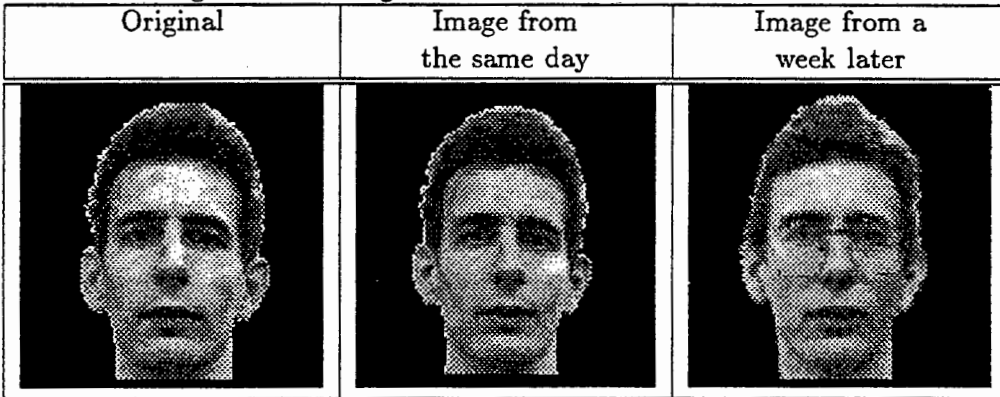
Using the simplistic model described, we found that the faces not in the database were always successfully rejected.

For faces in the database, 55 were correctly identified. 4 were incorrectly identified as not being in the database while the remaining image was incorrectly identified.

This corresponds to a recognition rate of 92%. For a database of our size, this may appear a little low, but the images in the test set contain considerably more variation in terms of light intensity than those in the training set, illustrating the need for some good solution to this problem.

This test case is not particularly realistic. For a slightly more realistic test, we use images of 20 different individuals from a database collected by the University of Manchester. This database includes 4 to 10 images of the same person taken over a period of several weeks. For our purposes, we use two images of the same person taken on the same day and one from a week later (see figure 2.39).

Figure 2.39: Images from the manchester database



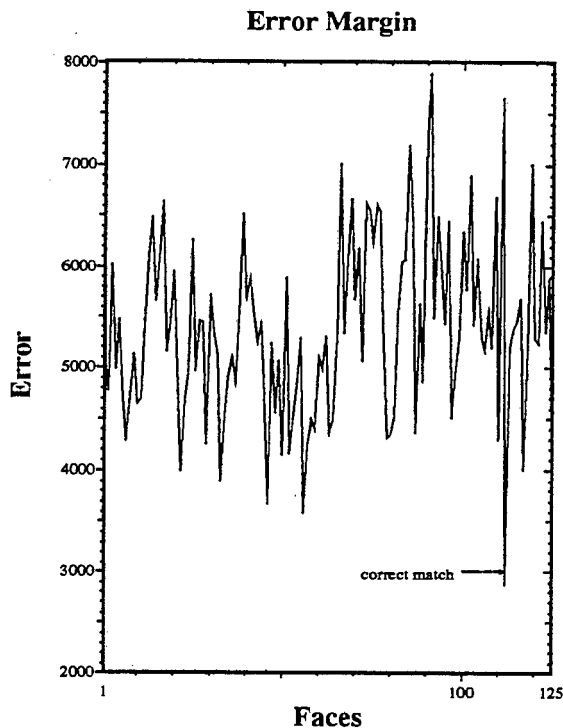
For the purposes of the test, we add the images of 15 different individuals to our training set. For our database, we then use 1 image of each individual. We then test using the other images taken on the same day and those taken a week later.

We are able to correctly identify 19 of the images taken on the same day. However, of the images taken a week later, we are only able to identify 17. Our training set is not able to describe all the variations in the images. This illustrates the need for a good training set.

Ideally, we need some measure of the confidence in a match. If we plot the actual margins by which we accept that matches, we good decide on a confidence measure based on the how many other images are also close matches. If we consider the figure 2.40, which shows the levels for correctly identifying the second image in figure 2.39, we can clearly see that there are no comparable match, so we would have a great deal of confidence in that match. However, figure 2.41, which is the case where we incorrectly identified the individual, is not a match we would have a great deal of confidence in since the error with the face chosen as the best match is not particularly small relative to the other images.

Of course, our test set cannot be considered representative. Thus it is difficult to draw any firm conclusions about how this system will perform in practice.

Figure 2.40: Error margins for a correct match



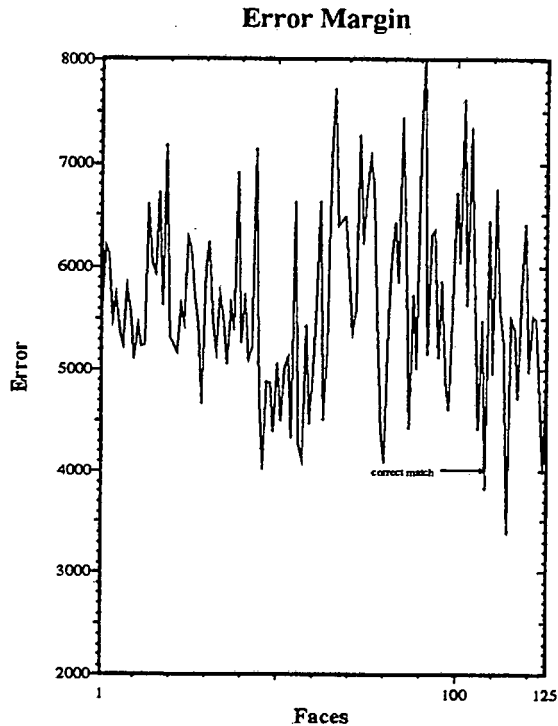
2.8.2 Comments

This chapter has covered the calculation of the eigenfaces and used them to represent images. This new compact representation greatly simplifies recognition and segmentation.

The technique, though, is based on two assumptions. First, we assume that the images in the database are well described by the eigenfaces. This implies that we need a training set that is representative of the images in our database. In section 2.7, we saw that we can modify the eigenfaces to account for poorly represented images, but in most practical applications, we would not be in a position to change the eigenfaces. However, the ability to quickly recalculate the eigenfaces should be very useful in determining a representative training set.

We also assume that the images are well normalised. Since variations in the position of the face will obviously alter the representation, some preprocessing is required to centre the face in the image and to normalise other sources

Figure 2.41: Error margins for incorrect match



of variation, such as the variation in lighting conditions. While we examined a technique to locate the face, the other problems still need attention.

In our test case, our simplistic recognition rule performs well. But that is no reason to suppose that this choice is a particularly good one. The differences between the two representations of the same person were not equally distributed amongst the different coefficients (see figure 2.21). In addition, while the first few coefficients can differ greatly between two different faces, since they describe such general features, we do encounter cases where the first few coefficients are similar, but the faces are quite different. This suggests that we need to weigh the different coefficients in some manner. We have not experimented with this here, since with a small database and a limited test set, it is impossible to draw any conclusions as to how generally applicable any improved system we design will be. We will look into the this weighting in more depth in section 3.5.

In the literature, recognition rates of over 90% have been reported for larger test cases (see the FERET database report [14]), but it is unclear how closely

the conditions of this test correspond to those that will be encountered in practice.

To thoroughly test the system, we will need a large database, but dealing with a large database will introduce a variety of new problems. To examine some of these, we turned to a situation where we can easily acquire a large database, namely Optical Character Recognition.

Chapter 3

Optical Character Recognition

3.1 Introduction

Although the eigenface technique was developed for face recognition, the only assumption we made about the images was that they were "sufficiently" similar. So there is no reason to suppose that this technique cannot be extended to other image recognition problems which satisfy this condition. One such problem is that of optical character recognition (OCR). However, the OCR problem is also sufficiently different to make this an interesting test of the technique's flexibility.

It should be pointed out that eigenfaces are an unusual approach to the OCR problem since in most OCR applications one is dealing with binary images, whereas the eigenpicture technique was developed with greyscale images in mind. In this sense there is a major difference between OCR and the face recognition problem, so it is not immediately clear that the eigenpicture technique is applicable. However, the basic premise of the technique - that the images are correlated with each other - still holds, so it is a problem worth investigating.

Furthermore, because of the difficulties associated with obtaining a large collection of facial images, our experiments to date have used a small database of images. Since acquiring a large database of characters is not nearly as problematic, we can investigate how the technique scales to larger databases. One of the specific issues we wish to consider is the possibility of subdividing the database into distinct classes. The advantage of this idea is that the images in each class will be well represented by a very small number of coefficients, which should improve the recognition efficiency.

For the purposes of these experiments, we use the complete upper and lower case alphabet (52 characters in all) from 10 different fonts as our training set. The fonts are standard \TeX fonts, with a size of 10 pt. The training sets are listed in appendix B. The letters were scanned at a resolution of 300 dots per inch and padded to 50×50 pixels. In this case, $N = 2500$. The padding was necessary since the widest letters (W and M) are 45 pixels wide, and it is desirable to have some white space surrounding every character to ensure that we do not lose significant information due to cropping effects. Since we wish to treat all the images as belonging to the same vector space, they need to all be the same size, so all the images were padded.

3.2 Subdividing the Training Set

3.2.1 Problem Description

For the eigenpicture technique to work, the images in the training set must be similar. However, the different letters do not all resemble each other. Letters tend to fall into distinct groups of similar shapes. Thus, while the letters “o” and “e” resemble each other, neither letter resembles “F” or “E”. Applying the eigenpicture technique to the entire set of images will probably be inefficient. So we consider subdividing the dataset, as mentioned in the introduction to this chapter.

If we can split the data into separate classes of similar images, we can apply the technique to each class separately, representing each letter with the “eigenletters” derived for that specific class. Since we can measure how well a letter is represented by a given set of eigenletters, it should be easy to decide which class should be used to describe a new letter.

However, one has to determine the classification of the different letters from the different fonts. One possibility is the “natural” classification, where we use the alphabetical order, i.e. placing all capital O’s in one class, all capital E’s in another and so on. However, this uses an abstract definition of the similarity between the letters and does not exploit mathematical similarities between different groups of letters, such as the resemblance between “o” and “c”.

Thus we need to develop an automatic classification technique which will exploit such similarities.

3.2.2 Classification Background

The classification problem is the subject of a great deal of research in Statistics. There are many algorithms available. Among those we considered were the h-means, k-means and e-means algorithms (see Späth [18]), the UPGMA algorithm and Ward's minimum variance method (see Romesburg [19]) and the pruning, splitting and pairwise nearest neighbour algorithms (see Ger-sho [20]). All of these algorithms have different strengths and weaknesses and also can vary considerably in terms of computational cost. Generally, the better the resulting classification, the more computationally expensive the algorithm.

However, all the algorithms require some relevant measure of how similar two objects are, which determines whether two objects belong to the same class or not. As mentioned in the introduction, we cannot easily decide if two images are similar. So the first problem is to find a suitable measure. Initially we represent all the letters by a single set of eigenletters. We then use the Euclidean distance between representations as our measure of similarity, as in section 2.5.

Although there are many classification algorithms to choose from, the two we consider here are frequently used when dealing with large datasets, such as this one.

3.2.3 Leader Principle with K-means Refinement

The idea of this algorithm is to construct a rough initial classification of the data into a given number of classes and then refine this classification until we arrive at the "best possible" division of the data.

The major problem is that we need to know the number of classes from the outset.

Normally, deciding on a suitable number of classes uses a cost function which involves the number of classes and the spread of data within each class. Usually we look for the minimum number of classes which gives us sufficiently homogeneous classes. However, since we need to calculate the eigenletters of each class, we also need to consider the number of elements of each class, since, if there are too few elements in a class, we will be unable to find a good set of eigenletters for that class. By using a trial and error approach, where we tried several possible choices and compared the separation of the classes in each case, we settled on 40 classes.

As our simple classifier, we use a variation of the leader principle (see Späth [18]). If we want to split the dataset into n classes, we first find n elements

which span the entire dataset and are uniformly scattered throughout the dataset, while also being well separated from one another. We call these the nodes of the dataset and use them as the “seeds” of our classes. Finding the first two nodes is easy as they are merely the two elements which are furthest apart. To find additional nodes, we use the following recursive algorithm. If we have found m nodes, then the $m+1$ 'th node is that element of the dataset with the largest minimum distance from the other nodes (i.e., that element which is the furthest removed from any other node). An element of the dataset is assigned to class l if it is nearer to class l 's node than it is to any other node. Each element of the dataset is assigned to a class.

The method described above does not update the class seeds to account for spread of the data within the class and outliers can have a disproportionate effect on the classification. Consequently the classification will not be optimal and we now describe how to improve it.

The measure of in-class “spread” most suitable for our purposes is the sum of the squared distances of the class elements from the class mean, which is known as the class SSD. The sum of the class SSD's is known as the total SSD and gives a measure of the average variance in each class. We wish to partition our dataset into n classes so that the total SSD is a minimum. Let C_i be the set of the elements of the i 'th class and m_i as the number of elements in C_i . Then the vectors $\mathbf{x}_{ji} \in C_i$ are the elements of the i 'th class¹ where $j = 1, \dots, m_i$, with the class mean

$$\bar{\mathbf{x}}_i = \frac{1}{m_i} \sum_{j=1}^{m_i} \mathbf{x}_{ji}. \quad (3.1)$$

The quantity to be minimised, the total SSD, is given by

$$\text{total SSD} = \sum_{i=1}^n \sum_{j=1}^{m_i} \|\mathbf{x}_{ji} - \bar{\mathbf{x}}_i\|^2.$$

In order to find the global minimum for the total SSD we would need to consider all possible divisions of the dataset. This is computationally very expensive and we are forced to accept a local minimum found by successively refining our initial classification. The method we use to do this is known either as the k-means algorithm (Späth [18]) or generalised Lloyd algorithm (Gersho [20]).

Since we will modify the classes, we need to know how these changes will affect the total SSD. We define e_q to be the SSD of q 'th class, C_q , i. e.

¹In our case, the \mathbf{x} 's will be the images of our letters.

$$e_q = \sum_{j=1}^{m_q} \|\mathbf{x}_{jq} - \bar{\mathbf{x}}_q\|^2.$$

From the definition of the mean, we know that

$$\sum_{i=1}^m (y_i - \bar{y}) = 0$$

where $\bar{y} = \frac{1}{m} \sum_{i=1}^m y_i$, so it follows that

$$e_q = \sum_{i=1}^{m_q} \|\mathbf{x}_{iq}\|^2 - m_q \|\bar{\mathbf{x}}_q\|^2. \quad (3.2)$$

Let us now consider 3 classes, C_j , C_k and C_p , where C_j is a class of at least 2 elements, C_k is a non-empty but proper subset of C_j (i.e. C_k is a subset of C_j and does not contain all the elements of C_j) and C_p is the complement of C_k in C_j (C_p is non-empty). Formally, $C_k \neq \phi$, $C_k \neq C_j$, $C_k \subset C_j$ and $C_j = C_p \cup C_k$.

From (3.1) it follows that

$$\begin{aligned} \bar{\mathbf{x}}_j &= \frac{1}{m_j} \left(\sum_{i=1}^{m_k} \mathbf{x}_{ik} + \sum_{l=1}^{m_p} \mathbf{x}_{lp} \right) \\ &= \frac{1}{m_j} (m_k \bar{\mathbf{x}}_k + m_p \bar{\mathbf{x}}_p) \end{aligned} \quad (3.3)$$

and

$$\bar{\mathbf{x}}_p = \frac{1}{m_j - m_k} (m_j \bar{\mathbf{x}}_j - m_k \bar{\mathbf{x}}_k)$$

since $m_p = m_j - m_k$.

Then, substituting into expression (3.2) for e_p , it follows that

$$\begin{aligned} e_p &= \sum_{i=1}^{m_p} \|\mathbf{x}_{ip}\|^2 - m_p \|\bar{\mathbf{x}}_p\|^2 \\ &= \sum_{i=1}^{m_j} \|\mathbf{x}_{ij}\|^2 - \sum_{l=1}^{m_k} \|\mathbf{x}_{lk}\|^2 - m_p \left\| \frac{m_j \bar{\mathbf{x}}_j - m_k \bar{\mathbf{x}}_k}{m_p} \right\|^2 \end{aligned}$$

$$\begin{aligned}
&= (e_j + m_j \|\bar{x}_j\|^2) - (e_k + m_k \|\bar{x}_k\|^2) - m_p \left(\frac{\|m_j \bar{x}_j - m_k \bar{x}_k\|^2}{m_p^2} \right) \\
&= e_j - e_k + m_j \|\bar{x}_j\|^2 - m_k \|\bar{x}_k\|^2 - \frac{1}{m_j - m_k} \|m_j \bar{x}_j - m_k \bar{x}_k\|^2.
\end{aligned}$$

Since

$$\|m_j \bar{x}_j - m_k \bar{x}_k\|^2 = (m_j \|\bar{x}_j\|)^2 + (m_k \|\bar{x}_k\|)^2 - 2m_j m_k \bar{x}_j^T \bar{x}_k$$

it follows that

$$\begin{aligned}
e_p &= e_j - e_k + \left(m_j - \frac{m_j^2}{m_j - m_k} \right) \|\bar{x}_j\|^2 + \left(\frac{m_k^2}{m_j - m_k} - m_k \right) \|\bar{x}_k\|^2 \\
&\quad + \frac{2m_j m_k}{m_j - m_k} \bar{x}_j^T \bar{x}_k \\
&= e_j - e_k - \frac{m_j m_k}{m_j - m_k} \|\bar{x}_j - \bar{x}_k\|.
\end{aligned}$$

Similarly, we can show that

$$e_j = e_p + e_k + \frac{m_p m_k}{m_p + m_k} \|\bar{x}_p - \bar{x}_k\|^2. \quad (3.4)$$

Let us consider the case where $m_k = 1$ (i.e., where C_k has only 1 element, x_{1k}). Then the above results are written as follows

$$\bar{x}_p = \frac{m_j \bar{x}_j - x_{1k}}{m_j - 1}$$

$$\bar{x}_j = \frac{m_p \bar{x}_p + x_{1k}}{m_p + 1}$$

$$e_p = e_j - \frac{m_j}{m_j - 1} \|\bar{x}_j - x_{1k}\|^2$$

$$e_j = e_p + \frac{m_p}{m_p + 1} \|\bar{x}_p - x_{1k}\|^2.$$

These results describe how the mean and the squared distance of a class will change if we add to or remove an element from the class.

The k-means algorithm uses these results to determine how the total SSD would change if an element is moved to another class. We determine the change for moving an element to each other class and move the element only if the move decreases the total SSD. If there are several such moves, we will choose the move which leads to the greatest reduction in the total SSD.

Let us consider \mathbf{x}_{ir} , which is currently assigned to class C_r . For all the classes C_j , $j \neq r$, we calculate $\frac{m_j}{m_j+1} \|\bar{\mathbf{x}}_j - \mathbf{x}_{ir}\|^2$. If

$$\frac{m_j}{m_j+1} \|\bar{\mathbf{x}}_j - \mathbf{x}_{ir}\|^2 \geq \frac{m_v}{m_v+1} \|\bar{\mathbf{x}}_v - \mathbf{x}_{ir}\|^2 \quad \forall j$$

and

$$\frac{m_r}{m_r-1} \|\bar{\mathbf{x}}_r - \mathbf{x}_{ir}\|^2 > \frac{m_v}{m_v+1} \|\bar{\mathbf{x}}_v - \mathbf{x}_{ir}\|^2$$

then we move \mathbf{x}_{ir} to C_v . After the re-assignment, the total SSD becomes

$$\sum e_j = \sum_{j \neq r,v} e_j + \tilde{e}_r + \tilde{e}_v - \frac{m_r}{m_r-1} \|\bar{\mathbf{x}}_r - \mathbf{x}_{ir}\|^2 + \frac{m_v}{m_v+1} \|\bar{\mathbf{x}}_v - \mathbf{x}_{ir}\|^2$$

(where \tilde{e}_r and \tilde{e}_v are the original values, i.e. before the reassignment). This leads to the largest possible reduction in the total SSD.

Note that in one iteration of the k-means algorithm, each element of every class is tested against all the other classes. We repeat this process until there are no more moves which reduce the total SSD. Thus the running time for any given iteration will depend on the number of elements in our dataset and the number of classes, but the total running time will depend on our initial classification. Also note that this algorithm will only find a local minimum for the SSD, which will depend on the initial classification.

Since the heuristic method presented in the previous section ensures that the various classes are well separated, we expect to get good results using it as a starting point for the k-means algorithm. To test this, we tried several other initial classifications of the data and compared the results of refining them. Our other initial classifications included random and pattern-based² assignments, which are both quite frequently used with the k-means algorithm (see Späth [18]). In all cases, we get similar final classifications, but using the leader principle frequently resulted in the k-means algorithm converging faster, indicating that it is a good starting point. All the results concerning the k-means algorithm reported here were found using the leader principle as the initial classification.

²i.e. assigning every 3rd element to the same class and similar classification rules

3.2.4 Pairwise Nearest Neighbor

The major problem with the k-means algorithm is that, being iterative in nature, running times are uncertain and can vary dramatically with the initial classification. It is also slow since all the elements must be tested against all the other classes.

A common alternative classification method is the pairwise nearest neighbor algorithm (PNN). Here we start by assigning each element to its own class. Although the total SSD is zero in this case, we have as many classes as we have elements in our database. The idea is to merge some of these classes in such a way as to minimise the effect on the total SSD. Thus we can gradually build up a set of classes which should give a good classification of the data (see Gersho [20]).

This algorithm is also computationally expensive. However, Equitz [21] presented the following fast approximation to the PNN classification. He suggested we do not use the optimal merge at any given step. Instead, we only need to find a sufficiently good merge.

Also, in the full PNN algorithm, we start each step from scratch. In searching for the next merge, we repeat many of the comparisons of the previous search. But the second best possible merge from the previous search should still be a good candidate for a merge since the classes involved do not vary much from step to step. Thus it is possible to speed up the algorithm considerably by performing several merges at each step.

In deciding on an appropriate merge, we use the criteria derived for the k-means algorithm. We choose the merge that will result in the least increase in the total SSD. Since our classes generally contain more than one element, this is described by equation (3.4). The new contribution to the error is given by the term

$$\Delta\text{SSD} = \frac{m_p m_k}{m_p + m_k} \|\bar{x}_p - \bar{x}_k\|^2$$

and we choose merges for which ΔSSD is as small as possible. After merging two classes, the mean of the new class is calculated using equation (3.3).

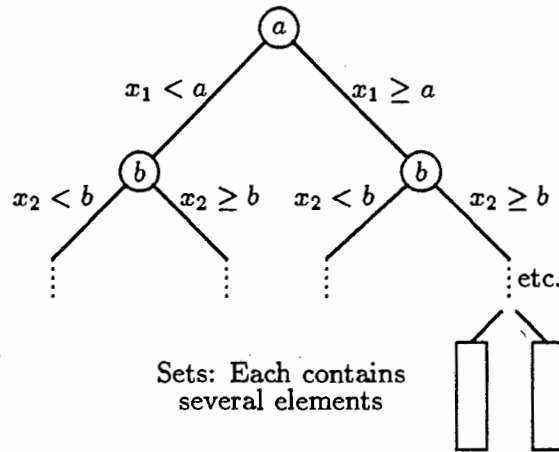
To use this algorithm, we need to be able to find good merges quickly. So we group our dataset into smaller sets where we know that the elements are reasonably near to each other. Since these sets are much smaller than our entire database, the best merge within each set can be found quickly.

Following the example of Equitz [21], we use a data structure known as a k-dimensional tree or kd-tree (see Samet [26]) to group the data. Specifically we use an incomplete kd-tree since we wish to divide the data into sets where

the elements are close to each other. We illustrate this structure in figure 3.1.

A kd-tree is a modification of the conventional binary tree. At each level, we split the data into two parts. However, unlike the binary tree, we do not use only one key to determine the division of our datasets. Instead, since we are dealing with vector data, we use a different component of the vector as our key at each level. For example, if we consider the vector $\mathbf{x} = [x_1 \ x_2 \ x_3 \ x_4]$, at the first level, we might decide that \mathbf{x} belongs in the left half of the tree if $x_1 < a$. At the next level, we might use $x_2 \lesseqgtr b$ as our division rule.

Figure 3.1: Diagram of a kd-tree



In the complete kd-tree, we would continue this process until each element is placed in a node of its own. However, in our case, we stop before this point and assign each element to a set. The elements of a set will have similar values for many of the vector components. Thus, to find our possible merges, we search each set for the best possible merge in that set. Since each set contains a fairly small subsection of the data, the search time is greatly reduced.

Of course, the different sets are not well separated. So, while we can find the best possible merge from each set, it will not necessarily be a particularly good merge. For this reason, we use only half of the possible merges in order to prevent us merging elements which are too far apart. After merging, we replace the two merged classes with the new class formed from the merge.

For good performance, we must keep the tree well balanced (i.e. ensure that all the sets have a similar number of elements). So we rebalance the

tree after each set of merges by reassigning elements from the larger sets to smaller sets. Balancing a binary tree is a standard operation and there are many methods which can be used (see Samet [26]), so we will not discuss this aspect here. This balancing of the tree, since it rearranges the elements in the various sets, ensures that, although we will not find merges which involve elements from different sets on any one step, these merges will be found at some later stage.

We continue until we have the required number of classes. Since we can easily keep track of the total SSD, we can also stop when this reaches some threshold. In this case, since we wish to compare the two classification algorithms, we decided to use 40 classes for this technique as well.

3.2.5 Eigenpicture Refinement

Once we have a good classification, we now turn to the final stages of processing our classes.

We calculate the eigenletters for each class separately. Thus we produce several eigenletter sets, with each set describing a particular class. Each letter should hopefully be well described by one of the classes.

One complication is that the classification algorithms allow small classes. Since the eigenpicture technique does not describe small datasets well, we cannot calculate the eigenletters for classes which are too small. Instead, we re-assign the members of the small classes to larger classes where they are well described. This will cause a few of our classes to change. The eigenletters of the changed classes are then re-calculated in order to improve the description of these new elements.

The presence of a small class might indicate that we have some region of the image-space which is poorly described, suggesting that the training set should include more information about that region.

3.3 Basic Classification Implementation

3.3.1 Algorithms for Classification

3.3.1.1 Leader Principle (Heuristic classification)

Our very basic classification algorithm takes the following form.

The actual code is listed as *classify.r* in appendix A.2.1.1.

```

// We assume we already have got the full
// eigenpicture set using all the letters
// and that  $D$  is our database
// Dsize is the number of letters
// We assume we are trying to find 40 classes

Begin
  Find  $k, l$  so  $\|D_i - D_j\|$  is maximum
  Let  $s_1 = k$ 
  Let  $s_2 = l$  // First 2 seeds

  repeat

    loop through database
      // finding seed  $j$  ( $s_j$ )

      for each  $i$  find  $\text{dist} = \|D_i - D_{s_k}\|$  so  $\text{dist}$  is a
        minimum for all  $k$ 

      if  $\text{dist}$  is a maximum for all  $i$ 
        then  $s_j = i$  // new seed
      end loop

  until we have 40 seeds

  // Okay, we have the seeds
  loop through entire database
    // We store the elements of each class in  $P$ 
    // so that  $P_i = \text{class of } D_i$ 
    find  $j$  so that  $\|D_i - s_j\|$  is a minimum
     $P_i = j$ 
  end loop

End

```

3.3.1.2 K-means Refinement

We used the following algorithm for the k-means refinement.

The actual code is listed as *k-means.r* in appendix A.2.1.2.

```

// We assume database is in  $D$ , with size Dsize
// Classification information is given in

```

```

// the list P.  $P_i =$  class of  $D_i$ 
// We use  $\downarrow$  SSD to indicate the decrease in SSD
// and  $\uparrow$  SSD the increase caused by a given move.

```

```
Begin
```

```
   $Q_i =$  number of elements in class  $i$ 
```

```
   $A_i =$  average of class  $i$ 
```

```
  // We now have the class means and the
```

```
  // starting class allocation
```

```
  repeat
```

```
    loop through database
```

```
      //  $i$  current item
```

```
       $j = P_i$ 
```

```
      if ( $Q_j = 1$ ) skip // don't destroy classes
```

```
       $\downarrow$  SSD =  $\frac{Q_j}{Q_{j-1}} \|D_i - A_j\|^2$  // From 3.4
```

```
      find  $k$  so that  $\uparrow$  SSD =  $\frac{Q_k}{Q_{k+1}} \|D_i - A_k\|^2$ 
        is a minimum ( $k \neq j$ )
```

```
      if  $\uparrow$  SSD  $<$   $\downarrow$  SSD then
```

```
        decreasing total SSD, so move
```

```
        // change  $Q$ 's and  $A$ 's
```

```
         $A_j = \frac{Q_j A_j - D_i}{Q_j - 1}$  and  $Q_j = Q_j - 1$  // From 3.3
```

```
         $A_k = \frac{Q_k A_k + D_i}{Q_k + 1}$  and  $Q_k = Q_k + 1$ 
```

```
      end if
```

```
    end loop
```

```
  until we cannot decrease SSD any further
```

```
End
```

3.3.1.3 PNN Classification

We illustrate the basic principles of the PNN algorithm. For details on the kd-tree, see Samet [26].

We assume that the tree remains well balanced as elements are added and removed from it. We also assume the tree will be automatically resized should it be necessary.

The code for a simple kd-tree, as well as the actual code for our implementation of the PNN algorithm is listed in appendix A.2.1.3 as *kdtree.r* and *pnn.r* respectively.

The algorithm is

```
// We assume we already have got the full
// eigenpicture set using all the letters together
// D is our database of letter images
// Dsize is the number of letters

Begin

  Create a tree with a depth of
    (integer part of  $(\log_2(\text{Dsize}) - 4)$ )
  // We want at least 16 elements in each bucket
  // to start with3
  Add all elements to tree

  while we have more than 40 classes

    for each bucket of the tree
      find classes a and b in this bucket
        which make best merge
        where a has  $N_a$  elements
        and mean  $X_a$ 
        and b has  $N_b$  elements
        and mean  $X_b$ 
      Add a and b to list of possible merges
    end for

    for the best 50% of the possible merges
      Remove  $X_a$  and  $X_b$  from tree
       $X_{\text{new}} = \frac{N_a X_a + N_b X_b}{N_a + N_b}$  // new class average
      Add  $X_{\text{new}}$  to tree
    end for

    Rebalance tree if required
  end while
```

³Since at each split, we divide the data in to sections, a tree of depth N has 2^N leaves. Thus, to have 16 elements at each leaf, we need $2^N = \frac{\text{Dsize}}{16}$. This then leads to $N = \log_2(\text{Dsize}) - \log_2(16)$, the formula used here. We round down to make sure we do not have less than 16 elements in each bucket.

End

3.3.1.4 Eigenpicture Refinement

Having classified the data, we need to calculate their eigenletters. We use the following algorithm to do this. The actual code is listed as *eigclassify.r* in appendix A.2.1.4.

```

// we assume images loaded in I
// We assume  $Q_i$  = size class  $i$ 

Begin
  // We will ignore small classes

  loop over all classes
    if ( $Q_i < 4$ ) then skip
    calculate average of class  $i$ ,  $A_i$ 
    calculate eigenletters of class  $i$ 
    and store them in  $U_i$ .
  end loop

  loop over all images
    find  $j$  so that  $\|I_i\|^2 - \|U_j^T (I_j - A_j)\|$  is a minimum
    assign  $I_i$  to class  $j$ 
  end loop

  recalculate eigenfaces of classes which have
  acquired new elements from the small classes
End

```

3.3.2 Classification Results





We will first consider the results of using the k-means algorithm to classify our data. We will then compare these with the results of the PNN algorithm.

In each case, we initially separate the data into 40 classes which we then process to obtain the eigenletters. This refinement further reduces the number of classes to 37. A full listing of the classes obtained by both methods is included in appendix C. In this section, we highlight some of the more interesting results.

3.3.2.1 Heuristic and K-means Classification

Figure 3.2 shows the average of 4 different classes obtained from this algorithm. Our original images were binary images. We chose to consider black as 0 and white as 255. Then our average letters are greyscale images with values ranging from 0 to 255.

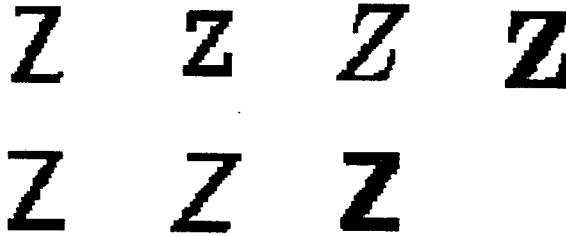
Figure 3.2: Table of class averages

Class 1	Class 8	Class 27	Class 35
			

We note that in all cases, the average is very letterlike. Class 1 obviously contains combinations of capital E's and capital B's, while class 35 contains capital A's and class 27 appears to mix c's and e's. This is exactly the behavior we were trying to achieve.

We list all the elements of class 8 in figure 3.3. Not surprisingly, this class contains the Z's from seven different fonts.

Figure 3.3: Elements of class 8.



Class 8 is one of the smaller classes, with only 7 elements. In comparison class 27 has 38 elements, as listed in figure 3.4.

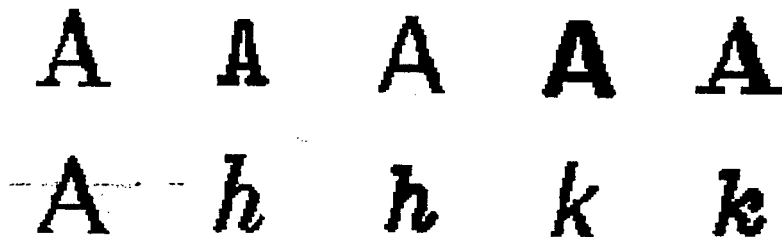
Note that all the letters in class 27 are very similar. The presence of the s's might be surprising, but on reflection they do share many characteristics with the e's. The only other oddity is the presence of one "u" and one "n" in this class. However, the combination of those 2 letters resembles an italic "o", consequently this is a good class to describe those letters. It should be mentioned that these letters were added in the eigenpicture refinement stage, from some other small class.

In figure 3.5, we list the elements of class 35. Surprisingly, the class contains 4 letters which we would not normally classify as being similar to capital A's.

Figure 3.4: Elements of class 27.



Figure 3.5: Elements of class 35.



However the slope of the vertical stroke in the italic letters is very similar to that of the left stroke of the A's. So there is considerable similarity between

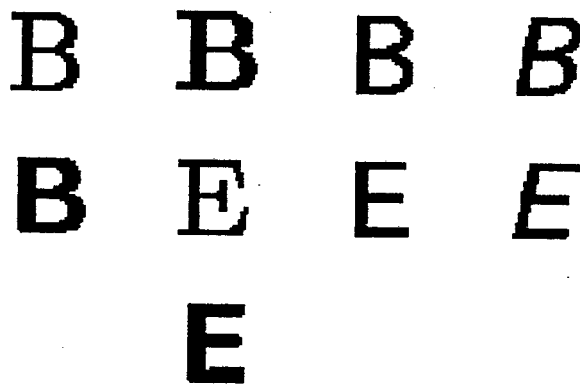
the letters in this class.

3.3.2.2 PNN Classification

Using the pairwise nearest neighbor, we get fairly similar results to those of the k-means based classifier. However, since the classes are numbered differently, we cannot merely compare class number 8 of the two methods. The PNN classes are also listed in appendix C, next to the closest corresponding class from the k-means algorithm. For instance the PNN class which closely approximates class 27 from the k-means algorithm is class 17. The PNN class which actually best matches class 8 is class 21 (these two are in fact identical).

However, most of the other classes differ by a few letters. For instance, comparing class 1 of the k-means algorithm (figure 3.6) with its closest matching class from the PNN algorithm (figure 3.7), which is class 35, we see that, in addition to the B's and E's which created class 1 in the k-means algorithm, we also acquire 3 P's. However, all the letters from the k-means class are included in this case.

Figure 3.6: Elements of k-means class 1

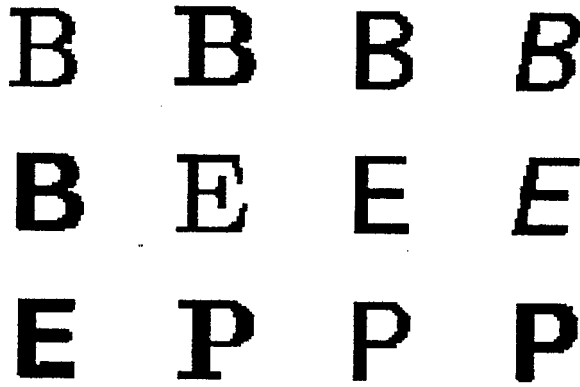


B B B B
B E E E
E

While the majority of the classes are fairly similar, there are a few cases of special interest. In some cases, as listed in the appendix, there are classes from the PNN algorithm which do not correspond with any class from the k-means algorithm. For instance, class 13 from the PNN classification does not correspond with any of the k-means classes.

Likewise, there are a few cases where the separation of the PNN classes differs markedly from that of the k-means algorithm. For instance, although there is no k-means class which closely corresponds with class 13 from the PNN classification, we note that the letters in classes 12, 13 and 14 from

Figure 3.7: Elements of PNN class 35



the PNN algorithm form only 2 classes in the k-means algorithm, namely 23 and 27. So we can see that the two algorithms divide the letters differently.

This indicates that we probably do not have an optimal classification in either case, but since the differences are small, both classifications are probably close to the optimal one, so we would expect the two classifications to give similar results.

3.3.2.3 Comparative Reconstruction Performance

The justification of the classification process is that a class based system should perform better than the eigenletter description derived from the full training set. Consequently, we wish to compare these alternatives.

To do this, we compare the ability of the classes from the k-means and PNN algorithms to reconstruct letters against that of the full eigenpicture set. For additional comparison, we also use the eigenletters calculated from the "natural" classification we defined earlier.

For the full set of eigenletters, we use 40 eigenletters in creating the reconstructions. For the various classes, we use between 3 and 10 eigenletters depending on the class size. For the "natural" classification, we use 5 eigenletters for each class.

It is important to point out that the more compact representation of the elements of each class improves the efficiency of the recognition rule considerably.

We tabulate representative results in figure 3.8. From this table, it is clear that the class based methods give much better representations of all the

letters. In all cases, we can improve the representation by using more eigen-letters, but this results in a more cumbersome recognition rule.

Figure 3.8: Reconstruction results using the different methods

Original	Full	"Natural"	K-means	PNN
A				
A				
K				
Q				
S				
b				
m				
n				
n				

Also note that the automatic classifications tend to have a more uniform

error spread, whereas the "natural" classification produces very localised errors (such as seen in the capital A's). In some cases, the representation of the "natural" classification is better than that of the automatic classification. This occurs when the "natural" classification's class is a subset of the class found by the automatic classification. It is therefore better described in the "natural" classification.

There are only small differences between the reconstructions from the two automatic classification methods. In some cases, such as the letter "m", they produce virtually identical reconstructions. In other cases, such as the italic "A", the PNN based classification performs better, while there are letters for which the reconstruction using the k-means classification is better. Thus there is little to choose between these classifications in terms of performance. However, the PNN algorithm can be implemented very efficiently and would be the preferred choice for larger datasets. In any case, we can always use the k-means algorithm to improve the PNN classification.

3.4 A Simple OCR System

3.4.1 Implementation Details

To test the recognition performance of these systems, we designed a very simple OCR system. There are two problems here. First, we have to locate the letter on the page. Only then can we try and identify it.

It is possible to use the location method described for faces in section 2.6 to locate the letters. However, as there are many letters on any given page, and as we will be searching in several different classes, this approach is unacceptably slow on the available hardware. Thus, to locate a possible letter, we search for a region which is well represented by some class (see the DFFS measure discussed in section 2.4.3). This region then becomes a candidate letter and we can search for a matching letter in that class.

One of the problems is that letters are not equally sized, and modern printing technology uses variable spacing between the letters. For instance the letter "i" will occupy a much smaller area than the letter "W". However, we are extracting image sections which are all of equal size. So, when extracting a letter from a word, there is a risk that we will include sections of the surrounding letters. This will adversely affect our error measurement, since we will be unable to reconstruct this extra noise. We need then to filter out this extraneous information.

Obviously, each class will require its own filter. We need to retain the information relevant to the eigenpictures of the class. Thus, the filter has to

have a value of 1 wherever any of the eigenpictures is non-zero. Note that we cannot completely ignore the surrounding information, since that would result in matching subsections of letters. Therefore, we design the filter so that it tails off gradually from the region of interest. We do this by using a fairly simple blurring method.

In figure 3.9, we show the filters constructed for classes 27 and 35.

Figure 3.9: Filters for classes 27 and 35.



In figure 3.10, we have an example of how the filtering process works. On the left, have a section from some document where the letters are close together. By using the correct filter, we can eliminate the extra characters and extract the "B". In practice, the average letter is subtracted from the image before any noise caused by other letters is removed by the filter.

Figure 3.10: Before and after filtering

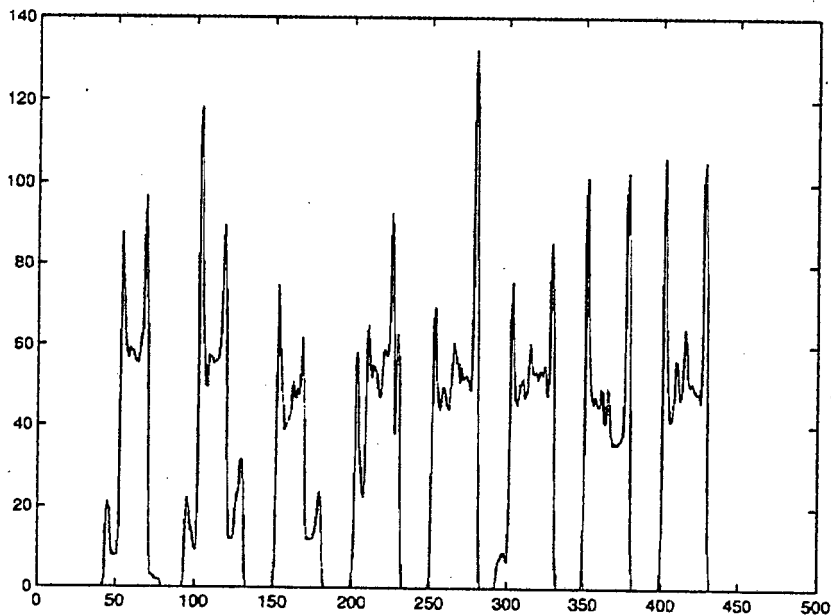


This filtering operation brings with it the additional risk that we will extract only a subsection of a letter, for which we could then find a match. For instance, if we were to extract only one stroke of the letter H, it would match the letter I. Thus we bring in the concept of letter "weight", which is a measure of how large an area the letter occupies. Thus "W" has a large weight, while "i" has a very small weight. If we are faced with two possible matches at one location with fairly similar probabilities, we will favor the match of a letter with more weight, thus compensating for the risk of matching sub-sections of a letter.

The final issue that needs to be discussed is what to do after we have successfully identified the letter. In a commercial OCR system, we then have the problem of mapping this newly matched letter onto a document. This has to account for the variable spacing between the different letters. Since we are concerned with the ability of the system to correctly identify letters, we have not pursued this issue. However, we note that the Radon transform allows us to easily determine the correct vertical placement. Using the Radon transform, we project the image onto a line at a given angle. If we

reverse the image, so a value of zero represents the white background and project onto a line parallel to the y-axis, the space separating two lines of text shows up as an area where the resulting projection is zero. This allows us to segment the text into the different lines.

Figure 3.11: Radon transform of 3.12 at 90°



As an example we use the the Radon transform on figure 3.12, which is one of our standard test cases, to get figure 3.11. Since the blank space separating the text show up as regions where the graph is zero, we can easily determine a range of Y coordinates which correspond to any one line of text.

Once we have the individual lines of text, it is possible to use the Radon transform to extract the letters from each line. However, since we are concerned with how the eigenpicture technique can be extended to the OCR problem, we have chosen not to use this method.

3.4.2 Algorithm for OCR System

To create the filters, we use the following algorithm.

The actual code is part of *setup.r* listed in appendix A.2.2.1.

```
// We assume we are dealing with only one class
// eigenpictures are stored in U
// We assume images are Dim x Dim
```

```

Begin
  set filter = 1 wherever  $U$  non-zero
  make filter matrix
  // can then work in definite directions
  // Blur horizontally
  filterij =  $\frac{\sum_{k=i-5}^{i+5} \text{filter}_{kj}}{6} + \text{filter}_{ij}$  for  $5 \leq i \leq \text{Dim} - 5$ 
  // add average of surrounding area

  for  $i < 5$  we set filterij = filter5j -  $\frac{1}{i}$ 
    and similarly for  $i > \text{Dim} - 5$ 

  if (filterij > 1) then filterij = 1

  if (filterij < 0) then filterij = 0
    vertical smoothing works the same way.
End

```

We wish to locate the potential letters on a page. We will illustrate the algorithm for only one class, as when using multiple classes, we will merely repeat the steps for each class in turn. For the sake of clarity, we will not illustrate a multi-scale search, but this merely adds an additional loop.

The actual code is part of *findchar.r* listed in appendix A.2.2.1.

```

// We assume  $U$  = class eigenletters
// and  $A$  = class average
// Image is the image
// we looking at sections of size Dim x Dim

Begin

  loop across Image
    I = section of interest from Image
    Test = I - A
    apply filter to Test
    y =  $U^T \text{Test}$ 
    if  $(\|\text{Test}\|^2 - \|y\|^2) < \text{InterestMargin}$  then
      Add to list of locations
    end loop

End

```

We process the list and extract matches using the following algorithm.

A slight modification of this algorithm is implemented as part of *findchar.r* in appendix A.2.2.1.

```

// We assume that we are dealing
// with one possible location
// I is image which may contain letter

Begin

  loop over all classes
    U = eigenletters of current class
    A = Average of current class
    Test = I - A
    apply filter to Test
    y = UTTest

    if ||Test||2 - ||y||2 is minimum for all classes
      then found best class, so search
        D = elements for this class
        find i so ||Di - y|| is minimum
        if ||Di - y|| < Threshold then return match
    end if

  end loop

End

```

This rather simple algorithm can be improved in many ways. For instance, if we have an image that is somewhat noisy, it might be difficult to decide in which class a section of the image belongs. Thus in practice, we take the best three possibilities at each location, search for matches in all three classes and assume that the best match is the correct one.

3.4.3 Results

For the purpose of comparing the effectiveness of our classification algorithms at recognising letters, we compare them with both the “natural” classification and the case where we treat all the letters as a single training set.

Since any OCR system has to be able to recognise letters from different fonts, we show the results of testing the system on a font, shown in figure

3.12, which was part of the training set and on another font, shown in figure 3.17, which was not part of the training set.

Figure 3.12: Font in training set

A B C D E F
G H I J K L
M N O P Q
R S T U V W
X Y Z w x y z
a b c d e f
g h i j k l m n
o p q r s t u v

3.4.3.1 Recognising a Font in the Training Set.

As our first test, we use the image in figure 3.12, which consists of the full alphabet of one of the fonts in our training set. The results are summarised in table 3.1⁴. It is immediately clear that for this case, no classification

⁴The percentages quoted in these tables are based on dividing the number of elements in each category by 52, the number of letters in the image. However, since there are cases where a character is mistakenly identified as two or more characters, the percentages do not necessarily add up to 100%.

method works best.

Table 3.1: Results of using four different classification methods on fig. 3.12.

Classification Method	Percentage Correct Identifications	Percentage Unidentified	Percentage Misidentifications
None	86	14	0
"Natural"	71	11	18
K-means	81	14	7
PNN	81	14	9

Figure 3.13 shows the results of using the full set of eigenpictures on this example. We get reasonable results, with successful identifications of most characters.

Figure 3.13: Results using eigenletters from full alphabet on fig. 3.12

```

      B   C D   E   F
G   H   I J K   L
M   N   O   P   Q
R           T   U
      Y   Z w x   y z
a b   c           e f
g h   i j k l m   n
o p           r   s t u v

```

Figure 3.14 shows the results of using the "natural" classification of the letters. This performs very poorly, and produces many misidentification. This is clearly a poor choice.

Figure 3.14: Results using eigenletters from "natural" classification on fig. 3.12

```

A   B   C D   E   I
G   I I I J K   i
      N   O   P Q
I   i I   V   W
X   Y           x   v z
a b   c d I e f
g h   i j k i m   n
o p   q           t u v

```

Figure 3.15, shows the results of using the classification derived from the

k-means algorithm. It is comparable with the results from the full test, but we note the the errors are differently arranged.

Figure 3.15: Results using eigenletters from k-means classification on fig. 3.12

A	B	C	D	E	F
	Y	I	J	K	m
M	m	N		P	Q
R		T		V	l
X	Y	Z	w	y	
a	b	c	d	f	
g	h	i	j	k	l
o	p	q	r	t	u

Figure 3.16 uses the classes derived from the PNN algorithm. In terms of the results, it is virtually identical to those obtained by the k-means classes shown in figure 3.15. Considering how similar the actual classifications were, this is not a surprising result. It should also be clear that there is much room for improvement in all cases.

Figure 3.16: Results using eigenletters from PNN classification on fig. 3.12

A	B	C	D	E	F
	Y	I	J	K	r
M	m	N	i	P	Q
R		T		V	l
X	Y	Z	w	y	z
a	b	c	d	f	
g	h	i	j	k	l
o	p	q	r	t	u

3.4.3.2 Recognising a Font not in the Training Set.

To test the system's ability to handle fonts which are not part of the training set, we used the image in figure 3.17. This font, although similar to those in the training set, is not part of the training set.

The results are summarised in table 3.2. In this case, the k-means and PNN based systems are clearly superior.

Figure 3.17: Font not in training set.

A B C D E F G H I v
 J K L M N O P Q R S
 T U V W X Y Z w x y
 a b c d e f g h i j k l z
 m n o p q r s t u

Table 3.2: Results of using 4 classifications methods on fig. 3.17.

Classification Method	Percentage Correct Identifications	Percentage Unidentified	Percentage Misidentifications
None	16	80	4
"Natural"	46	18	40
K-means	60	18	21
PNN	60	20	21

Figure 3.18 show the results of using the full eigenpicture system on this new font. While it successfully identifies some of the characters, clearly many have been completely missed. Thus it performs quite poorly under these circumstances.

Figure 3.18: Results using eigenletters from full alphabet on figure 3.17.

B C
 l M O Q
 T w
 b d h j l
 m s
 o u

Figure 3.19 shows the results from the “natural” classification. Once again we see very poor results, which suggests that the natural classifier is a poor way of approaching this problem.

Figure 3.19: Results using eigenletters from “natural” classification on fig. 3.17.

	B		I		E	i		i	I	i	v				
i	I		I	i	M		i		m		S				
	U							Y	Z	w	x				
a	I	b	c	G	C	e	i	o	h		i	j		i	z
m		n	G	o	o	q	I			t	u				

Figure 3.20 shows the results based on the k-means classification. While there are still many errors, we can see a marked improvement in the results.

Figure 3.20: Results using eigenletters from k-means classification on fig. 3.17.

A			m	D		E	F		G	l		l	v
l	K		L		M				O	P		R	S
F		J	V		W	l			Y	Z	w		V
a			c	d	Y	e		o	m		i	j	l
m		m		o	p	q	Y		s	t	u		

Figure 3.21 shows the results of using the PNN classes. Once again they are similar to those of the k-means classes.

Figure 3.21: Results using eigenletters from PNN classification on fig. 3.17

A			m	D		E	F		G	l		l	v
l	K	m	L		M				O	P		R	S
F	m	J	V		W	l			Y	Z	w		V
a			c	d	Y	e		o	m		i	j	l
m		m		o	p	q			s	t	u		

3.4.3.3 Comments

It is worth noting that the results of the k-means and PNN classifiers are virtually identical and usually differ only in the nature of the mismatches. Thus it appears that the results improve with the use of classification, but are fairly robust against variations in the actual classification.

It is curious that, although it manages to reconstruct some letters very well, the “natural” classification performs so poorly on recognition. This is because we don’t have a good separation between the classes. While some letters are fairly poorly represented by their correct class, others are well represented in a range of classes. It is therefore difficult to determine suitable thresholds.

We also note that in some cases, the full alphabet and natural classification techniques correctly identify letters missed by the automatic classification. However, in the automatic classification, we try to ensure that all letters are equally well described, a statement that cannot be made for the other methods. In the full alphabet, we try to ensure that any given letter is “well” described, but this can lead to much better descriptions of certain letters.

The full alphabet is also better at distinguishing letters from whitespace in terms of the early stages in letter location step. This is not surprising since at that stage we are only interested in whether a letter is present in a given location or not and so a general description of letter-like features is sufficient.

Since the class based systems are able to correctly identify many of the letters if given the correct location, it is probable that by improving the letter location algorithm, we can improve the performance of these systems. We will discuss one possibility in the next section.

3.4.3.4 K-means with Improved Segmentation Algorithm.

The experiments of the previous section indicated that the performance of the class based systems can be improved by modifying the letter location algorithm. One obvious possibility is that, since the full set of eigenletters provides the best segmentation performance, we use these eigenletters to segment the image, rather than the eigenletters from the various classes.

In figures 3.22 and 3.23, we illustrate the results of implementing this idea. We summarise our results in table 3.3.

Table 3.3: Results of using improved segmentation algorithm.

Figure	Percentage Correct Identifications	Percentage Unidentified	Percentage Misidentifications
3.12	88	9	5
3.17	64	18	19

Interestingly, while in figure 3.22, the results have improved somewhat, there

Figure 3.22: Results using k-means eigenletters with improved segmentation algorithm on fig. 3.12.

A		B		C	D		E		F
		H		I	J	K	m	L	
M	m	N		O		P		Q	
R			T			U	V		l
X		Y		Z	w		y		z
a	b		c	d		e	f		
g	h		i	j	k	l	m		n
o	p		q	r			t	u	v

Figure 3.23: Results using k-means eigenletters with improved segmentation algorithm on fig. 3.17.

A	B		m	D		E	F		G	l		l	v	
l	K		L		M				O	P	Q		R	S
F		J	V		W	l			Y	Z	w			V
a			c	d	Y	e		o	m		i	j		l
m		m		o	p	q			s	t	u			

is hardly any improvement in figure 3.23.

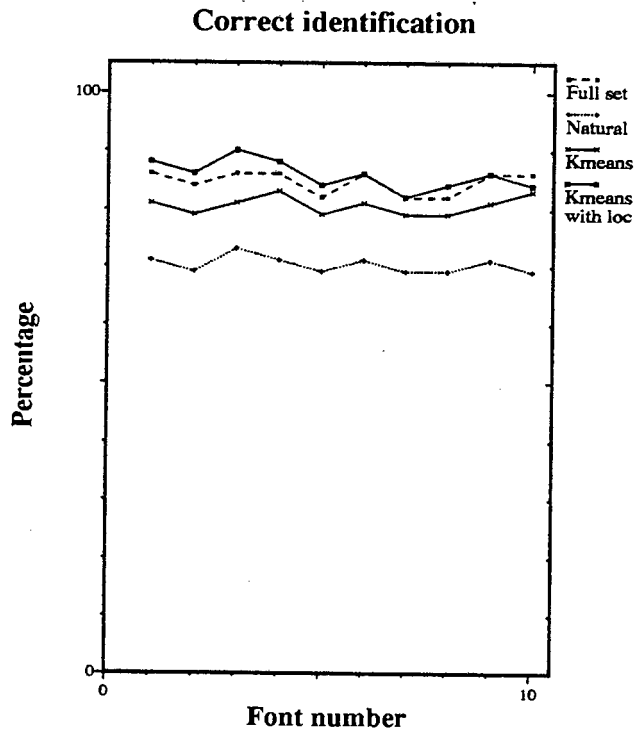
This is best illustrated if we graph the relative performance of our 4 systems. In figure 3.24, we graph the recognition performance for each of the fonts in the training set. The bad performance of the natural classification can clearly be seen, while the the full system outperforms the straight k-means classification.⁵ However, we can also see that with the improved location rule, the k-means classes gives comparable performance to the full system (around 86% correct).

We can also see that all the systems perform slightly worse on the italic fonts, although the drop in performance is minor and is no doubt due to the nature of our training set.

In figure 3.25, we show the comparative performance for 3 fonts not in the training set. Both the correct identifications and the incorrect ones are shown. Only the cases where a character is actually misidentified are included, i.e. those cases where the system does not locate a character are ignored since a mismatch.

⁵We have not included the PNN classes in these graphs as the results for this system do not differ significantly from the k-means classes.

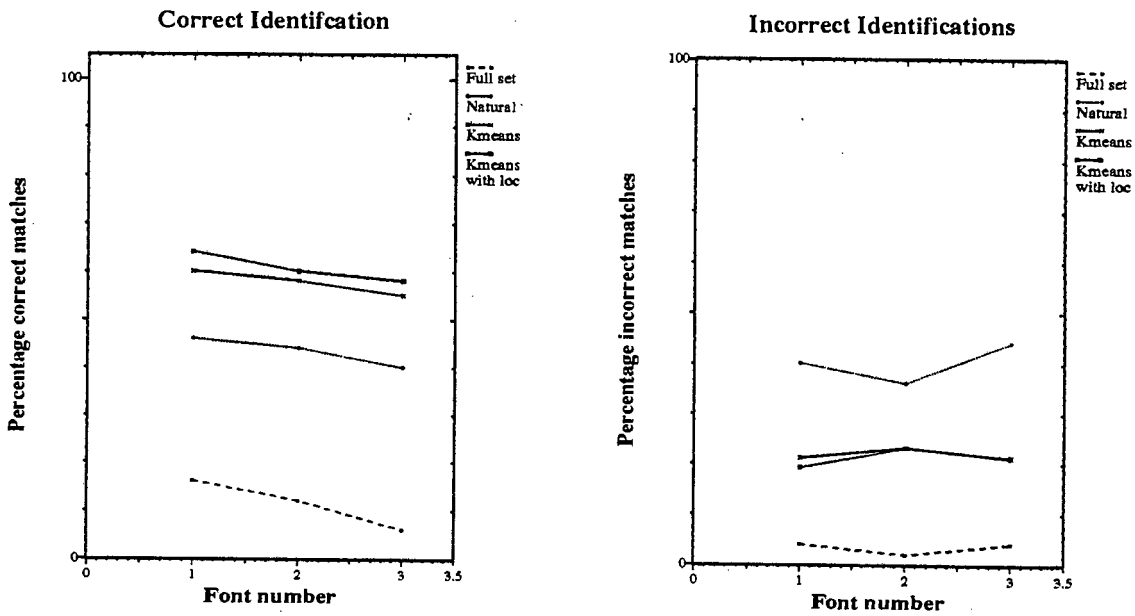
Figure 3.24: Recognition performance for fonts in the training set.



From its performance on the out of training set examples, we can clearly see that the natural classification is unacceptable. Although we correctly identify more characters than with the full set, the number of incorrect identifications is unacceptably high. This is again due to the poor separation of the classes we discussed earlier.

The k-means classes clearly give the best performance. They extend to fonts outside the training set successfully if the fonts are similar to those in the training set. Not surprisingly, the number of errors increases the more the font differs from those in the training set. However, there is little improvement here if we use our improved location rule.

Figure 3.25: Recognition performance for fonts not in the training set.



3.5 Improving the Recognition Rule

3.5.1 Description

While we can improve the letter location algorithm, it is also possible to improve the actual recognition rule.

The OCR system described in the previous section uses the same recognition rule we used for recognising faces, where we needed to find the matching image. However, in an OCR application, we want to identify the character. In some application we wish to create a document which resembles the original, so we do need to know the correct font, but it is still more important to correctly identify the letter, and even in these applications, it is frequently sufficient to know that the letter is from an italic font rather than the precise details of whether the font is Times or Courier. So the important information is that the letter is "A", not which font the letter is from. Building this into our recognition rule should lead to improved performance.

Using the same notation from the faces, we know that if I is an image of object O , then I has a representation $\mathbf{y} = [y_1 \ y_2 \ \cdots \ y_P]$, where y_i is the i 'th coefficient in the eigenpicture representation.

If we have a different image $\tilde{\mathbf{I}}$ of the object O , then we can write $\tilde{\mathbf{I}} = \mathbf{I} + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ is the error. Then the representation of $\tilde{\mathbf{I}}$ can be written as

$$\tilde{\mathbf{y}} = \begin{bmatrix} y_1 + \epsilon_1 & y_2 + \epsilon_2 & \cdots & y_P + \epsilon_P \end{bmatrix}$$

where ϵ_i is the error in the i 'th coefficient. We assume that $\mathbf{E}(\boldsymbol{\delta}) = 0$, i.e. that the error is unbiased. Thus $\epsilon_i = \mathbf{u}_i^T \boldsymbol{\delta}$, where \mathbf{u}_i is the i 'th eigenpicture. Since the transformation is linear, $\mathbf{E}(\boldsymbol{\delta}) = 0$ implies that $\mathbf{E}(\epsilon_i) = 0$.

Let us define a linear weight function $\mathbf{w}(\mathbf{y})$, where $\mathbf{w}(\mathbf{y}) = \begin{bmatrix} w_1 y_1 & w_2 y_2 & \cdots & w_P y_P \end{bmatrix}$, and a recognition rule based on $\|\mathbf{w}(\mathbf{y}) - \mathbf{w}(\tilde{\mathbf{y}})\|^2$ where

$$\begin{aligned} \|\mathbf{w}(\mathbf{y}) - \mathbf{w}(\tilde{\mathbf{y}})\|^2 &= \left\| \begin{bmatrix} w_1 y_1 - w_1 (y_1 + \epsilon_1) & \cdots & w_P y_P - w_P (y_P + \epsilon_P) \end{bmatrix} \right\|^2 \\ &= w_1^2 \epsilon_1^2 + w_2^2 \epsilon_2^2 + \cdots + w_P^2 \epsilon_P^2. \end{aligned} \quad (3.5)$$

We need a threshold value in order to determine whether to accept a match or not. The expected value of (3.5) should therefore be a constant (where we take the expectation in terms of the random variables ϵ_i), which for convenience is written as K .

Since (3.5) is a linear combination of random variables (ϵ_i^2), its expected value is the linear combinations of the expected values of the ϵ_i^2 's, so it follows that

$$\begin{aligned} K &= \mathbf{E}(\|\mathbf{w}(\mathbf{y}) - \mathbf{w}(\tilde{\mathbf{y}})\|^2) \\ &= w_1^2 \mathbf{E}(\epsilon_1^2) + \cdots + w_P^2 \mathbf{E}(\epsilon_P^2) \\ &= w_1^2 \mathbf{Var}(\epsilon_1) + \cdots + w_P^2 \mathbf{Var}(\epsilon_P) \end{aligned}$$

where the last step follows from the fact that the $\mathbf{E}(\epsilon_i^2) = \mathbf{Var}(\epsilon_i)$, since $\mathbf{E}(\epsilon_i) = 0$.

Since we do not want our system to be overly sensitive to changes in any particular coefficient, we choose

$$w_i^2 \mathbf{Var}(\epsilon_i) = k \quad \forall i.$$

Thus $K = kM$ and

$$w_i^2 = \frac{K}{M\text{Var}(\epsilon_i)}. \quad (3.6)$$

Using the Euclidean distance as a measure, as in our earlier calculations, implies that the weights are assumed to be equal. From the above calculation, it is now clear that this amounts to assuming that the variances of the errors ϵ_i are equal, i.e., that the variance of the error is independent of the "direction" of the error, and thus independent of the amount of the image described by that direction.

For letters, since the later coefficients primarily describe those variations which distinguish one font from another, we can argue that these coefficients should play a much smaller role in identifying the letter. Thus variations in these later coefficients should have less effect on the choice of the match. This is equivalent to assuming the errors of these coefficients have larger variances. Thus $\text{Var}(\epsilon_i) = f(\lambda_i)$, where f is a decreasing function of λ .

A simple model which satisfies this is $f(\lambda) = \frac{1}{\lambda}$. Using (3.6), it follows that our weights are

$$w_i = \sqrt{\lambda_i}$$

We modify our recognition rule to use these weights.

3.5.2 Modified Algorithm

The only major change to our previous algorithm will be the addition of this new recognition rule.

This is the recognition rule we implement in *findchar.r* listed in appendix A.2.2.1.

In our previous algorithm, (see pp. 3.4.2), we minimised $\|D_i - y\|$. Now we minimise E where

$$E = \sqrt{\left[\sum_{j=1}^P \lambda_j (y_j - D_{ij})^2 \right]}$$

and P is the number of eigenletters we use to describe that class.

3.5.3 Results

To demonstrate this new rule, we use the same test cases as in section 3.4.3. With this recognition rule, the PNN classes and k-means classes give identical final results. The only slight differences are in the actual margins by which a match is accepted. The results shown in this section thus apply to both sets of classes.

In figure 3.26, we show the results of using this rule on figure 3.12. All the letters are now recognised perfectly.

Figure 3.26: Results of using modified recognition rule on fig. 3.12.

```

A      B      C D      E      F
G      H      I J K      L
M      N      O      P      Q
R      S      T U      V      W
X      Y      Z w x      y z
a b      c d      e f
g h      i j k l      m n
o p      q r      s t u v

```

In figure 3.27, we show the results of using the modified rule on the out of training set example, figure 3.17. While we still have errors, there is a vast improvement compared to our previous results. The results are summarised in table 3.4.

Figure 3.27: Results of using modified recognition rule on fig. 3.17.

```

A      B C      D      E F      G H L l v
J K      L      M      l      O P      Q      R S
      T U      V W      Y Z      w x      y
a      b c o      e      o      h i j k l z
m r n      o p      q r s t      u

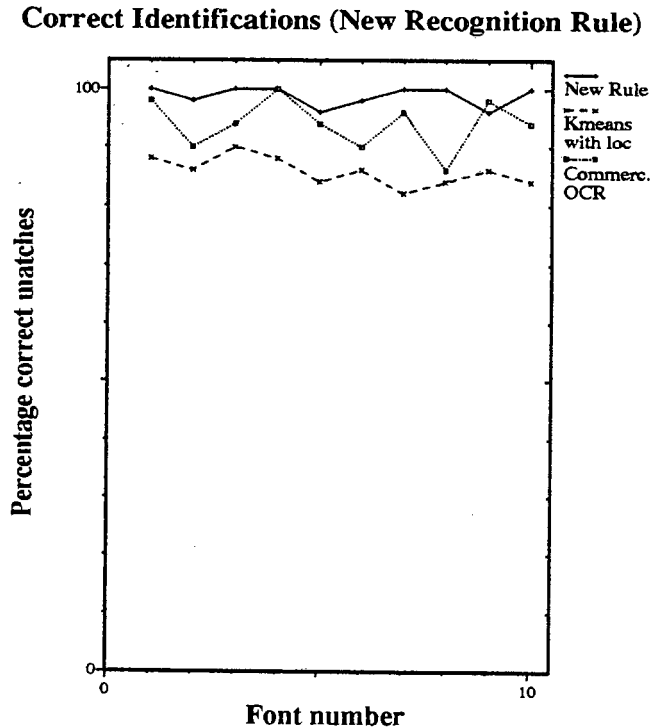
```

Table 3.4: Results of using the modified recognition rule.

Figure	Percentage Correct Identifications	Percentage Unidentified	Percentage Misidentifications
3.12	100	0	0
3.17	87	4	11

The improvement with this new rule can be best illustrated by comparing the results with our best previous system, which was using the k-means classes with the modified location rule.

Figure 3.28: Recognition performance for fonts in the training set

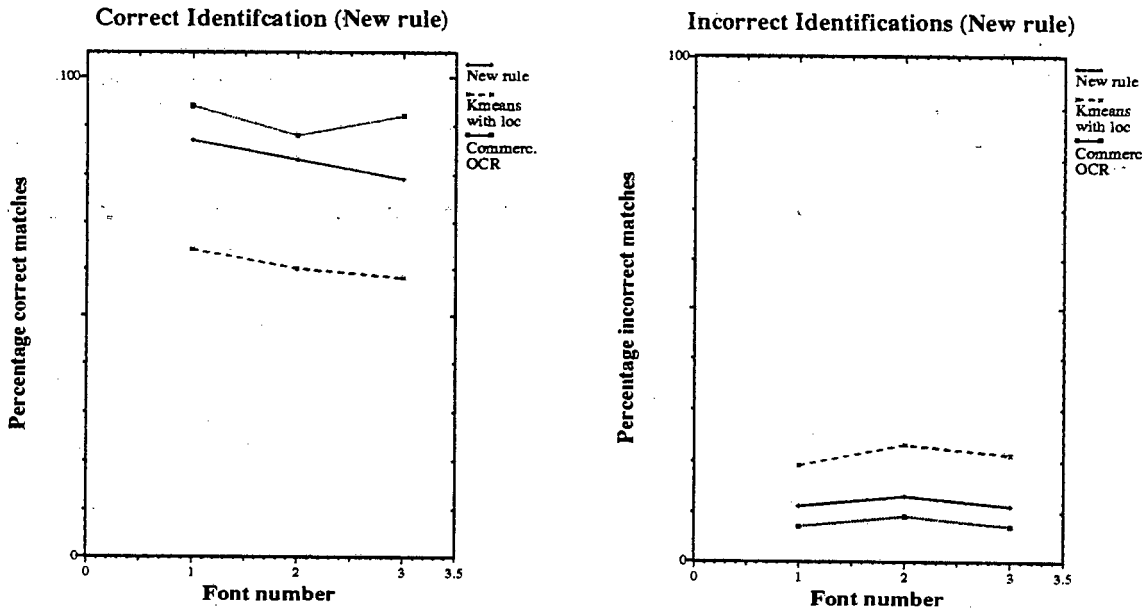


In figures 3.28 and 3.29 we compare some performances as in section 3.8. The new rule performs significantly better in all cases. For fonts in the training, we obtain almost perfect results. The performance on fonts outside the training set is still inadequate demonstrating the need for a more descriptive training set.

For comparison, we also include the results of using a commercial package, Textbridge Classic from Xerox on our test cases. For fonts in training set examples we can see that we actually outperform the commercial system, while the commercial system performs better for fonts outside the training set. However, the results indicate that we are not far off the performance level of at least some commercial systems.

In all our tests to date, we have been using perfect examples, where the data was scanned with zero noise. We wish to investigate the systems per-

Figure 3.29: performance for fonts not in training set.



formance under more realistic conditions.

3.6 Recognising Imperfectly Scanned Images

In order to test how the system performs in cases of imperfectly scanned images, which is a more realistic test case, consider the text in figure 3.30.

Figure 3.30: Original image which will be scanned

**THE QUICK RED
FOX JUMPED
OVER THE LAZY
BROWN DOG**

For these tests, we scanned this image at two different resolutions.

In figure 3.31, we scanned the image using a resolution of 100×100 dpi with a scanner that was slightly out of focus. Figure 3.32 shows that the system struggles with this image. Nevertheless several letters are correctly identified.

In figure 3.33, we scanned text at 200×200 dpi, which is closer to the

Figure 3.31: Figure 3.30 scanned at 100×100 dpi.

THE QUICK RED
 FOX JUMPED
 OVER THE LAZY
 BROWN DOG

Figure 3.32: Results of recognizing fig. 3.31.

t H U I D
 F
 J T H r

examples in our database. The results of using this image are shown in figure 3.34. Although we still a number of letters that are not identified, there is a considerable improvement over the results in figure 3.32.

We summarise the results in table 3.5.

Table 3.5: Results using actual scanned data.

Image	Percentage Correct Identifications	Percentage Number Unidentified	Percentage Number Misidentifications
3.31	21	74	5
3.33	51	44	5

Note the tendency to match the letter "D" as the letter "O". This illustrates a desirable feature of this system. While it does produce incorrect matches, there is a visual similarity between the mismatched letters and the correct match. This makes error correction easier.

Closer examination of the results shows that we successfully locate most of the characters as being potential letters, but fail to correctly identify many

Figure 3.33: Figure 3.30 scanned at 200 × 200 dpi.

**tHE QUICK RED
FOX JUMPED
OVER THE LAZY
BROWN DOG**

Figure 3.34: Results of using OCR system on fig. 3.33

```

t H           U I           0
F  O X       J U     P
  O V         T H     L A     Y
                0     N   O O

```

of them since the noise picked up in scanning the data makes finding a match difficult.

The system has greatest difficulty recognizing the larger letters, such as “W” and “M”. This is because there are few letters in our training set similar to these, thus we have comparatively less information about these letters. Consequently our recognition performance depends more on font specific features than would otherwise be the case. This demonstrates that a much larger training set is needed for a practical implementation of such a system.

If we use a image scanned at 300 × 300, we get nearly perfect recognition, which is not surprising since all the images in our training set were taken at this resolution. It suggests that we can improve the results on the scanned images by including the same fonts scanned at different resolutions.

3.7 Using the Update Method

3.7.1 Problem Description

Earlier, in section 2.7, we derived a method for updating our eigenfaces to adjust to those cases where we have a badly represented image. It seems natural to try extending this to the letters.

To see how the method performs in this case, we decided to add 3 more fonts to the k-means classes. Each letter is processed separately in order to find the best class to describe the letter. If the letter is not sufficiently well represented, the eigenletters of that class are then updated using the same technique as for the eigenfaces. The database is converted to use the new eigenletters as described in section 2.7.8.

3.7.2 Algorithm

We use the following algorithm to update the eigenletters.

The actual code is listed as *update.r* in appendix A.2.2.2

```
// We assume that in each case, we are starting with
// the class that best represents the letter
// U = current eigenpictures
// A = current Ave
// I = Image we are working with

Begin
  T = I - A
  y = UTT

  if  $\frac{\|T\|^2 - \|y\|^2}{\|T\|^2}$  too large then letter is
    poorly represented so
    calculate L
    get eigenvectors and eigenvalues of L
    update eigenletters
    (See section 2.7)
  Convert database of class to new
    eigenletters
```

```
        Find representation of I in terms
            of the new eigenletters
    end if

    // Sufficiently well represented
    Add new letter to database

End
```

3.7.3 Results

We want to compare the performance of our updated eigenletters with those from the full recalculation. For the full recalculation, we used the PNN classifier to divide the 13 fonts into 40 classes prior to eigenpicture refinement.

The eigenpicture refinement then leaves us with 38 classes as opposed to the 37 classes we found earlier. Since we don't allow our update method to create new classes, we still use 37 classes for our updated eigenletters.

We cannot compare the classifications directly, as we do not have the same number of classes in each case. So we compare how the two classifications perform in reconstructing and recognising letters.

3.7.3.1 Reconstruction Performance

Figure 3.35 shows the reconstruction of several letters with both sets of eigenletters. Not surprisingly, the reconstructions using the updated eigenpictures are somewhat noisier than those using the full recalculation. However, the differences are generally quite small.

3.7.3.2 Recognition Performance

In order to test the recognition performance, we took the following two test cases.

In figure 3.36, we have a font that was part of the training set when we used only 10 fonts. In figure 3.37, we have the familiar example of a font that was not in our original training set. However, it is one of the fonts we add to our training set, so we expect to see improved performance in this case.

We summarise the results in table 3.6.

Figure 3.38 shows the results of using the fully recalculated set of eigenpictures on figure 3.36. We have perfect recognition, which fits with the earlier results we obtained with the smaller training set.

Table 3.6: Results using updated and fully recalculated eigenletters.

Eigenletter Set	Figure	Percentage Correct	Percentage Unidentified	Percentage Incorrect
Full Recalculation	3.36	100	0	0
	3.37	100	0	2
Updated	3.36	96	0	4
	3.37	92	0	10

In figure 3.39, we used the updated set of eigenpictures to recognise 3.36. While we do not get perfect recognition, there only errors are the misidentification of the letter “l” as the letter “I” and the letter “O” as the letter “Q”. These is not particularly surprising errors, especially as in this font these letters are rather similar. Since we approximate the new eigenletters, we expect some drop in performance.

In figure 3.40, we show the results of using the full set of eigenletters to recognise figure 3.37. Again, we get nearly perfect recognition. The only surprising error is the presence of a “r” between the “m” and the “n”. This is due to incorrectly trying to recognise a subsection of one of the two letters. Although we implemented measures to try and prevent it, it is difficult to eliminate all these errors.

In figure 3.41, we use the updated set of eigenpictures to recognise figure 3.37. Although there are a few more errors, we obtain good results. For example, the system incorrectly identifies “I” as “l” and “l” as “J”, as well as making a few other errors. These errors can again be blamed on the errors introduced by approximating the correct eigenletters.

Comparing the two systems in figure 3.42, we see that, although the fully recalculated set of eigenletters performs better than the updated eigenletters, the difference in performance is quite small. In our tests, the difference in performance is usually between 2% and 5%, and never exceeded 7%.

Thus combining the update method with recalculating the database allows us to describe the original letters well enough to still obtain good results, as well as describing the new letters sufficiently well to obtain reasonable results.

Since we use the approximation a number of times and are still able to obtain acceptable result, it suggests that this approximation is fairly robust. This experiment also illustrates that we probably do not need to use all the letters to calculate the eigenletters. By updating the eigenletters, it is possible to find those letters which make a significant contribution to the eigenletters, which we could then flag and use in the full recalculation. This would

allow us to use far fewer images in the recalculation phases, making it more efficient.

Figure 3.35: Reconstruction using full set of eigenletters and the updated eigenletters.





















Original	Full	Updated
A		
A		
B		
K		
Q		
b		
g		
m		
n		
v		

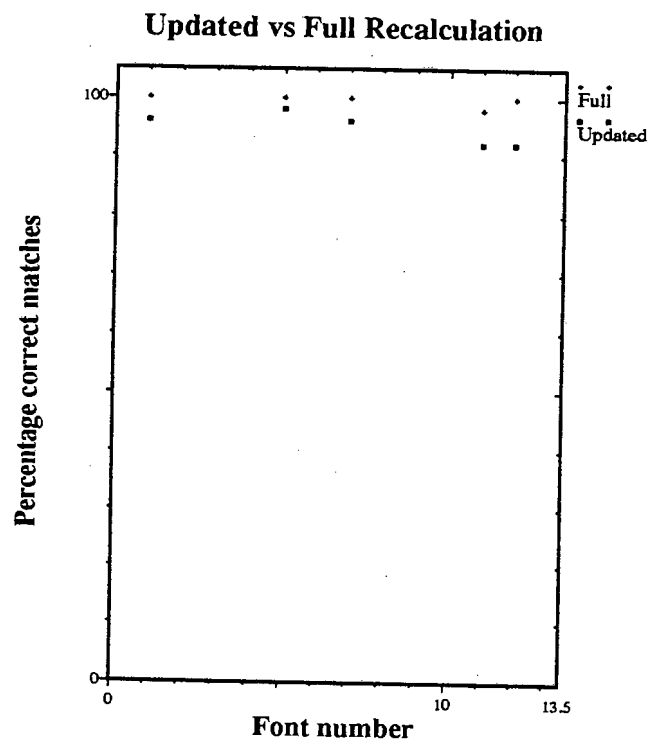
Figure 3.36: Alphabet in original training set.

A B C D E F G H
I J K L M N O P Z
Q R S T U V W X Y
a b c d e f g h
i j k l m n o p q
r s t u v w x y z

Figure 3.37: Alphabet newly included into training set.

A B C D E F G H I v
J K L M N O P Q R S
T U V W X Y Z w x y
a b c d e f g h i j k l z
m n o p q r s t u

Figure 3.42: Comparative performance of updated eigenletters against full recalculation.



Chapter 4

Conclusions

The Karhunen-Loeve transform is very efficient in extracting the correlation between different random images. In the case of facial images we have seen that it is possible to represent 10 000 dimensional images with about 40 values. Admittedly a relatively small dataset of facial images was used, however, it does point to an very substantial reduction, a fact that is corroborated by other studies. One may therefore conclude that the eigenface technique distills the essential information contained in facial images in a very compact form. It is therefore not surprising that many facial recognitions systems are under development (see the FERET report [14]) based on the eigenface technique.

A major issue in this field is the lack of detailed information about many of the systems. Many proprietary systems claim good performance figures, but little or no information is available about the details of the algorithms used and in many cases about the conditions under which they have been tested.

Although we have briefly investigated the possibilities of a facial recognition system based on eigenfaces, our database of some 100 faces is simply too small to allow us to draw any firm conclusions. We note however, that the recent FERET report where facial recognition systems based on a database of some 14 000 images of 1000 different people were tested, indicates that the most reliable system is the one developed by the Media Lab at MIT which is based on eigenfaces. Unfortunately and quite understandably, their published papers do not describe the implementation details in much depth. In fact, we strongly suspect that the details of the implementation are crucial. For instance, it is briefly mentioned in some papers (see [3] or [4]) that it is important to isolate the local features of a face, such as the eyes, nose or mouth, rather than comparing full faces including variable elements like hairstyles, etc. Since no details are given, it is something that has to be

investigated independently.

On a more fundamental level, it is not clear how many eigenfaces are needed for a truly representative dataset, nor, for that matter, is it known how large a representative dataset should be. It is also not known how the dataset is affected by a particular population mixture, for instance, as encountered in South Africa. In our opinion a viable strategy is where the dataset is subdivided into smaller, more homogeneous subsets where each subset has its own set of eigenfaces. Thus, the criterion for subdivision is the efficient representation of the images in each subset, by the eigenfaces associated with that subset.

One should also be concerned with the fact that in practice perfect data will seldom be available. For instance variable lighting conditions should be anticipated. That this is potentially a serious problem is clear if we keep in mind that the grayscale images we work with depend exclusively how light interact with different parts of the face, i.e. a grayscale facial image does not contain any direct information about the physical features of the face. Thus an efficient lighting normalization procedure removing the vagaries of different lighting situations is essential.

Thus, although we have looked in detail at many aspects of a facial recognition system, and even obtained very encouraging test results, many questions remain unanswered, questions that can only be investigated if a large dataset is available. Since it is a very time consuming process to develop a truly representative dataset with multiple images of a large number of people, obtained under different conditions, we decided to investigate some of the questions raised above in a different setting. An additional advantage is that one can test the ideas in a more familiar situation and one where the amount of data is not overwhelming. For these reasons we turned to the development of an OCR system, based on eigenletters.

Although we did not develop a full implementation of an OCR system (for example, we did not pay any attention to mapping characters back onto the page), we did investigate the recognition part in detail. In the first place we investigated the possibility of subdividing the characters into mathematically homogeneous subsets. This indeed led to a major improvement in performance. More precisely, during the process of locating a character in an 'image', one is looking for a general character-like profile. This is best described by the eigenfaces derived from the full dataset. Consequently, we find that the location problem is more reliable if the eigenletters derived from the full dataset are used. However, during the recognition stage, better performance is obtained by using a subdivided dataset.

A thorny issue of the subdivision into subsets is the number of subsets in the subdivision. All subdivision strategies we are aware of assume that the

criteria for deciding on a good number of subsets are well known. Working with characters one might have an intuitive feeling for the appropriate number; in the case of facial images it is not clear at all. However, it is clear from our experience with the OCR system that the number of subsets should not be too large.

The OCR system also showed significant improvement if the recognition rule places more emphasis on the earlier eigenletters. The reason for this is that the early eigenletters contain the information characteristic of the particular subset, i.e. the higher eigenletters contain more information about the differences between different fonts representing the same letter. With eigenfaces the emphasis is likely to be reversed—the first eigenfaces contain general information about the faces in the subset. Individual characteristics, which are required for identification, are described by the higher order eigenfaces.

Although our goal with the OCR system was to learn more about the eigenpicture technique and to try out certain ideas which we are not able to test on eigenfaces without a representative database, our investigations indicate that it might be possible to develop a commercial OCR system based on eigenpictures. A comparison of our still crude OCR system with some commercial software is encouraging.

It should also be noted that most OCR systems are extremely specialised in the sense that the recognition rules are based on the assumption that they have to distinguish between letters, and thus these techniques do not generalise well to other recognition problems (see for example Impedovo [24]). In comparison, the eigenpicture technique is much more general and can be applied to any situation where there is a strong correlation between the images.

An OCR system is in many aspects very different from a facial recognition system – the different characters certainly show more variation than different faces. Therefore, not all the ideas developed for the OCR system will necessarily apply equally well to eigenfaces. However, if the significant improvement obtained by a classification of the letters is any indication, we believe that a classification of faces, based on their eigenface representation, should also lead to a marked improvement in performance. Of course there is no guarantee that this will be the case and only further research will answer this question... We also believe that the OCR application has been a valuable source of experience in dealing large training sets.

We are already in the process of investigating many of the questions raised above. This is the topic of a further study.

Appendix A

Code for Algorithms

We present some of the programs used in our experiment. The sample presented though it not in any way intended to be complete as much of the detail involved in linking the various sections together has not been included since we felt it was not particularly informative.

A.1 Face Recognition

The programs in this section were all written for Matlab.

A.1.1 Calculating Eigenfaces

eigface.m

```
M=64
Meig=40
Dim=112;
% working with 112 x 112 images
Images=zeros([Dim*Dim M]);
U=zeros([Dim*Dim Meig]);
for I=1:M
    name=sprintf('112x112/training/face%dn',I)
    [tempIm,map]=gifread(name);
    Images(:,I)=tempIm(:);
end

A=zeros(size(Images(:,1)));
```

```

for I=1:M
    A=A+Images(:,I);
end
A = A./M;
for I=1:M
    Images(:,I)=Images(:,I)-A;
    Images(:,I)=Images(:,I)./norm(Images(:,I));
end

L=zeros(M);
L=Images'*Images;
[V,Lamba] = eig(L);
e=diag(Lamba);
[eigval,index]=sort(e);
e=eigval(M:-1:1);
Vnew=V;

for I=1:M
    V(:,M-I+1)=Vnew(:,index(I));
end
for J=1:Meig
    U(:,J)=Images*V(:,J);
    U(:,J)=U(:,J)./sqrt(e(J));
end
eigval=e./M;
clear Vnew Lamba tempIm name

```

A.1.2 Representing and Reconstructing Images

process.m

```

ImageT=zeros(size(A));
ImageNo=input('Image to work with:');
temp=zeros(Dim);
name=sprintf('112x112/training/face%dn.gif',ImageNo);
[Images,map]=gifread(name);

Y= U'*((Images(:)-A));
ImageT=(U*Y)+A;
figure(1); % reconstruction
colormap(map)
temp(:)=ImageT;

```

```

imshow(temp)

figure(2); % original
colormap(map)
temp(:)=Images(:);
imshow(temp)

```

A.1.3 Recognising an Image

findmatch.m

```

% Ys is our database
ImageT=zeros(size(Images(:,1)));
ImageNo=input('Image to work with:');
name=sprintf('112x112/test/face%dtm',ImageNo);
[ImageN,map1]=gifread(name);
Y= U'*((ImageN(:)-A));
% Search for best match
error=sqrt(norm(ImageNo(:))^2-Y'*Y);
if (error)/norm(ImageNo(:)) > 0.2
    disp('Not a face')
end;
minerror=Inf;
error=0;
match=0;

for i=1:M
    error=norm(Y-Ys(:,i));
    if error<minerror
        minerror=error;
        match=i;
    end;
end;

if (maxerror>2000)
    disp('Not in database?')
end;
figure(1)
Immtch=(U*Ys(:,match))+A;
temp(:)=Immtch;
imshow(temp);
colormap(map);

```

A.1.4 Locating the Face

locface.m

```

name=input('File to work with : ');
[X,map]=gifread(name);
%pad image;
Q=zeros([224 112]);
Q(56:167,:)=X;
P=zeros(224);
P(56:167,:)=Q';
P=P';
cX=112;
cY=112;
%image zero padded
currentmax=[-Inf,0,0,0];
% We assume face near centre

for scale=40:1:60
    for stepx=-25:1:25
        for stepy=-25:1:25
            face=imcrop(P,[cX-stepx-scale,
                           cY-stepy-scale,
                           2*scale-1,
                           2*scale-1]);
            imfull=imresize
                (face,[Dim Dim],'bilinear');
            alpha=(U(:,1:10))'*(imfull(:)-A);
            Prob=0;
            temp=0;

            for k=1:10
                temp=temp+((alpha(k)*
                             alpha(k))/eigval(k));
            end

            Prob=(-1/2*temp);
            temp=(work*work'-alpha'*alpha)/(-2*recp);
            Prob=Prob+temp;

```

```

        if (Prob>currentmax(1))
            currentmax(1:1:4)=
                [Prob,stepx,stepy,scale];
            maxface=imfull;
        end;
    end;
end;

% Got best location
figure;
imshow(maxface,map);

```

A.1.5 Updating the Eigenfaces

eigupdate.m

```

% Got original eigenfaces, now we update it by 1
temp=gifread('112x112/training/newface.gif');
ImageNew=temp(:);
Anew=(M*A + ImageNew)./(M+1);
Uold=U;
T=(ImageNew-Anew)./norm(ImageNew-Anew);
T=T./sqrt(M+1);
L=zeros([Meig+1 Meig+1]);

for i=1:Meig
    L(i,i)=eigval(i);
    U(:,i)=U(:,i)./sqrt(eigval(i));
    L(i,Meig+1)=T'*U(:,i);
    L(Meig+1,i)=L(i,Meig+1);
end

L(Meig+1,Meig+1)=T'*T;
[V,Lamba]=eig(L);
e=diag(Lamba);
[e,index]=sort(e);
eigval=e(Meig+1:-1:1)
Vnew=V;

for I=1:Meig+1
    V(:,Meig-I+2)=Vnew(:,index(I));

```

```

end
Unew=Uold;
for J=1:Meig
    Unew(:,J)=U*V(:,J);
    Unew(:,J)=Unew(:,J)./norm(Unew(:,J));
end
% Unew now the new eigenfaces.

```

convert.m

```

% Convert database (Ys)
Conv=Unew'*U;
B=Unew'*(A-Anew);
for i=1:M
    Ys(:,i)=Conv*Ys(:,i)+B;
end

```

A.2 OCR

The programs in this section were written for R-Lab.

A.2.1 Classification

For the classification programs, we assume that we have calculated the letter's eigenpictures. We do this by slightly modifying eigface.m in appendix A.1.1 to deal with letters.

A.2.1.1 Leader Principle

classify.r

```

J=1;
NumKept=20;
for (I in 1:M) {
    if (sqrt(Ys[1:NumKept;I]'*Ys[1:NumKept;I])>0.6) {
        database[1:NumKept;J]=Consts[I]*Ys[1:NumKept;I];
        database[NumKept+1;J]=I;
        J=J+1; }
}

```

```

Origdelta=1e+03;
// Dsize is database size
Dsize=J-1;
compcount=0;
// Nmax = maximum number of classes at this step
NMax=2;
Delta=Origdelta;
// P = assignment vector
P=zeros(Dsize,1);
// Distance [I,J]=Distance between elements
// I and J where I > J;
J=1;
Distance=zeros(Dsize,Dsize);
for (I in 1:Dsize-1) {
    p=database[1:NumKept;(I+1):Dsize] -
        database[1:NumKept;I]*ones(1,Dsize-I);
    Distance[(I+1):Dsize;I]=sqrt(diag(p'*p));
}

while ((NMax<M/2)&&(NMax<80) {
    if (NMax>2) {
        Qold=Q;
        Q=zeros(NumKept+1,NMax);
        P=zeros(size(P));
        Q[:,1:NMax-1]=Qold[:,1:NMax-1];
        P[Q[NumKept+1;1:NMax-1]]=1:NMax-1;
        Q[1:NumKept;NMax]=
            database[1:NumKept;Furtherest];
        Q[NumKept+1;NMax]=Furtherest;
        P[Furtherest]=NMax;
        Furtherest=0;
        MaxDistance=-1;
    }
    else
        Q=zeros(NumKept+1,NMax);
        P=zeros(size(P));
        // Q contains initial class elements.
        K=maxi(max(Distance));
        J=maxi(max(Distance'));
        P[K]=1;
        Q[1:NumKept;1]=
            database[1:NumKept;K];
        Q[NumKept+1;1]=K;

```

```

P[J]=2;
Q[1:NumKept;2]=
    database[1:NumKept;J];
Q[NumKept+1;2]=J;
Furtherest=0;
MaxDistance=-1;
// First 2 classes assigned.
}

for (J in 1:Dsize) {
    if (P[J]==0) {
        // unassigned, so search for class
        ThisD=zeros(1,NMax);
        for (K in 1:NMax) {
            if (J>Q[NumKept+1;K]) {
                ThisD[K]=Distance[J;Q[NumKept+1;K]];
            }
            else {
                ThisD[K]=Distance[Q[NumKept+1;K];J]; }
        }
        Nearest=mini(ThisD);
        P[J]=Nearest;
        if (ThisD[Nearest]>MaxDistance) {
            MaxDistance=ThisD[Nearest];
            Furtherest=J;
            // Keep track of Maximum Distance
            // from class.
            // Element furtherest from its class seed
            // becomes next new seed }
        }
    }
    compcount=compcount+1;
    completed[compcount;]=[NMax,MaxDistance];
    DoneP[:,compcount]=P;
    NMax=NMax+1;
}

```

A.2.1.2 K-means Refinement

k-means.r

```
NumberNMaxes=max(size(NMaxes));
```

```

for (K in 1:NumberNMaxes) {
    NMax=NMaxes[K];
    E=zeros(NMax,1);
    Q=zeros(NMax,1);
    P=DoneP[:,NMax-min(completed[1;])+1];
    Avg=zeros(NumKept,NMax);

    for (I in 1:Dsize) {
        R=P[I];
        Q[R]=Q[R]+1;
        Avg[:,R]=Avg[:,R]+database[1:NumKept;I];
    }
    for (I in 1:NMax) {
        R=Q[I];
        Avg[:,I]=Avg[:,I]./R;
    }

    for (I in 1:Dsize) {
        R=P[I];
        p = database[1:NumKept;I]-Avg[:,R];
        tempD=p'*p;
        E[R]=E[R]+tempD;
    }

    D=sum(E);
    I=0;
    IT=0;

    while (IT <= Dsize) {
        I=I+1;
        if (I > Dsize) { I=I-Dsize; }
        R=P[I];
        Size=Q[R];

        if (Size > 1) {
            p=database[1:NumKept;I]-Avg[:,R];
            A=(Size/(Size-1))*(p'*p);
            info=zeros(1,NMax);
            for (J in 1:NMax) {
                Size=Q[J];
                p=database[1:NumKept;I]-Avg[:,J];
                F[J]=p'*p;
            }
        }
    }
}

```

```

        if (J!=R) {
            info[J]=(Size/(Size+1))*F[J];
        else
            info[J]=A+1; }
    }

    B=min(info);
    if (B<A) {
        W=mini(info);
        IT=0;
        E[R]=E[R]-A;
        E[W]=E[W]+B;
        D=D+B-A;
        moving=database[1:NumKept;I];
        Avg[;R]=( (Q[R]*Avg[;R]) - moving)
                /(Q[R]-1);
        Avg[;W]=( ( Q[W]*Avg[;W]) + moving)
                /(Q[W]+1);

        P[I]=W;
        Q[R]=Q[R]-1;
        Q[W]=Q[W]+1;
    else
        IT=IT+1; }
}
}
DoneD[K]=D;
DoneQ[;K]=Q;
DoneE[;K]=E;
DoneP2[;K]=P;
}

```

A.2.1.3 PNN Based Classifier

The implementation of the kd-tree here is very simplistic and not at all efficient. We include this merely for the sake of completeness.

kdtree.r

```

CreateTree = function (depth,level) {
    // Do recursive create
    if (!(exist(depth))) { depth=1; }
    if (!(exist(level))) { level=1; }
}

```

```

    if (depth > 0) {
        A.R = CreateTree (depth-1,level+1);
        A.L = CreateTree (depth-1,level+1);
    }
    else
        A.L=0;
        A.R=0;
}

A.Depth = depth;
A.Var = level;
A.BranchVal = 0;
A.Items=0;
A.Value=0;
return A; }

```

```

AddTree = function (item,tree) {
    // Do recursive add
    A = tree;
    A.Items = A.Items+1;
    if (exist((A.L.L))) {
        if (item[A.Var] < A.BranchVal) {
            A.L=AddTree (item,A.L);
        }
        else
            A.R=AddTree (item,A.R);
    }
    else
        A.Value[A.Items;]=item;
}
return A; }

```

```

DelTree = function (item,tree) {
    // Assume item is actually in the tree
    A = tree;
    A.Items = A.Items-1;
    if (exist((A.L.L))) {
        if (item[A.Var] < A.BranchVal) {
            A.L=DelTree (item,A.L);
        }
        else
            A.R=DelTree (item,A.R);
    }
    else
        i=1;
}

```

```

while ( sum(A.Value[i;]!=item)>0) {
    i=i+1;
    if (i>A.Items+1) {
        disp ("Item not a member of tree");
        return A;
    }
}
A.Value[i;]=[];
}
return A; }

RebalanceTree = function (tree) {
    // Here we rebalance the tree.
    // We will do so very crudely
    // By dumping all elements into an array and
    // creating a suitably balanced tree

    A = tree;
    if (exist(A.L.L)) {
        // Has subtrees, so we need to rebalance
        // split on the median
        Total=0;
        for (i in 1:2^A.Depth) {
            Data=DumpBucket(A, i);
            TotalData[Total+1:Total+Data.Size;]=
                Data.Value;
            Total=Total+Data.Size;
        }
        B=CreateTree(A.Depth,A.Var);
        B.BranchVal = median(TotalData[:,A.Var]);
        for (i in 1:Total) {
            B=AddTree(TotalData[i;],B);
        }
        clear(A);
        A=B;
        A.R = RebalanceTree(A.R);
        A.L = RebalanceTree(A.L);
    }
    return A; }

```

```

DumpBucket = function (tree, bucket) {
  // returns the contents of the
  // x'th bucket of the tree

  A = tree;
  if (exist(A.Depth)) {
    midpoint = 2^(A.Depth-1);
  } else
    midpoint=0;
  if ( exist(A.L.L) ) {
    if (bucket <= midpoint) {
      Contents = DumpBucket(A.L, bucket);
    } else
      Contents = DumpBucket(A.R, bucket - midpoint);
    }
  else
    Contents.Value = A.Value;
    Contents.Size = A.Items;
  }
  return Contents; }

```

This program implements the PNN classification.

pnn.r

```

J=1;
NumKept=20;
for (I in 1:M) {
  if (sqrt(Ys[1:NumKept;I])*
    Ys[1:NumKept;I])>0.6) {

    database[1:NumKept;J]=Consts[I]
      *Ys[1:NumKept;I];
    database[NumKept+1;J]=1;
    // No. of Items
    J=J+1;
  }
}
Dsize=J-1;
// We now have database of all
// well represented items.

TreeDepth= int(log(Dsize/12)/log(2));

```

```

DataTree = CreateTree(TreeDepth);
for (i in 1:Dsize) {
    DataTree=AddTree(database[:,i]',DataTree); }
poss=0;
InB = 0;

while (DataTree.Items > 40) {
    clear(poss);
    poss=zeros(1,2*NumKept+3);
    Totalposs = 0; // Number of poss.

    for (i in 1:2^(DataTree.Depth)) {
        InBucket=DumpBucket(DataTree,i);
        clear(Bucket);
        SSize=InBucket.Size;
        Bucket=InBucket.Value; // Get Data
        clear(InBucket);
        clear (InB);
        InB = zeros(MtoFind,2*NumKept+3);
        InB[1;2*NumKept+3]=inf();

        for (j in 1:SSize-1) {
            Xa=Bucket[j;1:NumKept];
            Na=Bucket[j;NumKept+1];
            for (k in (j+1):SSize) {
                Xb=Bucket[k;1:NumKept];
                Nb=Bucket[k;NumKept+1];

                dist = ((Na*Nb)/(Na+Nb))*
                    vnorm(Xa-Xb,2)^2;

                if (dist<InB[1;2*NumKept+3]) {
                    InB[1;1:NumKept]=Xa;
                    InB[1;NumKept+1]=Na;
                    InB[1;(NumKept+2):(2*NumKept+1)]=Xb;
                    InB[1;2*NumKept+2]=Nb;
                    InB[1;2*NumKept+3]=dist;
                }
            }
        }
    }
}

```

```

// Insert amongst previous merges
if (Totalposs > 0) {
  // Not first addition
  k = 0;
  InBC = 1;
  j=1;
  oldposs=poss;
  while (k < Totalposs+MtoFind) {
    k=k+1;
    if ((InBC<MtoFind+1)) {
      // Still from InB to consider

      if (j < Totalposs+1) {
        // also from oldposs
        if (InB[InBC;2*NumKept+3] <
            oldposs[j;2*NumKept+3]) {

          poss[k;]=InB[InBC;];
          InBC=InBC+1;
          // take from InB
        else
          poss[k;]=oldposs[j;];
          j=j+1; // Take from oldposs
        }

        else
          poss[k;]=InB[InBC;];
          InBC=InBC+1;
        }

        else
          poss[k;]=oldposs[j;];
          j=j+1;
        }
      }

    Totalposs=Totalposs+MtoFind;
  else
    // is first addition, so add.
    poss[Totalposs+1:Totalposs+MtoFind;] =
      InB;
    Totalposs=Totalposs+MtoFind;
  }

```

```

    }
    // Do merges (only do half)
    for (i in 1:round(Totalposs/2)) {
        Ia=poss[i;1:NumKept+1];
        Xa=Ia[1:NumKept];
        Na=Ia[NumKept+1];
        Ib=poss[i;NumKept+2:2*NumKept+2];
        Xb=Ib[1:NumKept];
        Nb=Ib[NumKept+1];
        New=(Na*Xa+Nb*Xb)/(Na+Nb); // New median
        New[NumKept+1]=Na+Nb;
        DataTree = DelTree(Ia,DataTree);
        DataTree = DelTree(Ib,DataTree); // Remove old
        DataTree = AddTree(New,DataTree); // Add new
    }
    if (int(DataTree.Items/(2^(DataTree.Depth)))
        < 8) {
        if (DataTree.Depth > 0) {
            B = DataTree;
            clear(DataTree); // Resize Tree
            DataTree = CreateTree(B.Depth-1);
            for (i in 1:2^(B.Depth)) {
                Data=DumpBucket(B, i);
                for (j in 1:Data.Size) {
                    DataTree = AddTree
                        (Data.Value[j;],DataTree);
                }
            }
        }
    }
    DataTree = RebalanceTree (DataTree);
    // Rebalance tree
}
// Now we assign every item to its nearest group
// We then ready to either feed into k-means
// or eigclassify
k=0;
Medians=zeros(1,20);
for (i in 1:2^DataTree.Depth) {
    Bucket=DumpBucket(DataTree,i);
    Data=Bucket.Value;

```

```

    for (j in 1:Bucket.Size) {
        k=k+1;
        Medians[k;]=Data[j;1:NumKept];
    }
}

P=zeros(1,M);
for (i in 1:M) {
    test=Consts[i]*Ys[1:NumKept;i]';
    for (j in 1:DataTree.Items)
        { Mtest[j;]=test; }
    Z=Medians-Mtest;
    dist=sqrt(diag( Z*Z' ));
    P[i]=mini(dist);
}

```

A.2.1.4 Eigenpicture Refinement

eigclassify.r

```

// We scrap all classes with less than 5 element
MinClassSize=5;
M=520;
// We have vector lets which contains
// all lets, and points to info on
// letter position in image.
I=1;
for (L in lets) {
    for (J in 1:13) {
        sprintf(name,"letters/data/%s%d.pgm",L,J);
        temp=loadpgm(name);
        close(name);
        Images[;I]=temp[:];
        I=I+1;
    }
}

K=0;
List=zeros(NMax,(Dim*Dim));
for (I in 1:NMax) {
    Q[I]=0;
    A[;I]=zeros(size(Images[;1]));
    for (J in 1:M) {

```

```

    if (P[J]==I) {
        Q[I]=Q[I]+1;
        A[:,I]=A[:,I]+Images[:,J];
        List[I;Q[I]]=J;
    }
}

if (Q[I]>MinClassSize) {
    K=K+1; // One more valid class
    A[:,I]=A[:,I]./Q[I];
    Theta=zeros((Dim*Dim),Q[I]);
    Consts=zeros(1,Q[I]);
    for (J in 1:Q[I]) {
        Theta[:,J]=Images[:,List[I;J]]-A[:,I];
        Consts[J]=vpnorm(Theta[:,J],2);
        Theta[:,J]=Theta[:,J]./Consts[J];
    }

    L=zeros(Q[I],Q[I]);
    L=Theta'*Theta;
    Eig=eig(L);
    V=real(Eig.vec);
    e=real(Eig.val);
    e=e./Q[I];
    sorted=sort(e);
    e=sorted.val[Q[I]:1:-1];

    Vnew=V;
    for (J in 1:Q[I]) {
        V[:,Q[I]-J+1]=Vnew[:,sorted.ind[J]];
    }
    clear (Vnew,sorted);
    U=zeros((Dim*Dim),Q[I]-1);
    for (J in 1:Q[I]-1) {
        U[:,J]=Theta*V[:,J]./sqrt(Q[I]*e[J]);
    }
}

```

```

        sprintf(name,"vector13/class%i.mat",K);
        Ave=A[:,I];
        Size=Q[I];
        MemberList=List[I,:];
        flag=0;
        NumKept=Size;
        write(name, U, Ave, Size,
              MemberList, e, NumKept);
        close(name);
        clear(Ave,Size,MemberList);
    }
}

Pold=P;
Qold=Q;
NMax=K // we now have only
        // K classes of significance
Class=3; // We try 3 eigenpictures.
clear (V);
for (I in 1:NMax) {
    sprintf(name,"vector13/class%i.mat",I);
    read(name,test);
    V.[I]=test.U[:,1:Class];
    A[:,I]=test.Ave;
}

clear(classify);
for (I in 1:M) {
    for (J in 1:NMax) {
        Theta=Images[:,I]-A[:,J];
        temp=real(V.[J]')*Theta;
        classify[J]=
            (Theta'*Theta)-(temp'*temp);
    }
    P[I]=mini(real(classify));
}
// Repeat 1st part to recalculate eigenletters

```

At the end of this program, we have finished the classification process.

A.2.2 OCR Systems

Since our early efforts were merely a subsection of the later improved system, we felt there was no point in including both systems. We thus include only the final system.

A.2.2.1 Improved Recognition Rule

This program creates the filters and does some other miscellaneous processing required before we can start the actual recognition process.

setup.r

```

Class=3;
for (I in 1:NMax) {
    sprintf(name,"classes/class%i.mat",I);
    read(name,data);
    V.[I]=data.V;
    V.[I].Ave=data.Ave;
    V.[I].e=data.e;
    V.[I].NumKept=data.NumKept;

    Len=max(size(data.e));
    List=data.MemberList;
    V.[I].recp=sum(data.e[Class+1:Len])
        /(Dim*Dim-Class);
    sprintf(name,"classes/data%i.mat",I);
    read(name,data);

    database.[I]=data.Ys;
    database.[I].Size=data.Ysize;
    database.[I].List=List[1:data.Ysize];
    database.[I].AvgErr=data.AvgErr;
    database.[I].match=data.match;
    database.[I].x=data.xlocs;
    database.[I].y=data.ylocs;
    database.[I].Variance=
        sqrt(V.[I].e[1:V.[I].NumKept]);
}

```

```

//We now need to calculate filter
filt=( sum((V.[I]'!=0)))';
filt = (filt > 0);
//We now have binary filter, which we must blur
filt2=zeros(Dim,Dim);
for (J in 1:Dim) {
    filt2[:,J]=filt[(J-1)*Dim+1:J*Dim];
}
for (J in 1:Dim) { // Horizontal
    for (K in 6:Dim-5) {
        filt3[J;K]=sum(filt2[J;K-5:K+5])/4 +
            filt2[J;K];
        if (filt3[J;K]>1) { filt3[J;K]=1; }
    }
    for (K in 1:5) {
        filt3[J;K]=filt3[J;6]-1/K;
        if (filt3[J;K]<0) { filt3[J;K]=0; }
        filt3[J;Dim-K+1]=filt3[J;Dim-5]-1/K;
        if (filt3[J;Dim-K+1]<0) {
            filt3[J;Dim-K+1]=0; }
    }
}

file2=filt3;
for (K in 1:Dim) { // Vertical
    for (J in 6:Dim-5) {
        filt3[J;K]=sum(filt2[J-5:J+5;K])/4 +
            filt2[J;K];
        if (filt3[J;K]>1) { filt3[J;K]=1; }
    }
    for (J in 1:5) {
        filt3[J;K]=filt3[6;K]-1/J;
        if (filt3[J;K]<0) { filt3[J;K]=0; }
        filt3[Dim-J+1;K]=filt3[Dim-5;K]-1/J;
        if (filt3[Dim-J+1;K]<0) {
            filt3[Dim-J+1;K]=0; }
    }
}
filt2=filt3[:];
V.[I].filter=filt2;
}
// Filter complete
read("classesone/class1.mat",data);

```

```

Vloc=data.V[:,1:10];
Vloc.Ave=data.Ave;
Vloc.e=data.e;
Vloc.NumKept=40;
read("classesone/data1.mat",data);
Vloc.AvgErr=data.AvgErr;

clear(data);
IntrestMargin = 20;
SearchMargin= 6;
VarMargin = 300;
AcceptMargin=900;
Stepsize=4;
ClassCon = 3;
InClass = 1; // Setup various constants

```

findchar.r

```

sprintf (name,"test-images/scana%i.pgm",Z);
ImageScan = loadpgm (name);
close(name);
sprintf (name,"test-images/lines%i.mat",Z);
read (name,lines); // load line info.
for (J in 1:200) {
  for (K in 1:200) { Result[J;K]=" "; }
} // Initialise results array (very badly done)

Result.Prob=zeros(200,200);
Result.X=-100*ones(200,200);
Result.Y=Result.X;
Result.Width=-Result.X;
Result.Weight=Result.Prob;

for (I in 1:-1:-1) {
  newsize= int ((1+I/10)*size(ImageScan));
  PageOrig=imresize(ImageScan,newsize);
  Page=255*ones(size(PageOrig)+[60,60]);
  // pad page
  Page[31:PageOrig.nr+30;31:PageOrig.nc+30] =
    PageOrig;

```

```

XMax=Page.nc;
YMax=Page.nr;
YstepMax=(int((YMax-80)/Stepsize)+1) *
    Stepsize+40;
while (YstepMax-(YMax-40)>=Stepsize) {
    YstepMax=YstepMax-Stepsize; }
XstepMax=(int((XMax-80)/Stepsize)+1) *
    Stepsize+40;
while (XstepMax-(XMax-40)>=Stepsize) {
    XstepMax=XstepMax-Stepsize; }
XstepMin=40;
YstepMin=40;
ListCount=1;
List.X=0;
List.Y=0;
K=0;
Im=0;

for (Y in YstepMin:YstepMax:Stepsize) {
    for (X in XstepMin:XstepMax:Stepsize) {
        ImOrig = Page[Y-24:Y+25;X-24:X+25][:];
        Im2 = ImOrig-Vloc.Ave;
        Im=Im2;
        y = Vloc*Im; // Try initial classification
        Err = ( (Im*Im)-(y*y) )/(y*y);
        if (Err < TotalInterestMargin) {
            List.X[ListCount]=X; // possible
            List.Y[ListCount]=Y; // letter
            ListCount=ListCount+1; }
    }
}

ListCount=ListCount-1;
MinScale=48;
MaxScale=52;
Cl=0;

for (K in 1:ListCount) {
    XOrig=List.X[K];
    YOrig=List.Y[K];

```

```

ClassList=zeros(1,ClassCon);
ClassList.Err=inf()*ones(1,ClassCon);
ClassList.Im=zeros(size(ImOrig),ClassCon);
ClassList.X=zeros(1,ClassCon);
ClassList.Y=zeros(1,ClassCon);
ClassList.Scale=zeros(1,ClassCon);

clear(Cl);
for (J in 1:NMax) {
    Cl.[J]=zeros(1,InClass);
    Cl.[J].Err=inf()*ones(1,InClass);
    Cl.[J].Im=zeros(size(ImOrig),InClass);
    Cl.[J].X=zeros(1,InClass);
    Cl.[J].Y=zeros(1,InClass);
    Cl.[J].Scale=zeros(1,InClass);
}

for (X in -2:2) {
    for (Y in -2:2) {
        for (Scale in MinScale:MaxScale:2) {
            ImOrig = Page [
                int((YOrig+Y)-(Scale/2-1))
                :int((YOrig+Y)+(Scale/2));
                int((XOrig+X)-(Scale/2-1))
                :int((XOrig+X)+(Scale/2)) ];
            Im = imresize(ImOrig,[50;50]);
            ImOrig = Im[:];

            for (J in 1:NMax) {
                Im2 = ImOrig - V.[J].Ave;
                Im = imfilter (Im2,V.[J].filter);
                y = V.[J]'*Im; // Get y
                Err=( (Im'*Im)-(y'*y) )/(y'*y);
                Match=0;
                j=0;

                while ((!Match)&&(j<InClass)) {
                    j=j+1;
                    Match=(Err<Cl.[J].Err[j]);
                }

                if (Match) {

```

```

    for (i in InClass:(j+1):-1) {
        Cl.[J][i] =
            Cl.[J][i-1];
        Cl.[J].Err[i] =
            Cl.[J].Err[i-1];
        Cl.[J].Im[i] =
            Cl.[J].Im[i-1];
        Cl.[J].X[i] =
            Cl.[J].X[i-1];
        Cl.[J].Y[i] =
            Cl.[J].Y[i-1];
        Cl.[J].Scale[i] =
            Cl.[J].Scale[i-1];
    }

    Cl.[J][j] = J;
    Cl.[J].Err[j] = Err;
    Cl.[J].Im[j] = Im;
    Cl.[J].X[j] = X;
    Cl.[J].Y[j] = Y;
    Cl.[J].Scale[j] = Scale;
}
}
}
}
}
for (J in 1:NMax) {
    for (X in 1:InClass) {
        j=0;
        Match=0;
        while ((!Match)&&(j<ClassCon)) {
            j=j+1;
            Match=Cl.[J].Err[X] <
                ClassList.Err[j]; }

        if (Match) {

```

```

    for (i in ClassCon:(j+1):-1) {
        ClassList[i] =
            ClassList[i-1];
        ClassList.Err[i] =
            ClassList.Err[i-1];
        ClassList.Im[;i] =
            ClassList.Im[;i-1];
        ClassList.X[i] =
            ClassList.X[i-1];
        ClassList.Y[i] =
            ClassList.Y[i-1];
        ClassList.Scale[i] =
            ClassList.Scale[i-1];
    }
    ClassList[j] =
        Cl.[J][X];
    ClassList.Err[j] =
        Cl.[J].Err[X];
    ClassList.Im[;j] =
        Cl.[J].Im[;X];
    ClassList.X[j] =
        Cl.[J].X[X];
    ClassList.Y[j] =
        Cl.[J].Y[X];
    ClassList.Scale[j] =
        Cl.[J].Scale[X];
}
}
}

```

```

for (i in ClassCon:1:-1) {
    J=ClassList[i];
    Im = ClassList.Im[;i];
    Err=ClassList.Err[i];
    X=ClassList.X[i];
    Y=ClassList.Y[i];
    Scale=ClassList.Scale[i];
    y = V.[J]*Im; // Get y
    if ( Err < SearchMargin *
        database.[J].AvgErr) {
        MinMatch=inf();
        MinL=0;
    }
}

```

```

for (L in 1:database.[J].Size) {
    testy=database.[J][;L];
    My=y;
    Ty=testy;

    for (p in 1:V.[J].NumKept) {
        My[p]=sqrt(V.[J].e[p])*My[p];
        Ty[p]=sqrt(V.[J].e[p])*Ty[p];
    }

    Match=vpnorm(Ty-My,2);
    MinErr=vpnorm(testy-y,2);

    if ((Match<MinMatch) &&
        (MinErr<AcceptMargin)) {
        MinMatch=Match;
        MinL=L;
    }
}

if (MinMatch<VarMargin) {
    testy=database.[J][;MinL];
    L=MinL;
else
    continue;
}

OldErr=vpnorm(testy-y,2);
MinErr=MinMatch;
ListNum = database.[J].List[L];
LetterNum=int((ListNum-1)/10)+1;
letter=database.[J].match[L];
Font=ListNum-(LetterNum-1)*10;
AcX = (X+XOrig-30)/(1+I/10);
AcY = (Y+YOrig-30)/(1+I/10);

ResultX=round(((AcX)
    +database.[J].x[L])/25) + 1;
ResultY=0;
while (AcY<lines[ResultY]) {
    ResultY=ResultY+1;
}
// Okay we have potential mapping to page

```

```

LetWidth=2*(25-database.[J].x[L]);
TIm=V.[J]*testy+V.[J].Ave;
TIm=TIm./255;
Weight=(TIm'*TIm)/(max(size(TIm)))*100;
replace=0;
for (PX in ResultX-1:ResultX+1) {
    CurrX=Result.X[PY;PX];
    if ((!replace) &&
        (abs((XOrig+X)-CurrX)<15))) {
        ResultX=PX;
        replace=1;
    }
}

if (Result[ResultY;ResultX]==" ") {
    // no previous letter
    Result[ResultY;ResultX] =
        letter;
    Result.Prob[ResultY;ResultX] =
        MinErr;
    Result.X[ResultY;ResultX] =
        XOrig+X;
    Result.Y[ResultY;ResultX] =
        YOrig+Y;
    Result.Width[ResultY;ResultX] =
        LetWidth;
    Result.Weight[ResultY;ResultX] =
        Weight;
}
else
    // replace previous letter?
    Margin = MinErr - Result.Prob
        [ResultX;ResultY];

```



```

V.[I]=0;
V.[I].V=data.V;
V.[I].Ave=data.Ave;
V.[I].e=data.e;
V.[I].NumKept=data.NumKept;

Len=max(size(data.e));
List=data.MemberList;
sprintf(name,"classes/data%i.mat",I);
read(name,data);

database.[I]=0;
database.[I].Ys=data.Ys;
database.[I].Size=data.Ysize;
database.[I].List=List[1:data.Ysize];
database.[I].AvgErr=data.AvgErr;
database.[I].Err=data.Err;
database.[I].match=data.match;
database.[I].xlocs=data.xlocs;
database.[I].ylocs=data.ylocs;
} // Have old info

K=1;
NewImages=zeros(Dim*Dim,156);
for (J in lets) {
    for (I in 11:13) {
        sprintf(name,"letters/data/%s%d.pgm",J,I);
        temp=loadpgm(name);
        close(name);
        NewImages[:,K]=temp[:];
        K=K+1; // Load new letters
    }
}
K=K-1;
NNew=K;

for (K in 1:NNew) {
    minErr = inf();
    for (I in 1:NMax) {
        T=NewImages[:,K]-V.[I].Ave;
        Y=V.[I].V'*T;
        Err=((T'*T)-(Y'*Y))/(T'*T);
    }
}

```

```

    if (Err < minErr) {
        minErr=Err;
        bclass=I;
        by=Y;
    }
}

I=bclass; // Have best class
if (sqrt(minErr)>0.3) {
    // need to update V.
    Anew=(database.[I].Size*V.[I].Ave +
        NewImages[:,K])/(database.[I].Size+1);
    newLsize=V.[I].NumKept+1;
    L=zeros(newLsize,newLsize);
    eigval=V.[I].e;
    U=V.[I].V;
    T=NewImages[:,K]-Anew;
    Th = T./((sqrt(database.[I].Size+1)) *
        vnorm(T,2));

    for (i in 1:newLsize-1) {
        L[i;i]=eigval[i];
        U[:,i]=U[:,i]*sqrt(eigval[i]);
    }

    for (i in 1:newLsize-1) {
        L[newLsize;i]=Th'*U[:,i];
        L[i,newLsize]=L[newLsize;i];
    }

    L[newLsize;newLsize]=Th'*Th;
    Eig=eig(L);
    e=real(Eig.val);
    vecs=real(Eig.vec);
    clear(Eig);
    sorted=sort(e);
    e=sorted.val;
    index=sorted.ind;
    clear (sorted);
    eigval=e[newLsize:1:-1];
    vecsnew=vecs;
}

```

```

for (i in 1:newLsize) {
    vecs[;newLsize-1-i+2]=vecsnew[;index[i]];
}
clear(vecsnew,e);
Unew=U;

for (i in 1:newLsize-1) {
    Unew[;i]=U*vecs[1:newLsize-1;i] +
        vecs[newLsize;i]*Th;
    Unew[;i]=Unew[;i]./vpnorm(Unew[;i],2);
}
U=Unew;
clear(Unew); // have new eigenletters

// Need to convert the database
database.[I].Size=database.[I].Size+1;
B=U'*(V.[I].Ave-Anew);
Conv=U'*V.[I].V;

for (i in 1:database.[I].Size-1) {
    database.[I].Ys[;i]= Conv *
        database.[I].Ys[;i]+B;
}
database.[I].Ys[;database.[I].Size] =
    U'*T;
ind=(int ((K-1)/3))+1;
database.[I].match[database.[I].Size] =
    lets[ind];
database.[I].xlocs[database.[I].Size] =
    lets[ind].x;
database.[I].ylocs[database.[I].Size] =
    lets[ind].y;

V.[I].e=eigval;
V.[I].Ave=Anew;
V.[I].V=U;

P[(ind-1)*13+10+(K-(ind-1)*3)]=I;
clear(U);

```

```
else
  // Just add to database
  database.[I].Size=database.[I].Size+1;
  ind=(int ((K-1)/3))+1;
  P[(ind-1)*13+10+(K-(ind-1)*3)]=I;

  database.[I].match[database.[I].Size] =
    lets[ind];
  database.[I].xlocs[database.[I].Size] =
    lets[ind].x;
  database.[I].ylocs[database.[I].Size] =
    lets[ind].y;
  database.[I].Ys[;database.[I].Size] =
    by;
}
```

Appendix B

OCR Training Sets

B.1 The Original 10 Fonts in the Training Set

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

B.2 The 3 New Fonts

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

Appendix C

Full Listing of All the Classes

K-means		PNN	
No.	Class	Class	No.1
1	B B B B B E E E E	B B B B E E E E P P P	35
2	A L b b b f f h i k l r t t t t t t t t t	i l r r r r r r t t t t t t t t t t z	3
3	V V V V V V	V V V Y Y Y Y Y Y	5
4	A J d d d d d d d	A A A J J J d d d	11
5	C G H H K K M M N O O Q R R S U W W W d d H	C C D D D D H H G G G O O O O O O O Q Q Q Q Q Q Q U U	22
6	B E L L b b k b b b h h h k h h h k k k k k	b b b b b h h h h h h h	32
7	A a m m m w w w w w w w w	m m m r r w w x z x x x x z z x e x z z z z	18
8	Z Z Z Z Z Z Z	Z Z Z Z Z Z Z	21
9	C D G G G O	a a e n o r r s s s s s s S u v V z z z	13

K-means		PNN	
No	Class	Class	No
25	C C D G G O O O O O Q Q Q Q Q Q U U	C C G H M N N O O O Q R R U U U W W o	37
26	B H K K M N R S U	L L L L U V V k k	27
27	a a a a a a c c c c c c c c C e e e e e e e e n o o o o o o s s s s s s s u	c c c c c c c c c c c e e e e e o o o o o	12
28	m m m m m m m	m m m m m m m	1
29	E E F F P P R	E E E F F P R Y F F F F F L P	23
30	D D H H O Q U	B B H H H H H U	30
31	B D E F F F K P P P R g p p P P P P P y y	B F F F P P P R R R R R U g P P P P P P y y	10
32	V V V g q q v v v v v v y y y y y y y y y	V V Y Y Y f g q v y y y y y y y y	9
33	K K K K R R R	K K K K K K K N R X X	25
34	Q Q q q q q q q q q	Q Q q q q q q q q q	33
35	A A A A A A h h k k	A K b b b h k h k k k k k k	14
36	F L L L T T T b f f f i	I I L L T T T b b f f f f h i k l	6
37	I I I I T T T T f f f i i i i i l l l l	I I I I T T T T f f f i i i i j l l l l l	2

The following classes do not actually correspond, but are listed together.

K-means Number	PNN Number
9	13
16	15
21	28
26	27

In viewing the classes, it should be remembered that we classified based on centred images of the letters. Thus while it may appear surprising that the letter "g" is grouped with the letter "S", if we consider the actual images (figure C.1), we can see that they share several strokes, which makes the grouping reasonable.

Figure C.1: Letter g against letter S

Appendix D

Bibliography

1. SIROVICH, I. and KIRBY, M, Low-dimensional Procedures for the Characterization of Human Faces, *J. Opt Soc. Am. A*, 4, pp. 519-524 (1987)
2. O'TOOLE, A.J., ABDI, H., DEFFENBACHER, K. A. and VALENTIN, D, Low-dimensional Representation of Faces in Higher Dimensions of the Face Space, *J. Opt. Soc. Am. A.*, 10, pp. 405-410, (1993)
3. MOGHADDAM, B. and PENTLAND, A, An Automatic System for Model-Based Coding of Faces, *IEEE Data Compression Conference, Snowbird, Utah*, (1995)
4. PENTLAND, A., MOGHADDEN, B., and STARNER, T, View-Based and Modular Eigenspaces for face recognition, *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, (1994)
5. TURK, M. and PENTLAND, A, Representing Face for Recognition, *Technical Report #192, Vision and Modeling group, MIT*, (1990)
6. MOGHADDAM, B. and PENTLAND, A, A subspace method for Maximum Likelihood Target Detection, *IEEE International conference on Image Processing, Washington DC*, (1995)
7. MOGHADDEN, B. and PENTLAND, A, Probabilistic Visual Learning for Object Detection, *The 5th International conference on Computer Vision, Cambridge, MA*, (1995)
8. PENTLAND, A., PICARD, R. W. and SCLAROFF S, Photobook: Content-Based Manipulation of Image Databases, *International Journal of Computer Vision*, pp. 233-254 (1996)

9. PENTLAND, A., STARNER, T., ETCOFF N., MASOIU, A., OLIYIDE, O. and TURK, M, Experiments with Eigenfaces, *M.I.T Media Laboratory Perceptual Computing Technical Note No. 194*, (1992)
10. TURK, M. and PENTLAND, A, Eigenfaces for Recognition, *Journal of Cognitive Neuroscience*, pp. 71-86 (1991)
11. PENEV, P and ATICK J, Local Feature Analysis: A general statistical theory for object representation, *Network: Computation in Neural Systems* ,7, pp 477-500, (1996)
12. GONZALEZ, R.C and WOODS, R.E., Digital Image Processing, *Addison-Wesley, Reading, Massachusetts*, (1992)
13. JOLLIFFE, I. T, Principal Component Analysis, *Springer-Verlay, New York*, (1986)
14. PHILLIPS, J., MOON, H., RAUSS P. and RIZVI, S, The FERET September 1996 Database and Evaluation Procedure, *Proceedings of First International Conference on Audio and Video-based Biometric Person Authentication* (1997)
15. FEINSTEIN, A, Foundations of Information Theory, *McGraw-Hill, New York*, (1958)
16. CSISZÁR, I. and KÖRNER, J, Information Theory : Coding Theorems for Discrete Memoryless Systems, *Academic Press, New York*, (1981)
17. COVER, T. M. and THOMAS, J. A, Elements of Information Theory, *John Wiley & Sons, New York*, (1994)
18. SPÄTH, H, Cluster Analysis Algorithms for data reduction and classification of objects, *Ellis Horwood, Chichester*, (1980)
19. ROMESBURG, H. C, Cluster Analysis for Researchers, *Lifetime Learning Publications, Belmont, California*, (1984)
20. GERSHO, A. and GRAY, R. M, Vector Quantization and Signal Compression, *Kluwer Academic Publishers, Boston*, (1992)
21. EQUITZ, W, A New Vector Quantization Clustering Algorithm, *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37, pp. 1568-1575 (1989)
22. GOLOB, G. H. and VAN LOAN, C.F, Matrix computations - 2nd Edition, *John Hopkins University Press, Baltimore* (1989)
23. TREFETHEN, L. N. and BAU, III, D, Numerical Linear Algebra, *Society for Industrial and Applied Mathematics, Philadelphia* (1997)

24. IMPEDOVO, S., OTTAVIANO, L. and OCCHINEGRO, S, Optical Character Recognition - A survey, in CHARACTER AND HANDWRITING RECOGNITION, EXPANDING FRONTIERS, *World Scientific Publishing, Singapore*, pp. 1-24 (1991)
25. RICE, J. A, Mathematical Statistics and Data Analysis, *Duxbury Press, Belmont, California* (1988)
26. SAMET, H, The Design and Analysis of Spatial Data Structures, *Addison-Wesley, Reading, Massachusetts* (1989)