PCASYS - A Pattern-level Classification Automation System for Fingerprints

G. T. Candela
P. J. Grother
C. I. Watson
R. A. Wilkinson
C. L. Wilson

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Visual Image Processing Group
Gaithersburg, MD 20899

August 1, 1995

Abstract

This report describes a system we have developed that automatically classifies images of fingerprints into six pattern-level classes. Automatic classification is useful in an Automated Fingerprint Identification System (AFIS) because it can be used to partition the database of fingerprint cards and thereby reduce the amount of work that must be performed by the fingerprint matcher. Our program takes gray-level images of fingerprints as input, and for each fingerprint it produces a hypothesized classification as *arch*, *left loop*, *right loop*, *scar*, *tented arch*, or *whorl*, as well as a number indicating how much confidence should be assigned to its classification decision. The system performs these processing steps: image segmentation; image enhancement; feature extraction; registration; application of a linear transform that both applies a pattern of regional weights and reduces dimensionality; running of a main classifier, which is a Probabilistic Neural Net, and of an auxiliary whorl-detecting classifier that traces and analyzes pseudoridges (approximate trajectories through the ridge flow); and finally, the combining of the outputs of the main and auxiliary classifiers so as to decide on a hypothesized class and a confidence level. The program's memory and disk space requirements can be met by a typical desktop workstation. The distribution consists of source code, data files, a demonstration set of 2700 fingerprint images, and documentation.

Contents

List of Figures

List of Tables

1 Introduction

Automatic fingerprint classification is a subject of interest to developers of an Automated Fingerprint Identification System (AFIS). In such a system, there is a database of *file* fingerprint cards against which incoming *search* cards must be efficiently matched. Automatic matchers now exist; they compare fingerprints on the basis of their patterns of ridge endings and bifurcations (the minutiae). However, if the file is very large, then exhaustive matching of search fingerprints against file fingerprints may require so much computation as to be impractical. In such a case, the efficiency of the matching process may be greatly increased by partioning the file on the basis of classification of fingerprints. Once the class of each print of the search card has been determined, the set of possibly matching file cards can be restricted to those whose 10-tuple of classes matches that of the search card, thereby reducing the number of comparisons that must be performed by the minutiae-matcher.

Some fingerprint indentification systems currently use manual classification followed by automatic minutiae-matching; the standard Henry classification system, or a modification or extension of it, is often used. (The handbook [1] provides a complete description of the manual classification system used by the FBI.) Automating the classification process should improve its speed and cost-effectiveness. However, producing an accurate automatic fingerprint classifier has proved to be a very difficult task. The object of the research leading to PCASYS is to build a prototype classifier that separates fingerprints into basic pattern-level classes known as *arch*, *left loop*, *right loop*, *scar*, *tented arch*, and *whorl*. Figure 1 shows example fingerprints of the several classes; figure 2 is a second example print of the whorl class, which will be used for subsequent figures illustrating the stages of processing. The pattern-level classes comprise a version of the top level of classification involved in a Henry system. Although this small set of classes does not produce as fine a partioning of a database as does a full Henry system, it seems reasonable to use only the pattern-level classes in this research; automatic classification into a finer system of classes closer to the Henry system could be a subject of future research.

PCASYS is a prototype/demonstration pattern-level fingerprint classification program. It is provided in the form of a source code distribution and is intended to run on a desktop workstation. The program reads and classifies each of a set of fingerprint image files, optionally displaying the results of the several processing stages in graphical form. The disk contains 2700 fingerprint images that may be used to demonstrate the classifier; it can also be run on user-provided images.

The basic method used by the PCASYS fingerprint classifier consists of, first, extracting from the fingerprint to be classified an array (a two-dimensional grid in this case) of the local orientations of the fingerprint's ridges and valleys, and second, comparing that orientation array with similar arrays made from prototype fingerprints ahead of time. The comparisons are actually performed between low-dimensional feature vectors made from the orientation arrays, rather than using the arrays directly, but that can be thought of as an implementation detail. Orientation arrays or matrices like the ones used in PCASYS were produced in early fingerprint work at Rockwell, CALSPAN, and Printrak; the detection of local ridge slopes came about naturally as a side effect of binarization algorithms that were used to preprocess scanned fingerprint images in preparation for minutiae detection. Early experiments in automatic fingerprint classification using these orientation matrices were done by Rockwell, improved upon by Printrak, and work was done at NIST (then NBS); Wegstein, of NBS, who produced the R92 registration algorithm that is used by PCASYS, did important early automatic classification experiments.

This report is organized as follows. Section 2 describes the algorithms used. Section 3 describes a method that can be used to train (optimize) the parameters of the classifier. Section 4 provides some accuracy and timing results. Section 5 lists some possible topics for future work. Section 6 concerns the procedure for installing and running the programs.

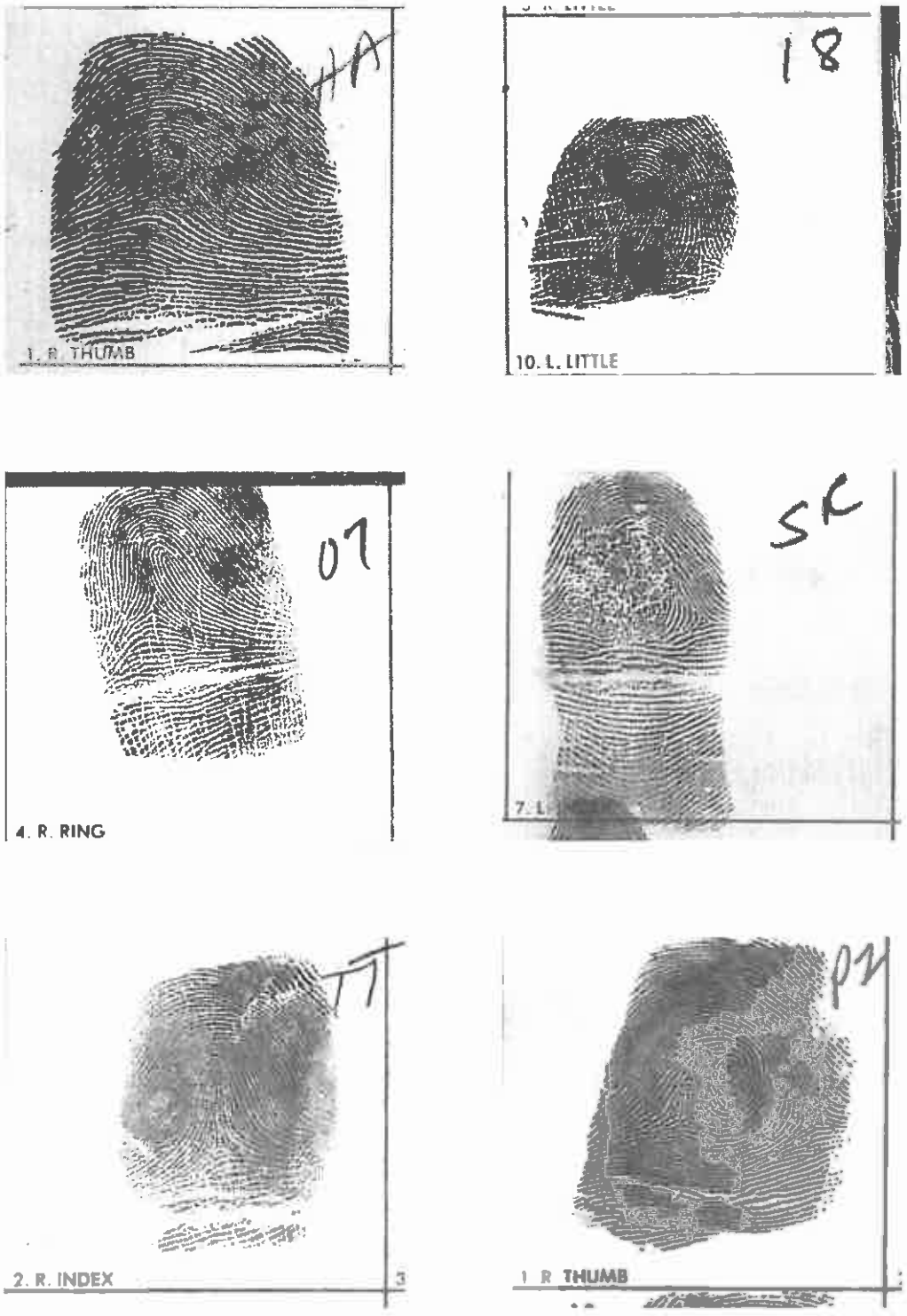

Figure 1: Example fingerprints of the six pattern-level classes. Top row: arch (A), left loop (L). Middle row: right loop (R), scar (S). Bottom row: tented arch (T), whorl (W). (These are NIST Special Database 14 images s0024501.wsq, s0024310.wsq, s0024304.wsq, s0026117.wsq, s0024372.wsq, and s0024351.wsq; they are among the demonstration images provided with PCASYS.)

Figure 2: A second example fingerprint of the whorl class (s0025236.wsq). This is the fingerprint used for subsequent figures.

2 Description of the Algorithms

This section describes the algorithms the system uses in its several processing phases. Each subsection shows, at its beginning, the pertinent source code file(s).

2.1 Segmentor

(Source file: lib/pca/sgmnt.c)

The segmentor routine performs the first stage of processing needed by the classifier. It reads the input fingerprint image, which must be an 8-bit grayscale raster of width at least 512 pixels and height at least 480 pixels, scanned at about 500 dots per inch; this image is assumed to depict a predefined region of a fingerprint card, corresponding to one of the ten boxes in which the individual fingers are rolled. The segmentor produces, as its output, an image that is 512 × 480 pixels in size, by cutting a rectangular region of these dimensions out of the input image. The sides of the rectangle that is cut out are not necessarily parallel to the corresponding sides of the original image. The segmentor attempts to position its cut rectangle on the impression made by the last joint of the finger, and it attempts to define the rotation angle of the cut rectangle so as to remove any rotation that the finger impression had to start with. Cutting out this smaller rectangle is helpful because it reduces the amount of data that has to undergo subsequent processing (especially the compute-intensive image enhancement); and the removal of rotation may be helpful since it removes a source of variation between prints of the same class. [1]

The segmentor decides which rectangular region of the image to snip out by performing a sequence of processing steps. This processing, and all processing after segmentation as well, will be illustrated using the fingerprint of figure 2 as an example; figure 3, which is one of the windows produced by the graphical version of the system, shows the results of the segmentor's processing steps, which are as follows.

First, the segmentor produces a small binary (two-valued or logical-valued) image whose pixels indicate which 8×8-pixel blocks of the original image should be considered to be the *foreground*, that is, the part of the image that contains ink, whether from the finger impression itself or from printing or writing on the card. To produce this foreground-image, it first finds the minimum pixel value for each block and the global minimum and maximum pixel values. Then, for each of a fixed set of **factor** values between 0 and 1, the routine produces a candidate foreground-image based on **factor** as follows:

> **threshold = global_min + factor** × **(global_max - global_min)**
>
> Set to **true** each pixel of candidate foreground-image whose corresponding pixel of the array of block minima is ≤ **threshold**, and count resulting **true** pixels.
>
> Count the transitions between **true** and **false** pixels in the candidate foreground-image, counting along all rows and columns. Keep track of the minimum, across candidate foreground-images, of the number of transitions.

Among those candidate foreground-images whose number of **true** pixels is within predefined limits, pick the one with the fewest transitions. (If **threshold** is too low, there tend to be many white holes in what should be solid blocks of black foreground; if **threshold** is too high, there tend to be many black spots on what should be solid white background. If **threshold** is about right, there are few holes and few spots, hence few transitions.) The top picture of figure 3 shows the resulting foreground-image.

Next, the routine performs some cleanup work on the foreground-image, the main purpose of which is to delete those parts of the foreground that correspond to printing or writing rather than the finger impression. The routine does three iterations of erosion[2], then it deletes every connected set of **true** pixels except the one whose number of **true** pixels is largest, and as the final step of cleanup it sets to **true**, in each row, every pixel between the leftmost and rightmost **true** pixels in that row, and similarly for columns. The routine then computes the centroid

[1] The images produced by the segmentor are similar to those of NIST Special Database 4, in which the corrections for translation and rotation were done manually.

[2] Erosion consists of changing to false each **true** pixel that is next to a false pixel.

of the cleaned-up foreground-image, for later use. The second picture of figure 3 shows the result of this cleanup processing.

The routine next finds the left, top and right edges of the foreground, which usually has a roughly rectangular shape. Because the preceding cleanup work has removed noise **true** pixels caused by printed box lines or writing, the following very simple algorithm is sufficient for finding the edges. Starting at the middle row of the foreground-image and moving upward, the routine finds the leftmost **true** pixel of each row in turn, considering the resulting pixels to trace the left edge; but, to avoid going around the corner onto the top edge, the routine stops as soon as it encounters a row whose leftmost **true** pixel has a horizontal distance of more than 1 from the leftmost **true** pixel of the preceding row. The routine finds the bottom part of the left edge by using the same process but moving downward from the middle row; and it finds the top and right edges similarly. The third, fourth, and fifth pictures of figure 3 depict these edges.

Next, the routine uses the edges to calculate the overall slope of the foreground, as follows. First it fits a straight line to each edge by linear regression; naturally, it fits the left and right edges, which are expected to be roughly vertical, to lines of the form $x = my + b$, and it fits the top edge to a line of the form $y = mx + b$. The next to last picture of figure 3 shows these fitted lines. The overall slope is defined to be the average of the slopes of the left-edge line, the right-edge line, and a line perpendicular to the top-edge line.

Having measured the foreground slope, the segmentor now knows the angle to which it should rotate its cutting rectangle so as to nullify the existing rotation of the fingerprint; but it still must decide the location at which to cut. To decide this, it first finds the foreground top, in a manner more robust than the original finding of the top edge and resulting fitted line. It finds the top by considering a tall rectangle, whose width corresponds to the output image width, whose center is at the previously computed centroid of the foreground-image, and which is tilted in accordance with the overall foreground slope. Starting at the top row of this tall, narrow tilted rectangle and moving downward, the routine counts the **true** pixels of each row in turn, and stops at the first row which both fits entirely on the foreground-image and has at least a threshold number of **true** pixels. The routine then finishes deciding where to cut by letting the top edge of the rectangle correspond to the foreground top it has just detected. The cut out image thus will be tilted so as to cancel out the existing rotation of the fingerprint, and will be positioned so as to hang from the top of the foreground, as it were. The bottom picture of figure 3 is the (cleaned-up) foreground with an outline superimposed on it showing where the segmentor has decided to cut. The segmentor finishes by actually cutting out the corresponding piece of the input image; figure 4 shows the resulting segmented image. (The routine also cuts out the corresponding piece of the foreground-image, for use later by the pseudoridge analyzer.)

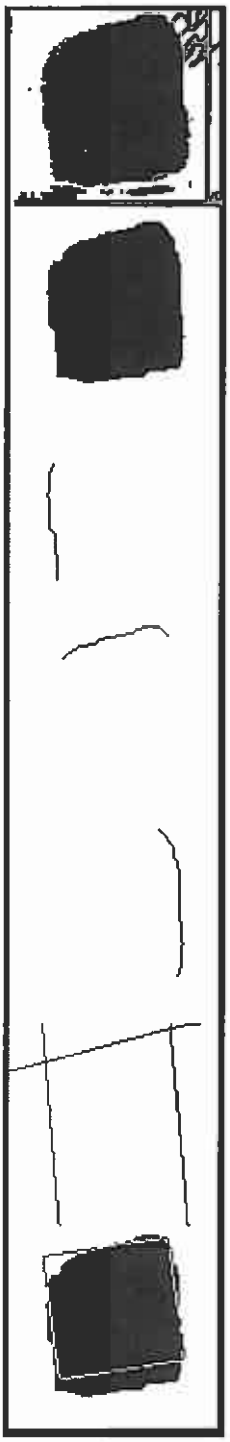

Figure 3: How the segmentor decided the angle and location at which to cut from the example fingerprint image.

Figure 4: The image cut out of the example fingerprint image by the segmentor.

2.2 Image Enhancer

(Source files: lib/pca/enhnc.c, lib/fft/*.c)

This routine enhances the segmented fingerprint image. The algorithm used is basically the same as the enhancement algorithm described in [2], and pp. 2-8 – 2-16 of [3] provide a description of other research that independently produced this same algorithm. The routine goes through the image and snips out a sequence of squares each of size 32×32 pixels, with the snipping positions spaced 24 pixels apart in each dimension to produce overlapping. Each input square undergoes a process that produces an enhanced version of its middle 24×24 pixels, and this smaller square is installed into the output image in a non-overlapping fashion relative to other output squares. (The overlapping of the input squares reduces boundary artifacts in the output image.)

The enhancement of a square of input is done by first performing the forward two-dimensional fast Fourier transform (FFT) to convert the data from its original (spatial) representation to a frequency representation; then applying a nonlinear function that tends to increase the power of useful information (the overall pattern, and in particular the orientation, of the ridges and valleys) relative to noise; and finally, performing the backward 2-d FFT to return the enhanced data to a spatial representation before snipping out the middle 24×24 pixels and installing them into the output image.

The filter's processing of a square of input pixels can be described by the following equations. First, produce the complex-valued matrix $A + iB$ by loading the square of pixels into A and letting B be zero. Then, perform the forward 2-d discrete Fourier transform, producing the matrix $X + iY$ defined by

$$X_{jk} + iY_{jk} = \sum_{m=0}^{31} \sum_{n=0}^{31} (A_{mn} + iB_{mn}) \exp\left(\frac{-2\pi i}{32}(mj + nk)\right).$$

Change to zero a fixed subset of the Fourier transform elements corresponding to low and high frequency bands which, as discussed below, can be considered to be noise. Then take the *power spectrum* elements $X_{jk}^2 + Y_{jk}^2$ of the Fourier transform, raise them to the 0.3 power, and multiply them by the Fourier transform elements, producing a new frequency-domain representation $U + iV$:

$$U_{jk} + iV_{jk} = (X_{jk}^2 + Y_{jk}^2)^{0.3}(X_{jk} + iY_{jk}).$$

Return to a spatial representation by taking the backward Fourier transform of $U + iV$.

$$C_{mn} + iD_{mn} = \sum_{j=0}^{31} \sum_{k=0}^{31} (U_{jk} + iV_{jk}) \exp\left(\frac{2\pi i}{32}(jm + kn)\right),$$

and then finish up as follows: find the maximum absolute value of the elements of C (the imaginary matrix D is zero), and cut out the middle 24×24 pixels of C and install them in the output image, but first applying to them an affine transform that maps 0 to 128 (a middle gray) and that causes the range to be as large as possible without exceeding the range of 8-bit pixels (0 through 255). (The DC component of the Fourier transform is among the low-frequency elements that are zeroed out, so the mean of the elements of C is zero and it is therefore reasonable to map 0 to the middle of the available range.)

However, for greater efficiency, the enhancer routine actually does not simply implement these formulas directly. Instead, it uses fast Fourier transforms (FFTs), and in fact takes advantage of the purely real nature of the input matrix by using 2-d *real* FFTs. But the output is no different than if the above formulas had been translated straight into code.

We have found that enhancing the segmented image with this algorithm, before extracting the orientation features (next section), increases the accuracy of the resulting classifier. The table and graphs on pp. 24-6 of [4] show the accuracy improvement caused by using this filter (localized FFT filter), as well as the improvements caused by various other features. The nonlinear function applied to the frequency-domain representation of the square of pixels has the effect of increasing the relative strength of those frequencies that were already stronger than the others; and these originally stronger frequencies correspond to the ridges and valleys in most cases, so that the enhancer strengthens the important aspects of the image (the ridges and valleys) at the expense of noise such as

small details of the ridges, breaks in the ridges, and ink spots in the valleys. This is not simply a linear filter that attenuates certain frequencies, although part of its processing does consist of eliminating low and high frequencies: the surviving frequencies go through a nonlinear function which adapts to variations as to which frequencies are most powerful. This aspect of the filter is helpful because the ridge wavelength can vary considerably between fingerprints and between areas within a single fingerprint.[3]

Figure 5 shows the enhanced version of the segmented image. At first glance, a noticeable difference between the original and enhanced versions is the increase in contrast, but actually the more important change caused by the enhancer is the improved smoothness and stronger ridge/valley structure of the image, which are apparent upon closer examination. Discontinuities are visible at the boundaries of some output squares despite the overlapping of input squares, but these apparently have no major harmful effect on subsequent processing.

Figure 5: Enhanced image from the example fingerprint.

[3] A different FFT-based enhancement method, the directional FFT filter of [4], uses global rather than local FFTs and uses a set of masks to selectively enhance regions of the image that have various ridge orientations. This enhancer was more computationally intensive than the localied FFT filter, and did not produce better classification accuracy than the localized filter.

2.3 Ridge-Valley Orientation Detector

(Source file: lib/pca/ridge.c)

This routine detects, at each pixel location of the fingerprint image, the local orientation of the ridges and valleys of the finger surface, and produces an array of regional averages of these orientations. This is the basic *feature extractor* of the classification system; its output is used by both the main PNN classifier and the auxiliary classifier.

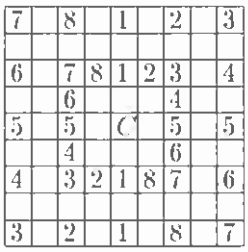

Figure 6: Pattern of slits used by the orientation detector.

The routine is based on the *ridge-valley* fingerprint binarizer described in [5]. That binarizer uses the following algorithm to reduce a grayscale fingerprint image to a binary (black and white only) image. For each pixel of the image, denoted C in figure 6, the binarizer computes slit sums s_i, $i = 1 \ldots 8$, where each s_i is the sum of the values of the slit of four pixels labled i in the figure. The binarizer uses local thresholding and slit comparison formulas. The local thresholding formula sets the output pixel to white if the value of the central pixel, C, exceeds the average of the pixels of all slits, that is, if

$$C > \frac{1}{32} \sum_{i=1}^{8} s_i. \tag{1}$$

Local thresholding such as this is better than using a single threshold everywhere on the image, since it ignores gradual variations in the overall brightness.

The slit comparison formula sets the output pixel to white if the average of the minimum and maximum slit sums exceeds the average of all the slit sums, that is, if

$$\frac{1}{2}(s_{min} + s_{max}) > \frac{1}{8} \sum_{i=1}^{8} s_i. \tag{2}$$

The motivation for this formula is as follows. If a pixel is in a valley, then one of its eight slits will lie along the (light) valley and have a high sum, whereas the other seven slits will each cross several ridges and valleys and these slits will therefore have roughly equal, lower sums, so that the average of the two extreme slit sums will exceed the average of all eight slit sums and the pixel will be binarized correctly to white. Similarly, the formula causes a pixel lying on a ridge to be binarized correctly to black. This formula uses the slits to detect long structures (ridges and valleys), rather than merely using their constituent pixels as a sampling of local pixels as formula 1 does; it is able to ignore small ridge gaps and valley blockages, since it concerns itself only with entire slits and not with the value of the central pixel.

The authors of [5] found that they obtained good binarization results by using the following compromize formula, rather than using either (1) or (2) alone: the output pixel is set to white if

$$4C + s_{min} + s_{max} > \frac{3}{8} \sum_{i=1}^{8} s_i. \tag{3}$$

This is simply a weighted average of formulas (1) and (2), with the first one getting twice as much weight as the second.

This binarizer can be converted into a detector of the ridge or valley orientation at each pixel very trivially: merely consider each pixel that would have been binarized to black (a ridge pixel) to have the orientation of its minimum-sum slit, and consider each pixel that would have been binarized to white (a valley pixel) to have the orientation of its maximum-sum slit. However, the resulting array of pixelwise orientations is large, noisy, and coarsely quantized (only 8 different orientations are allowed). Therefore, the pixelwise orientations are reduced to a much smaller array of *local average* orientations, each of which is made from a 16×16 square of pixelwise orientations. The averaging process reduces the volume of data, lessens noise, and produces a finer quantization of orientations.

The ridge angle θ at a location is here defined to be $0°$ if the ridges are horizontal, increasing towards $180°$ as the ridges rotate counterclockwise, and snapping back to $0°$ when the ridges become horizontal again: $0° \leq \theta < 180°$. When pixelwise orientations are averaged, the quantities averaged are actually not the pixelwise ridge angles θ, but rather the pixelwise *orientation vectors* $(\cos 2\theta, \sin 2\theta)$. The orientation finder produces an array of these averages of pixelwise orientation vectors.[4] Since all pixelwise vectors have length 1 (being cosine, sine pairs), each average vector has a length of at most 1. If a square of the image does not have a well-defined overall ridge and valley orientation, for example because of blurring or because the orientation is highly variable within this square, then the orientation vectors of its 256 pixels will tend to cancel each other out and produce a short average vector. The length of an average vector is thus a measure of orientation strength. (The routine also produces as output the array of pixelwise orientation indices, to be used by a later routine which produces a more finely spaced array of average orientations.) Figure 7 depicts the local average orientations that were detected in the segmented and filtered image from the example fingerprint.

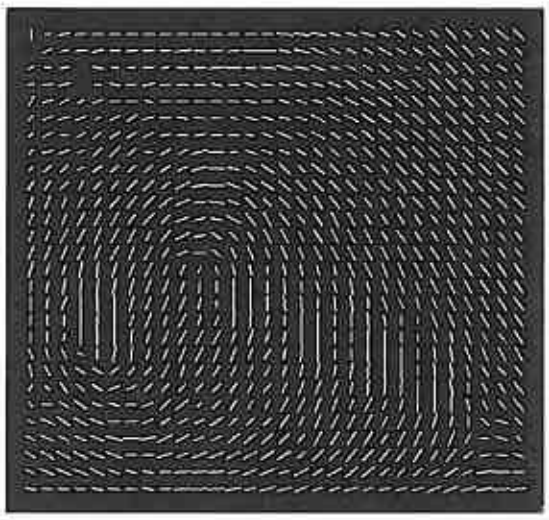

Figure 7: Array of local average orientations of the example fingerprint. Each bar, depicting an orientation, is approximately parallel to the local ridges and valleys.

2.4 Registration

(Source files: lib/pca/r92.c, lib/pca/ridge.c)

Registration is a process that the classifier uses in order to reduce the amount of translational variation between similar orientation arrays. If the arrays from two fingerprints are such that one is approximately a translation of the other, then the feature vectors that later processing steps will produce from these orientation arrays may be very different, because of the translation, even though the ridge flow patterns are similar. This problem can be

[4] Averaging a set of local orientation *angles* can produce absurd results, because of the unremovable point of discontinuity that is inherent in an angular representation, so it is better to use the vector representation. Also, the resulting local average orientation vectors are an appropriate representation to use for later processing steps (except for the R92 registration program, which requires converting the vectors into angles of a different format), because these later steps require that Euclidean distances between entire arrays of local average orientations produce reasonable results.

ameliorated by *registering* each array: finding a consistent feature and essentially translating the array so as to bring that feature to a standard location.

To find the consistent feature that is required, we use the R92 algorithm of Wegstein [6]. This procedure finds, in an array of ridge angles, a feature that is located approximately at the *core* of a loop fingerprint, or at the more upright of the two cores that often occur in a whorl; the algorithm also finds a well-defined feature in arch and tented arch fingerprints, even though these types of prints do not have true cores. After R92 finds this feature, which will be denoted the *registration point*, the registration process is completed by taking the array of pixelwise orientations produced earlier and averaging 16×16 squares from it in order to make a new array of average orientations; the averaging is done the same way it was done to make the first array of average orientations (which became the input to R92), except that now the squares upon which averaging is performed are translated by the vector that is the registration point minus a *standard registration point* defined as the componentwise median of the registration points of a sample of fingerprints. The result is a registered array of average orientations.[5]

The R92 algorithm begins by analyzing the matrix of angles in order to build the "K-table". (R92 processes the orientations in angular form, and it defines the angles to range from 0° to 90° as a ridge rotates from horizontal counterclockwise to vertical, and from 0° to −90° for clockwise rotation, rather than having the angles range from 0° to 180° as the ridge rotates counterclockwise from horizontal, the definition used earlier.) This table lists the first location in each row of the matrix where the ridge orientation changes from positive slope to negative slope to produce a well-formed arch. Associated with each K-table entry are other elements that are used later to calculate the registration point. The ROW and COL values are the position of the entry in the orientation matrix. The SCORE is how well the arch is formed at this location. The BC SUM is the sum of this angle with its east neighbor, while the AD SUM is BC SUM plus the one angle to the west and east of the BC SUM. SUM HIGH and SUM LOW are summations of groups of angles below the one being analyzed. For these two values, five sets of four angles are individually summed, and the lowest and highest are saved in the K-table.

With the K-table filled in, each entry is then scored. The score indicates how well the arch is formed at this point. The point closest to the core of the fingerprint is intended to get the largest score. If scores are equal, the entry closest to the bottom of the image is considered the winner. Calculating a score for a K-table entry uses six angles and one parameter, $RK3$. $RK3$ is the minimum value of the difference of two angles. For this work, the parameter was set to 0 degrees, which is a horizontal line. The six angles are the entry in the K-table, the two angles to its left and the three angles to its right. So if the entry in the K-table is (i, j), then the angles are at positions $(i, j-2)$, $(i, j-1)$, (i, j), $(i, j+1)$, $(i, j+2)$, and $(i, j+3)$. These are labeled M, A, B, C, D, and N, respectively. For each of the differences, $M - B$, $A - B$, $C - N$, and $C - D$, which are greater than $RK3$, the score is increased by one point. If A has a positive slope, meaning the angle of A is greater than $RK3$, then the score is increased by one point and another if $M - A$ is greater than $RK3$. If D has a negative slope, meaning the angle of D is less than $RK3$, then the score is increased by one point and another if $D - N$ is greater than $RK3$. If N has a negative slope, then the score is increased by one point. All these comparisons form the score for the entry.

Using the information gathered about the winning entry, a registration point is produced. First, it is determined whether the fingerprint is possibly an arch; if so, the registration point (x, y) is computed as:

$$
\begin{aligned}
x &= \alpha \times \left(\frac{A(R,C)}{A(R,C) - A(R,C+1)} + C - 1 \right) + \beta \\
y &= \frac{(\alpha \times R + \beta) \times ts(k+1) + (\alpha \times (R-1) + \beta) \times ts(k) + (\alpha \times (R-2) + \beta) \times ts(k-1)}{ts(k+1) + ts(k) + ts(k-1)}
\end{aligned}
$$

where A is the angle at an entry position, R is the row of the entry, C is the column of the entry, k is the entry number, ts is a sum of angles, α is 16 (the number of pixels between orientation loci), and β is 8.5. The ts value is calculated by summing up to six angles. These angles are the current angle and the five angles to its east. While the angle is not greater than 99 degrees its absolute value is added to ts. For angles 89, 85, 81, 75, 100, and 60, the sum would be 330 (89 + 85 + 81 + 75). Since 100 is greater than 99, the summation stops at 75.

[5] The new array is made by re-averaging the pixelwise orientations with translated squares, rather than merely translating the first average-orientations array, because the registration point found by R92, and hence the prescribed translation vector, is defined more precisely than the crude 16-pixel spacing corresponding to one step through the average orientation array.

2.5 Feature Set Transformation

(Source file: lib/pca/trnsfrm.c)

This routine applies a linear transform to the registered orientation array. Transformation accomplishes two useful processes: the reduction of the dimensionality of the feature vector from its original 1680 dimensions to 64 dimensions, and the application of a fixed pattern of regional weights which are larger in the important central area of the orientation array.

2.5.1 Karhunen-Loève Transform

The Probabilistic Neural Network (PNN) classifier, discussed in section 2.6, must[6] compute the squared distances between a *feature vector* representing the fingerprint to be classified, and stored feature vectors representing each of a large number of *prototype* fingerprints. The feature vector representation of a fingerprint could be defined to be the registered orientation array, treated as a single vector of 1680 elements (28×30 orientation vectors $\times$ two components per orientation vector). If this were done, the resulting squared Euclidean distance between two feature vectors would be independent of the (consistent) ordering of the 1680 elements, and could be thought of as the sum of 840 (28×30) squared distances between corresponding pairs of orientation vectors in the two arrays. This way of defining the distance between two orientation arrays, amounting to template matching in which one array is overlaid on the other, would be reasonable in terms of the quality of the results. However, it would not be a particularly reasonable distance computation in practical terms: because of the high dimensionality of these vectors, storing a large number of them as prototypes would requires excessive memory and computing distances between such vectors would be slow.

It would be helpful to transform these high-dimensional feature vectors into much lower-dimensional ones in such a way that Euclidean distances between vectors were approximately preserved. Fortunately, the Karhunen-Loève (K-L) transform [7] does exactly that. To produce the matrix that implements the K-L transform, the first step is to make the sample covariance matrix of a set of typical original feature vectors, the registered orientation arrays in our case. Then, a diagonalization routine is used to produce a subset of the eigenvectors of the covariance matrix, corresponding to the largest eigenvalues; let m denote the number of eigenvectors produced. (The diagonalization process can be sped up by arranging to produce only, say, the first 100 or so eigenvectors, which will probably be sufficient.) Then, for any $n \leq m$, the matrix $\mathbf{\Psi}$ can be defined to have as its columns the first n eigenvectors; each eigenvector has as many elements as an original feature vector, 1680 in the case of orientation arrays. A version[7] of a K-L transform, which reduces an original feature vector $\mathbf{u}$ (an orientation array, thought of as a single 1680-dimensional vector) to a vector $\mathbf{w}$ of n elements, can then be defined as follows:

$$\mathbf{w} = \mathbf{\Psi}^t \mathbf{u}.$$

The K-L transform thus may be used to reduce the orientation array of a fingerprint to a much lower-dimensional vector which may be sent to the PNN classifier, producing approximately the same classification results as would be obtained without the use of the K-L transform but with large savings in memory and compute time. A reasonable value of n, the number of eigenvectors used and hence number of elements in the feature vectors produced, can be found by trial and error; usually n can be much smaller than the original dimensionality. (We have found 64 to be a reasonable n.) In earlier versions of our fingerprint classifier, we produced low-dimensional feature vectors in this manner, using the arrays of 28×30 orientation vectors as the original feature vectors. Later experiments revealed, however, that significantly better classification accuracy could be obtained by modifying the production of the feature vector, in various ways, so as to cause the important central region of the fingerprint to have more weight than the outer regions; these experiments culminated in the production of the *regional weights* discussed below.

[6] in its naive form, which PCASYS uses

[7] Usually the sample mean vector is subtracted from the original feature vector before applying $\mathbf{\Psi}^t$, but we omit this step because doing so simplifies the computations and has no effect on the ultimate results. The PNN classifier uses squared distances between feature vectors, so any affine transform (a linear transform plus a constant vector) used in the production of feature vectors can be replaced by its corresponding linear transform without changing the resulting squared distances, since the affine part would cancel out between the two vectors being compared anyway.

2.5.2 Regional Weights

During testing it was noted that the uniform spacing of the orientation measurements throughout the picture area could probably be improved upon by replacing it with a nonuniform spacing that concentrated the measurements more closely together in the important central area of the picture, with a sparser distribution in the clearly less important outer regions. We tried this [8], keeping the total number of orientation measurements the same as before (840) in order to make a fair comparison, and the result was indeed a significant lowering of the classification error rate.

Eventually it occurred to us that these improved results might not really have been caused by the nonuniform *spacing* but rather by the mere assignment of effectively greater *weight* to the central region, caused by placing a larger number of measurements there. We tested this hypothesis by reverting to the uniformly-spaced array of orientation measurements, but now with a nonuniform pattern of *regional weights* to be applied to the orientation array before performing the K-L transform and computing distances. The application of a fixed pattern of weights to the features before computing distances between feature vectors is equivalent to the replacement of the usual Euclidean distance by an alternative distance. In [9], Specht improves the accuracy of PNN in basically the same manner: pp. I-765-6 describe the method used to produce a separate σ value for each dimension (feature).

To keep the number of weights reasonably small and thus control the amount of runtime that would be needed to optimize them, we decided to assign a weight to each 2×2 block of orientation-vectors, for a total of 210 (14×15) weights, rather than assigning a separate weight to each of the 840 orientation-vectors. Optimization of the weights was done using a very simple form of gradient descent, as discussed in section 3.5. The resulting optimal (or nearly optimal) weights are depicted in figure 9. The gray tones represent the absolute values of the weights (their signs have no effect), with the values mapped to tones by a linear mapping that maps 0 to black and the largest absolute value that occured, to white. These weights can be represented as a diagonal matrix $\mathbf{W}$ of order 1680, and then their application of the weights to an original feature vector (orientation array) $\mathbf{u}$, to produce a weighted version $\tilde{\mathbf{u}}$, is given by the matrix multiplication

$$\tilde{\mathbf{u}} = \mathbf{W}\mathbf{u}.$$

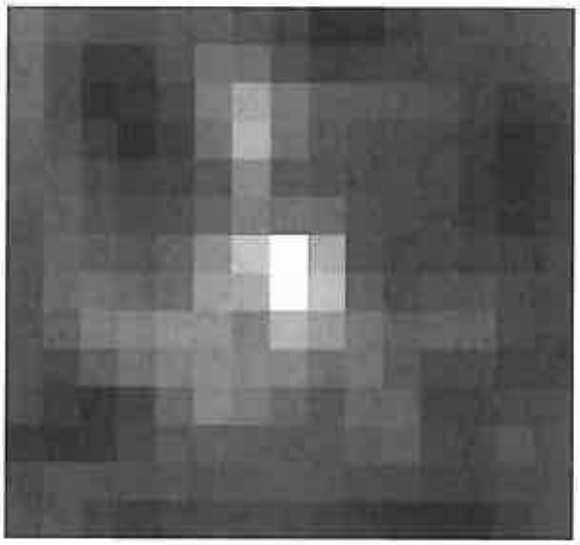

Figure 9: Absolute values of the optimized regional weights. Each square represents one weight, associated with a 2×2 block from the registered orientation array.

We have tried optimizing a set of weights to be applied directly to the K-L features, but this produced poor generalization results. The regional weights we describe here are not equivalent to any set of weights (diagonal

matrix) that could be applied to the K-L features: their use is approximately equivalent to the application of the non-diagonal matrix $\mathbf{\Psi}^t\mathbf{W}\mathbf{\Psi}$, mentioned in 3.5, to the K-L feature vectors. We also have tried optimizing a completely unconstrained linear transform (matrix) to be applied to the K-L feature vectors before computing distances; that produced impressive lowering of the error during training but disastrous generalization results. Among our experiments involving the application of linear transforms prior to PNN distance computations, we obtained the best results by using regional weights.

2.5.3 Combined Transform

Clearly it is reasonable to apply the optimized regional weights $\mathbf{W}$, and then to reduce dimensionality with $\mathbf{\Psi}^t$, before letting the PNN classifier compute distances. An efficient way to do this is to make the combined transform matrix $\mathbf{T} = \mathbf{\Psi}^t\mathbf{W}$ and then when running the classifier on a fingerprint, to use

$$\mathbf{w} = \mathbf{T}\mathbf{u}$$

to convert its orientation array directly into the final feature-vector representation.[8]

[8] After optimizing the weights $\mathbf{W}$, we could have made new eigenvectors from the covariance matrix of the *weighted* original-feature vectors; the $\mathbf{\Psi}^t\mathbf{W}$ resulting from this new $\mathbf{\Psi}$ would presumably have then produced a more efficient dimensionality reduction than we now obtain, allowing the use of fewer features. But we decided not to bother with this, since the memory and time requirements of the current feature vectors are reasonable.

2.6 Main Classifier: Probabilistic Neural Network

(Source file: lib/pca/pnn.c)

This routine takes as its input the low-dimensional feature vector that is the output of the transform discussed in the preceding section, and it determines the class of the fingerprint. The Probabilistic Neural Network (PNN) is described by Specht in [10]. The algorithm classifies an incoming feature vector by computing the value, at its point in feature space, of spherical Gaussian kernel functions centered at each of a large number of stored prototype feature vectors. These prototypes were made ahead of time from a training set of fingerprints of known class by using the same preprocessing and feature extraction that was used to produce the incoming feature vector. For each class, an an activation is made by adding up the values of the kernels centered at all prototypes of that class; the hypothesized class is then defined to be the one whose activation is largest. The activations are all positive, being sums of exponentials. Dividing each activation by the sum of all activations produces a vector of normalized activations, which, as Specht points out, can be used as estimates of the posterior probabilities of the several classes. In particular, the largest normalized activation, which is the estimated posterior probability of the hypothesized class, is a measure of the confidence that may be assigned to the classifier's decision.[9]

In mathematical terms, the above definition of PNN can be written as follows, starting with notational definitions:

N	=	number of classes (6 in PCASYS)
M_i	=	number of prototype prints of class i $(1 \le i \le N)$
$\mathbf{x}_j^{(i)}$	=	feature vector from j^{th} prototype print of class i $(1 \le j \le M_i)$
$\mathbf{w}$	=	feature vector of the print to be classified
β	=	a smoothing factor
a_i	=	activation for class i
$\tilde{a}_i$	=	normalized activation for class i
h	=	hypothesized class
c	=	confidence

For each class i, the PNN computes an activation:

$$a_i = \sum_{j=1}^{M_i} \exp\left(-\beta \left(\mathbf{w} - \mathbf{x}_j^{(i)}\right)^t \left(\mathbf{w} - \mathbf{x}_j^{(i)}\right)\right).$$

It then defines h to be the i for which a_i is greatest, and defines c to be the h^{th} normalized activation:

$$c = \tilde{a}_h = a_h / \sum_{i=1}^{N} a_i.$$

Figure 10 is a bar graph of the normalized activations produced for the example fingerprint. (Although the PNN only needs to normalize one of the activations, namely the largest, to produce the confidence, all 6 normalized activations are shown here.) The whorl (W) class has won and so is the hypothesized class (correctly as it turns out), but the left loop (L) class has also received a fairly large activation and therefore the confidence is only moderately high.

[9] This naive version of PNN must compute the distance of the incoming feature vector from each of the many prototype feature vectors, possibly using many cycles. Various methods have been found for increasing the speed of nearest- neighbors classifiers, a category which PNN may be considered to fall into (see, for example, [11], and [12] for a very fast tree method), but the classification accuracy of fast approximations to the naive PNN may suffer at high rejection levels. For that reason, and because the naive PNN takes only a small fraction of the total time used by the PCASYS classification system (image enhancement takes much longer), we have used the naive version.

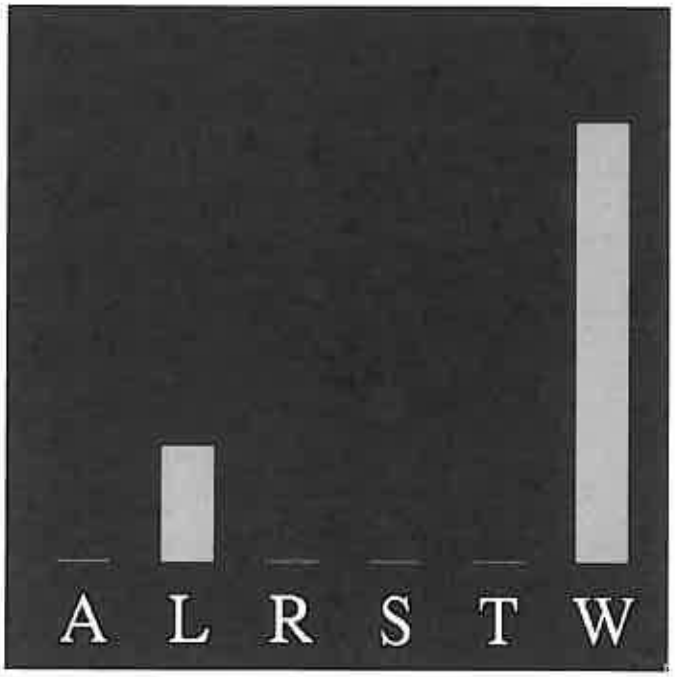

Figure 10: Normalized output activations that the Probabilistic Neural Network produced for the example fingerprint. A = *arch*, L = *left loop*, R = *right loop*, S = *scar*, T = *tented arch*, W = *whorl*.

2.7 Auxiliary Classifier: Pseudoridge Tracer

(Source file: lib/pca/pseudo.c)

This routine takes a grid of ridge orientations of the incoming fingerprint and traces *pseudoridges* [13], which are trajectories that approximately follow the flow of the ridges. By testing the pseudoridges for concave-upward shapes, the routine detects some whorl fingerprints that are misclassified by the PNN classifier. We were motivated to produce a whorl-detector when we realized, upon examining the prints misclassified by the PNN even at high rejection levels, that many of them were whorls.

The routine takes as an input an array of local average ridge orientations[10] Another input is a small binary image that shows the region of the segmented image comprising the inked foreground rather than the lighter background. First, the routine changes to zero vectors any of the orientation vectors that either are not on the foreground, or are smaller than a threshold in squared length. (Small squared length indicates uncertainty as to the orientation.) Next, it performs a few iterations of a smoothing algorithm, which merely replaces each vector by an average of itself and its four neighbors: this tends to improve anomalous or noisy vectors. Then, it finds out which vectors are either off the foreground or, in their now smoothed state, smaller than a threshold in squared length, and it marks these locations as bad, so that they will not be used later. The program also makes some new representations of the orientation vectors – as angles, and as step-vectors of specified length - which it uses later for efficient tracing of the pseudoridges (an implementation detail).

Having finished with this preliminary processing, the program then starts to trace pseudoridges. Starting at each of a block of initial locations in the orientation array, it makes a pseudoridge by following the orientation flow, first in one of the two possible directions away from the initial point and then in the other direction. (For example, if the ridge orientation at an initial point is "northeast-southwest", then the program starts out in a northeast direction, and later comes back to the initial point and starts out in a southwest direction.) If a location has been marked as bad, then no trajectores are started there. A trajectory is stopped if it reaches a limit of the array of locations, or if it reaches a bad location, or if the turn required to continue the trajectory is excessively sharp, or if a specified maximum number of steps has been taken. The two trajectories traced out from an initial point are, in effect, joined end to end, producing a finished pseudoridge. The pseudoridge only approximately follows the ridges and valleys, and is insensitive to details such as bifurcations or small scars. The left part of figure 11 shows some pseudoridges found in the example fingerprint.

After the routine has finished tracing a pseudoridge, it goes through it from one end to the other and finds each maximal segment of turns that are either all left (or straight) turns, or all right turns. These segments can be thought of as lobes, each of which makes a sweep in a constant direction of curvature. If a lobe's orientation at its point of sharpest curvature (vertex) is sufficiently close to being horizontal, and in such a sense that the lobe could be approximately concave upward (not concave downward), and if additionally the lobe has, on each side of its vertex, at least a minimum amount of cumulative curvature, then the lobe qualifies as a concave upward shape. The routine checks each lobe of the current pseudoridge to find out if the lobe qualifies as a concave upward shape. If no lobe qualifies, it advances to the next location in the block of initial points and makes a new pseudoridge. The routine stops when it either finds a concave upward shape, or exhausts all lobes of all pseudoridges without finding one; its output is one bit of information, namely whether or not it found a concave upward shape. The right part of figure 11 shows a concave upward shaped lobe that was found.

This pseudoridge tracer is useful as a detector of whorls. It only very rarely produces a false positive, defined as finding a concave upward lobe in a print that is not a whorl, but often does produce a false negative, defined as not finding a concave upward lobe even though a print is a whorl. The next section describes a simple rule that is used to combine the pseudoridge tracer's output with the output of the main (PNN) classifier, thereby producing a hybrid classifier that is more accurate than the PNN alone.

[10] The array used has its constituent orientation vectors spaced half as far apart as those comprising the arrays used earlier, and it does not undergo registration.

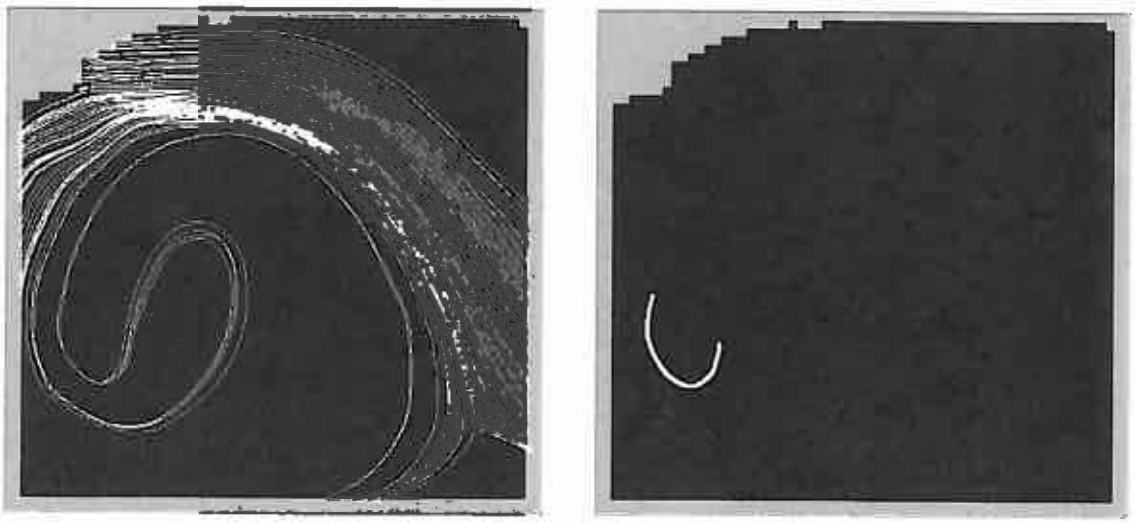

Figure 11: Left: Pseudoridges found in the example print until search was stopped by the finding of a concave-upward lobe; black region is the foreground (inked) part of the segmented image. Right: The concave-upward lobe that was found.

2.8 Combining the Two Classifiers

(Source file: lib/pca/combine.c)

This final processing module takes the outputs of the main Probabilistic Neural Network classifier and the auxiliary pseudoridge tracer, and makes the decision as to what class, and confidence, to assign to the fingerprint.

The PNN produces a hypothesized class and a confidence. The pseudoridge tracer produces a single bit of output, whose two possible values can be interpreted as *the print is a whorl* and *it is not clear whether the print is a whorl.* (The pseudoridge tracer is never sure that a print is *not* a whorl.) A simple rule was found to be sufficient for combining the PNN and pseudoridge tracer results, so as to produce a classifier more accurate than the PNN alone. The rule is described by this pseudocode:

```
if(pseudoridge tracer says whorl) {
  hypothesized_class = whorl
  if(pnn_hypothesized_class == whorl)
    confidence = 1.
  else
    confidence = .9
}
else { /* pseudoridge tracer says not clear whether whorl */
  hypothesized_class = pnn_hypothesized_class
  confidence = pnn_confidence
}
```

This is a reasonable way to use the pseudoridge tracer as a whorl detector, because as noted in the preceding section, this detector has very few false positives but a fair number of false negatives. So, if the whorl detector "fires" then the print is classified as a whorl even if the PNN disagrees, although disagreement results in slightly lower confidence, since firing implies that the print is almost certainly a whorl. If the whorl detector does not fire, then the PNN is allowed to set the classification and confidence. In this case, the PNN is allowed to hypothesize any class, even whorl, since non-firing of the whorl detector does *not* imply that the print is almost certainly not a whorl, but only that it is not the case that it almost certainly *is* a whorl.

Since the whorl detector fired for the example print and the PNN also classified this print as a whorl, the combining rule caused the final output of the classifier to be a hypothesized class of whorl and a confidence of 1. As it turns out, this is the best possible result that could have been obtained for this print, since it actually is a whorl.

The pseudoridge tracer improves the result for some prints that the PNN would have correctly classified as whorls anyway (such as the example print), by increasing the classification confidences. It also improves the result for some whorls that the PNN misclassifies, by causing them to be correctly classified as whorls. The tracer harms the result only for a very small number of prints, the non-whorls that it mistakenly detects to be whorls. The overall effect of combining the pseudoridge tracer with the main PNN classifier is a lowering of the error rate, compared to the rate obtained using the PNN alone.

3 Training the Classifier

The Probabilistic Neural Network classifier itself does not require a complex optimization process, but the full PCASYS classifier requires optimization of the regional weights that it uses to produce improved feature vectors for use by the PNN. After the regional weights are optimized, an additional parameter must be optimized, namely an overall smoothing factor for the PNN.

The following sections describe a training process. Each section shows the name of the command that should be used. The arguments or, for some commands, parameters files, are too lengthy to show here; complete usage instructions may be found by consulting the online *man pages*.

3.1 Make the Orientation Arrays

mkoas

This command reads Ihead image files of fingerprints and, from each image, extracts the orientation array (oa). This should be run on the full set of prototype prints that will ultimately be used to make the prototype feature vectors to be used by the PNN classifier.

3.2 Make the Covariance Matrix

meancov

This command reads a set of oas and computes their sample mean and sample covariance matrix.[11] It should be run on at least a reasonably large subset of the oas made by mkoas.

3.3 Diagonalize the Covariance Matrix

diag

This diagonalization program reads the covariance matrixand computes a subset of (largest) eigenvalues, and the corresponding eigenvectors. The eigenvalues are not needed in the training process, but may be of theoretical interest. The program calls a sequence of EISPACK routines[14].

3.4 Run the Karhunen-Loève Transform

lintran

This command applies a specified linear transform to a set of vectors. It should be run on a subset of the oas, with the transform matrix consisting of the eigenvectors made in the preceding step, i.e. each row of the transform matrix is an eigenvector. The resulting set of low-dimensional Karhunen-Loève (K-L) vectors will be used as data by optrws (optimize regional weights command, below). Note that these K-L vectors do not represent the entire prototype set: producing them for only a subset of the set of prototype prints keeps the number of these K-L vectors small enough so that the optrws command (optimize regional weights, below) will not take an unreasonably long time to run. Note also that these vectors are *not* of the form that will be taken by the ultimate prototype feature vectors that will be used by the finished classifier: they are merely intermediate representations, which will be useful when optimizing the regional weights. After that optimization is finished they will no longer be needed.

[11] The mean is not needed for further processing, but is computed because if multiple processors are available, it may be possible to save time by running several simultaneous instances of meancov on different subsets of the oas, then combining the resulting output files, using the **cmbmcs** command. To combine several covariances, cmbmcs needs the means as wells as the covariances of the subsets.

3.5 Optimize the Regional Weights

optrws

This command optimizes the regional weights. First it finds an optimal single value to which to set all the weights. Having thus defined an initial point in weight space, the program finishes the optimization by a very simple version of gradient descent. First it estimates (by secant method) the gradient of the activation error rate, when classifying a set of fingerprints by PNN, using the same set of prints as the PNN prototypes but leaving out of the proto set the particular one being classified. Then it searches the line pointing in the anti-gradient direction from the initial point, using a very simple method to find the minimum (or at least a local minimum) of the error along this line. The program then estimates the gradient there and does another downhill search. It stops after a specified number of iterations. A reasonable number of iterations is three or four, which may take several hours of time to run on a typical workstation if using a few thousand prints as the data. If several processors are available, it may be possible to save optrws runtime by setting its parameters so that, in one of its phases of processing, it spawns several processes to divide up the work. Consult the man page and the default parameters file mentioned in the man page to find out about this. If your operating system does implement fork() and execl(), which are required by the severalprocesses version of optrws, then optrws can be caused to link properly (i.e. without the fork and execl calls becoming unresolved references) by adding the argument -DNO_FORK_AND_EXECL to the definition of CLAGS in src/bin/optrws/Makefile. That will cause a different subset of the source code file to be compiled (conditional compilation).

In order to efficiently evaluate the error function at a point in regional-weights space, optrws produces the square matrix $\Psi^t \mathbf{W} \Psi$ of order NFEATS from the eigenvectors Ψ and the diagonal matrix $\mathbf{W}$ that is equivalent to the regional weights, then applies this matrix to all the K-L feature vectors before computing distances. This is only an approximation to the direct use of the regional weights, because of the use of only a partial set of eigenvectors, which also are not recomputed each time the weights are changed; but the results seem satisfactory, and the total runtime is much smaller than for direct methods.

3.6 Make the Transform Matrix

mktran

Reads the optimized regional weights made by optrws, and the eigenvectors, and makes the transform matrix $\Psi^t \mathbf{W}$, which will be used in the next step.

3.7 Apply the Transform Matrix

lintran

Lintran should next be run on the entire set of prototype oas made earlier, using the transform matrix made by mktran. The resulting feature vectors will be the prototype feature vectors used by the finished PNN classifier. The transform matrix both applies the optimal pattern of regional weights, and uses the eigenvectors to accomplish dimension reduction. (When the finished classifier runs on an incoming print, it applies this same transform matrix to the oa made from the print and then sends the resulting feature vector to the PNN. This approximately duplicates the effect that would have resulted if PNN had been used on the oas themselves, but with the optimized regional weights pattern applied before the distance computation.)

3.8 Optimize the Overall Smoothing Factor

optosf

Optimizes an overall smoothing factor (osf) that the PNN classifier will use. As noted above, the optimization of the regional weights should be done using the K-L vectors of only a subset of the prototype prints, to save time. Since the full set of prototypes will be used in the finished classifier, though, it is expected that better accuracy

will be obtained if the classifier uses an osf that is slightly larger than 1, rather than using 1 in effect as is the case during optimization since no overall smoothing factor is used then. This corresponds to Specht's observation [10] that as the number of prototypes increases, the optimal smoothing parameter σ decreases; increasing the osf corresponds to decreasing σ. (If the full prototype set was used to optimize the regional weights, then optosf should not be run: the osf should just be set to 1.)

Completing the above optimization process results in the finished PNN classifier data, consisting of prototype feature vectors, a transform matrix that will be applied to the oas of incoming prints, and the overall smoothing factor. The classification system then consists of the combination of the PNN classifier and the pseudoridge tracer. The latter program has many parameters that may be experimented with if desired, and also there is a parameter of the rule used to combine the PNN and the pseudoridge program. These parameters may be optimized by trial and error; reasonable values for them are provided in the default parameters files.

After all aspects of the classification system are settled, it can be tested by running it on a test set of fingerprints. The test set should of course be disjoint from the set of prints used in optimization and should even be disjoint from the alternate rollings of such prints, since the alternate rolling of a print is generally much more similar to it than is a randomly selected print. (In SD14, each "f" first rolling print has an alternate rolling, namely the corresponding "s" second rolling print. These two prints were made from the same finger of the same person, on different occasions.) To get the maximum value from whatever fingerprints are available, it is probably best to perform several instances of optimization/testing, each time using a different subset of the fingerprints for optimization and a different subset disjoint from the first subset, as the test set. That will help to overcome the problem of the measured error rate being strongly affected by the characteristics of one small test set, and will produce a better esimate of the error rate that could be expected if the classifier were to be run on a large number of randomly occuring prints.

4 Accuracy and Timing Results

4.1 Accuracy Results

The fingerprint images used to train and test the PCASYS classifier were taken from NIST Special Database 14 (SD14) [15]. This database consists of images scanned from 2700 pairs of standard fingerprint cards. Each pair of cards contains fingerprints taken from a single individual, but captured on two different occasions. One of them is the card stored in the FBI file for this person and is denoted the *file* card, and the other card is one that had been sent in to be searched against the database and is denoted the *search* card. Each card used was scanned at 500 dots per inch, and the resulting image was then used to produce 10 smaller images, one for each finger impression, by cutting out rectangles of predefined locations and dimensions, corresponding to the printed boxes in which the rolled finger impresssions are made.

We trained (optimized) the main PNN classifier using file prints f0000001.wsq through f0024300.wsq of SD14, which ultimately became the PNN prototypes. Training consisted of optimizing the regional weights and the overall smoothing factor, by the criterion of minimizing an activation error rate (a continuous kind of error rate, more suitable for minimization than a fraction of prints misclassified would be) when fingerprints were classified by PNN. Then, the finished classification system was made by adding to the optimized PNN the pseudoridge tracer, with its parameters set to values that had been arrived at much earlier as a result of testing. There is one additional parameter of the system, namely a confidence value that is assigned to a print's classification as *whorl* when the pseudoridge tracer finds that it is a whorl but the PNN has concluded that it is not a whorl; this was set to 0.9. With all aspects of the classification system settled, we then tested its accuracy by running it on search prints s0024301.wsq through s0027000.wsq of SD14, with the PNN part of the system using feature vectors of f0000001.wsq through f0024300.wsq as prototypes. The test set that was used is provided on the PCASYS CD in directory demofngs, in the form of the original fingerprint rasters. The classifier may be run on this entire set if desired, so as to duplicate the test results, or it may be run on a subset of these prints or on other prints provided by the user. The 24,300 prints from which the PNN prototype feature vectors are derived are not provided on the CD because there would not be enough space, but the prototype feature vectors themselves are provided.

The result of the test was an error rate (fraction of the test prints misclassified) of 7.78%. More insight into the behavior of the classifier can be obtained by examining the *confusion matrix* of table 1. This matrix has a row for each actual class and a column for each hypothesized class, and it shows, as the unparenthesized numbers, how many test prints fell into each (actual class, hyp. class) cell. For example, it shows that 779 of the L (left loop) prints were classified as L, and that 2 of them were classified as R. Each parenthesized number is the percentage that its corresponding count comprises of the sum of the counts in that row; for example, the parenthsized numbers show that 96.9% of the L prints were classified as L, and that 0.2% of them were classified as R. The entries shown in boldface correspond to correct classifications. Obviously this classifier is not at all good at classifying *scar* prints, but they were nevertheless left in the training and test sets, for simplicity and to avoid any possible optimistic bias.

The 7.78% error rate, and the confusion matrix, pertain to the use of the classifier without rejection: it is required to produce a hypothesized class for every print. However, if the classifier is allowed to reject some prints, i.e. to indicate that it is uncertain as to their class and that it does not accept the hypothesized class, then it can achieve an error rate much lower than 7.78% on the prints that it accepts. The confidence number produced by the classifier is used to provide an adjustable rejection level. To implement rejection, it is sufficient to set a confidence threshold, then reject all prints for which the classifier produces a confidence below the threshold. The larger a threshold is used, the greater is the percentage of the prints that are rejected (obviously), but also the smaller is the percentage of the accepted prints that are misclassified. The solid curve in figure 12 is an *error vs. reject* curve that summarizes this behavior, produced from the results of the test run. The dashed curve on the same graph is for a classifier consisting of the PNN alone, without the help of the pseudoridge analyzer; clearly the hybrid classifier is more accurate than the PNN alone, at all rejection levels.

Actual class	Hypothesized class					
	A	L	R	S	T	W
A	**43 (87.8)**	1 (2.0)	2 (4.1)	0 (0.0)	3 (6.1)	**0** (0.0)
L	4 (0.5)	**779 (96.9)**	2 (0.2)	0 (0.0)	6 (0.7)	13 (1.6)
R	4 (0.5)	11 (1.5)	**696 (94.7)**	0 (0.0)	8 (1.1)	16 (2.2)
S	**0** (0.0)	3 (60.0)	**1** (20.0)	**0 (0.0)**	0 (0.0)	1 (20.0)
T	20 (23.8)	25 (29.8)	15 (17.9)	0 (0.0)	**24 (28.6)**	0 (0.0)
W	2 (0.2)	45 (4.4)	28 (2.7)	0 (0.0)	0 (0.0)	**948 (92.7)**

Table 1: Confusion matrix. Unparenthesized: counts of how many prints fell into each cell. Parenthesized: percentages of row sums of counts. Boldface: correct classifications.

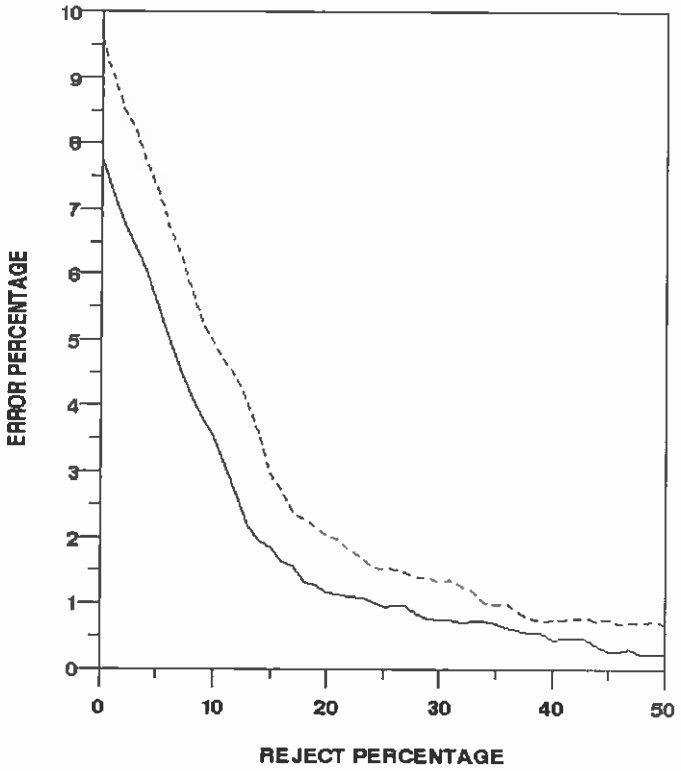

Figure 12: Percentage of accepted prints erroneously classified vs. percentage of prints rejected. Solid: hybrid classifier (PNN combined with pseudoridge-analyzing whorl detector). Dashed: PNN alone.

4.2 Timing Results

The classifier (in its non-graphical form) was timed on several computer models[12]: the results are shown in table 2. The default run of pcasysgn was run (i.e. no user parameters file), which classifies the first 20 prints of the demo set. The clock time reported by "time" was multiplied by the usage percentage and divided by the number of prints to produce the average seconds per print. Note that PCASYS is intended only as a prototype and demonstration program, and is not expected to achieve a throughput rate sufficient for use in a working AFIS.

Computer Model	Single-Precision and Optimization Options	Seconds
DEC Alpha	-fsingle -O2	7.0
HP 9000/735	+O3	5.0
IBM 7012/370	-fsingle -O3	6.7
SGI Challenge (IP19), using only one processor	-O2	7.5
Sun SPARCstation 2 w/ Weitek 80MHz CPU	-fsingle -O3	17.7
Sun SPARCstation 10	-fsingle -O3	16.7
Sun SPARCserver 4/470	-fsingle -O3	32.9

Table 2: Timing results for various workstation models.

[12] Various commercial equipment may be identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology.

5 Possible Future Work

The PCASYS program implements the best fingerprint classification algorithms that we have found as a result of our experiments, but we think that considerable accuracy improvement may still be possible without radically changing the overall structure of the system. The following are some of the possible avenues of further research that seem promising:

- A Multilayer Perceptron (MLP) with a sine activation function could be used in place of the PNN. We have already tried an MLP with a sigmoid activation function, producing results slightly worse than those of the PNN, but the sinusoidal version has better training dynamics than the sigmoidal version, and has already produced very good results in experiments done at NIST [16, 17, 18].
- Error-correcting codes could be used as an output representation for an MLP (probably the sinusoidal version), instead of the usual one-node-per-class representation. This has been found to improve accuracy [19].
- More information could be extracted from the pseudoridges than is done in our current system, such as by using histograms of the cumulative turning angles or by using rules designed to detect patterns other than concave-upward shapes. These are among the possible uses of pseudoridges pointed out by Smith [13].

6 Installing and Running the Classifier

6.1 Installing the Classifier

Instructions for installing the classifier demos and other commands are contained in the readme.txt file on the distribution CD-ROM. That file contains instructions for running the installation script, install.sh, and indicates the setup procedures that are needed in order to be able to run the commands.

6.2 PCASYS Data Files

For the purpose of conveniently storing and transporting data, formats have been defined for three types of data files:

matrix A matrix of real numbers.

covariance A covariance matrix of real numbers. This format saves disk space by storing only the nonstrict lower triangle, which is sufficient because a covariance matrix is symmetric.

classes A list of classes, thought of as unsigned characters. For use with fingerprints in pcasys, class values 0 through 5 denote arch, left loop, right loop, scar, tented arch, and whorl. A classes file can be used for any classification situation with no more than 256 classes, though.

Each type of file can exist in either an ascii or a binary storage mode. A data file contains header information followed by the data itself. The header information consists of: a description string (can be of any length, but must contain no newlines; can be left empty); code bytes indicating the file type and storage mode; and additional information specific to the file type (if matrix, the two dimensions; if covariance, the order, i.e. what both dimensions of the symmetric matrix are, and the number of vectors used to build the covariance; if classes, the number of elements.) The **datainfo** command can be run on any PCASYS data file; it writes a report of the header information to the standard output. (Caution: datainfo produces meaningful output only if for matrix, covariance, and classes files. Do not run it on IHead image rasters, text files, source code files, binary executables, etc.)

The directory pcpftm on the CD contains three data files needed by the classifier demos: the prototype feature vectors used by the PNN (profvs.asc, a matrix file with each feature vector stored as one row); their classes (procls.asc, a classes file); and a transform matrix that the demos apply to the orientation array of each demo fingerprint to produce the feature vector to be sent to the PNN (tranmat.asc, a matrix file). These files are provided in ascii form as a way of transporting the data to any machine that PCASYS is to be installed on, regardless of the binary data storage format used by that machine. Machines can vary in the byte ordering used and in the format of floating-point numbers. However, files containing large quantites of data (in particular, the prototype feature vectors) are best stored in binary format, to save i/o time. So, the installation script converts the ascii data files on the CD to binary form and stores the resulting files in pcasys/data. (Because the ascii to binary converter command, asc2bin, will have been compiled locally during the compilation phase of installation, it will produce binary files appropriate for the local machine.)

6.3 Commands

Installation of PCASYS results in the production of the following commands, shown here with short descriptions. For a complete description and usage instructions for any of these commands, consult the provided online *man page*. (To prepare to use the man pages, either edit .cshrc or .profile so that *wherever_pcasys_is_installed*/pcasys/man is added to MANPATH, or make an alias for man -M *that_directory*, which can then be used to consult only PCASYS man pages).

6.3.1 Classifier Demos

pcasysgn non-graphical demo

pcasysgy graphical demo

6.3.2 Training (Optimization) Commands

diag finds some eigenvalues and eigenvectors

lintran runs a linear transform on a set of vectors

meancov makes mean and covariance from a set of vectors

mkoas makes orientation arrays from fingerprints

mktran makes transform matrix incorporating the optimized regional weights

optosf optimizes the overall smoothing factor

optrws optimizes the regional weights

6.3.3 Utility Commands

asc2bin converts an ascii data file to binary

bin2asc converts a binary data file to ascii

chgdesc changes the description string of a data file

cmbmcs combines several mean/covariance file pairs

datainfo reports the header info of a data file to standard output

dpyimage displays an IHead image file on the terminal screen

oas2pics makes IHead pictures of orientation arrays

rwpics makes IHead pictures of regional weights or estimated gradients

stackms stacks several *matrix* files together

sun2ihdr converts a Sun raster file to an IHead file

6.4 Running the Classifier

6.4.1 Graphical and Non-graphical Versions

The classifier has a graphical version (command name pcasysgy) and a non-graphical version (pcasysgn). The graphical version, which requires the X Window System, produces windows on the screen containing graphics which show the results of the phases of processing used to classify each fingerprint. (Many of the illustrations in this report were made from screen dumps of the graphical demo.) The non-graphical version classifies the fingerprints but produces no graphics; it is suitable if you do not have X Windows, or for greatest running speed. Both versions optionally produce a stream of messages on the terminal showing which fingerprint the classifier is working on and what phase of processing it is performing, and both versions produce an output file.

6.4.2 Default Parameters and Specifying Parameters

After installation, each user who wants to run the pcasys commands should edit .cshrc or .profile so as to add the pcasys/bin directory to the path, and should make a directory .pcasys in user's home directory, containing two symbolic links. These should be: cd_mount_point, pointing to wherever the CD is mounted; and inst_parentdir, pointing to the parent directory of the pcasys installation hierarchy. (The demos need to know the CD mount point to find the demo fingerprint images on the CD, and they need to know the installation parent dir so they can find the binary files installed in pcasys/data.) Then, to run the classifier in default mode, it is sufficient to type the command name, for the graphical or the non-graphical version, without arguments: "pcasysgy" or "pcasysgn". In default mode, the demo runs on the first 10 of the 2700 provided demo fingerprints.

Alternatively, the user may produce a parameter file and use its name as an argument: "pcasysgy *prsfile*" or "pcasysgn *prsfile*". Whether or not the user provides a parameter file, the demo reads two default parameter files in the pcasys/data/dfprs hierarchy, first oas.prs (parameters affecting the making of orientation arrays) and then add/pcasysgy.prs or add/pcasysgn.prs as appropriate (additional parms for demo). If the user provides a parameter file, the demo then reads it, and whatever values are specified in it override those found in the default files. Consult the default files to find out what the parameters are and as an example of the format for a parameters file (a name-value pair on each line, with a pound sign to indicate that the rest of a line is a comment).

6.4.3 Output File

The output file produced by running the classifier in its default mode is as follows:

```
s0024301.wsq: is W; pnn: hyp R, conf 0.66; conup y; hyp W, conf 0.90; right
s0024302.wsq: is R; pnn: hyp R, conf 0.84; conup n; hyp R, conf 0.84; right
s0024303.wsq: is R; pnn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024304.wsq: is R; pnn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024305.wsq: is R; pnn: hyp R, conf 0.99; conup n; hyp R, conf 0.99; right
s0024306.wsq: is L; pnn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024307.wsq: is L; pnn: hyp L, conf 0.85; conup n; hyp L, conf 0.85; right
s0024308.wsq: is L; pnn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
s0024309.wsq: is L; pnn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
s0024310.wsq: is L; pnn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024311.wsq: is R; pnn: hyp R, conf 0.99; conup n; hyp R, conf 0.99; right
s0024312.wsq: is W; pnn: hyp W, conf 0.99; conup y; hyp W, conf 1.00; right
s0024313.wsq: is R; pnn: hyp R, conf 0.99; conup n; hyp R, conf 0.99; right
s0024314.wsq: is R; pnn: hyp R, conf 0.89; conup n; hyp R, conf 0.89; right
s0024315.wsq: is R; pnn: hyp R, conf 0.94; conup n; hyp R, conf 0.94; right
s0024316.wsq: is L; pnn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024317.wsq: is L; pnn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
s0024318.wsq: is L; pnn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024319.wsq: is W; pnn: hyp W, conf 0.99; conup y; hyp W, conf 1.00; right
s0024320.wsq: is L; pnn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right

pct error: 0.00

      A          L          R          S          T          W
A     0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )
L     0(  0.0)   9(100.0)   0(  0.0)   0(  0.0)   0(  0.0)   0(  0.0)
R     0(  0.0)   0(  0.0)   8(100.0)   0(  0.0)   0(  0.0)   0(  0.0)
S     0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )
T     0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )   0(  -  )
W     0(  0.0)   0(  0.0)   0(  0.0)   0(  0.0)   0(  0.0)   3(100.0)
```

The output file has a line for each of the fingerprints that were classified. Each line shows: the last component of the fingerprint image filename; its actual class (A, L, R, S, T, and W stand for the pattern-level classes arch, left loop, right loop, scar, tented arch, and whorl); the output of the main Probabilistic Neural Network (PNN) classifier (a hypothesized class and a confidence); the output of the auxiliary pseudoridge-tracing whorl detector (whether or not a concave-upward shape, a conup, was found); the final output of the hybrid classifier, which is a hypothesized class and a confidence; and whether this hypothesized class was right or wrong.

The last part of the output file is a brief summary of the results. First, there is the percent error, i.e. the percentage of the fingerprints that were classified incorrectly. Following this is a *confusion matrix*. It has the same format as table 1, described in 4.1.

7 Acknowledgements

The authors thank M. Gilchrist and R. Chellappa for helpful advice and suggestions, and R. Boisvert, W. Rogers, and S. Janet for help with the software development and testing. The makefiles are derived from ones developed by M. Garris. We thank the Federal Bureau of Investigation for supporting our research.

A The IHead Image File Format

After digitization, certain attributes of an image are required to correctly interpret the 1-dimensional pixel data as a 2-dimensional image. Examples of such attributes are the pixel width and pixel height of the image. These attributes are stored in a machine readable header prefixed to the raster bit stream. A program that manipulates the raster data of an image is able to first read the header containing these attributes and determine the proper interpretation of the data that follows it.

NIST has designed, implemented, and distributed images based on this paradigm. A header format named IHead has been developed for use as an image interchange format. Numerous image formats exist; some are widely supported on small personal computers, others supported on larger workstations; most are proprietary formats, few are public domain. IHead is an attempt to design an open image format which can be universally implemented across heterogeneous computer architectures and environments. IHead has been successfully ported and tested on several systems including: UNIX workstations and servers, DOS personal computers, and VMS mainframes. Both documentation and source code for the IHead format are publicly available. IHead has been designed with an extensive set of attributes in order to: adequately represent both binary and gray level images; represent images captured from different scanners and cameras; and satisfy the image requirements of diversified applications, including but not limited to, image archival/retrieval, character recognition, and fingerprint classification.

The IHead structure definition, written in the C programming language, is as follows:

```
/*********************************************************/
/*         File Name: IHead.h                            */
/*         Package:   NIST Internal Image Header         */
/*         Author:    Michael D. Garris                  */
/*         Date:      2/08/90                            */
/*********************************************************/
/* Defines used by the ihead structure */
#define IHDR\_SIZE 288 /* len of hdr record (always even bytes) */
#define SHORT\_CHARS  8 /* # of ASCII chars to represent a short */
#define BUFSIZE 80 /* default buffer size */
#define DATELEN 26 /* character length of date string */

typedef struct ihead{
   char id[BUFSIZE]; /* identification/comment field */
   char created[DATELEN]; /* date created */
   char width[SHORT\_CHARS]; /* pixel width of image */
   char height[SHORT\_CHARS]; /* pixel height of image */
   char depth[SHORT\_CHARS]; /* bits per pixel */
   char density[SHORT\_CHARS]; /* pixels per inch */
   char compress[SHORT\_CHARS]; /* compression code */
   char complen[SHORT\_CHARS]; /* compressed data length */
   char align[SHORT\_CHARS]; /* scanline multiple: 8|16|32 */
   char unitsize[SHORT\_CHARS]; /* bit size of image memory units */
   char sigbit; /* 0->sigbit first | 1->sigbit last */
   char byte\_order; /* 0->highlow | 1->lowhigh*/
   char pix\_offset[SHORT\_CHARS]; /* pixel column offset */
   char whitepix[SHORT\_CHARS]; /* intensity of white pixel */
   char issigned; /* 0->unsigned data | 1->signed data */
   char rm\_cm; /* 0->row maj | 1->column maj */
   char tb\_bt; /* 0->top2bottom | 1->bottom2top */
   char lr\_rl; /* 0->left2right | 1->right2left */
   char parent[BUFSIZE]; /* parent image file */
   char par\_x[SHORT\_CHARS]; /* from x pixel in parent */
```

```
   char par\_y[SHORT\_CHARS]; /* from y pixel in parent */
}IHEAD;

/* General Defines */
#define UNCOMP 0
#define CCITT\_G3 1
#define CCITT\_G4 2
#define LZW 3
#define RL\_LZW 4
#define RL 5
#define JPEG 6
#define WSQ 7
#define MSBF '0'
#define LSBF '1'
#define HILOW '0'
#define LOWHI '1'
#define UNSIGNED '0'
#define SIGNED '1'
#define ROW\_MAJ '0'
#define COL\_MAJ '1'
#define TOP2BOT '0'
#define BOT2TOP '1'
#define LEFT2RIGHT '0'
#define RIGHT2LEFT '1'

#define BYTE\_SIZE 8.0

extern IHEAD *calerahdr();
extern IHEAD *cuthdr();
extern void nullihdr();
extern void parseihdrid();
extern void printihdr();
extern IHEAD *readihdr();
extern void writeihdr();
```

The following listing shows the header values from an IHead file, corresponding to these structure members. This header information belongs to demo fingerprint s0024301.wsq:

```
IMAGE FILE HEADER
-----------------
Identity :  s0024301.wsq m i do
Header Size :  288 (bytes)
Date Created :  Wed Apr 21 22:37:17 1993
Width :  832 (pixels)
Height :  768 (pixels)
Bits per Pixel :  8
Resolution :  500 (ppi)
Compression :  7 (code)
Compress Length :  37446 (bytes)
Scan Alignment :  8 (bits)
Image Data Unit :  8 (bits)
Byte Order :  High-Low
MSBit :  First
Column Offset :  0 (pixels)
```

```
White Pixel :  255
Data Units :  Unsigned
Scan Order :  Row Major,
   Top to Bottom,
   Left to Right
Parent :  tape205.n0407065.01 4096x1536
X Origin :  0 (pixels)
Y Origin :  0 (pixels)
```

The first attribute field of IHead is the identification field, id. This field uniquely identifies the image file, typically by a file name. In the fingerprint files of NIST Special Database 14 (of which the provided demo fingerprints are a subset), the id field contains, after thte file name, various information about the fingerprint: sex (m or f), inked or live scan (i or l), and the classification of the fingerprint (in this case, do, meaning double loop whorl with outer tracing).

The attribute field, created, is the date on which the image was captured or digitized. The next three fields hold the image's pixel width, height, and depth. A binary image has a pixel depth of 1 whereas a gray scale image containing 256 possible shades of gray has a pixel depth of 8. The attribute field, density, contains the scan resolution of the image; in this case, 500 ppi. The next two fields deal with compression.

In the IHead format, images may be compressed with virtually any algorithm. Whether the image is compressed or not, the IHead is always uncompressed. This enables header interpretation and manipulation without the overhead of decompression. The compress field is an integer flag which signifies which compression technique, if any, has been applied to the raster image data which follows the header. If the compression code is zero, then the image data is not compressed, and the data dimensions: width, height, and depth, are sufficient to load the image into main memory. However, if the compression code is nonzero, then the complen field must be used in addition to the image's pixel dimensions. For example, the image described here has a compression code of 7. By convention, this signifies that WSQ compression has been applied to the image data prior to file creation. In order to load the compressed image data into main memory, the value in complen is used to load the compressed block of data into main memory. Once the compressed image data has been loaded into memory, WSQ decompression can be used to produce an image which has the pixel dimensions consistent with those stored in its header.

The attribute field, align, stores the alignment boundary to which scan lines of pixels are padded. Pixel values of binary images are stored 8 pixels (or bits) to a byte. Most images, however, are not an even multiple of 8 pixels in width. In order to minimize the overhead of ending a previous scan line and beginning the next scan line within the same byte, a number of padded pixels are provided in order to extend the previous scan line to an even byte boundary. Some digitizers extend this padding of pixels out to an even multiple of 8 pixels, other digitizers extend this padding of pixels out to an even multiple of 16 pixels. This field stores the image's pixel align- ment value used in padding out the ends of raster scan lines.

The next three attribute fields identify binary interchanging issues among heterogeneous computer architectures and displays. The unitsize field specifies how many contiguous pixel values are bundled into a single unit by the digitizer. The sigbit field specifies the order in which bits of significance are stored within each unit: most significant bit first or least significant bit first. The last of these three fields is the byte_order field. If unitsize is a multiple of bytes, then this field specifies the order in which bytes occur within the unit. Given these three attributes, binary incompatibilities across computer hardware and binary format assumptions within application software can be identified and effectively dealt with.

The pix_offset attribute defines a pixel displacement from the left edge of the raster image data to where a particular image's significant image information begins. The whitepix attribute defines the value assigned to the color white. For example, the binary image described in Figure 7 is black text on a white background and the value of the white pixels is 0. This field is particularly useful to image display routines. The issigned field is required to specify whether the units of an image are signed or unsigned. This attribute determines whether an image with a pixel depth of 8, should have pixels values interpreted in the range of -128 to +127, or 0 to 255. The orientation of the raster scan may also vary among different digitizers. The attribute field, rm_cm, specifies whether the digitizer captured the

image in row-major order or column-major order. Whether the scan lines of an image were accumulated from top to bottom, or bottom to top, is specified by the field, tb bt, and whether left to right, or right to left, is specified by the field, rl_lr.

The final attributes in IHead provide a single historical link from the current image to its parent image; the one from which the current image was derived or extracted. In the example, the parent field shows the name of a large raster, scanned from a fingerprint card, from which the current raster, depicting one box of the card, was cut. In general, the x origin and y origin field contain the upper left hand corner pixel coordinate from where the extraction took place from the parent image, but for these fingerprint images the x origin and y origin fields have been set to zeros. We believe that the IHead image format contains the minimal amount of ancillary information required to successfully manage binary and gray scale images.

References

[1] *The Science of Fingerprints*. U. S. Department of Justice, Washington, DC, 1984.

[2] Automated classification system reader project (ACS). Technical report, DeLaRue Printrak Inc., February 1985.

[3] Automated Fingerprint Classification Study, Phase I Final Report. Technical report, Ektron Applied Imaging, May 1985.

[4] C. I. Watson, G. T. Candela, and P. J. Grother. Comparison of FFT Fingerprint Filtering Methods for Neural Network Classification. Technical Report NISTIR 5493, National Institute of Standards and Technology, September 1994.

[5] R. M. Stock and C. W. Swonger. Development and evaluation of a reader of fingerprint minutiae. *Cornell Aeronautical Laboratory*, Technical Report CAL No. XM-2478-X-1:13-17, 1969.

[6] J. H. Wegstein. An automated fingerprint identification system. *National Institute of Standards and Technology*, NBS Special Publication 500-89, Febuary 1982.

[7] Anil K. Jain. *Fundamentals of Digital Image Processing*, chapter 5.11, pages 163 174. Prentice Hall Inc., prentice hall international edition, 1989.

[8] C. L. Wilson, G. T. Candela, and C. I. Watson. Neural-network fingerprint classification. *Journal of Artificial Neural Networks*, 1(2):203-228, 1994.

[9] Donald F. Specht. Enhancements to Probabilistic Neural Networks. In *International Joint Conference on Neural Networks*, pages I-761 - I-768, June 1992.

[10] D. F. Specht. Probabilistic neural networks. *Neural Networks*, 3(1):109-118, 1990.

[11] B. V. Dasarathy, editor. *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*. IEEE Computer Society Press, 1991.

[12] P. J. Grother, G. T. Candela, and J. L. Blue. Fast Implementations of Nearest Neighbor Classifiers. *IEEE PAMI*, March 1995, to be published.

[13] W. R. Smith. Improved Feature Set for Fingerprint Image Classification. In *Proceedings from the Research in Criminal Justice Information Services Technology Symposium*, pages C-111 - C-127, September 1993.

[14] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix System Routines - EISPACK Guide*. Springer-Verlag, 1976.

[15] C. I. Watson. Mated Fingerprint Card Pairs 2. Technical Report Special Database 14, **MFCP2**, National Institute of Standards and Technology, September 1993.

[16] Charles L. Wilson, James L. Blue, and Omid M. Omidvar. Improving Neural Network Performance for Character and Fingerprint Classification by Altering Network Dynamics. Technical Report NISTIR 5695, National Institute of Standards and Technology, 1995.

[17] Charles L. Wilson, James L. Blue, and Omid M. Omidvar. Improving Neural Network Performance for Character and Fingerprint Classification by Altering Network Dynamics. In *World Congress on Neural Networks Proceedings II*, pages 151 - 158, Washington DC, July 1995.

[18] James L. Blue. Sine Activation in Neural Networks. Technical report, National Institute of Standards and Technology, 1995, to be published.

[19] Thomas G. Dietterich and Ghulum Bakiri. Solving Multiclass Learning Problems via Error-Correcting Output Codes. *Journal of Artificial Intelligence Research 2*, pages 263-286, 1995.