User's Guide to
NIST Biometric Image Software (NBIS)

Craig I. Watson (cwatson@nist.gov)

Michael D. Garris (mgarris@nist.gov)

Elham Tabassi (elham.tabassi@nist.gov)

Charles L. Wilson (cwilson@nist.gov)

R. Michael McCabe (mccabe@nist.gov)

Stanley Janet (sjanet@nist.gov)

Kenneth Ko (kenko@nist.gov)

National Institute of Standards and Technology
Bldg. 225, Rm. A216
100 Bureau Drive, Mail Stop 8940
Gaithersburg, MD 20899-8940

ACKNOWLEDGEMENTS

We would like to acknowledge the Federal Bureau of Investigation and the Department of Homeland Security who provided funding and resources in conjunction with NIST to support the development of this fingerprint image software.

NOTE TO READER

This document provides guidance on how the NIST Biometric Image Software (NBIS) non-export controlled packages are installed and executed. Its content and format is one of user's guide and reference manual. Some algorithmic overview is provided, but more details can be found in the cited references.

The Table of Contents provides the reader a map into the document, and the hyperlinks in the electronic version enable the reader to effectively navigate the document and locate desired information. These hyperlinks are unavailable when using a paper copy of the document. Any updates to this software will be posted NIST Image Group's Open Source Sever (NIGOS).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

User's Guide to NIST Biometric Image Software (NBIS)

C. I. Watson, M. D. Garris, E. Tabassi, C. L. Wilson, R. M. McCabe, S. Janet and K. Ko

ABSTRACT

This report documents the biometric image software distribution developed by the National Institute of Standards and Technology (NIST) for the Federal Bureau of Investigation (FBI) and Department of Homeland Security (DHS). Provided is a collection of application programs, utilities, and source code libraries. The NBIS software is organized in two categories: non-export controlled and export controlled. The non-export controlled NBIS software is organized into five major packages: 1. `PCASYS` is a neural network based fingerprint pattern classification system; 2. `MINDTCT` is a fingerprint minutiae detector; 3. `NFIQ` is a neural network based fingerprint image quality algorithm, 4. `AN2K` is a reference implementation of the ANSI/NIST-ITL 1-2000 "Data Format for the Interchange of Fingerprint, Facial, Scar Mark & Tattoo (SMT) Information" standard; and 5. `IMGTOOLS` is a collection of image utilities, including encoders and decoders for Baseline and Lossless JPEG and the FBI's WSQ specification. The export controlled NBIS software is organized into two major packages: 1. `NFSEG` is a fingerprint segmentation system useful for segmenting four-finger plain impressions, 2. `BOZORTH3` is a minutiae based fingerprint matching system. **The `NFSEG` and `BOZORHT3` software are subject to U.S. export control laws** It is our understanding that `NFSEG` and `BOZORTH3` software fall within ECCN 3D980, which covers software associated with the development, production or use of certain equipment controlled in accordance with U.S concerns about crime control practices in specific countries. This source code is written in ANSI "C", and has been developed to compile and execute under the Linux operating system and MAC OS-X operating system using the GNU `gcc` compiler and `gmake` utility. The source code may also be installed to run on Win32 platforms that have the Cygwin library and associated tools installed. A Reference Manual describing each program in the distribution is included at the end of this document.

Keywords: ANSI/NIST, DHS, FBI, fingerprint, image, JPEG, fingerprint segmentation, minutiae detection, minutiae matching, fingerprint image quality, neural network, pattern classification, WSQ, wavelet scalar quantization.

1. INTRODUCTION

This report documents the open source release of the biometric image software distribution developed by the National Institute of Standards and Technology (NIST) for the Federal Bureau of Investigation (FBI) and Department of Homeland Security (DHS). Its content and format is one of user's guide and reference manual. While some algorithmic overview is provided, the cited references contain more details of how these fingerprint software technologies work.

As background, the FBI has been utilizing computer technology to help capture, store, and search fingerprint records since the late 70's. In the early 90's, they began developing a system to enable the electronic exchange of fingerprint records and images by law enforcement agencies and to handle electronic transactions with these agencies. This system is called the Integrated Automated Fingerprint Identification System (IAFIS) and it is currently in operation in Clarksburg, West Virginia.

IAFIS has been primarily designed to process fingerprints that have been captured at a booking station of a jail or that are being submitted for a civilian background check. These types of fingerprints are typically taken by inking and rolling the fingertip onto a paper fingerprint card or captured from the surface of a live scan device. Traditionally these fingerprints have been referred to as *tenprints*, as all ten fingers are typically captured.

Over the years, the FBI has accumulated more than 40,000,000 fingerprint cards on file, and they handle up to 60,000 fingerprint-related requests a day. This demand originated from the need to support criminal investigations, but through the successful development and deployment of technology to meet this demand, legislation continues to be passed progressively increasing the demand for conducting civilian checks and clearances. The workload, which was once 80 % criminal and 20 % civilian, is quickly approaching 50 % criminal and 50 % civilian, and demand is projected to rapidly grow. In light of this situation, the FBI must continue to pursue the development and exploitation of new technologies, and NIST is partnered with the FBI in support of this pursuit.

NIST has a long-standing relationship with the FBI. Researchers at NIST began work on the first version of the FBI's AFIS system back in the late 60's. Over the years, NIST has conducted fingerprint research, developed fingerprint identification technology and data exchange standards, developed methods for measuring the quality and performance of fingerprint scanners and imaging systems, and produced databases containing a large repository of FBI fingerprint images for public distribution.[1]-[30], [56]-[60]

NIST is equally pleased to simultaneously support DHS. DHS is tasked with protecting the nations borders against persons attempting to enter the United States illegally. The importance of securing the nations borders was significantly increased after the terrorist attacks of September 11, 2001. Since 9-11, the phrase, “Everything has changed,” has been frequently stated. This is no less true for the Image Group at NIST. Within a couple of months, new initiatives were started that redirected work focused on law enforcement to new work focused on border control.

After 9-11 Congress passed several new laws to improve border security. These include the USA PATRIOT Act (PL 107-56) and the Enhanced Border Security and Visa Entry Reform Act

(PL 107-173) which directly cited participation and contribution from NIST in the area of biometric standards development and certification.

The USA PATRIOT Act as amended by the Enhanced Border Security and Visa Entry Reform Act, directs that the Attorney General and the Secretary of State jointly, through NIST "shall develop and certify a technology standard, including appropriate biometric identifier standards, that can be used to verify the identity of persons applying for a United States visa or such persons seeking to enter the United States pursuant to a visa for the purposes of conducting background checks, confirming identity, and ensuring that a person has not received a visa under a different name."

The Enhanced Border Security and Visa Entry Reform Act states that "In the development and implementation of the data system under this subsection, the President shall consult with the Director of the National Institute of Standards and Technology (NIST) and any such other agency as may be deemed appropriate."

These standards apply to visa documents issued by the US government. A visa waiver country is required "to issue to its nationals machine-readable passports that are tamper-resistant and incorporate biometric and document authentication identifiers that comply with applicable biometric and document identifying standards established by the International Civil Aviation Organization."

To carry out the legislated requirements of Congress NIST required a versatile *open* system for conducting applied research. Much of the new code in this release of NBIS is from the Verification Test Bed (VTB) system [56] that is being used by NIST. This system allows NIST to develop fingerprint evaluation methods and protocols for evaluating baseline technology, provide a large computation capacity for conducting fingerprint matches, segment four-finger plain impressions so that rolled vs. plain studies can be conducted, build large databases, and conduct automated data quality checks to create repositories for use in future evaluations and on prototype production systems. The VTB performance is being compared against other matchers in ongoing studies: its role at NIST is to serve as a minimum standard baseline for fingerprint matcher performance, and to allow comparative analysis of different types of fingerprint data.

The software technology contained in this distribution is a culmination of more than a decade's worth of work for the FBI and DHS at NIST. Provided is a collection of application programs, utilities, and source code libraries that are organized in two categories: non-export controlled and export controlled. The non-export controlled NBIS software includes five major packages: (`PCASYS`, `MINDTCT`, `NFIQ`, `AN2K`, and `IMGTOOLS`). The export controlled NBIS software includes two major packages: (`NFSEG and BOZORTH3`).

The Non-Export Control NBIS source code is managed using Perforce[1] source code management system on the NIST Image Group Open Source Sever (NIGOS). To obtain the latest version of NBIS non-export controlled source code (`PCASYS`, `MINDTCT`, `NFIQ`, `AN2K`, and `IMGTOOLS`), please visit http://www.itl.nist.gov/lad/894.03/nigos/nigos.html for the instructions on accessing NIGOS. Also, the non-export controlled NBIS source code is available as a zip file which is

[1] Specific software products and equipment identified in this paper were used in order to adequately support the development of the technology described in this document. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the software or equipment identified is necessarily the best available for the purpose.

updated daily at http://www.itl.nist.gov/lad/894.03/nigos/nigos.html. The export controlled NBIS source code (NFSEG and BOZORTH3) is only available on CD-ROM upon request. For detailed information on how the request the export controlled NBIS source code, please follow the procedure in APPENDIX B.

The first open source package, PCASYS, is a pattern classification system designed to automatically categorize a fingerprint image as an arch, left or right loop, scar, tented arch, or whorl. Identifying a fingerprint's class effectively reduces the number of candidate searches required to determine if a fingerprint matches a print on file. For example, if the unknown fingerprint is an arch, it only needs to be compared against all arches on file. These types of "binning" strategies are critical for the FBI to manage the searching of its fingerprint repository. Section 3.1 describes this package in more detail, and Section 5.1 provides an algorithmic overview and functional description of the PCASYS.

The second open source package, MINDTCT, is a minutiae detection system. It takes a fingerprint image and locates features in the ridges and furrows of the friction skin, called *minutiae*. Points are detected where ridges end or split, and their location, type, orientation, and quality are stored and used for search. There are 100 minutiae on a typical tenprint, and matching takes place on these points rather than the 250,000 pixels in the fingerprint image. Section 3.2 describes this package in more detail, and Section 5.2 provides an algorithmic overview and functional description of the MINDTCT.

Techniques of fingerprint pattern classification and minutiae detection typically share some functionality. For example, both applications typically derive and analyze ridge flow within a fingerprint image.[31] This is true of NFSEG, PCASYS and MINDTCT, as all conduct image binarization of the fingerprint as well. It should be noted that these systems were developed independently of each other, so although these processing steps are in common, different algorithms are applied in each. Further study is required to determine if one system's algorithmic approach is better than the other.

The third open source package, NFIQ, is a fingerprint image quality algorithm. It takes a fingerprint image and analyzes the overall quality of the image returning an image quality number ranging from 1 for highest quality to 5 for lowest. The quality of the image can be extremely useful in knowing the likely performance of a fingerprint matcher on that image. Section 3.3 describes the package in more detail, and Section 5.3 provides an algorithmic description of the NFIQ.

A significant contribution NIST has made to the FBI, and to the fingerprint and law enforcement communities at large, is the development of the ANSI/NIST-ITL 1-2000 "Data Format for the Interchange of Fingerprint, Facial, Scar Mark & Tattoo (SMT) Information" standard.[30] This standard defines a common file format, available to law enforcement agencies in the U.S. since 1986, for the electronic exchange of fingerprint images and related data.[9] Today, it supports other types of images as well, including palmprints, mugshots, scars, and tattoos. This standard has been adopted by all major law enforcement agencies in the U.S., including the FBI, and has strong support and use internationally. IAFIS is implemented on this data interchange standard. For the purposes of abbreviation, this standard will be referred to in this documentation as the "ANSI/NIST" standard.

The fourth open source package, AN2K, contains a suite of utilities linked to a reference library implementation of the ANSI/NIST-ITL 1-2000 standard. These utilities facilitate reading, writing, manipulating, editing, and displaying the contents of ANSI/NIST files. Section 3.4 describes this package in more detail.

The last open source package, IMGTOOLS, is a large collection of general-purpose image utilities. Included are image encoders and decoders supporting Baseline JPEG, Lossless JPEG, and the FBI's specification of Wavelet Scalar Quantization (WSQ). There are utilities supporting the conversion between images with interleaved and non-interleaved color components; colorspace conversion between RGB and YCbCr; and the format conversion of legacy files in NIST databases. Section 3.5 describes this package in more detail.

The first export controlled package, NFSEG, is a fingerprint segmentation system. It takes a four-finger plain impression fingerprint image (called a slap) and segments it into four separate fingerprint images. These single finger plain impression images can then be used for single finger matching versus either rolled images or other plain impression fingerprint images. NFSEG will also take a single finger rolled or plain impression image and isolate the fingerprint area of the image by removing the white space. Details of the NFSEG package including the algorithmic description is provided on CD-ROM.

The second export controlled package, BOZORTH3, is a fingerprint matching system. It uses the minutiae detected by MINDTCT to determine if two fingerprints are from the same person, same finger. It can analyze fingers two at a time or run in a batch mode comparing a single finger (probe) against a large database of fingerprints (gallery). Details of the BOZORTH3 package are also provided on CD-ROM.

The source code in this distribution has been developed using the GNU project's gcc compiler and gmake utility.[32] The software has been tested under Linux, MAC OS-X and under Windows 2000 using the Cygwin library [33] and associated tools.

The baseline JPEG code in NBIS/Main/ijg/src/lib/jpegb uses the Independent JPEG Group's compression/decompression code. For details on its copyright and redistribution, see the included file NBIS/Main/ijg/src/lib/jpegb/README.

This software was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibilities associated with its operation, modification, and maintenance

In this document, Section 2 provides instructions for installing and compiling the distribution's source code; Section 3 presents an overview of the algorithms used in the PCASYS, MINDTCT and NFIQ systems; Section 4 discusses the export control packages; Section 5 discusses each package in greater detail; and manual pages are included in the Reference Manual in APPENDIX C. Source code references have been provided throughout this document to direct the reader to specific files and routines in the distribution. The source code should be considered the ultimate documentation. Technical questions are welcomed and any problems with this distribution should be directed to the authors via email. Any updates to this software will be posted at NIST Image Group's Open Source Sever (NIGOS).

2. INSTALLATION GUIDE

This section describes the organization and installation of the distribution. This distribution contains a combination of software, documentation, and test examples/data all stored in a hierarchical collection of directories stored on the NIST Image Group's Open Source Sever (NIGOS). To obtain the latest version of NIST Biometric Image Software (NBIS), please visit http://www.itl.nist.gov/lad/894.03/nigos/nigos.html for the instructions on accessing NIGOS via Perforce. The top-level directory is `NBIS`. The `NBIS` directory contains `Main` and `Test` directories. `Main` contains NBIS packages and the software build utilities to compile applications. Documentation is also provided under `Main` in the `doc` and `man` directories. The `Test` directory contains testing scripts, input data files, and example results. The distributed source code has been written in ANSI "C," and compilation scripts for the Bourne shell are provided. The source code and compilation scripts have been designed and tested to work with the free, publicly available Linux operating system and GNU `gcc` compiler and `gmake` utility.[32][34] The software is also compiled and tested to work with Mac OS-X operating system. The software may also be compiled to run on computers running the family of Win32 operating systems by first installing the free, publicly available Cygwin library and associated tools.[33] The porting of the software to other operating systems, compilers, or compilation environments is the responsibility of the recipient.

2.1. Software Installation

The software can be installed and compiled by first syncing the NBIS source code to a write-able disk partition on your computer using a Perforce source code management client. The directory to which you copy is referred to as the *installation_directory*. Please visit http://www.itl.nist.gov/lad/894.03/nigos/nigos.html for the instructions on using Peforce to access NIGOS.

A Bourne Shell (sh) script has been provided (*installation_directory* `/NBIS/Main/setup.sh`) to setup the NBIS build environment on your local computer. The script may be invoked from a shell. Within the directory containing the script, type the following:

```
% sh setup.sh <FINAL INSTALLATION DIR> [--without-X11]
```

where the text `<FINAL INSTALLATION DIR>` is replaced by your specific directory path where you desire to install the NBIS applications, libraries, percomputed runtime data and manuals. This is accomplished during the "make install" process. This path will be referred to as the *final installation directory*. The default setting for the setup script enables compilation of NBIS source code with X11 support. If you would like to disable X11 support, you can run the setup script with option "–-without-X11".

Once the NBIS build environment is successfully set up, the software can be compiled by executing the following commands in the *installation_directory*`/NBIS/Main`:

```
% make config
% make it
% make install
```

```
% make catalog
```

Successful compilation will produce 55 executable files stored in the `bin` directory under *final_installation_directory*, if compiled with X11 disable; otherwise, 52 executable files stored in the `bin` directory under *final_installation_directory*. To invoke these utilities you can specify a full path to these files, or you may add *final_installation_directory*`/bin` to your environment's execution path. Table 1 lists these utilities broken out by package. The sixth package listed, `IJG`, contains utilities provided with the Independent JPEG Group's compression/decompression source code library.[35] To learn more about the utilities in this package, refer to the Reference Manual in APPENDIX C.

Table 1. NBIS Non-Export Controlled utilities listed by package

NBIS Non-Export Controlled Package Utilities					
PCASYS	**NFIQ**	**MINDTCT**	**AN2K**	**IMGTOOLS**	**IJG**
asc2bin	nfiq	mindtct	an2k2iaf	cjpegb	cjpeg
bin2asc	fing2pat		an2k2txt	cjpegl	djpeg
chgdesc	znormdat		an2ktool	cwsq	jpegtran
cmbmcs	znormpat		dpyan2k*	diffbyts	rdjpgcom
datainfo			iaf2an2k	djpegb	wrjpgcom
eva_evt			txt2an2k	djpegl	
fixwts				djpeglsd	
kltran				dpyimage*	
lintran				dwsq	
meancov				dwsq14	
mkoas				intr2not	
mktran				not2intr	
mlp				rdwsqcom	
mlpfeats				rgb2ycc	
oas2pics				sd_rfmt	
optosf				wrwsqcom	
optrws				ycc2rgb	
optrwsgw					
pcasys					
pcasysx*					
rwpics					
stackms					

Note: The NBIS Package Utilities marked with an arsterisk (*) are only generated when compiled with the X11 option set.

A manual page is provided for each utility in the `man` directory under *final installation directory*. To view a man, type:

```
% man -M <install_dir>/man <executable>
```

where the text `<install_dir>` is replaced by your specific *final_installation_directory* path and `<executable>` is replaced by the name of the utility of interest. These manual pages are also included at the end of this document, in the Reference Manual in APPENDIX C.

2.2. Data and Testing Directories

Package directory `pcasys` contains various runtime files required by the non-graphical utility `pcasys` and the graphical X Window version `pcasysx`. The graphical version utilizes the image files provided in `/NBIS/Main/pcasys/runtimedata/images`, while both utilities require the parameters files and pre-computed feature vectors and weights in `/NBIS/Main/pcasys/runtimedata/parms` and `/NBIS/Main/pcasys/runtimedate/weights`. For more information see Section 5.1.4. Package directory `nfiq` contains files used for training `NFIQ`. See Section 5.3.1 for more information.

The directory `/NBIS/Test` is provided to give examples of how to invoke each of the utilities in the distribution. A simple PERL script is provided for each utility along with sample input data and resulting output. It should be noted that some utilities may, when run on other computer architectures, produce slightly different results due to small differences in floating point calculations and round off.

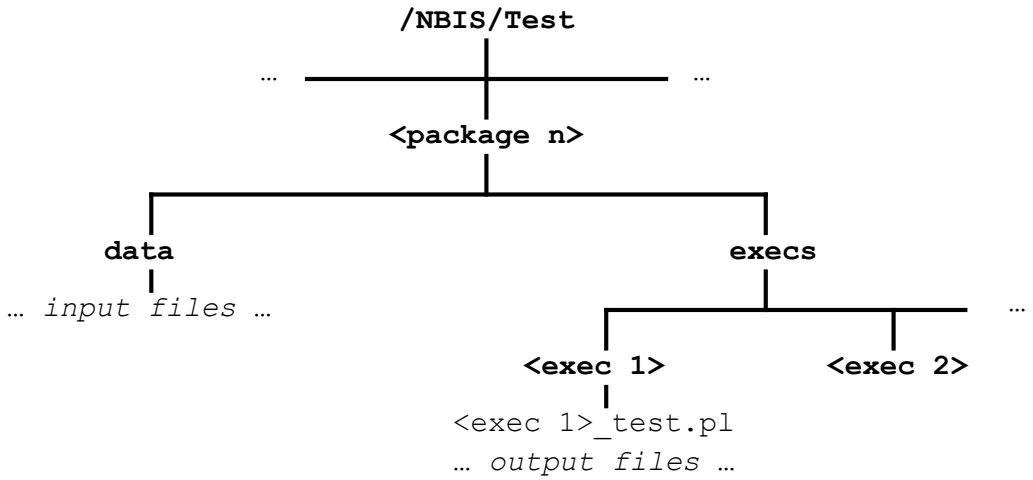


Figure 1. Hierarchical organization of testing directory.

The `Test` directory is organized hierarchically as illustrated in Figure 1. The utilities are organized by packages within the `Test` directory, one subdirectory for each package. Within each package, there are two directories. The `data` directory contains sample input files for the utilities in the package. The `execs` directory contains one subdirectory for each utility in the package. Within each utility subdirectory under `execs`, there is a simple PERL script (with extension "`pl`") and example output files produced by the script.

Included among the input data files used for testing are color and grayscale versions of a face image stored as an uncompressed pixmap, compressed using Baseline JPEG, and compressed using Lossless JPEG. Similarly, versions of a grayscale fingerprint image are provided uncompressed, compressed using Lossless JPEG, and compressed using WSQ. In addition, a set of 2700 WSQ compressed grayscale fingerprint images are included to support the testing of `PCASYS`.

3. OPEN SOURCE PACKAGES ON NIGOS

The packages described in this section have been deemed non-export controlled and are made freely available from NIGOS.

3.1. PCASYS – Fingerprint Pattern Classification

Automatic fingerprint classification is a subject of interest to developers of an Automated Fingerprint Identification System (AFIS). In an AFIS system, there is a database of *file* fingerprint cards, against which incoming *search* cards must be efficiently matched. Automatic matchers now exist that compare fingerprints based on 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 partitioning the file fingerprints based on classification. Once the class for each fingerprint of the search card has been determined, the set of possible matching file cards can be restricted to those whose 10-tuple (one for each finger) of classes matches that of the search card. This greatly reduces the number of comparisons that must be performed by the minutiae-matcher.

Some fingerprint identification systems 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 [36] provides a complete description of a manual classification system. Automating the classification process would 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 2 shows example fingerprints of the several classes. The system performs these steps: image segmentation and enhancement; feature extraction, registration, and dimensionality reduction; running of a main classifier, either a Probabilistic or Multi-Layer Perceptron Neural Network and an auxiliary whorl-detector that traces and analyzes pseudo-ridges; and finally, creation of a hypothesized class and confidence level.

`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 several processing stages in graphical form. This distribution 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. 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.

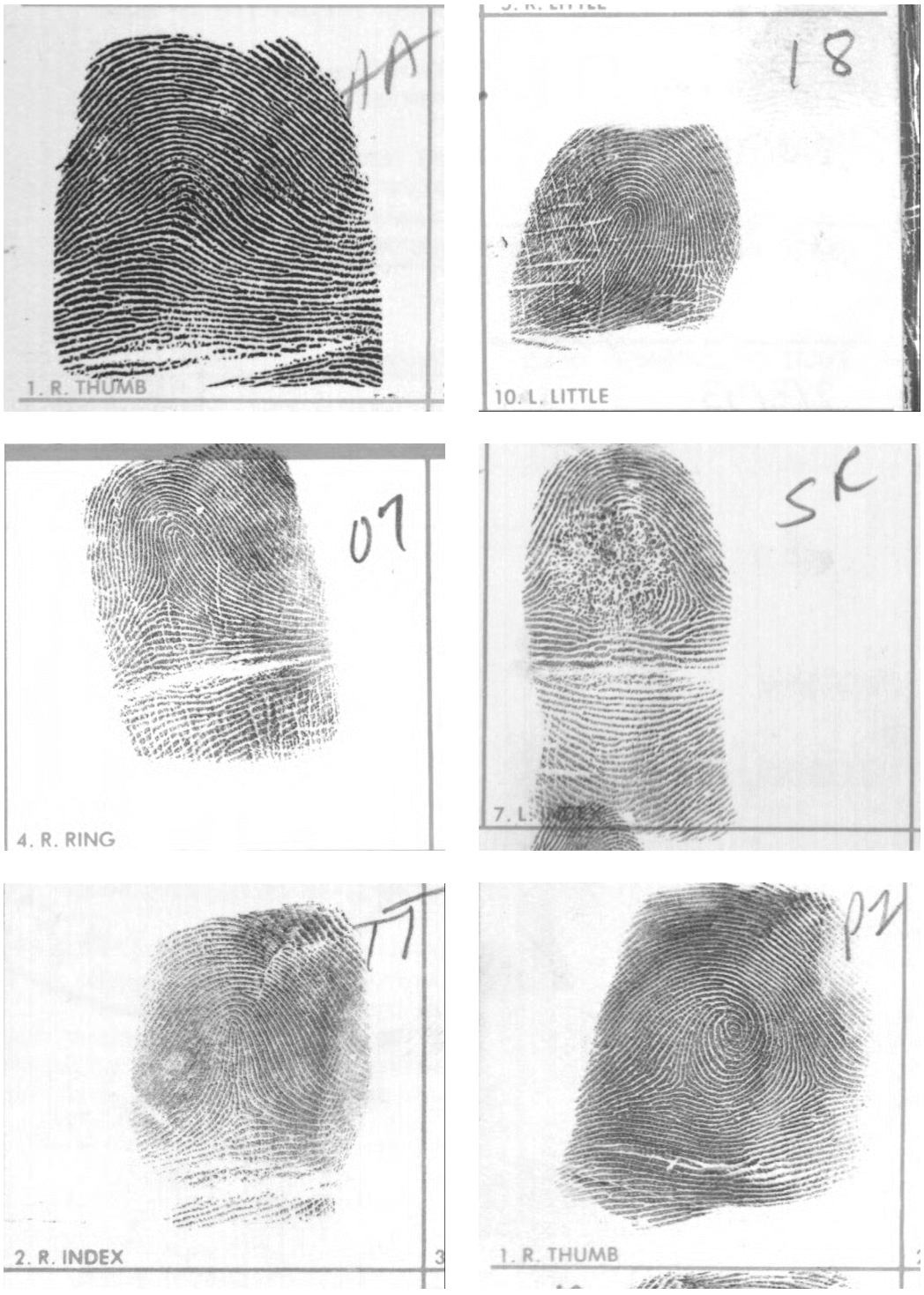


Figure 2. Example fingerprints of the six pattern-level classes. Going left-right, top-bottom, arch [A], left loop [L], right loop [R], scar [S], tented arch [T], and whorl [W]. These are NIST Special Database 14 images `s0024501.wsq`, `s0024310.wsq`, `s0024304.wsq`, `s0026117.wsq`, `s0024372.wsq`, and `s002351.wsq`, and they are among the fingerprint images in `/NBIS/Test/pcasys/data/images`.

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 (formerly NBS). Wegstein, of NBS, produced the R92 registration algorithm that is used by PCASYS and did important early automatic classification experiments.[8]

The algorithms used in PCASYS are described further in Section 5.1 and in References [17] and [22]-[24].

3.2. MINDTCT – Minutiae Detection

Another software system provided in this distribution is a minutiae detection package called, MINDTCT. This section first describes what fingerprint minutiae are, and then some background is provided as to why this package was developed for the FBI.

3.2.1. Definition of Minutiae

Traditionally, two fingerprints have been compared using discrete features called minutiae. These features include points in a finger's friction skin where ridges end (called a *ridge ending*) or split (called a ridge *bifurcation*). Typically, there are on the order of 100 minutiae on a tenprint. In order to search and match fingerprints, the coordinate location and the orientation of the ridge at each minutia point are recorded. Figure 3 shows an example of the two types of minutiae. The minutiae are marked in the right image, and the tails on the markers point in the direction of the minutia's orientation.

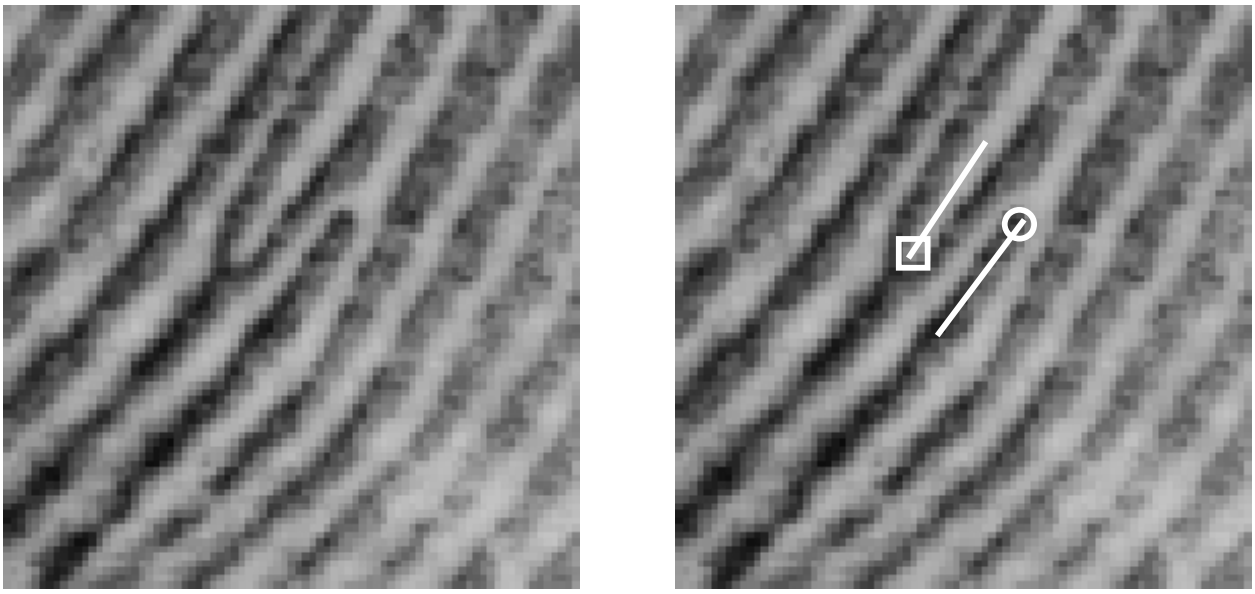


Figure 3. Minutiae: bifurcation (square marker) and ridge ending (circle marker).

The location of each minutia is represented by a coordinate location within the fingerprint's image. Different AFIS systems represent this location differently. The ANSI/NIST standard specifies units of distance in terms of 0.01 mm from an origin in the *bottom left* corner of the image. For example, a 500×600 pixel image scanned at 19.69 pixels per millimeter (ppmm) has dimensions 25.39×30.47 mm which in standard units of 0.01 mm is

$$2539 \times 3047 \quad = \quad \frac{500}{19.69 * 0.01} \times \frac{600}{19.69 * 0.01}$$

Thus, the pixel coordinate (200, 192) will be represented in standard units at

$$(1016, 2071) \quad = \quad \left(\frac{200}{19.69 * 0.01}, 3047 - 1 - \frac{192}{19.69 * 0.01} \right)$$

where the Y-coordinate is measured from the bottom of the image upward.

Minutiae orientation is represented in degrees, with zero degrees pointing horizontal and to the right, and increasing degrees proceeding counter-clockwise. The orientation of a ridge ending is determined by measuring the angle between the horizontal axis and the line starting at the minutia point and running through the middle of the ridge. The orientation of a bifurcation is determined by measuring the angle between the horizontal axis and the line starting at the minutia point and running through the middle of the intervening valley between the bifurcating ridges.

The minutiae plotted in Figure 4 illustrate the line to which the angle of orientation is measured. Each minutia symbol is comprised of a circle or square, marking the location of the minutia point, and the line or tail proceeding from the circle or square is projected along either the ridge ending's ridge, or the bifurcation's valley. The angle of orientation as specified by the ANSI/NIST standard is marked as angle "A" in the illustration.

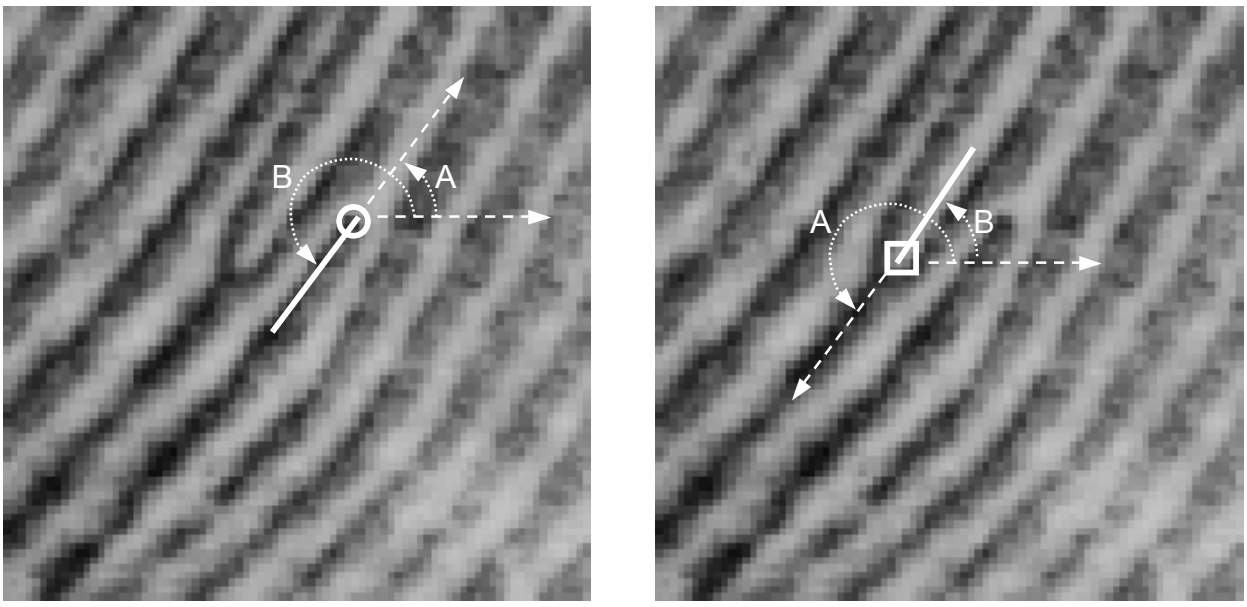


Figure 4. Minutiae orientation.
A. standard angle, B. FBI/IAFIS angle

3.2.2. Latent Fingerprints

In addition to tenprints, there is a smaller population of fingerprints also important to the FBI. These are fingerprints captured at crime scenes that can be used as evidence in solving criminal cases. Unlike tenprints, which have been captured in a relatively controlled environment for the expressed purpose of identification, crime scene fingerprints are by nature incidentally left behind. They are often invisible to the eye without some type of chemical processing or dusting. It is for this reason that they have been traditionally called *latent* fingerprints.

As one would expect, the composition and quality of latent fingerprints are significantly different from tenprints. Typically, only a portion of the finger is present in the latent, the surface on which the latent was imprinted is unpredictable, and the clarity of friction skin details are often blurred or occluded. All this leads to fingerprints of significantly lesser quality than typical tenprints. While there are 100 minutiae on a tenprint, there may be only a dozen on a latent. Figure 5 shows a "good" quality latent on the left and its matching tenprint on the right.

Figure 5. Latent fingerprint (left) with matching tenprint (right).

Due to the poor conditions of latent fingerprints, today's AFIS technology operates poorly when presented a latent fingerprint image. It is extremely difficult for the automated system to accurately classify latent fingerprints and reliably locate the minutiae in the image. Consequently, human fingerprint experts, called latent examiners, must analyze and manually mark up each latent fingerprint in preparation for matching. This is a tedious and labor intensive task.

To support the processing of latent fingerprints, the FBI and NIST collaboratively developed a specialized workstation called the Universal Latent Workstation (ULW). This workstation has been designed to aid the latent examiner in preparing a latent fingerprint for search. In addition, the workstation provides for interoperability between different AFIS systems by functioning as a vendor-independent front-end interface. These two aspects of the ULW contribute significantly to the advancement of the state-of-the-art in latent fingerprint identification and law enforcement

in general. As such, the FBI has chosen to distribute the ULW freely upon request. To receive more information regarding ULW, please contact:

Tom Hopper (thopper@leo.com)
FBI, JEH Bldg.
CJIS Div / Rm 11192E
935 PA Ave., NW
Washington, DC 20537-9700
202-324-3506

The successful application of the ULW is primarily facilitated by its use of the ANSI/NIST-ITL 1-2000 standard. NIST also developed some its underlying core technology, including the minutiae detection package in this software distribution.

`MINDTCT` takes a fingerprint image and locates all minutiae in the image, assigning to each minutia point its location, orientation, type, and quality. The command, `mindtct`, reads a fingerprint image from an ANSI/NIST, WSQ, baseline JPEG, lossless JPEG file, or IHead formatted file. A command line option [-b] can be selected that allows `mindtct` to perform image enhancement on low contrast images. Next it detects the minutiae in the image and outputs them to a file.

If the input file format was ANSI/NIST, it encodes the results into a Type-9 minutiae record [30], combines the record with the input data, and writes it all out to a new ANSI/NIST file. If the input file is not ANSI/NIST, the results are printed to a formatted text file. Details of this are included in the `mindtct` man page in APPENDIX C. The default is for `mindtct` to output the minutiae based on the ANSI/NIST standard as described in Section 3.2.1 but it has the option to write the minutiae according to the M1 (ANSI INCITS 378-2004) representation. M1 has the pixel origin in the top left of the image and directions pointing up the ridge ending or bifurcation valley consistent with the IAFIS minutiae representation illustrated in Figure 3.

An algorithmic description of `MINDTCT` is provided in Section 5.2.

3.3. NFIQ – Fingerprint Image Quality

Fingerprint image quality is a valuable tool in determining a fingerprint matchers ability to match a fingerprint against other fingerprints. It has been shown that as fingerprint image quality degrades so does matcher performance.[57][59] Knowing the fingerprint image quality can dramatically change the way fingerprints are currently processed. For example, multiple attempts could be made to obtain a higher quality fingerprint at the time of image capture. If the quality of fingerprints were know in advance, a system could handle the fingerprints of different quality in different ways. For example, a faster matcher could be used on good quality images without a drop in accuracy and a slower more accurate matcher could be used on lower quality images to improve matching accuracy. This would greatly improve the overall performance of the system.

`NFIQ` is an implementation of the NIST "Fingerprint Image Quality" algorithm as described in.[57] It computes a feature vector using the quality image "map" and minutiae quality statistics produced by the `MINDTCT` minutiae detection algorithm. The feature vector is then used as inputs to a Multi-Layer Perceptron (MLP) neural network classifier, and the output activation

level of the neural network is used to determine the fingerprint's image quality value. There are five quality levels with 1 being the highest quality and 5 being the lowest quality.

Please note that NFIQ has been designed to assign a level of quality to a grayscale fingerprint image. It was not designed to distinguish fingerprint images from all possible non-fingerprint images, and it will not necessarily assign non-fingerprint image types to the worst quality level five.

A detailed description of the NFIQ algorithm is included in Section 5.3.

3.4. AN2K – Standard Reference Implementation

The AN2K package is a reference implementation of the ANSI/NIST-ITL 1-2000 standard.[30] This package contains utilities for reading, writing, and manipulating the contents of ANSI/NIST data files. These files are comprised of a sequence of data fields and image records. Source code is provided to parse ANSI/NIST files into memory, manipulate designated fields, and write the sequence back to file. The utility an2ktool does this in *batch* mode. Logical data units are referenced on the command line, and the specified contents may be printed, inserted, substituted, or deleted from the file.

Alternatively, two other utilities are provided to support *interactive* editing of the contents of an ANSI/NIST file. The command an2k2txt converts the textual content of an ANSI/NIST file into a formatted text report, and any binary data (including images) are stored to temporary files and externally referenced in the text report. In this way, the text report can then be loaded into any common text editor and ASCII information can be added, deleted, or changed. When all edits are complete, the command txt2an2k is run on the edited version of the text report, externally referenced binary data files are incorporated, and a new ANSI/NIST file is written.

One of the many types of records in an ANSI/NIST file is the Type-9 record designed to hold minutiae data for fingerprint matching. Currently there is no global consensus on how fingerprint minutiae should be numerically represented. Different fingerprint systems use different sets of attributes and representation schemes. To manage this, the fields of the Type-9 record have been divided into blocks, where each block is assigned to a registered vendor, and the vendor defines how he will represent his minutiae data. In the standard, the first 4 fields of the Type-9 record are mandatory and must always be filled. Fields 5 through 12 are fields in the standard defined by NIST to hold among other things, the fingerprint's core, delta, and minutiae locations, along with neighbors and intervening ridge counts. The FBI's IAFIS is assigned fields 13 through 23. The definition of these fields is specified in the FBI's Electronic Fingerprint Transmission Specification (EFTS).[37]

Unfortunately, these two blocks of fields are different. Two utilities are provided in the AN2K package to facilitate the conversion between these blocks of fields in a Type-9 record. The command an2k2iaf translates the minutiae data stored in NIST fields 5-12 into the FBI/IAFIS fields 13-23. The command iaf2an2k reverses the process. An X Windows ANSI/NIST file image previewer is included in the package. The utility dpyan2k is designed to parse an ANSI/NIST file, locating and displaying each image in the file to a separate window. In addition, if any minutiae are included in a corresponding Type-9 record, then the minutiae points are plotted on top of the fingerprint image.

3.5. IMGTOOLS – General Purpose Image Utilities

NIST has distributed several fingerprint databases [14],[18]-[20] over the past decade for use in evaluating fingerprint matching systems. The images in these databases are formatted as NIST IHead [14], [18] files using either Lossless JPEG or WSQ compression. The IHead format uses a 296 byte header to store basic information about the image (i.e. pixel width, height, depth, compression type, compressed length, etc.). Displaying these images is problematic as common image viewing utilities do not support this format. Using utilities in the `IMGTOOLS` package, users are able to take NIST legacy database files and convert them into standard compliant formats, including Baseline JPEG which is widely supported.

Another issue is that these legacy files are not standard compliant. The utility `sd_rfmt` takes a legacy database file and reformats it. For example, legacy IHead WSQ files are converted so that they can be decoded with an FBI compliant WSQ decoder. The command `dwsq14` decompresses fingerprint images distributed with *NIST Special Database 14*, while the command `djpeglsd` decompresses images distributed with *NIST Special Database 4*, *9*, *10*, & *18*.[25]

`IMGTOOLS` also contains a collection of standard compliant and certifiable image encoders and decoders. The utilities `cjpegb` and `djpegb` encode and decode Baseline JPEG files respectively. The utilities `cjpegl` and `djpegl` encode and decode Lossless JPEG files. This represents one of the only available implementations of the standard Lossless JPEG algorithm. Finally, the utilities `cwsq` and `dwsq` encode and decode FBI WSQ files. An X Window application, `dpyimage`, is provided to view these different file compression formats, including IHead images and raw pixmaps.

Users should exercise caution when using these encoders and decoders in succession. The decoders generate uncompressed, reconstructed image pixmaps that can be subsequently re-encoded. Both Baseline JPEG and WSQ are *lossy* compression schemes, so taking their decoded pixmaps and re-encoding them may produce undesirable results. The amount of image degradation caused by lossy compression can be analyzed using the utility `diffbyts` to compare the pixels in an original image to those returned by one of the decoders.

All three compression algorithms in this distribution support internal comment blocks. Applications typically need easy access to various image attributes. These attributes include generic statistics such as pixel width, height, depth, and scan resolution, but often it is desirable to store and retrieve application-specific information such as fingerprint type, mugshot pose, or age/sex of the individual. To support applications, a structure called a `NISTCOM` has been defined, containing a text-based attribute list of (name, value) pairs. The encoders in `IMGTOOLS` accept an optional `NISTCOM` file, and if provided, embed its contents into a comment block within the encoded bytestream. The decoders on the other hand, search the encoded bytestream for the presence of a `NISTCOM`, and if found, merge its contents with those attributes the decoder derives itself and writes the resulting attribute list to a separate text file with extension "`ncm`." For more information on the `NISTCOM` convention, please refer to the Reference Manual in APPENDIX C. A `NISTCOM` stored in a JPEG or WSQ file does not interfere with other standard compliant decoders because it is contained in a standard comment block.

Several commands are provided to support `NISTCOM` and comment blocks in general. The utilities `rdjpgcom` and `wrjpgcom` read and write comments blocks to and from both Baseline

and Lossless JPEG files. Similarly, `rdwsqcom` and `wrwsqcom` read and write comment blocks to and from WSQ files.

Two other capabilities are included in `IMGTOOLS`. The first handles the interleaving and non-interleaving of color components in an image. The command `intr2not` takes an interleaved color image and separates the components into individual planes, whereas the command `not2intr` reverses the process. The second capability handles converting between RGB and YCbCr colorspaces. The command `rgb2ycc` converts from RGB to YCbCr, and `ycc2rgb` reverses the process.

4. EXPORT CONTROL PACKAGES

It is our understanding that this software falls within ECCN 3D980, which covers software associated with the development, production or use of certain equipment controlled in accordance with U.S concerns about crime control practices in specific countries.

4.1. NFSEG – Four-Finger Plain Segmentation

NFSEG source code is subject to U.S. export laws. This package is only available on CD-ROM upon request. The `NFSEG` package detail information and the algorithmic description is provided on the CD-ROM. For detail information on how to request this export controlled source code, please follow the procedure on APPENDIX B.

4.2. BOZORTH3 – Fingerprint Matcher

BOZORTH3 source code is subject to U.S. export laws. This package is only available on CD-ROM upon request. The `BOZORTH3` package detail information and the algorithmic description is provided on the CD-ROM. For detail information on how to request this export controlled source code, please follow the procedure on APPENDIX B.

5. ALGORITHMS

In this section, the open source algorithms used in the PCASYS, MINDTCT, and NFIQ packages are described. The source code in this distribution is the ultimate documentation for these systems; therefore, source code references are provided in the subheadings below for the various steps in these systems. Each reference is listed within square brackets, and they point to a specific file in the source code followed by the name of the subroutine primarily responsible for executing the process step. These references are provided as a road map into the source code.

5.1. PCASYS

NIST released its first version of PCASYS to the public in 1995. The two main changes in this new distribution are the addition of the multi-layer perceptron (MLP) classifier and replacing EISPACK routines with more stable CLAPACK routines [47] for computing eigenvalues and eigenvectors. Section 5.1.3.2 discusses the details of the MLP classifier. The CLAPACK routines have proven more stable and reliable across different operating systems.

A large portion of the code has also been modified to be more general with the parameters of the input data. For example, the original version required fingerprints to be at least 512×480 pixels, six output-classes in the classifier, and a 28×30 array of ridge directions. The new code allows for variations in these sizes. While most of the code is more general, the core detection algorithm still requires a 32×32 array of ridge directions and several parameters are tuned based on the old fixed dimensions that were used. For this reason many of the old "hard coded" dimensions are still used, but they are defined in include files that the user could easily change and then retune other parameters as needed.

Adjustments were also made to the image enhancement and ridge flow detection. Previously, the enhancement algorithm did local enhancement over the image every 24×24 pixels, starting at a preset location (near top-left of image). Similarly, the ridge flow algorithm used 16×16 pixel averages starting at a different preset location. The enhancement has been adjusted to work on 16×16 pixel windows that are aligned with the window used in ridge flow averaging. This helps minimize the effect of enhancement artifacts when computing ridge flow averages, which occur on the borders of the 16×16 window, causing erroneous ridge flow points in the final orientation array.

Interpolation was added to the final segmentation process but there was no improvement in the overall system performance, just an increase in preprocessing time. The old method of rounding to the nearest pixel was kept when doing segmentation. The interpolation routine is included in the source code (`sgmnt.c`) but not used.

Finally, given the changes that were made to the feature extraction process, the parameters for the PNN classifier, mainly the regional weights and overall smoothing factor, were optimized to the new feature set.

5.1.1. Algorithmic Description

This section describes the details of feature extraction and classification for each fingerprint. Sections 5.1.1.1 and 5.1.1.2 are preprocessing steps to prepare the fingerprint image for feature

extraction covered in Sections 5.1.1.3 - 5.1.1.5. Sections 5.1.1.6 - 5.1.1.9 discuss details of the neural network classifiers. Figure 6 is an example print of the whorl class and will be used for subsequent figures illustrating the stages of processing.

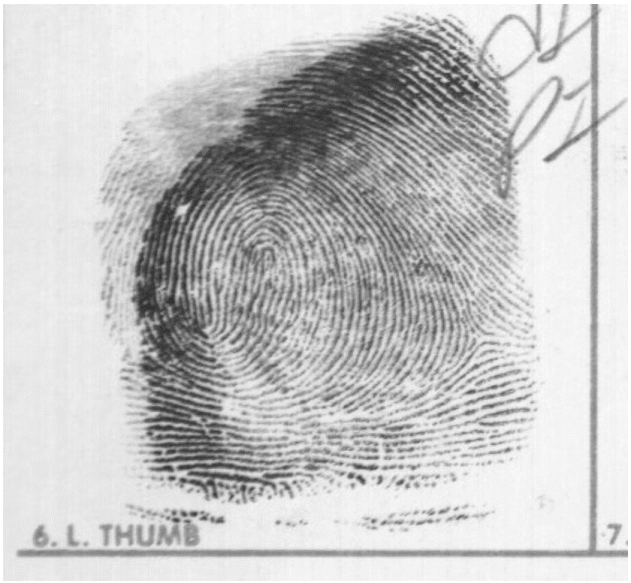


Figure 6. Fingerprint used to demonstrate the fingerprint classification process (`s0025236.wsq`).

5.1.1.1. Segmentor `[/NBIS/Main/pcasys/src/lib/pca/sgmnt.c; sgmnt()]`

The segmentor routine performs the first stage of processing needed by the classifier. It reads the input fingerprint image. The image must be an 8-bit grayscale raster of width at least 512 pixels and height at least 480 pixels (these dimensions can be changed in the file `include/pca.h`), and scanned at about 19.69 pixels per millimeter (500 pixels per inch). 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 first joint of the finger. It also attempts to define the rotation angle of the cut rectangle and 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). Removing rotation may help since it removes a source of variation between prints of the same class.[2]

(1) (2) (3) (4) (5) (6) (7)

Figure 7. Steps in the fingerprint segmentation process.

The segmentor decides which rectangular region of the image to snip out by performing a sequence of processing steps. This and all subsequent processing will be illustrated using the fingerprint from Figure 6 as an example. Figure 7 shows the results of the segmentor's processing steps.

First, the segmentor produces a small binary (two-valued or logical-valued) image. The binary image pixels indicate which 8×8-pixel blocks of the original image should be considered the *foreground*. Foreground 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 in the image. 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 first frame in Figure 7 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[3] then deletes every connected set of true pixels except the one whose number of true pixels is largest. The final cleanup step

[2] 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.

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

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 of the cleaned-up foreground-image, for later use. The second frame in Figure 7 shows the result of this cleanup processing.

Next, the routine 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 and uses the resulting pixels to trace the left edge. To avoid going around the corner onto the top edge, the routine stops when 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 frames in Figure 7 depict these edges.

Next, the routine uses the edges to calculate the overall slope of the foreground. First, it fits a straight line to each edge by linear regression. The left and right edges, which are expected to be roughly vertical, use lines of the form $x = my + b$ and the top edge use the form $y = mx + b$. The next to last frame in Figure 7 shows the 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 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 is tilted in accordance with the overall foreground slope. Starting at the top row of the rectangle and moving downward, the routine counts the **true** pixels of each row. It 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 will be tilted to cancel out the existing rotation of the fingerprint, and positioned to hang from the top of the foreground.

The last frame in Figure 7 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 8 shows the resulting segmented image. (The routine also cuts out the corresponding piece of the foreground-image, for use later by the pseudo-ridge analyzer.)

Figure 8. The sample fingerprint after segmentation.

5.1.1.2. Image Enhancement

```
[/NBIS/Main/pcasys/src/lib/pca/enhnc.c;
enhnc(),/NBIS/Main/pcasys/src/lib/fft/fft2dr.c;
fft2dr()]
```

This step enhances the segmented fingerprint image. The algorithm used is basically the same as the enhance-merit algorithm described in [38], and pp. 2-8 - 2-16 of [39] 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 16 pixels apart in each dimension to produce overlapping. Each input square undergoes a process that produces an enhanced version of its middle 16×16 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 an input square 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. Next, a nonlinear function is applied 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. Finally, the backward 2-d FFT is done to return the enhanced data to a spatial representation before snipping out the middle 16×16 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 $X_{jk} + Y_{jk}$ 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 $U + iV$:

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

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

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

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 16×16 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; therefore it is 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 takes advantage of the purely real nature of the input matrix by using 2-d *real* FFTs. 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, increases the accuracy of the resulting classifier. The table and graphs on pp. 24-6 of [24] 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 dominant. The dominant frequencies correspond to the ridges and valleys in most cases. So the enhancer strengthens the important aspects of the image 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 remaining frequencies go through a nonlinear function that adapts to variations as to which frequencies are most dominant. This aspect of the filter is helpful because the ridge wavelength can vary considerably between fingerprints and between areas within a single fingerprint.[4]

Figure 9 shows the enhanced version of the segmented image. At first glance, a noticeable difference seen between the original and enhanced versions is the increase in contrast. 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

[4] A different FFT-based enhancement method, the directional FFT filter of [24], 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 localized FFT filter, and did not produce better classification accuracy than the localized filter.

apparently have no major harmful effect on subsequent processing, due to alignment of the enhanced tiles with the orientation detection tiles.

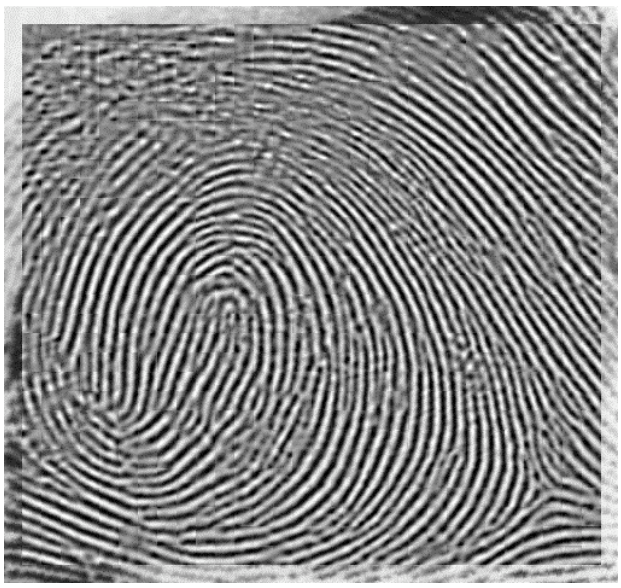


Figure 9. Sample fingerprint after enhancement.

5.1.1.3. Ridge-Valley Orientation Detector `[src/lib/pca/ridge.c; rors(), rgar()]`

This step 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.

7		8		1		2		3
6		7	8	1	2	3		4
		6				4		
5		5		**C**		5		5
		4				6		
4		3	2	1	8	7		6
3		2		1		8		7

Figure 10. Pattern of slits (i = 1-8) used by the orientation detector.

The routine is based on the ridge-valley fingerprint binarizer described in.[40] 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 10, the binarizer computes slit sums $s_i, i = 1...8$, where each s_i, is the sum of the values of the slit of four pixels labeled *i* (i.e., 1-8) 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

$$C > \frac{1}{32}\sum_{i=1}^{8} s_i \qquad (1)$$

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

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

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. The other seven slits will each cross ridges and valleys and have roughly equal lower sums. 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 [40] found that they obtained good binarization results by using the following compromise formula, rather than using either (1) or (2) alone: the output pixel is set to white if

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

This is simply a weighted average of (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. A pixel that would have been binarized to black (a ridge pixel) gets the orientation of its minimum-sum slit, and a pixel that would have been binarized to white (a valley pixel) gets the orientation of its maximum-sum slit. However, the resulting array of pixel-wise orientations is large, noisy, and coarsely quantized (only 8 different orientations are allowed). Therefore, the pixel-wise orientations are reduced to a much smaller array of *local average* orientations, each of which is made from a 16×16 square of pixel-wise orientations. The averaging process reduces the volume of data, decreases noise, and produces a finer quantization of orientations.

The ridge angle θ is defined to be 0° if the ridges are horizontal and increasing towards 180° as the ridges rotate counterclockwise ($0^\circ =< \theta < 180^\circ$). When pixel-wise orientations are averaged, the quantities averaged are not actually the pixel-wise ridge angles θ, but rather the pixel-wise *orientation vectors* (cos 2θ, sin 2θ). The orientation finder produces an array of these averages of pixel-wise orientation vectors.[5] Since all pixel-wise vectors have length 1 (being cosine, sine

[5] Averaging a set of local orientation *angles* can produce absurd results, because of the non-removable point of discontinuity that is inherent in an angular representation, so it is better to use the vector representation. Also, the

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, then the orientation vectors of its 256 pixels will tend to cancel each other out and produce a short average vector. This can occur because of blurring or because the orientation is highly variable within the square. The length of an average vector is thus a measure of orientation strength. The routine also produces as output the array of pixel-wise orientation indices, to be used by a later routine that produces a more finely spaced array of average orientations. Figure 11 depicts the local average orientations that were detected in the segmented and filtered image from the example fingerprint.

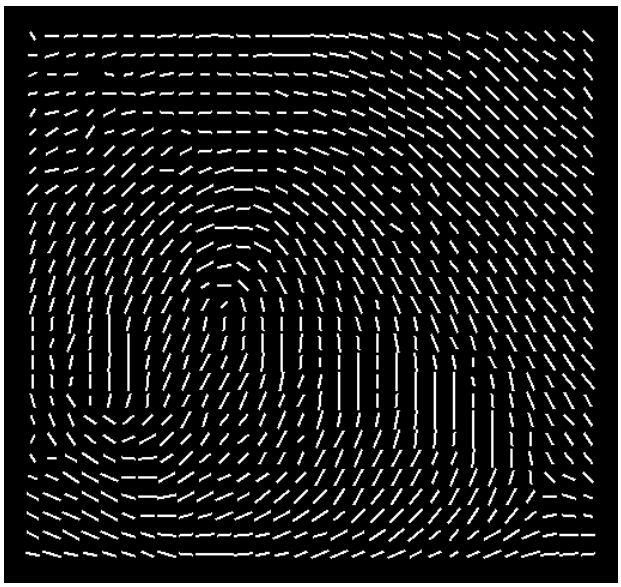


Figure 11. Array of local average orientations of the example fingerprint. Each bar, depicting an orientation, is approximately parallel to the local ridges and valleys.

5.1.1.4. Registration `[/NBIS/Main/pcasys/src/lib/pca/r92a.c; r92a()]`

Registration is a process that the classifier uses in order to reduce the amount of translation variation between similar orientation arrays. If the arrays from two fingerprints are similar except for translation, the feature vectors that later processing steps will produce from these orientation arrays may be very different because of the translation. This problem can be improved by registering each array (finding a consistent feature and essentially translating the array, bringing that feature to standard location).

To find the consistent feature that is required, we use the R92 algorithm of Wegstein.[8] The R92 algorithm finds, in an array of ridge angles, a feature that is used to register the fingerprint. The feature R92 detects in a loop and whorl fingerprint is located approximately at the *core* of the fingerprint. The algorithm also finds a well-defined feature in arch and tented arch fingerprints although these types of prints do not have true cores. After R92 finds this feature, which will be designated the *registration point,* registration is completed by taking the array of

---

resulting local average orientation vectors are an appropriate representation to use for later processing, because these later steps require that Euclidean distances between entire arrays of local average orientations produce reasonable results. Note: The R92 registration program requires converting the vectors into angles of a different format.

pixel-wise orientations produced earlier and averaging 16×16 squares from it 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). In addition, 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 component-wise median of the registration points of a sample of fingerprints. The result is a registered array of average orientations.[6]

The R92 algorithm begins by analyzing the matrix of angles in order to build the "K-table." R92 processes the orientations in angular form. It defines angle ranges from 0° to 90° as a ridge rotates from horizontal counterclockwise to vertical, and 0° to -90° for clockwise rotation. These angles differ from the earlier range 0° to 180° as the ridge rotates counterclockwise from horizontal. 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, *RK*3. *RK*3 is the minimum value of the difference of two angles. For this work, the parameter was set at 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*, greater than *RK*3, the score is increased by one point. If *A* has a positive slope, meaning the angle of *A* is greater than *RK*3, or if *M* - *A* is greater than *RK*3, the score is increased by one point. If *D* has a negative slope, meaning the angle of *D* is less than *RK*3, or if *D* - *N* is greater than *RK*3, then the score is increased by one point. 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:

$$x = \alpha\left(\frac{A(R,C)}{A(R,C) - A(R,C+1)} + C - 1 \right) + \beta$$

[6] The new array is made by re-averaging the pixel-wise orientations with translated squares, rather than just translating the first average-orientation array. This is done 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.

$$y = \frac{(\alpha R + \beta)ts(k+1) + (\alpha(R-1) + \beta)ts(k) + (\alpha(R-2) + \beta)ts(k-1)}{ts(k+1) + ts(k) + ts(k-1)}$$

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 isn't greater than 99°, 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.

For an image that is possibly something other than an arch, the computation of the registration point is slightly more complex:

$$dsp1 = A(R, C) - A(R, C+1)$$

$$dsp2 = A(R+1, JL) - A(R+1, JL+1)$$

$$dh1 = \begin{cases} \alpha & dsp1 \geq 90 \\ dsp1\,\alpha/90 & otherwise \end{cases}$$

$$dh2 = \begin{cases} (dsp2 - 90)\alpha/90 & dsp2 > 90 \\ 0 & otherwise \end{cases}$$

$$dh = (dh1 + dh2)/2$$

$$xx1 = \alpha \times \left(\frac{A(R, C)}{A(R, C) - A(R, C+1)} + C - 1 \right) + \beta$$

$$xx2 = \alpha \left(\frac{A(R+1, JL)}{dsp2} + JL - 1 \right) + \beta$$

$$x = \frac{dh(xx1 - xx2)}{ds} + xx2$$

$$y = \alpha R - dh$$

Where A, R, C, α, and β are as before and JL is the cross-reference point column.

The left picture in Figure 12 shows the orientation array of the example fingerprint with its registration point, and the standard registration point is marked; the right picture shows the resulting registered version of the orientation array. Obviously some lower and left areas of the registered array are devoid of data, which would have had to be shifted in from undefined regions. Likewise, data that were in upper and right areas have fallen off the picture and are lost; but the improved classification accuracy that we have obtained as a result of registration shows that this is no great cause for concern. The optimal pattern of regional weights, discussed later, also shows that outer regions of the orientation array are not very important. The test results in [24] show that registration improves subsequent classification accuracy.

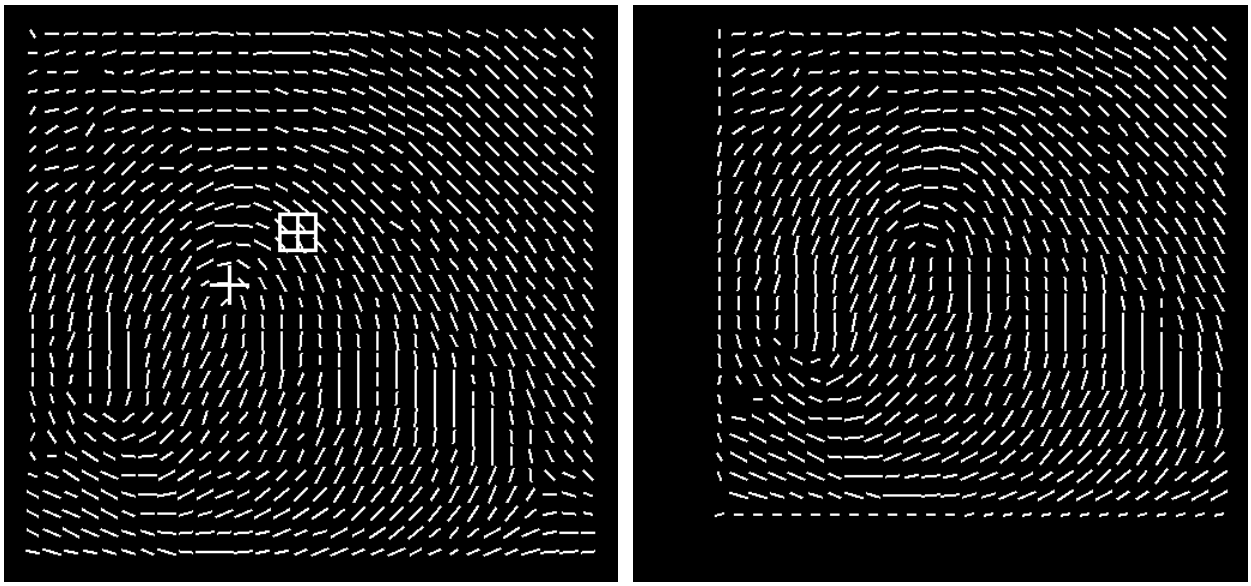


Figure 12. Left: Orientation array Right: Registered orientation array. The plus sign is registration point (core) found by R92, and plus sign in a square is standard (median) registration point.

5.1.1.5. Feature Set Transformation `[src/lib/pca/trnsfrm.c; trnsfrm()]`

This step applies a linear transform to the registered orientation array. Transformation accomplishes two useful processes. First, the reduction of the dimensionality of the feature vector from its original 1680 dimensions to 64 dimensions (PNN) and 128 dimensions (MLP). Second, the application of a fixed pattern of regional weights (PNN only) which are larger in the important central area of the orientation array.

5.1.1.5.1. Karhunen-Loève Transform

The size of the registered orientation array (oa) representing each fingerprint is 1680 elements (28×30 orientation vectors × two components per orientation vector). The size of these arrays makes it computationally impractical to use them as the feature inputs into either of the neural network classifiers (PNN/MLP).

It would be helpful to transform these high-dimensional feature vectors into much lower-dimensional ones in such a way that would not be detrimental to the classifiers. Fortunately, the Karhunen-Loève (K-L) transform [41] 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 routine is used to produce a set of eigenvectors of the covariance matrix, corresponding to the largest eigenvalues; let m denote the number of eigenvectors produced. Then, for any $n \leq m$, the matrix $\boldsymbol{w}$ 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 of a K-L transform[7], which reduces an original feature vector $\boldsymbol{u}$ (an orientation array, thought of as a single 1680-dimensional vector) to a vector $\boldsymbol{w}$ of n elements, can then be defined as follows:

$$\boldsymbol{w} = \boldsymbol{\Phi}^t \boldsymbol{u}$$

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 classifier. This dimension reduction produces 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 for PNN and 128 for MLP.

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. However, later experiments revealed that significantly better classification accuracy could be obtained by modifying the production of the feature vector. The modification allows the important central region of the fingerprint to have more weight than the outer regions; what we call *regional weights*. This is discuss in the next section.

5.1.1.5.2. Regional Weights

`[/NBIS/Main/pcasys/src/bin/optrws/optrws.c]`

During testing, it was noted that the uniform spacing of the orientation measurements throughout the picture area could probably be improved by using a non-uniform spacing. The non-uniform spacing concentrated the measurements more closely together in the important central area of the picture and had a sparser distribution in the outer regions. We tried this [23], 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, we realized that the improved results might not have been caused by the non-uniform *spacing* but rather by the mere assignment of 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 non-uniform pattern of *regional weights* 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 [42], Specht improves the accuracy of PNN in about the same manner: pp. 1-765-6 described 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. This produced 210 (14×15) weights, versus assigning a separate weight to each of the 840 orientation-vectors. Optimization of the weights was done using a very simple

[7] Usually the sample mean vector is subtracted from the original feature vector before applying $\boldsymbol{\Psi^t}$, but we omit this step because doing so simplifies the computations and has no effect on the final results of either classifier. If the user needs it, a full version, that subtracts the mean vector from each feature vector, is included in `src/bin/kltran`.

form of gradient descent, as discussed in Section 5.1.3.1.1. The resulting optimal (or nearly optimal) weights are depicted in Figure 13. 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 occurred, to white. These weights can be represented as a diagonal matrix $\boldsymbol{W}$ of order 1680. Their application to an original feature vector (orientation array) $\boldsymbol{u}$, to produce a weighted version $\tilde{\boldsymbol{u}}$, is given by the matrix multiplication

$$\tilde{\boldsymbol{u}} = \boldsymbol{W}\boldsymbol{u}$$

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 described 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 $\boldsymbol{\psi}^t\boldsymbol{W}\boldsymbol{\psi}$ mentioned in Section 5.1.3.1.1, 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.

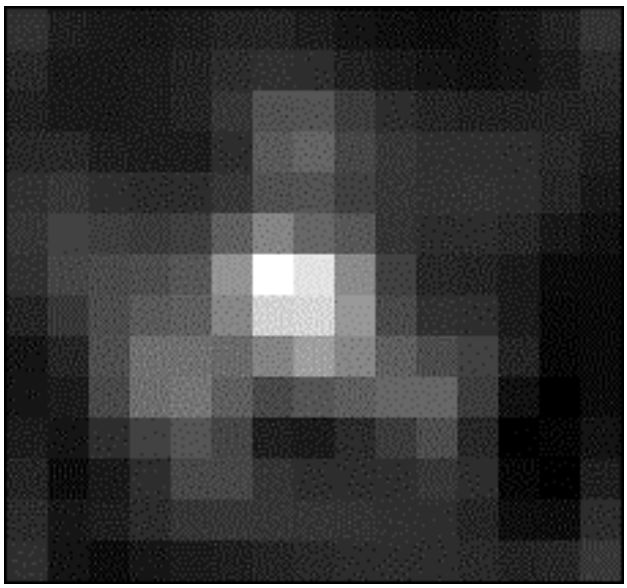


Figure 13. Absolute values of the optimized regional weights. Each square represents one weight, associated with a 2×2 block from the registered orientation array.

5.1.1.5.3. Combined Transform

`[/NBIS/Main/pcasys/src/bin/mktran/mktran.c]`

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

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

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

5.1.1.6. Probabilistic Neural Network Classifier [src/lib/pca/pnn.c; pnn()]

This step takes as its input the low-dimensional feature vector that is the output of the transform discussed in Section 5.1.3.1, and it determines the class of the fingerprint. The Probabilistic Neural Network (PNN) is described by Specht in [43]. 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 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 of the activations 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 ($1 < i < N$)
$\boldsymbol{x}_j^{(i)}$ = feature vector from j^{th} prototype print of class i ($1 \leq j \leq M_i$)
$\boldsymbol{w}$ = 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:

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

[9] This naive version of PNN must compute the distance of the incoming feature vector from each of the many prototype feature vectors, possibly many cycles. Various methods have been found for increasing the speed of nearest-neighbors classifiers, a category PNN may be considered to fall into (see, for example, [44], and [45] for a very fast tree method). 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.

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

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

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

Figure 14 is a bar graph of the normalized activations produced for the example fingerprint. Although 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.

Figure 14. PNN output activations for the example fingerprint.

Figure 15. MLP output activations for the example fingerprint.

5.1.1.7. Multi-Layer Perceptron Neural Network Classifier

`[/NBIS/Main/pcasys/src/lib/pca/mlp_sing.c; mlp_single()]`

This alternative classifier takes as input the low-dimensional feature vector, non-optimized, as discussed in Sections 5.1.1.5 and 5.1.2.4 and a set of MLP weights. The weights are the result of several training runs of `MLP` in which the weights are optimized to produce the best results with the given training data. Section 5.1.3 discusses the training process in more detail. The output of `mlp_single()` is a set of confidence levels for each of the possible output classes and an indication of which hypothetical class had the highest confidence. Figure 15 shows the MLP output activations for the example fingerprint. The whorl (W) class has the highest activation and is the correct answer.

5.1.1.8. Auxiliary Classifier: Pseudo-ridge Tracer

`[/NBIS/Main/pcasys/src/lib/pca/pseudo.c; pseudo()]`

This step takes a grid of ridge orientations of the incoming fingerprint and traces *pseudo-ridges* [46], which are trajectories that approximately follow the flow of the ridges. By testing the pseudo-ridges for concave-upward shapes, the routine detects some whorl fingerprints that are misclassified by the classifiers. We were motivated to produce a whorl-detector when we realized, upon examining the prints misclassified by the NN classifiers, that many of them were whorls.

The routine takes as input an array of local averaged 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

[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.

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 pseudo-ridges (an implementation detail).

Having finished with this preliminary processing, the process then traces pseudo-ridges. Starting at a block of initial locations in the orientation array, it makes a pseudo-ridge 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 goes in a southwest direction. If a location has been marked as bad, then no trajectories are started there. A trajectory is stopped if it reaches a limit of the array of locations, reaches a bad location, if the turn required in the trajectory is excessively sharp, or if a specified maximum number of steps have been taken. The two trajectories traced out from an initial point are joined end to end, producing a finished pseudo-ridge. The pseudo-ridge only approximately follows the ridges and valleys, and is insensitive to details such as bifurcations or small scars.

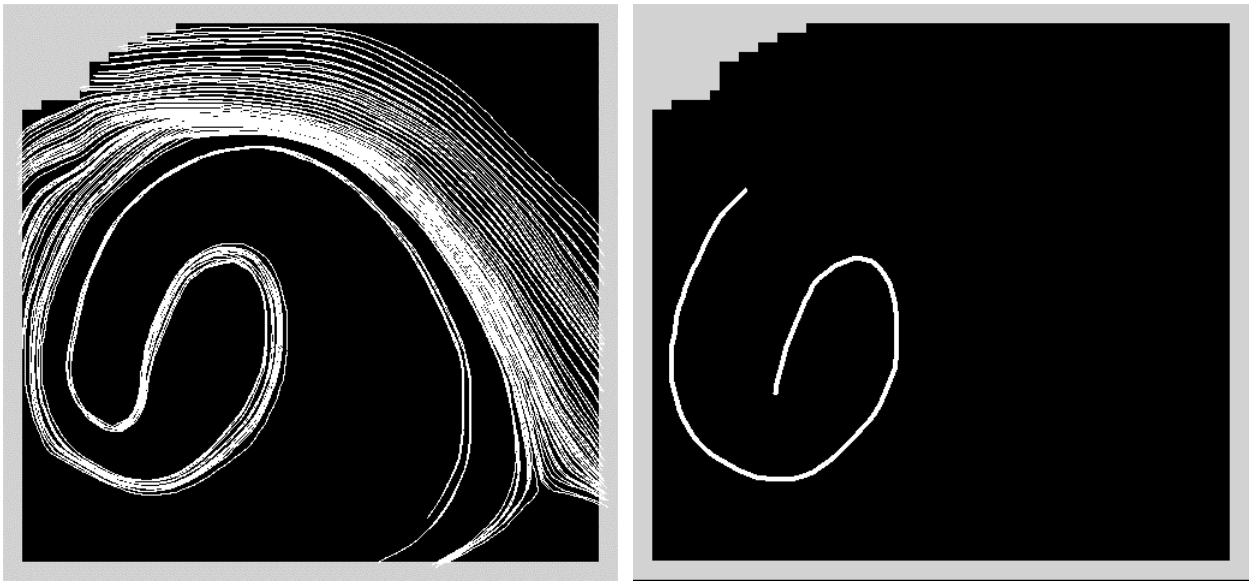


Figure 16. Left: Pseudo-ridges traced to find concave-upward lobe. Right: Concave-upward lobe that was found.

After the routine has finished tracing a pseudo-ridge, 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. A lobe qualifies as a concave upward shape, if it's orientation, at the sharpest point of curvature (vertex), is close to horizontal and concave upward, and it has a minimum amount of cumulative curvature on each side of it's vertex. The routine checks each lobe of the current pseudo-ridge 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 pseudo-ridge. The routine stops when it either finds a concave upward shape or exhausts all lobes of all pseudo-ridges without finding one. The final output shows if it did or did not find a concave upward shape. Figure 16 shows a concave upward shaped lobe that was found.

This pseudo-ridge tracer is useful as a detector of whorls. It rarely produces a false positive, defined as finding a concave upward lobe in a print that is not a whorl. It is more likely to produce a false negative, defined as not finding a concave upward lobe although a print is a whorl. The next section describes a simple rule that is used to combine the pseudo-ridge tracer's output with the output of the main classifier (PNN or MLP), thereby producing a hybrid classifier that is more accurate than the main classifier alone.

The ridge tracer has many parameters that may be experimented with if desired as well as the parameter for combining the classifier and ridge tracing results (Section 5.1.4.3.2), but reasonable values are provided in the default parameter file.

5.1.1.9. Combining the Classifier and Pseudo-ridge Outputs

`[/NBIS/Main/pcasys/src/lib/pcs/combine.c; combine()]`

This final processing module takes the outputs of the main Neural Network (NN) classifier and the auxiliary pseudo-ridge tracer, and makes the decision as to what class, and confidence, to assign to the fingerprint.

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

```
if(pseudo-ridge tracer says whorl) {
      hypothesized-class = whorl
      if(pnn_hypothesized_class == whorl)
            confidence = 1.
      else
            confidence = .9
}
else { /* pseudo-ridge tracer says not clear whether whorl */
      hypothesized_class = pnn_hypothesized_class
      confidence = pnn_confidence
}
```

This is a reasonable way to use the pseudo-ridge 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 detects a whorl, the print is classified as a whorl even if the NN disagrees, although disagreement results in slightly lower confidence, since whorl detection implies that the print is almost certainly a whorl. If the whorl detector does not detect a whorl, then the NN sets the classification and confidence.

Since the whorl detector detected a whorl for the example print and the NN classified this print as a whorl, the final output of the classifier had 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 pseudo-ridge tracer improves the result for some prints the NN 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 NN 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 pseudo-ridge tracer with the main NN classifier is a lowering of the error rate, compared to the rate obtained using the NN alone.

5.1.2. Computing Features

The following subsections describe the process used to get the features that will be independently optimized for each classifier. The name of the command is listed for each step. For details on the arguments and parameter files used, see the Reference Manual in APPENDIX C. Section 5.1.3 discusses how the features are optimized for each classifier.

5.1.2.1. Make the Orientation Arrays

`mkoas`

This command reads the fingerprint image files and extracts the orientation array (oa). This is run on the full set of fingerprints that will be used as the "training set" for the neural network classifier.

5.1.2.2. Make the Covariance Matrix

`meancov`

This command reads a set of oas and computes their sample mean and sample covariance matrix.[11] Is it typically run on the full set of orientation arrays from `mkoas` but could be run on just a reasonably large subset of the training set.

5.1.2.3. Make the Eigenvalues and Eigenvectors

`eva_evt`

This program reads the covariance matrix and computes the 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 CLAPACK routines.[47]

5.1.2.4. Run the Karhunen-Loève Transform

`lintran`

[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. The resulting output files are combined using the `cmbmcs` command, but to combine several covariance matrices, `cmbmcs` needs the means as wells as the covariance matrices of the subsets.

This command applies a specified linear transform to a set of vectors. The transform matrix is the eigenvectors from `eva_evt`. The set of vectors to which the transform matrix is being applied is the oas files, from `mkoas`, for the training fingerprints. This set of the resulting low-dimensional Karhunen-Loève (K-L) vectors will be used as the training set for the MLP classifier when optimizing the classifier weights. A subset of the K-L vectors will be used as data by `optrws` (optimize regional weights command, below) to help optimize the PNN classifier. Remember this version of the K-L transform does not subtract the mean vector from each feature vector. A complete version of the K-L transform is included in the command `kltran`.

5.1.3. Training the Neural Networks

This section explains how to optimize the features for the PNN and MLP classifiers. Optimization for PNN is done using the `optrws` (regional weights optimization) and `optosf` (optimize the overall smoothing factor) commands, described in Section 5.1.3.1. MLP uses the features from Section 5.1.2 as its input but does a series of "training" runs to optimize its set of neural network weights. Section 5.1.3.2 discusses the training process for MLP that results in the set of optimized weights used by the classifier.

5.1.3.1. Optimizing the Probabilistic Neural Network

Several steps are needed to optimize the feature set for PNN. First a set of regional weights are computed that place emphasis on the most significant regions of the fingerprint (typically the core area). These results are combined with the eigenvectors to produce a transform matrix to use when reducing the dimensionality of the original oa features. Finally, the overall smoothing factor (osf) for PNN is optimized.

5.1.3.1.1. Optimize the Regional Weights

`optrws`

This command optimizes the regional weights. First, it finds an optimal single value 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. It estimates (by secant method) the gradient of the activation error rate, using the PNN classifier and its prototype features. Classification on the prototype features is done by excluding the print being classified from the prototypes (i.e. leave-one-out). 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 are 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 the work. Consult the manual page in APPENDIX C and the default parameter file mentioned in the manual page to find about this. If your operating system does implement fork() and execl(), which are required by the several-processes version of `optrws`, then `optrws` can 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 `/NBIS/Main/pcasys/src/bin/optrws/makefile.mak`. 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 $\Psi^t W \Psi$ of order NFEATS from the eigenvectors Ψ and the diagonal matrix W that is equivalent to the regional weights. It 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. The results seem satisfactory, and the total runtime is much smaller than for direct methods.

5.1.3.1.2. Make the Transform Matrix

`mktran`

Reads the optimized regional weights made by `optrws`, and the eigenvectors, and makes the transform matrix $\Psi^t W$ used in the next step.

5.1.3.1.3. Apply the Transform Matrix

`lintran`

`Lintran` should 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 applies both 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 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.

5.1.3.1.4. Optimize the Overall Smoothing Factor

`optosf`

Optimizes an overall smoothing factor (osf) used by the PNN classifier. 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, better accuracy is expected if the classifier uses an osf that is slightly larger than 1, which is the value used during regional weights optimization. This corresponds to Specht's observation [43] 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 and the osf 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 fingerprints, and the overall smoothing factor. The PNN classification system then consists of a combination of the PNN classifier and the pseudo-ridge tracer.

5.1.3.2. Training the Multi-layer Perceptron Neural Network

`mlp`

The program `mlp` trains a 3-layer feed-forward linear perceptron [48] using novel methods of machine learning that help control the learning dynamics of the network. As a result, the derived minima are superior, the decision surfaces of the trained network are well formed, the information content of confidence values is increased, and generalization is enhanced. As a classifier, MLP is superior to the PNN classifier in terms of its memory requirements and classification speed. The theory behind the machine learning techniques used in this program is discussed in References [49], [50], & [51]. The main routine for this program is found in `/NBIS/Main/pcasys/src/bin/mlp/mlp.c` and the majority of supporting subroutines is located in the library `/NBIS/Main/pcasys/src/lib/mlp`.

Machine learning is controlled through a batch-oriented iterative process of training the MLP on a set of prototype feature vectors, then evaluating the progress made by running the MLP (in its current state) on a separate set of feature vectors. Training on the first set of patterns then resumes for a predetermined number of passes through the training data and then the MLP is tested again on the evaluation set. This process of training and then testing continues until the MLP has been determined to have satisfactorily converged. For details on the command line and specfile parameters see the included manual page in the Reference Manual in APPENDIX C.

This command trains or tests an MLP neural network suitable for use as a classifier. The network has an input layer, a hidden layer, and an output layer, each layer comprising a set of nodes. The input nodes are feed-forwardly connected to the hidden nodes, and the hidden nodes to the output nodes, by connections whose weights (strengths) are trainable. The activation function used for the hidden nodes can be chosen to be sinusoid, sigmoid (logistic), or linear, as can the activation function for the output nodes. Training (optimization) of the weights is done using either a Scaled Conjugate Gradient (SCG) algorithm [52], or by starting out with SCG and then switching to a Limited Memory Broyden Fletcher Goldfarb Shanno (LBFGS) algorithm.[53] Boltzmann pruning [54], i.e. dynamic removal of connections; can be performed during training if desired. Prior weights can be attached to the patterns (feature vectors) in various ways.

When `mlp` is invoked, it performs a sequence of runs. Each run does either training, or testing:

> **training run:** A set of patterns is used to train (optimize) the weights of the network. Each pattern consists of a feature vector, along with either a class or a target vector. A feature vector is a tuple of floating-point numbers, which typically has been extracted from some natural object such as a fingerprint image. A class denotes the actual class to which the object belongs, for example "whorl." The network can be trained to become a classifier: it trains using a set of feature vectors extracted from objects of known classes. Training runs finish by writing the final values of the network weights as a file. It also produces a summary file showing information about the run, and optionally produces a longer file that shows the results the final (trained) network produced for each individual pattern.
>
> **testing run:** A set of patterns is sent through a network, after the network weights are read from a file. The output values, i.e. the hypothetical classes are compared with target

classes or vectors, and the resulting error rate is computed. The program can produce a table showing the correct classification rate as a function of the rejection rate.

Output files generated from `mlp` training are provided in `/NBIS/Test/pcasys/execs/mlp/mlp_dir`. The specfile used by `mlp` to train the classifier on fingerprint images is `spec`. This specfile requires the input files `patnames`, `patwts`, `priors`, the training set `fv1-9mlp.kls`, and the testing set `sv10mlp.kls`, and it invokes 5 sequential pairs of `mlp` training/testing sessions. Three files are generated from each training/testing session. For example, from the first session: `trn1.err`, `trn1l.err`, `trn1.wts`, and `tst1.err` are created. `Trn1.err` is a report of the progressive error rates achieved on the training set. `Trn1l.err` is a file containing the output activations for each fingerprint in the training set. `Trn1.wts` is the resulting weights trained in the session. `Tst1.err` is a report of the error rate achieved on the testing set using the most recent set of weights from training.

For the next training/testing session, training resumes with the MLP network initialized to the weights contained in `trn1.wts`. The output files from this session are `trn2.err`, `trn2.err`, `trn2.wts`, and `tst2.err`. The weights file `trn2.wts` is then used as input to the next session and so on until the final session is complete. The files `trn5.err`, `trn5l.err`, and `trn5.wts` contain the final results of training and `tst5.err` contains the error rate achieved by using the final set of weights to classify the testing set contained in `sv10mlp.kls`. APPENDIX A gives more details about the output files of the `mlp` training process including formulas and sample data from `PCASYS` training.

There are numerous parameters (see the manual page for details on all the parameters) to be specified in the specfile for running the program `mlp`. A good strategy for training the MLP on a new classification problem is to first work with a single training/testing session. Try different combinations of parameter settings until a reasonable amount of training is achieved within the first 50 iterations, for example. This typically involves using a relatively high value for regularization (such as 2.0 with fingerprint classification); varying the number of hidden nodes in the network; and trying different levels of temperature, typically incrementing or decrementing by powers of 10. For fingerprint classification, the number of hidden nodes is typically set to equal or greater than the number of input KL features, and a temperature of 1.0e-5 works well.

Once reasonable training is achieved, these parameters should remain fixed, and successive sessions of training/testing are performed according to a schedule of decreasing regularization. For fingerprint classification it works well to specify about 50 iterations for each training session, and to use a regularization factor schedule starting at 2.0 and decreasing to 1.0, 0.5, 0.2, 0.1 for each successive training session. This process of multiple training/testing sessions initiates MLP training within a reasonable solution space. It also enables the machine learning to refine its solution so that convergence is achieved while maintaining a high level of generalization by controlling the dynamics of constructing well "behaved" decision surfaces. The intermediate testing sessions allow one to evaluate the progress made on an independent testing set, so that a judgment can be made as to whether incremental gains in training have reached diminishing returns. The theory behind the control of dynamical changes within the MLP learning process is discussed in References [49], [50], & [51].

Training the MLP in this fashion generates superior decision surfaces thus providing robust activations for use as confidence values when rejecting confusing classification. This training process is of course done once off-line, and then the resulting weight files are reused by the actual recognition system. In practice, the user could use the `mlp` command to do a batch run over a set of test data versus running the `PCASYS` commands and processing each test image individually. The `PCASYS` commands are merely for demonstrating the procedure used to get the final classification results and when possible allow the user to see graphics of the progress at each step along the way.

5.1.4. Running PCASYS

5.1.4.1. PCASYS Data Files

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

matrix: A matrix of real numbers.

covariance: A covariance matrix of real numbers. This format saves disk space by storing only the non-strict 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 respectively. A classes file can be used for any classification situation with no more than 255 classes.

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 contains a description string (can be of any length, but must contain no new lines; or can be left empty), code bytes indicating the file type and storage mode, and additional information specific to the file type. Additional information includes: 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; and if classes, the number of elements. The `datainfo` command can be run on any PCASYS data file. `Datainfo` writes a report of the header information to the standard output.

5.1.4.2. Commands

Installation of PCASYS provides the following commands, shown here with short descriptions. For a complete description and usage instructions for any of these commands, consult the manual pages in APPENDIX C.

5.1.4.2.1. Classifier Demos

`pcasys`	non-graphical demo
`pcasysx`	graphical demo

5.1.4.2.2. Training (Optimization) Commands

`eva_evt`	finds the eigenvalues and eigenvectors
`lintran`	runs a linear transform on a set of vectors
`meancov`	makes mean and covariance from a set of vectors
`kltran`	runs a Karhunen-Loève transform on 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

5.1.4.2.3. 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
`oas2pics`	makes IHead pictures of orientation arrays
`rwpics`	makes IHead pictures of regional weights or estimated gradients
`stackms`	stacks several *matrix* files together

5.1.4.3. Running the Classifier

5.1.4.3.1. Graphical and Non-graphical Versions

The classifier has a graphical version (`pcasysx`) and a non-graphical version (`pcasys`). The graphical version, which requires the X Window System, produces windows on the screen containing graphics showing 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.

5.1.4.3.2. Default Parameters and Specifying Parameters

The default files needed by the classifier are located in the distribution's top-level `/NBIS/Main/pcasys/runtimedata` directory. The subdirectory `runtimedata/images` contains a set of images used to create the screens when running the graphics version. The subdirectory `runtimedata/parms` has all the default parameter files used by the classifier. The `runtimedata/weights` directory is split into two subdirectories `pnn` and `mlp`, which contain the optimized prototypes for each of the classifiers. The 2700 sample images used by the classifier are located in `/NBIS/Test/pcasys/data/images`.

5.1.4.3.3. Output File

The output file has a line for each of the fingerprints that were classified. Each line shows: the fingerprint filename; the actual class (A, L, R, S, T, and W stand for the pattern-level classes arch, left loop, right loop, sear, tented arch, and whorl); the output of the classifier (a hypothesized class and a confidence); the output of the auxiliary pseudo-ridge 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 output showing the first and last 10 sample images using the PNN classifier is:

```
s0024301.wsq: is W; nn: hyp W, conf 0.59; conup y; hyp W, conf 1.00; right
s0024302.wsq: is R; nn: hyp R, conf 0.88; conup n; hyp R, conf 0.88; right
s0024303.wsq: is R; nn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024304.wsq: is R; nn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024305.wsq: is R; nn: hyp R, conf 0.99; conup n; hyp R, conf 0.99; right
s0024306.wsq: is L; nn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024307.wsq: is L; nn: hyp L, conf 0.94; conup n; hyp L, conf 0.94; right
s0024308.wsq: is L; nn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024309.wsq: is L; nn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
s0024310.wsq: is L; nn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
…
s0026991.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026992.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026993.wsq: is T; nn: hyp A, conf 0.79; conup n; hyp A, conf 0.79; wrong
s0026994.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026995.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026996.wsq: is W; nn: hyp W, conf 0.84; conup y; hyp W, conf 1.00; right
s0026997.wsq: is W; nn: hyp W, conf 0.75; conup y; hyp W, conf 1.00; right
s0026998.wsq: is L; nn: hyp L, conf 0.84; conup n; hyp L, conf 0.84; right
s0026999.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0027000.wsq: is W; nn: hyp W, conf 0.96; conup y; hyp W, conf 1.00; right

pct error: 7.07

A           L           R           S           T           W
A    41( 83.7)    3(  6.1)    0(  0.0)    0(  0.0)    4(  8.2)    1(  2.0)
L     3(  0.4)  784( 97.5)    3(  0.4)    0(  0.0)    5(  0.6)    9(  1.1)
R     7(  1.0)    6(  0.8)  699( 95.1)    0(  0.0)    5(  0.7)   18(  2.4)
S     0(  0.0)    4( 80.0)    0(  0.0)    0(  0.0)    1( 20.0)    0(  0.0)
T    19( 22.6)   26( 31.0)   14( 16.7)    0(  0.0)   25( 29.8)    0(  0.0)
W     1(  0.1)   35(  3.4)   27(  2.6)    0(  0.0)    0(  0.0)  960( 93.8)
```

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 2 and Table 3, described in the next section.

5.1.5. Classification Results

The fingerprint images used to train and test the `PCASYS` classifier were taken from NIST Special Database 14 (SD14).[20] 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 card is the card stored in the FBI file for this person and is denoted the *file* card. The other card was sent in to be searched against the database and is denoted the *search* card. Each card was scanned at 19.69 pixels per millimeter (500 pixels per inch), then parsed into individual fingerprint images, by cutting out rectangles of predefined locations and dimensions, corresponding to the printed boxes in which the rolled finger impressions were made.

We trained (optimized) the main classifiers using file prints `f0000001.wsq` through `f0024300.wsq` of SD14. Then, the finished classification system was made by adding to the

classifier the pseudo-ridge tracer, with its parameters set to values that had been arrived at much earlier as a result of testing. With all aspects of the classification system settled, we then tested its accuracy on search prints `s0024301.wsq` through `s0027000.wsq` of SD14. The test set that was used is provided in the directory `/NBIS/Test/pcasys/data/images`, in the form of the original fingerprint images. The classifier may be run on this entire set if desired, 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 NN training feature vectors are derived are *not* provided in the distribution, but the prototype feature vectors themselves are provided in `test/pcasys/data/oas`.

The result of the test was an error rate (fraction of the test prints misclassified) of 7.07 % for PNN and 8.19 % for MLP. More insight into the behavior of the classifiers can be obtained by examining the confusion matrix of Table 2 and Table 3. This matrix has a row for each actual class and a column for each hypothesized class, and it shows, as the non-parenthesized numbers, how many *test* prints fell into each (actual class, hypothetical class) cell. For example, it shows that 784 of the L (left loop) prints were classified as L and that 4 of them were classified as R (right loop). Each parenthesized number is the percentage that its corresponding count comprises of the sum of the counts in that row. For example, the parenthesized numbers show that 97.4 % of the L prints were classified as L, and that 0.5 % of them were classified as R. The entries shown in boldface correspond to correct classifications.

The 7.07 % (or 8.19 %) error rate and 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, indicating it is uncertain about the hypothesized class, it can achieve an error rate much lower than 7.07 % (or 8.19 %) 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 curves in Figure 17 are *error vs. reject* curves that summarize this behavior, produced from the results of the test runs. Curves are included for a classifier consisting of PNN or MLP alone or with the help of the pseudo-ridge analyzer; clearly the hybrid classifier is more accurate than the PNN or MLP alone, at all rejection levels.

Table 2. PNN Confusion matrix
Non-parenthesized: Actual count that occurred for that cell.
Parenthesized: Percentage of total row sums.

Actual		Hypothesized Class				
Class	A	L	R	S	T	W
A	**41 (83.7)**	3 (6.1)	0 (0.0)	0 (0.0)	4 (8.2)	1 (2.0)
L	3 (0.4)	**784 (97.5)**	3 (0.4)	0 (0.0)	5 (0.6)	9 (1.1)
R	7 (1.0)	6 (0.8)	**699 (95.1)**	0 (0.0)	5 (0.7)	18 (2.4)
S	0 (0.0)	4 (80.0)	0 (0.0)	**0 (0.0)**	1 (20.0)	0 (0.0)
T	19 (22.6)	26 (31.0)	14 (16.7)	0 (0.0)	**25 (29.8)**	0 (0.0)
W	1 (0.1)	5 (3.4)	27 (2.6)	0 (0.0)	0 (0.0)	**960 (93.8)**

Table 3. MLP Confusion matrix
Same layout as Table 1.

Actual		Hypothesized Class				
Class	A	L	R	S	T	W
A	**10 (20.4)**	20 (40.8)	18 (36.7)	0 (0.0)	0 (0.0)	1 (2.0)
L	0 (0.0)	**783 (97.4)**	4 (0.5)	0 (0.0)	0 (0.0)	17 (2.1)
R	0 (0.0)	9 (1.2)	**704 (95.8)**	0 (0.0)	0 (0.0)	22 (3.0)
S	0 (0.0)	3 (60.0)	2 (40.0)	**0 (0.0)**	0 (0.0)	0 (0.0)
T	0 (0.0)	43 (51.2)	40 (47.6)	0 (0.0)	**0 (0.0)**	1 (1.2)
W	0 (0.0)	29 (2.8)	12 (1.2)	0 (0.0)	0 (0.0)	**982 (96.0)**

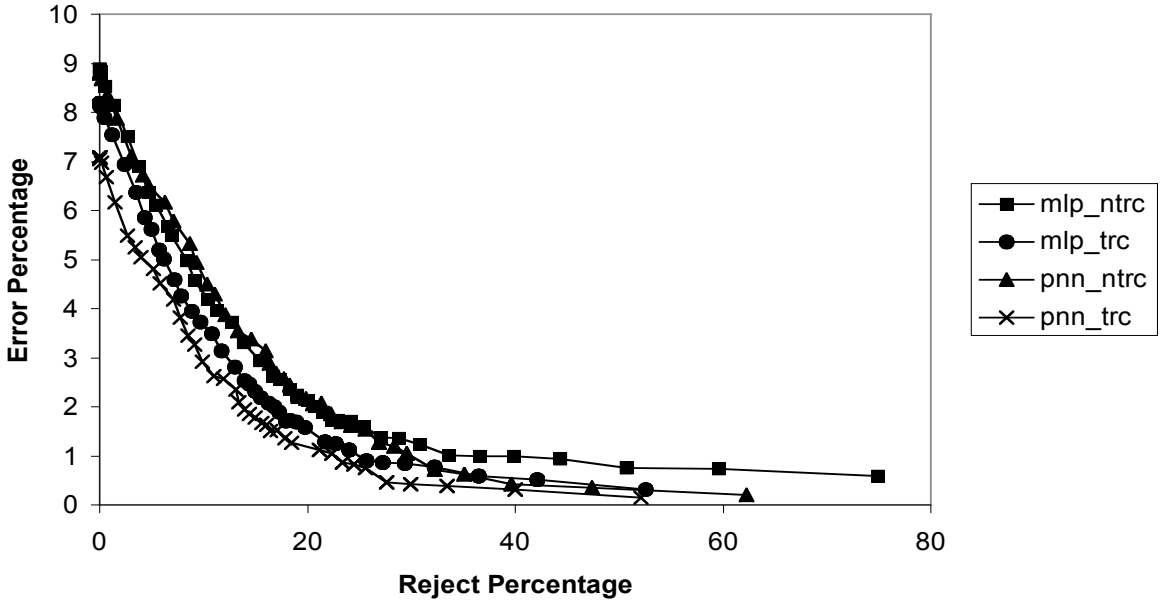


Figure 17. Error versus reject curves for PNN and MLP classifiers and hybrid combinations.

5.2. MINDTCT

The algorithms used in `MINDTCT` were inspired by the Home Office's Automatic Fingerprint Recognition System; specifically the suite of algorithms commonly referred to as "HO39."**[55]** The NIST software is an entirely original implementation exceeding the capabilities of HO39. It incorporates new algorithms, a modular design, dynamic allocation, and flexible parameter control, which provide a framework for supporting future enhancement and adaptation of the technology. It should be noted that the algorithms and software parameters have been designed and set to optimally process images scanned at 19.69 pixels per millimeter (ppmm) (500 pixels per inch) and quantized to 256 levels of gray.

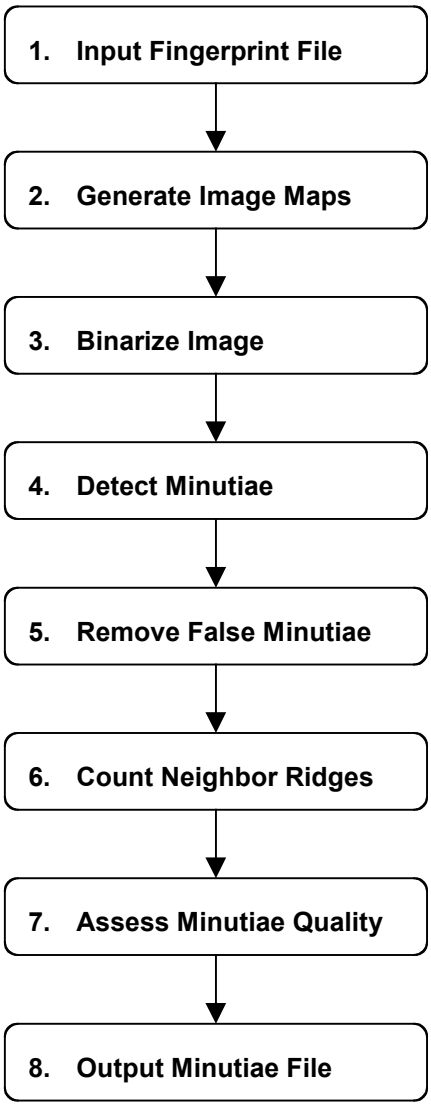


Figure 18. Minutiae detection process.

Once the software is successfully installed and compiled, the program, `mindtct`, is available for detecting minutiae in a fingerprint image. This section describes each of the major steps in the

minutiae detection process. It should be noted that two generations of minutiae detection have been developed prior to the release of this software. Thus, `mindtct` calls second generation (or Version 2) routines. Version 1 routines are included in the libraries for comparison, but in general, they will perform less satisfactorily. Figure 18 lists the functional steps executed.

The software has been designed in a modular fashion so that each of the steps listed in Figure 18 is primarily executed by a single subroutine. This permits other alternative approaches to be implemented and substituted into the process, and the overall impact on performance can be evaluated. To support the many required operating parameters, a single global control structure is used to record sizes, tolerances, and thresholds. This structure, `lfsparm_V2`, is automatically constructed and initialized in the file `/NBIS/Main/mindtct/src/lib/lfs/globals.c` and the values of its members are defined in `/NBIS/Main/mindtct/src/include/lfs.h`. Many of the principal control parameters are discussed in this section.

5.2.1. Input Fingerprint Image File

```
[/NBIS/Main/an2k/src/lib/an2k/fmtstd.c;
read_ANSI_NIST_file(),/NBIS/Main/imgtools/src/lib/image/imgdecod.c;
read_and_decode_grayscale_image()]
```

`Mindtct` inputs a fingerprint image and automatically detects minutiae on the fingerprint. The algorithms and parameters have been developed and set for images scanned at 19.69 ppmm and quantized to 256 levels of gray. The application can read files in ANSI/NIST, WSQ, JPEGB, JPEGL, and IHEAD formats.

In ANSI/NIST formatted files it searches the file structure for a grayscale fingerprint record. Once found, the fingerprint image in this record is processed. The application is capable of processing ANSI/NIST Type-4, Type-13, and Type-14 fingerprint image records.**[30]** Currently, only the *first* grayscale fingerprint record in the ANSI/NIST file is processed, but the application could be changed to process *all* grayscale fingerprints in the ANSI/NIST file.

`Mindtct` has an option that will allow it to enhance very low contrast images. If the option is selected, `mindtct` will evaluate the histogram of the input image. If the image is a very low contrast image, it is enhanced to improve the contrast otherwise it is not modified.

5.2.1.1. Generate Image Quality Maps

```
[/NBIS/Main/mindtct/src/lib/lfs/maps.c; gen_image_maps()]
```

Because the image quality of a fingerprint may vary, especially in the case of latent fingerprints, it is critical to be able to analyze the image and determine areas that are degraded and likely to cause problems. Several characteristics can be measured that are designed to convey information regarding the quality of localized regions in the image. These include determining the directional flow of ridges in the image and detecting regions of low contrast, low ridge flow, and high curvature. These last three conditions represent unstable areas in the image where minutiae detection is unreliable, and together they can be used to represent levels of quality in the image. Each of these characteristics is discussed below.

5.2.1.2. Direction Map [/NBIS/Main/mindtct/src/lib/lfs/dft.c; dft_dir_powers()]

One of the fundamental steps in this minutiae detection process is deriving a directional ridge flow map, or *direction map*. The purpose of this map is to represent areas of the image with sufficient ridge structure. Well-formed and clearly visible ridges are essential to reliably detecting points of ridge ending and bifurcation. In addition, the direction map records the general orientation of the ridges as they flow across the image.

To locally analyze the fingerprint, the image is divided into a grid of *blocks*. All the pixels within a block are assigned the same results. Therefore, in the case of the direction map, all the pixels in a block will be assigned the same ridge flow direction. Several considerations must be made when using a block-based approach. First, it must be determined how much local information is required to reliably derive the desired characteristic. This area is referred to as the *window*. The characteristic measured within the window is then assigned to each pixel in the block. It is typically desirable to share data used to compute the results assigned to neighboring blocks. This way some of the image that contributed to one block's results is included in the neighboring block's results as well. This helps minimize the discontinuity in block values as you cross the boundary from one block to its neighbor. This "smoothing" can be implemented using a system where a block is smaller than its surrounding window, and windows overlap from one block to the next. This is illustrated in Figure 19.

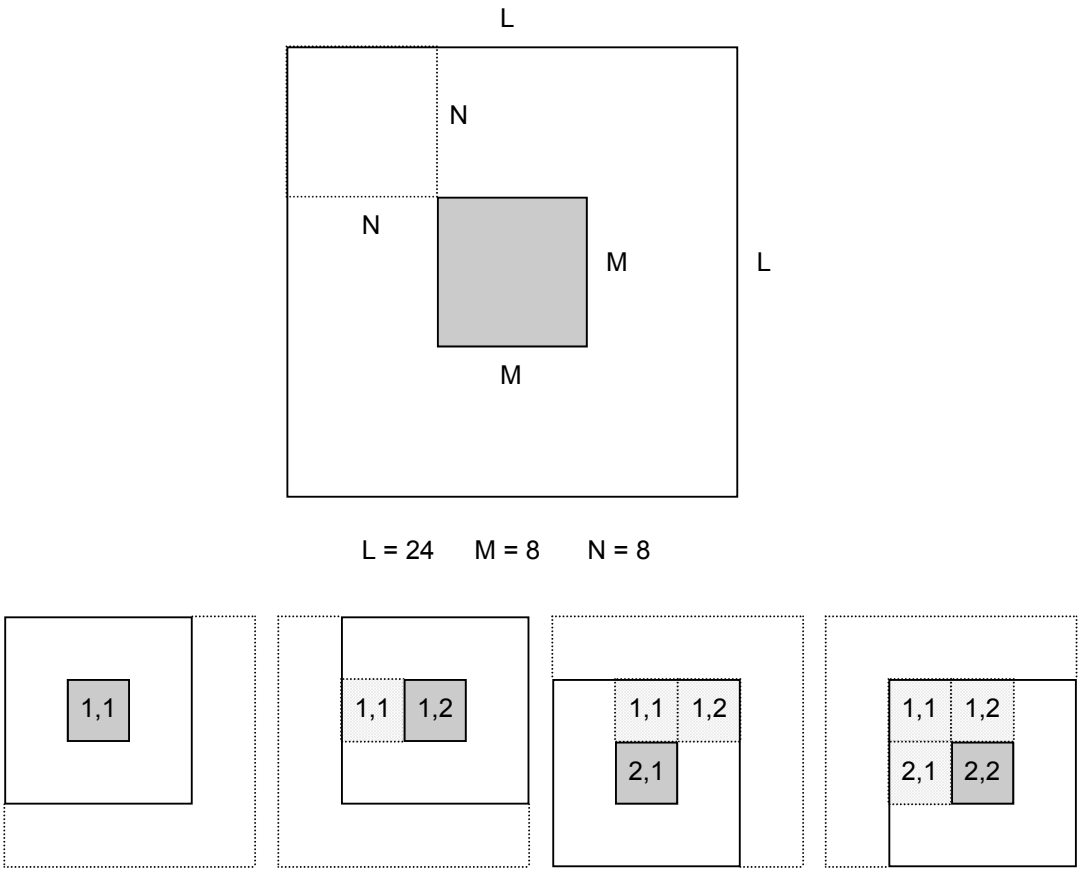


Figure 19. Adjacent blocks with overlapping windows.

The large frame at the top of the figure depicts a window (in white) surrounding a smaller block (in gray). Assuming that neighboring blocks are adjacent and non-overlapping, this scenario is defined by three parameters: the *window size* "L," the *block size* "M" and the *offset* of the block from the window's origin "N." In the global control structure, `lfsparms_V2`, these parameters are defined as `MAP_WINDOWSIZE_V2`=24, `MAP_BLOCKSIZE_V2`=8, and `MAP_WINDOWOFFSET_V2`=8 respectively. As a result, the image is divided up into a grid of 8×8 pixel blocks with each block being assigned a result from a larger surrounding 24×24 pixel window, and the area for windows of neighboring blocks overlap by up to 2/3.

The bottom row of frames in the Figure 19 illustrates how this works in practice. Designating the address of a block by its (row, column) indices, the left frame shows the first block (1,1) being computed. The next frame advances to the next adjacent block to the right, block (1,2). Correspondingly, its window is shifted 8 pixels, and the new block receives its results. Note that there are two copies of the image being used. Each window operates on the original image data, while block results are written to a separate output image. The third frame in the illustration depicts the window configuration for block (2,1), and the fourth frame shows its right neighbor being computed.

One additional consideration must be made when using blocks. It must be determined how to handle the edges of the image. The dimensions of the image will likely not be an even multiple of blocks, and the windows surrounding blocks along the perimeter of the image may extend off the image. In this software, the image is padded by a margin of medium gray pixels (set to intensity 128). This margin is sufficiently large to contain the perimeter windows in the image. The processing of partial blocks is also accounted for at the right and bottom of the image. This blocking scheme is implemented in `/NBIS/Main/mindtct/src/lib/lfs/block.c; block_offsets()`.

Given the above approach for computing block results with an overlapping window, the technique used for determining ridge flow direction in the image can be described. For each block in the image, the surrounding window is rotated incrementally and a Discrete Fourier Transform (DFT) analysis is conducted at each orientation. Figure 20 illustrates the incremental rotation of the window. The top left box in the figure depicts a window with its rows rotated 90° counterclockwise so that they are aligned vertically. This is considered orientation "0" in the software. The parameter `NUM_DIRECTIONS` in the global control structure, `lfspars_V2`, specifies the number of orientations to be analyzed in a semicircle. This parameter is set to 16, creating an increment in angle of 11.25° between each orientation. These orientations are depicted on the circle in the figure. The bottom row in the figure illustrates the incremental rotation of the window's rows at each defined orientation.

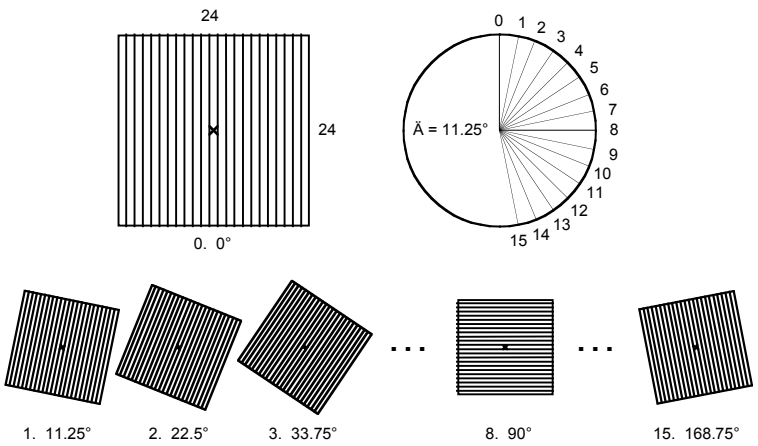


Figure 20. Window rotation at incremental orientations.

When determining the direction of ridge flow for a block, each of its window orientations is analyzed. Within an orientation, the pixels along each rotated row of the window are summed together, forming a vector of 24 pixel row sums. The 16 orientations produce 16 vectors of row sums. Each vector of row sums is convolved with 4 waveforms of increasing frequency. These are illustrated in Figure 21. The top waveform in the figure has a single period extending across the length of the entire vector. The second waveform's frequency is doubled from the first; the third is doubled from the second, and so forth. Discrete values for the sine and cosine functions at the 4 different frequencies are computed for each unit along the vector. The row sums in a vector are then multiplied to their corresponding discrete sine values, and the results are accumulated and squared. The same computation is done between the row sums in the vector and their corresponding discrete cosine values. The squared sine component is then added to the squared cosine component, producing a resonance coefficient that represents how well the vector *fits* the specific waveform frequency.

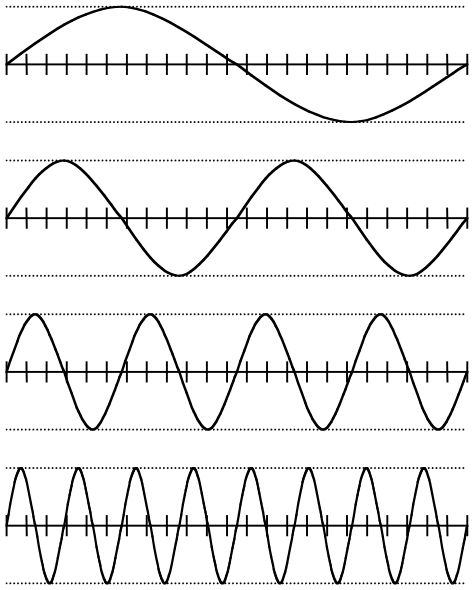


Figure 21. DFT waveform frequencies.

The spatial frequency of the top waveform in Figure 21 discretely represents ridges and valleys with a width of approximately 12 pixels. The second waveform represents 6 pixel wide ridges and valleys. The third waveform represents 3 pixel wide ridges and valleys. Finally, the fourth waveform represents 1.5 pixel wide ridges and valleys. Given an image scanned at 19.69 ppmm, these waveforms cover ridges and valleys ranging in width from 0.6 mm down to 0.075 mm.

The resonance coefficients produced from convolving each of the 16 orientation's row sum vectors with the 4 different discrete waveforms are stored and then analyzed. Generally, the dominant ridge flow direction for the block is determined by the orientation with maximum waveform resonance. The details are in the source code.

In Figure 22, an original fingerprint image is shown on the left. The image on the right, is the same fingerprint image annotated with the ridge flow directions recorded in the resulting direction map. Each direction in the map is represented as a rotated line segment centered within its corresponding 8×8 pixel image block.

Figure 22. Direction map results.

5.2.1.3. Low Contrast Map [/NBIS/Main/mindtct/src/lib/lfs/block.c; low_contrast_block()]

It is difficult, if not impossible, to accurately determine a dominant ridge flow in certain portions of a fingerprint image. This is true of low contrast areas that contain image background and smudges. It is desirable to detect these areas and prevent artificially assigning ridge flow directions where there really are no clearly defined ridges. To derive an arbitrary ridge flow strictly from the data within these areas is problematic.

An image map called the *low contrast map* is computed where blocks of sufficiently low contrast are flagged. This map separates the background of the image from the fingerprint, and it maps out smudges and lightly-inked areas of the fingerprint. Minutiae are not detected within low contrast blocks in the image.

One way to distinguish a low contrast block from a block containing well-defined ridges, is to compare their pixel intensity distributions. By definition, there is little dynamic range in pixel intensity in a low contrast area, so the distribution of pixel intensities will be very narrow. A block containing well-defined ridges will, on the other hand, have a considerably broader range of pixel intensities as there will be pixels ranging from very light in the middle of valleys to very dark in the middle of ridges.

In order to determine if a block is low contrast, this software computes the pixel intensity distribution within the block's surrounding window. A specified percent of the distribution's high and low tails are trimmed, and the width of the remaining distribution is measured. If the measured width is sufficiently small, then the block is flagged in the map as having low contrast.

In the global control structure, `lfsparms_V2`, the parameter `PERCENTILE_MIN_MAX`=10 causing the lowest and highest 10% of pixel intensities in the distribution to be trimmed. By trimming the tails, the subsequent width measurement is made in a much more stable portion of the distribution. The parameter `MIN_CONTRAST_DELTA`=5 is the pixel intensity threshold less than which indicates a low contrast block. This threshold was derived empirically from a training sample of low and high contrast blocks extracted from real fingerprint images. The image maps are actually computed in this software on a 6-bit pixel intensity image with 64 levels of gray. The threshold here of 5 actually corresponds to a threshold of 10 shades of gray in the original 8-bit pixel intensity image with 256 levels of gray. In other words, if the dynamic range of the center 80 % of a block's pixel intensity distribution is not larger than 10 shades of gray, it is determined to be low contrast.

The white cross marks in the corner of the fingerprint image in Figure 23 label blocks with sufficiently low contrast.

Figure 23. Low contrast map results.

5.2.1.4. Low Flow Map [/NBIS/Main/mindtct/src/lib/lfs/maps.c; gen_initial_maps()]

It is possible, when deriving the initial direction map, for some blocks to have no dominant ridge flow. These blocks typically correspond to low-quality areas in the image. Initially these blocks are not assigned an orientation in the direction map, but subsequently some of these blocks may be assigned an orientation by interpolating the ridge flow of neighboring blocks. The *low flow map* marks the blocks that could not initially be assigned a dominant ridge flow.

In the event that minutiae are detected in these blocks, their assigned quality is reduced because they have been detected within a less reliable part of the image. The white cross marks in the fingerprint image in Figure 24 label blocks with no dominant ridge flow.

Figure 24. Low flow map results.

5.2.1.5. High Curve Map [/NBIS/Main/mindtct/src/lib/lfs/maps.c; gen_high_curve_map()]

Another part of fingerprint image that is problematic when it comes to detecting minutiae reliably is in areas of high curvature. This is especially true of the core and delta regions of a fingerprint.[36] The *high curve map* marks blocks that are in high-curvature areas of the fingerprint. Two different measures are used. The first, called *vorticity*, measures the cumulative change in ridge flow direction around all the neighbors of a block. The second called, *curvature*, measures the largest change in direction between a block's ridge flow and the ridge flow of each of its neighbors. The details are in the source code.

In the event that minutiae are detected in these blocks, their assigned quality is reduced because they have been detected within a less reliable part of the image. The white cross marks in the fingerprint image in Figure 25 label blocks with high-curvature ridges.

Figure 25. High curve map results.

5.2.1.6. Quality Map [/NBIS/Main/mindtct/src/lib/lfs/quality.c; gen_quality_map()]

The final image map produced by this package is a *quality map*. As discussed, the low contrast map, low flow map, and the high curve map all point to different low quality regions of the image. The information in these maps is integrated into one general map, as shown in Figure 26, and contains 5 levels of quality. The quality assigned to a specific block is determined based on its proximity to blocks flagged in these various maps. The details are in the source code.

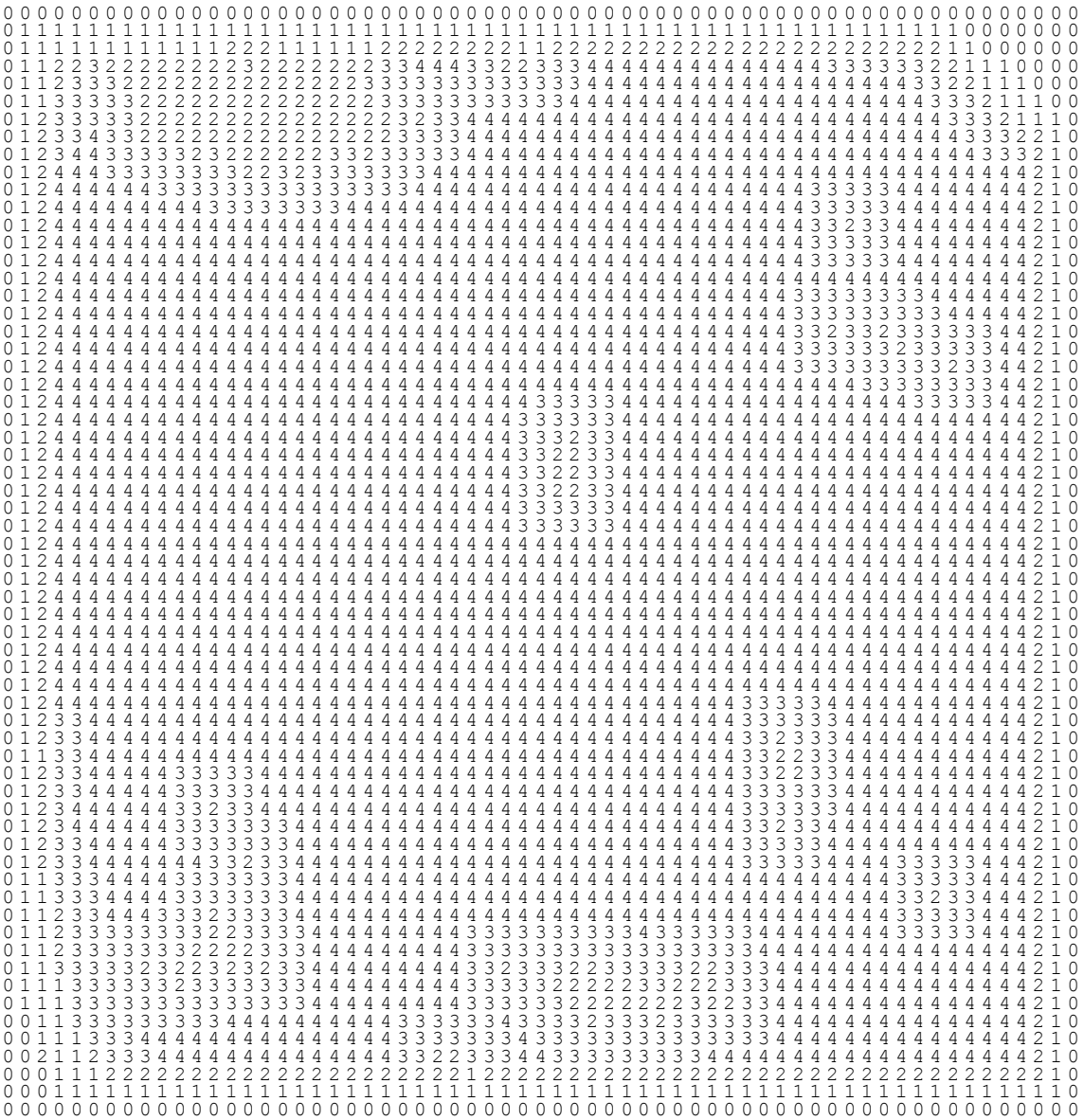


Figure 26. Quality map results.

5.2.3. Binarize Image [/NBIS/Main/mindtct/src/lib/lfs/binar.c; binarize_V2()]

The minutiae detection algorithm in this system is designed to operate on a bi-level (or binary) image where black pixels represent ridges and white pixels represent valleys in a finger's friction skin. To create this binary image, every pixel in the grayscale input image must be analyzed to determine if it should be assigned a black or white pixel. This process is referred to as image *binarization*.

A pixel is assigned a binary value based on the ridge flow direction associated with the block the pixel is within. If there was no detectable ridge flow for the current pixel's block, then the pixel is set to white. If there is detected ridge flow, then the pixel intensities surrounding the current pixel are analyzed within a rotated grid as illustrated in Figure 27.

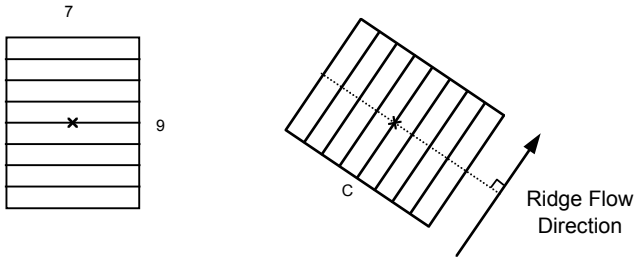


Figure 27. Rotated grid used to binarize the fingerprint image.

This grid is defined in the global control structure, `lfsparms_V2`, with column width (`DIRBIN_GRID_W`) set to 7 pixels and row height (`DIRBIN_GRID_H`) set to 9 pixels. With the pixel of interest in the center, the grid is rotated so that its rows are parallel to the local ridge flow direction. Grayscale pixel intensities are accumulated along each rotated row in the grid, forming a vector of row sums. The binary value to be assigned to the center pixel is determined by multiplying the center row sum by the number of rows in the grid and comparing this value to the accumulated grayscale intensities within the entire grid. If the multiplied center row sum is less than the grid's total intensity, then the center pixel is set to black; otherwise, it is set to white.

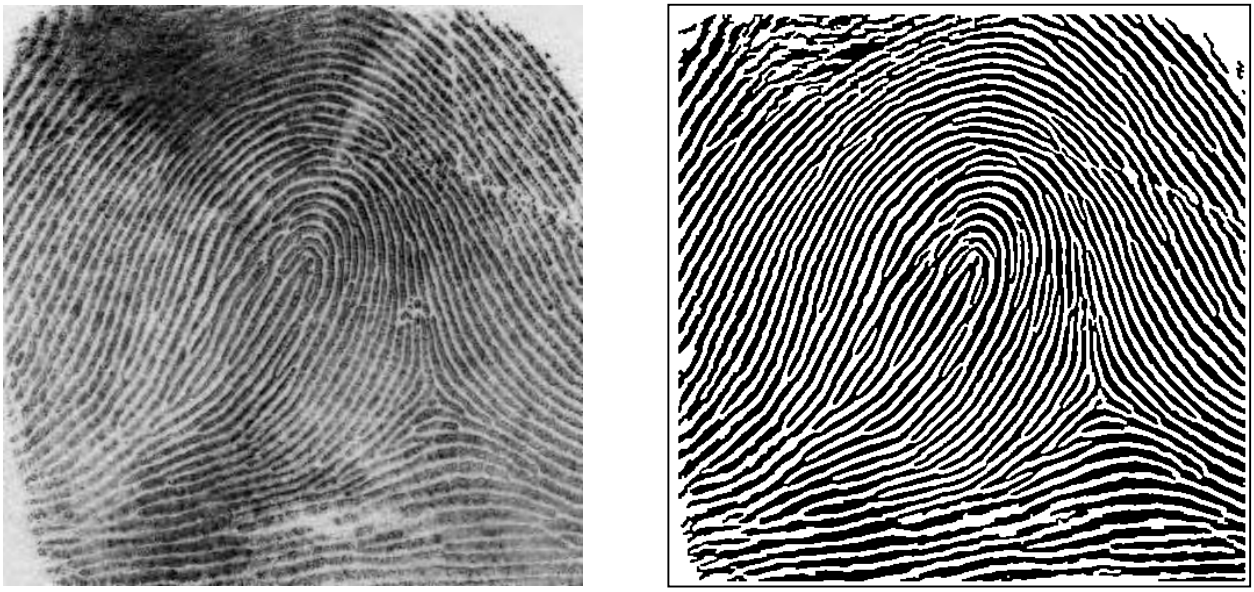


Figure 28. Binarization results

The results of binarization are shown in the Figure 28. The original grayscale image is on the left, and its binarization results are on the right.

It should be noted that the binarization step is critical to the successful detection of minutiae in this approach. The binarization results need to be *robust* in terms of effectively dealing with varying degrees of image quality and *reliable* in terms of rendering ridge and valley structures accurately. These are challenging, and at times conflicting goals. It is desirable to preserve as much image information and ridge/valley structure as possible so that minutiae are not missed, and yet it is undesirable to accentuate degraded areas in the image to the point of introducing *false* minutiae. Significant effort has been invested to promote both robust and reliable binary images, and yet the current system tends to produce a considerable number of false minutiae. This is particularly troublesome when processing latent fingerprint images.

5.2.4. Detect Minutiae `[/NBIS/Main/mindtct/src/lib/lfs/minutia.c; detect_minutiae_V2()]`

This step methodically scans the binary image of a fingerprint, identifying localized pixel patterns that indicate the ending or splitting of a ridge. The patterns searched for are very compact as illustrated in Figure 29. The left-most pattern contains six binary pixels in a 2×3 configuration. This pattern may represent the end of a black ridge protruding into the pattern from the right. The same is true for the next 2×4 pattern. The only difference between this pattern and the first one is that the middle pixel pair is repeated. In fact, this is true for all the patterns depicted. This "family" of ridge ending patterns can be represented by the right-most pattern, where the middle pair of pixels (signified by "∗") may repeat one or more times.

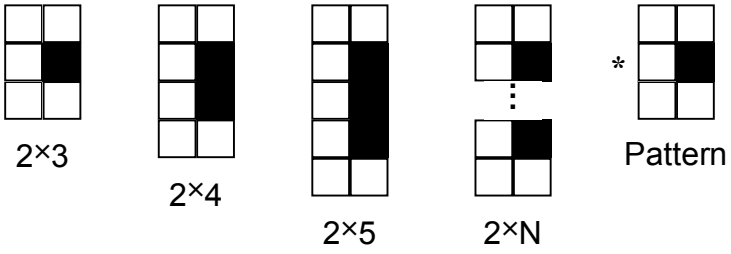


Figure 29. Pixel pattern used to detect ridge endings.

Candidate ridge endings are detected in the binary image by scanning consecutive pairs of pixels in the image looking for sequences that match this pattern. Pattern scanning is conducted both vertically and horizontally. The pattern as illustrated is configured for vertical scanning as the pixel pairs are stacked on top of each other. To conduct the horizontal scan, the pixel pairs are unstacked, rotated 90° clockwise, and placed back in sequence left to right.

Using the representation above, a series of minutiae patterns are used to detect candidate minutia points in the binary fingerprint image. These patterns are illustrated in Figure 30. There are two patterns representing candidate ridge endings, the rest represent various ridge bifurcations. A secondary attribute of appearing/disappearing is assigned to each pattern. This designates the direction from which a ridge or valley is protruding into the pattern. All pixel pair sequences matching these patterns, as the image is scanned both vertically and horizontally, form a list of candidate minutia points.

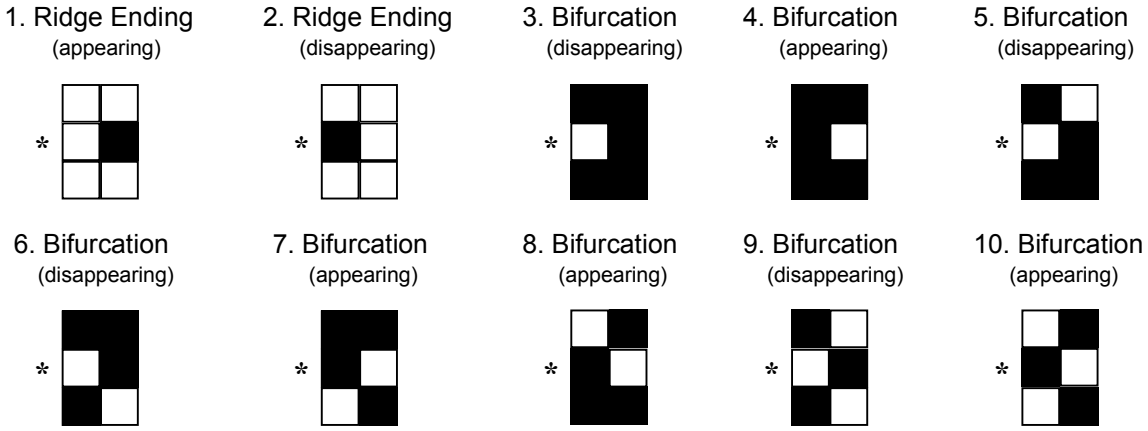


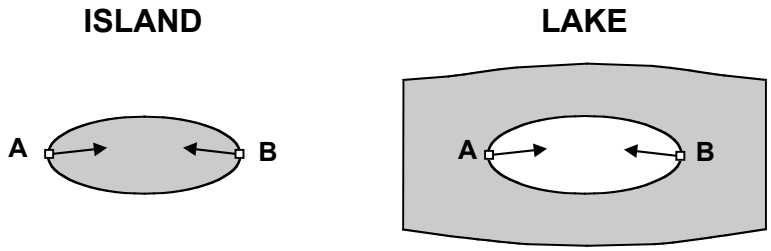
Figure 30. Pixel patterns used to detect minutiae.

5.2.5. Remove False Minutiae `[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_false_minutia_V2()]`

Using the patterns in Figure 30, candidate minutiae points are detected with as few as six pixels. This facilitates a particularly greedy detection scheme that minimizes the chance of missing true minutiae; however, many false minutiae are included in the candidate list. Because of this, much effort is spent on removing the false minutiae. These steps include removing islands, lakes, holes, minutiae in regions of poor image quality, side minutiae, hooks, overlaps, minutiae that

are too wide, and minutiae that are too narrow (pores). A short description of each of these steps is provided in the order in which they are executed.

5.2.5.1. Remove Islands and Lakes [src/lib/lfs/remove.c; remove_islands_and_lakes()]



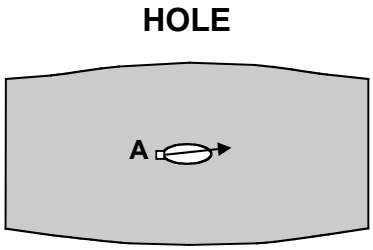
1. If (distance(A,B) <= 16 pixels) Then
    2. If (direction_angle(A,B) >= 123.75°) Then
        3. If (on_loop(A) && on_loop(B)) Then
            4. If (loop_length <= 60 pixels) Then
                5. remove(A,B)
                6. fill_loop()

Figure 31. Removal of islands and lakes.

In this step, ridge ending fragments and spurious ink marks (islands) along with interior voids in ridges (lakes) are identified and removed. These features are somewhat larger than the size of pores in the friction skin and they are often elliptical in shape; therefore, they typically will have a pair of candidate minutia points detected at opposite ends. An illustration of these types of features is shown in Figure 31.

Included at the bottom of the figure are the criteria used to detect islands and lakes. A pair of minutia must be within 16 pixels (MAX_RMTEST_DIST_V2) of each other. If so, then the directions of the two minutiae must be nearly opposite (≥ 123.75°) each other. Next, both minutiae must lie on the edge of the same loop, and the perimeter of the loop must be ≤ 60 pixels (MAX_HALF_LOOP_V2 × 2). If all these criteria are true, then the pair of candidate minutiae are removed for the list and the binary image is altered so that the island/lake is filled. Note that this is the *only* removal step that modifies the binary fingerprint image.

5.2.5.2. Remove Holes [/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_holes()]



1. If (on_loop(A)) Then
   2. If (loop_length <= 15 pixels) Then
      3. remove(A)

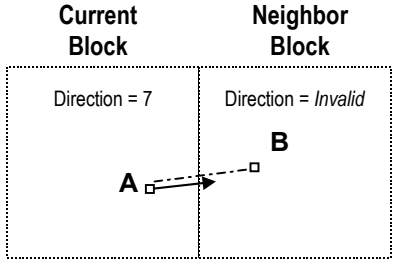
Figure 32. Removal of holes.

Here a hole is defined similarly to an island or lake, only smaller, and the loop need only have one minutia point on it. The criteria for removing a hole are illustrated in Figure 32. If a candidate minutia point lies on the edge of a loop with perimeter length ≤ 15 pixels (`SMALL_LOOP_LEN`), then the point is removed from the candidate list.

5.2.5.3. Remove Pointing to Invalid

Block `[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_pointing_invblock_V2()]`

This step and the next identify and remove candidate minutiae that are located near blocks that contain no detectable ridge flow. These blocks are referred to as containing *invalid* ridge flow direction and represent low-quality areas in the fingerprint image.



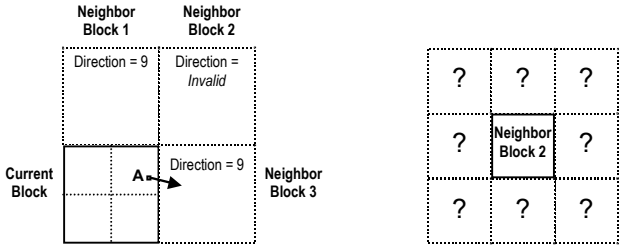
1. B = translate(A, 4 pixels, direction(A))
2. D = direction(block(B))
3. If (D == *Invalid*) Then
   4. remove(A)

Figure 33. Removal of minutia pointing to an invalid block.

This step is illustrated in Figure 33. A minutia point is translated 4 pixels (`TRANS_DIR_PIX_V2`) in the direction the minutia is pointing. If the translated point lies within a block with *invalid* ridge flow direction, then the original minutia point is remove from the list.

5.2.5.4. Remove Near Invalid Blocks

`[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_near_invblocks_V2()]`



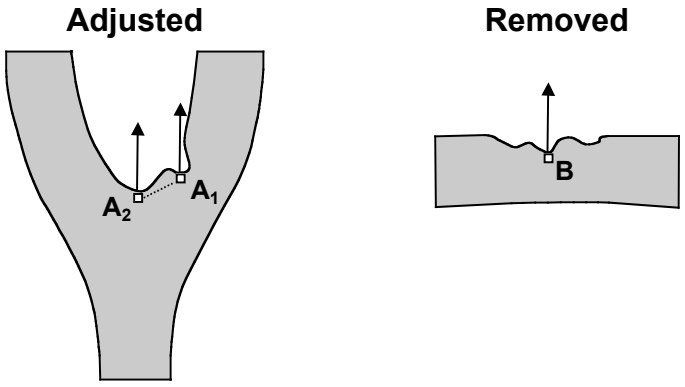
1. Nbrs = block_neighbors(A)
2. InvNbrs = invalid_directions(Nbrs)
3. Foreach Ni in InvNbrs
    4. Ni_Nbrs = neighbors(Ni)
    5. Ci = count_valid_directions(Ni_Nbrs)
    6. If (Ci < 7) Then
        7. remove(A)

Figure 34. Removal of minutia near invalid blocks.

Here, the proximity of a candidate minutia to a number of surrounding blocks with invalid ridge flow direction is evaluated. Given a minutia point, the blocks sufficiently close to the minutia (*details left to the source code*), and immediately neighboring the block in which the minutia resides, are tested in turn. If one of these neighboring blocks has invalid ridge flow direction, then its surrounding 8 neighbors are tested. The number of surrounding blocks with valid ridge flow direction are counted, and if the number of valid blocks is < 7 (`RM_VALID_NBR_MIN`), then the original minutia point is removed from the candidate list. Figure 34 illustrated this step.

5.2.5.5. Remove or Adjust Side Minutiae

`[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_or_adjust_side_minutiae_V2()]`



1. Pts = trace_contours(A, 7 pixels)
2. R_Pts = rotate_points_vertical(Pts, direction(A))
3. {Min_Ys,Max_Ys} = min_and_max_Ys(R_Pts)
4. If (#Min_Ys == 1) Then
   Adjust(A, Pts[Min_Y1])
5. Else If ({Min_Ys,Max_Ys} ==
   {Min_Y1,Max_Y1,Min_Y2}) Then
   Min_Y = point_at_min_Y(R_Pts, Min_Ys)
   Adjust(A, Pts[Min_Y])
6. Else revove(A)

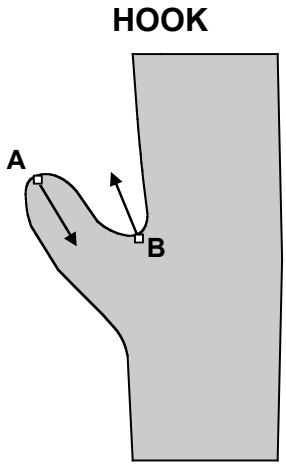
Figure 35. Removal or adjustment of minutiae on the side of a ridge or valley.

This step accomplishes two purposes. The first is to fine-tune the position of a minutia point so that it is more symmetrically placed on a ridge or valley ending. In the process, it may be determined that there is no clear symmetrical shape to the contour on which the candidate minutia lies. This is often the case with points detected along the *side* of a ridge or valley instead of the ridge or valley's ending. In this case, the misplaced minutia point is removed. In Figure 35, the illustration on the left depicts the adjustment of a minutia point from point A_1 to A_2. The illustration on the right depicts the removal of a side point, B.

To accomplish this, starting at the candidate minutia point, the edge of either the ridge or valley is traced to the right and to the left 7 pixels (`SIDE_HALF_CONTOUR`), producing a list of 15 contour points. The coordinates of these contour points are rotated so that the direction of the candidate minutia is pointing vertical. The rotated coordinates are then analyzed to determine the number and sequence of relative maxima and minima in the rotated y-coordinates. If there is only one y-coordinate minima, then the point of the minimum is assumed to lie at the bottom of a bowl-shaped rotated contour, and the candidate minutia is moved to correspond to this position in the original image. If there are more than one y-coordinate minima, then a specific sequence of minima-maxima-minima must exist, in which case the candidate minutia is moved to the point in the original image corresponding to the lowest y-coordinate minima. Again, this is assumed to be the bottom of a relatively bowl-shaped rotated contour. If there is more than one y-coordinate minima and there is not an exact minima-maxima-minima sequence along the rotated

contour, then the minutia point is determined to lie along the side of a ridge or valley, and it is removed from the candidate list.

5.2.5.6. Remove Hooks [/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_hooks()]



1. If (distance(A,B) <= 16 pixels) Then
    2. If (direction_angle(A,B) >= 123.75°) Then
        3. If (type(A) != type(B)) Then
            4. Pts = trace_contours(A, 30 pixels)
            5. If (in_points(Pts, B)) Then
                6. remove(A,B)

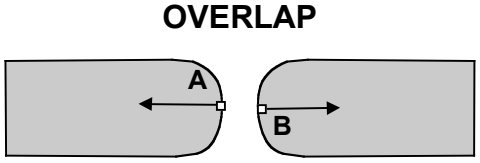
Figure 36. Removal of hooks.

A hook is a spike or spur that protrudes off the side of a ridge or valley. An example is illustrated in Figure 36. This feature typically has two minutiae of opposite type, one on a small piece of ridge and the other in a small valley, that are relatively close to each other. The two points must be within 16 pixels (MAX_RMTEST_DIST_V2) of each other, their directions must be nearly opposite (≥ 123.75°), they must be of opposite type, and they must lie on the same ridge/valley edge within 30 contour pixels (MAX_HOOK_LEN_V2) from each other. If all these are true, then the two minutia points are removed from the candidate list.

5.2.5.7. Remove Overlaps [/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_overlaps()]

In this step, an overlap is a discontinuity in a ridge or valley. These artifacts are typically introduced by the fingerprint impression process. A break in a ridge causes 2 false ridge endings to be detected, while a break in a valley causes 2 false bifurcations. The criteria for detecting an overlap are illustrated in Figure 37. Two minutia points must be within 8 pixels (MAX_OVERLAP_DIST) of each other, and their directions must be nearly opposite (≥ 123.75°). If so, then the direction of the line joining the two minutia points is calculated. If the difference between the direction of first minutia and the joining line is (≤ 90°), then the two minutiae are removed from the cadidate list. Otherwise, if the minutiae are within 6 pixels

(MAX_OVERLAP_JOIN_DIST) of each other, and there are no pixel value transitions along the joining line, then the points are removed from the candidate list.



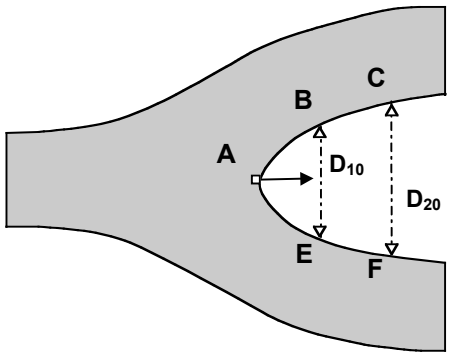
1. If (distance(A,B) <= 8 pixels) Then
    2. If (direction_angle(A,B) >= 123.75°) Then
        3. If(type(A) == type(B)) Then
            4. J = join_direction(A,B)
            5. If(direction_angle(180°-A,J) <= 90° Then
                6. remove(A,B)
            7. Else If (distance(A,B) <= 6 pixels &&
                free_path(A,B)) Then
                8. remove(A,B)

Figure 37. Removal of overlaps.

5.2.5.8. Remove Too Wide Minutiae

[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_malformations()]

The next two steps identify false minutiae that lie on malformed ridge and valley structures. A generalized ridge ending is comprised of a Y-shaped valley enveloping a black rod. The inverse is true for a generalized bifurcation. Simple tests are applied to evaluate the quality of this Y-shape.



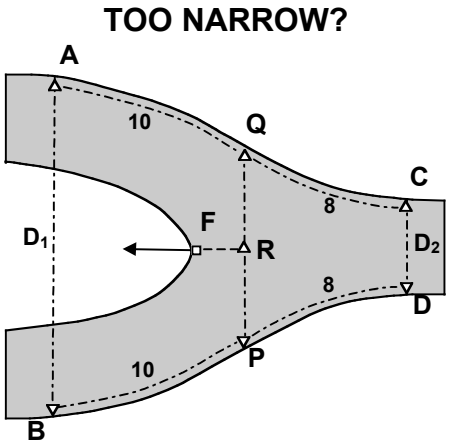
1. Pts1 = trace_contour(A, 20 pixels)
2. Pts2 = trace_contour(A, -20 pixels)
3. B = Pts1[10]; C = Pts1[20]
4. E = Pts2[10]; F = Pts2[20]
5. D10 = distance(B,E)
6. D20 = distance(C,F)
7. If ((D20/D10) > 2.0) Then
    8. remove(A)

Figure 38. Removal of too wide minutiae.

This step evaluates whether the structure enveloping a ridge or valley ending is relatively Y-shaped and not too wide. Figure 38 illustrates the criteria applied. The edge of the ridge or valley is traced to the left and to the right 20 pixels (MALFORMATION_STEPS_2), producing 2 lists of contour points. On each contour, coordinates at pixel index 10 (B&E) and at pixel index 20 (C&F) are stored. The distance between pixels at index 10 (MALFORMATION_STEPS_1) is computed as is the distance between pixels at index 20. The ratio of these two distances is then calculated (D_{20}/D_{10}), and if the ratio is larger than 2.0 (MIN_MALFORMATION_RATIO), then the minutia point is removed from the candidate list. It should be noted that based on these criteria the bifurcation in the illustration would *not* be removed.

5.2.5.9. Remove Too Narrow Minutiae

[/NBIS/Main/mindtct/src/lib/lfs/remove.c; remove_pores_V2()]



1. T = 180° - direction(F)
2. R = translate(F, 3 pixels, T)
3. Q = find_edge(R, Up, 12 pixels)
4. P = find_edge(R, Down, 12 pixels)
5. Pts = trace_contour(Q, 10 pixels)
6. A = Pts[10]
7. Pts = trace_contour(Q, -8 pixels)
8. C = Pts[8]
9. Pts = trace_contour(P, 10 pixels)
10. B = Pts[10]
11. Pts = trace_contour(P, -8 pixels)
12. D = Pts[8]
13. D1 = distance(A,B)
14. D2 = distance(C,D)
15. If ((D1/D2) <= 2.25) Then
    16. remove(F)

Figure 39. Removal of too narrow minutiae.

The previous step tests for candidate minutiae that are too wide. This step tests for points that are on structures that are too narrow. This is typical, for example, of pores in the friction skin. Figure 39 illustrates this test. Starting with the candidate minutia point, F, its coordinates are translated 3 pixels (PORES_TRANS_R) opposite the minutia's direction. The top edge and bottom

edges of the enveloping structure are then located at (Q&P). From these two points, the edge is traced to the left 10 pixels (PORES_STEP_FWD) and to the right 8 pixels (PORES_STEP_BWD). The points at the end of the 10 pixel contours are stored (A&B), and the points at the end of the 8-pixel contours are stored (C&D). Next, distances are computed between these pairs of points, and the ratio (D_1/D_2) is computed. If the ratio is ≤ 2.25 (PORES_MAX_RATIO), then the minutia point is removed from the candidate list. In fact, if the process fails to find any of the points in the illustration, then the candidate minutia is removed. It should be noted that, mindtct, only searches for minutiae that are too narrow within high-curvature regions or regions where ridge flow direction is non-determinable.

5.2.6. Count Neighbor Ridges [/NBIS/Main/mindtct/src/lib/lfs/ridges.c; count_minutiae_ridges()]

Fingerprint minutiae matchers often use information in addition to just the points themselves. Ancillary information usually includes the minutia's direction, its type, and it may include information pertaining to minutiae neighbors. Beyond a minutia's position, direction, and type, there are no standard neighbor schemes. Different AFIS systems use different neighbor topologies and attributes. One common attribute is the number of intervening ridges (called ridge crossings) between a minutia and each of its neighbors. For example, the FBI's IAFIS uses ridge crossings between a minutia and its 8 nearest neighbors, where each neighbor is the closest within a specified octant.[37]

The neighbor scheme distributed with this system has been directly inherited from HO39.**[55]** Up to 5 nearest neighbors (MAX_NBRS) are reported. Given a minutia point, the closest neighbors below (in the same pixel column), and to the right (within entire pixel columns) in the image are selected. These nearest neighbors are sorted in order of their direction, starting with vertical and working clockwise. Using this topology, ridge counts are computed and recorded between a minutia point and each of its nearest neighbors.

5.2.7. Assess Minutia Quality [/NBIS/Main/mindtct/src/lib/lfs/quality.c; combined_minutia_quality()]

One of the goals of developing this software package was to compute a quality/reliability to be associated with each detected minutia point. Even with the lengthy list of removal steps above, false minutiae potentially remain in the candidate list. A robust quality measure can help manage this in that false minutiae should be assigned a lower quality than true minutiae. Through dynamic thresholding, a trade off between retaining false minutiae and throwing away true minutiae may be determined. To this end, mindtct, computes and reports minutiae qualities.

Two factors are combined to produce a quality measure for each detected minutia point. The first factor, *L*, is taken directly from the location of the minutia point within the quality map described in Section 5.2.1.6. One of five quality levels is initially assigned, with 4 being the highest quality and 0 being the lowest.

The second factor is based on simple pixel intensity statistics (mean and standard deviation) within the immediate neighborhood of the minutia point. The size of the neighborhood is set to 11 pixels (RADIUS_MM). This is sufficiently large to contain generous portions of an average ridge and valley. A high quality region within a fingerprint image will have significant contrast

that will cover the full grayscale spectrum. Consequently, the mean pixel intensity of the neighbor hood will be very close to 127. For similar reasons, the pixel intensities of an ideal neighborhood will have a standard deviation ≥ 64.

Using this logic, the following reliability measure, R, is calculated given neighborhood mean, μ, and standard deviation, σ:

$$F_{\mu} = 1.0 - \frac{|\mu - 127|}{127}$$

$$F_{\sigma} = \begin{cases} 1.0 & if\ \sigma > 64 \\ \dfrac{\sigma}{64} & otherwise \end{cases}$$

$$R = \min(F_{\mu}, F_{\sigma})$$

Minutia quality, Q, is calculated using quality map level, L, and reliability, R, as:

$$Q = \begin{cases} .50 + (.49 * R) & if\ L = 4 \\ .25 + (.24 * R) & if\ L = 3 \\ .10 + (.14 * R) & if\ L = 2 \\ .05 + (.04 * R) & if\ L = 1 \\ .01 & if\ L = 0 \end{cases}$$

This results in a quality value on the range .01 to .99. A low quality value represents a minutia detected in a lower quality region of the image, whereas a high quality value represents a minutia detected in a higher quality region.

5.2.8. Output Minutiae File [/NBIS/Main/an2k/src/lib/an2k/fmtstd.c; write_ANSI_NIST_file(), /NBIS/Main/mindtct/src/lib/lfs/results.c; write_text_results()]

Upon completion, `mindtct`, takes the resulting minutiae and outputs them to a file. If the input file was an ANSI/NIST formatted file, `mindtct` adds two new records and writes a new ANSI/NIST formatted file to <oroot>.mdt, where <oroot> is passed as a parameter to `mindtct`. The new records are a Type-9 record, holding the detected minutiae, is constructed and inserted along with a Type-13 or Type-14 record, holding the image binarization results. If the input image is of a latent fingerprint, then the binarization results are stored in a Type-13 record; otherwise, the image results are stored in a Type-14 record. It should be noted that the minutiae in the Type-9 record are formatted in the NIST-assigned fields 5-12 according to the ANSI/NIST standard.**[30]** The utilities, `an2k2iaf` and `iaf2an2k`, as described in the Reference Manual of this document may be used to convert between these fields and the FBI/IAFIS-assigned fields 13-23.**[37]** If the input file is not in ANSI/NIST format, the resulting minutiae can be accessed in the text file <oroot>.min and there is no ANSI/NIST output file created but a raw pixel file is created that has the image binarization results.

For all input types the detected minutiae are also written to a text file <oroot>.xyt that is formatted for use with the `bozorth3` matcher. This file has one space delimited line per minutiae containing its x and y coordinate, direction angle theta, and the minutiae quality.

The output minutiae are in the ANSI/NIST format described in Section 3.2.1 which has the origin at the bottom left of the image and directions pointing out and away from the ridge ending or bifurcation valley. There is an option to output the minutiae in the M1 (ANSI INCITS 378-

3004) representation which has the pixel origin at the top left of the image and directions pointing up the ridge ending or bifurcation valley. If this option (-m1) is used when running `mindtct` it should also be used by `bozorth3` when matching the minutiae files.

A number of other output files are produced. These include a file for each of the image maps described in Section 5.2.1.1 and a log file listing all the detected minutiae and their associated attributes. All of these are text files and are created by `mindtct` in the current working directory with fixed file names. The direction map is stored in `<oroot>.dm`; the low contrast map is stored in `<oroot>.lcm`; the low flow map is stored in `<oroot>.lfm`; the high curve map is stored in `<oroot>.hcm`; and the quality map is stored in `<oroot>.qm`. The maps are represented by a grid of numbers, each corresponding to a block in the fingerprint image. The last text output file, `<oroot>.min`, contains a formatted listing of attributes associated with each detected minutiae in the fingerprint image. Among these attributes are the minutia's pixel coordinate location, its direction, and type. The format and all the attributes reported in this file are described within the `mindtct` manual page in APPENDIX C. Output files generated from `mindtct` are provided in `/NBIS/Test/mindtct/execs/mindtct`.

Figure 40 displays the detected minutiae for the example fingerprint.

Figure 40. Minutiae results.

5.3. NFIQ [/NBIS/Main/nfiq/src/lib/nfiq/{nfiq.c, nfiqggbls.c, nfiqread.c, znorm.c}]

The NFIQ algorithm is an implementation of the NIST "Fingerprint Image Quality" algorithm as described in.**[57]** It takes an input image that is in ANSI/NIST or NIST IHEAD format or compressed using WSQ, baseline JPEG, or lossless JPEG. NFIQ outputs the image quality value for the image (where 1 is highest quality and 5 is lowest quality) and its output activation level from the MLP neural network. The details and analysis of the NFIQ algorithm are in **[57]** and a copy of it has been included on this CDROM in the file doc/ir_7151.pdf. Appendices for ir_7151.pdf are available at http://fingerprint.nist.gov/NBIS. Information on training the neural network weights used by NFIQ is provided below in Section 5.3.1. Details on the fingerprint data used to create the current weights being used by NFIQ are provided in Section 4.2.1 of [57].

Please note that NFIQ has been designed to assign a level of quality to a grayscale fingerprint image. It was not designed to distinguish fingerprint images from all possible non-fingerprint images, and it will not necessarily assign non-fingerprint image types to the worst quality level five.

5.3.1. NFIQ Neural Network Training

Neural networks offer a very powerful and very general framework for representing non-linear mappings from several input variables to several output variables, where the form of the mapping is governed by a number of adjustable parameters (*weights*). The process of determining the values for these weights based on the data set is called *training* and the data set of examples is generally referred to as a *training set*. There is quite an art in training neural networks. The model is generally over parameterized, and the optimization problem is non-convex and unstable unless certain guidelines are followed.

This section summarizes some of the important issues in the training process of a MLP used to assess the quality of a gray-scale fingerprint image, as is done in NFIQ. The reader is encouraged to read the MLP documentation contained in Sections 5.1.3 and APPENDIX A.

5.3.1.1. Number of Hidden Units and Layers

Generally, it is better to have too many hidden units than too few. With too few hidden units, the model might not have enough flexibility to capture the non-linearities in the data; with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization and pruning is used. NFIQ training used 22 hidden units, which is twice the number of input nodes.

5.3.1.2. Training Set

For a successful training, the training set must be a true representation of real, practical data. It is important to design a training set balanced in terms of different finger positions of different quality, and if possible, of different operational settings. Our training set consisted of 5244 flat impressions of right/left index and thumbs with various level of quality as explained in Section 4.2.1 of [57].

5.3.1.3. Class-Weights

It is a common classification problem where large samples of rare classes may not be available. Our training set had more samples of high and good quality than samples of low and very low quality. We compensated for smaller samples of poor quality prints in our training set by setting the *class-weight* parameter of MLP. Prior class probabilities are used to normalize all error calculations so that statistically significant samples of the rare but important class of very poor quality fingerprint images; can be included without distorting the error surface.

5.3.1.4. Scaling of the Inputs

Since the scaling of the inputs determines the effective scaling of the weights in the bottom layer, it can have a large effect on the quality of the final solution. At the outset, it is best to standardize all inputs to have mean zero and standard deviation one. We computed mean and standard deviation of 240,000 fingerprint images (global mean and standard deviation) and used these statistics to standardize input patterns. These are stored in the file `/NBIS/Main/nfiq/znorm.dat`.

5.3.1.5. Boltzmann Pruning

The dimension of the weight space is constrained by using Boltzmann pruning. Boltzmann pruning also keeps the information content of the weights bounded at values, which are equal to or less than the information content of the training set.

5.3.1.6. Regularization

Regularization decreases the volume of weight space used in the optimization process. This is achieved by adding an error term, which is proportional to the sum of the squares of the weights. This reduces the average magnitude of the weights by creating a parabolic term in the error function that is centered on the origin.[49] A scheduled sequence of regularization values is used which starts with high regularization and decreases until no further change in the form of the error-reject curve is detected.

Training was carried out by selecting a Boltzmann temperature of 1.0e-4 and successively training the network at decreasing values of the regularization factor. More specifically, the file `nfiq/spec` contains the settings for the multiple training runs, which starts with the regularization factor at 1.0 and decreases gradually to $1.0e^{-6}$.

5.3.1.7. How to perform training

`Fing2pat` gets the list of gray-scale fingerprint images in the training set along with their class labels as input and computes patterns for MLP training and writes them to a binary file. Each pattern consists of a feature vector, along with a class vector. `Fing2pat` can also z-normalize each pattern based on statistics provided by an input file specified on the command line with the –z flag. The user can compute the global mean and standard deviation statistics using `znormdat` or use the set provided in `/NBIS/Main/nfiq/runtimedata/znorm.dat`. These global statistics can be applied to new pattern files using `nzormpat`. The user needs to write a `spec` file, setting parameters of the training runs that MLP is to perform. The spec file used in

the training of NIST Fingerprint Image Quality can be found in the file `/NBIS/Main/nfiq/runtimedata/spec`.

6. REFERENCES

(Some of the references listed can be downloaded at http://www.itl.nist.gov/iad/894.03.)

[1] J.H. Wegstein, "A Semi-automated Single Fingerprint Identification System," NBS Technical Note 481, April 1969.

[2] J.H. Wegstein, "Automated Fingerprint Identification," NBS Technical Note 538, August 1970.

[3] J.H. Wegstein, "Manual and Computerized Footprint Identification," NBS Technical Note 712, February 1972.

[4] R.T. Moore, "The Influence of Ink on The Quality of Fingerprint Impressions," NBS Technical Report NBSIR 74-627, December 1974.

[5] J.H. Wegstein, "The M40 Fingerprint Matcher," NBS Technical Note 878, July 1975.

[6] J.H. Wegstein, and J.F. Rafferty, "The LX39 latent Fingerprint Matcher," NBS Special Publication 500-36, August 1978.

[7] R.T. Moore, "Results of Fingerprint Image Quality Experiments," NBS Technical Report NBSIR 81-2298, June 1981.

[8] J.H. Wegstein, "An Automated Fingerprint Identification System," NBS Special Publication 500-89, February 1982.

[9] R.M. McCabe, and R.T. Moore, "Data Format for Information Interchange," American National Standard ANSI/NBS-ICST 1-1986, August 1986.

[10] R.T. Moore, "Automated Fingerprint Identification Systems - Benchmark Test of Relative Performance," American National Standard ANSI/IAI 1-1988, February 1988.

[11] R.T. Moore, "Automated Fingerprint Identification Systems – Glossary of Terms and Acronyms," American National Standard ANSI/IAI 2-1988, July 1988.

[12] R.T. Moore, R.M. McCabe, and R.A. Wilkinson, "AFRS Performance Evaluation Tests," NBS Technical Report NBSIR 88-3831, August 1988.

[13] "Minimum Image Quality Requirements for Live Scan, Electronically Produced Fingerprint Cards," Technical Report for the Federal Bureau of Investigation – Identification Division, November 1988.

[14] C. Watson, "*NIST Special Database 4*: 8-bit Gray Scale Images of Fingerprint Image Groups," CD-ROM & documentation, March 1992.

[15] C.L. Wilson, G.T. Candela, P.J. Grother, C.I. Watson, and R.A. Wilkinson, "Massively Parallel Neural Network Fingerprint Classification System," Technical Report NISTIR 4880, July 1992.

[16] R. McCabe, C. Wilson, and D. Grubb, "Research Considerations Regarding FBI-IAFIS Tasks & Requirements," NIST Technical Report NISTIR 4892, July 1992.

[17] G.T. Candela and R. Chellappa, "Comparative Performance of Classification Methods for Fingerprints," Technical Report NISTIR 5163, April 1993.

[18] C. Watson, "*NIST Special Database 9*: 8-Bit Gray Scale Images of Mated Fingerprint Card Pairs," Vol. 1-5, CD-ROM & documentation, May 1993.

[19] C. Watson, "*NIST Special Database 10*: Supplemental Fingerprint Card Data (SFCD) for NIST Special Database 9," CD-ROM & documentation, June 1993.

[20] C. Watson, "*NIST Special Database 14*: Mated Fingerprint Card Pairs 2," CD-ROM & documentation, September 1993.

[21] R.M. McCabe, "Data Format for the Interchange of Fingerprint Information," American National Standard ANSI/NIST-CSL 1-1993, November 1993.

[22] J.L. Blue, G.T. Candela, P.J. Grother, R. Chellappa, C.L. Wilson, "Evaluation of Pattern Classifiers for Fingerprint and OCR Application," in Pattern Recognition, 27, pp. 485-501, 1994.

[23] C.L. Wilson, G.T. Candela, C.I. Watson, " Neural Network Fingerprint Classification," in Journal for Artificial Neural Networks, 1(2), 203-228, 1994.

[24] C.I. Watson, J. Candela, P. Grother, "Comparison of FFT Fingerprint Filtering Methods for Neural Network Classification," Technical Report NISTIR 5493 September 1994.

[25] C. Watson, "*NIST Special Database 18*: Mugshot Identification Database of 8 bit gray scale images," CD-ROM & documentation, December 1994.

[26] G.T. Candela, P.J. Grother, C.I. Watson, R.A. Wilkinson, C.L. Wilson, "PCASYS - A Pattern-level Classification Automation System for Fingerprints," Technical Report NISTIR 5647 & CD-ROM, April 1995.

[27] R.M. McCabe, "Data Format for the Interchange of Fingerprint, Facial & SMT Information," American National Standard ANSI/NIST-ITL 1a-1997, April 1997.

[28] C. Watson, "*NIST Special Database 24*: Digital Video of Live-Scan Fingerprint Data," CD-ROM & documentation, July 1998.

[29] M.D. Garris and R.M. McCabe, "*NIST Special Database 27*: Fingerprint Minutiae From Latent and Matching Tenprint Images," CD-ROM & documentation, June 2000.

[30] R.M. McCabe, "Data Format for the Interchange of Fingerprint, Facial, Scar Mark & Tattoo (SMT) Information," American National Standard ANSI/NIST-ITL 1-2000, July 2000. Available from R.M. McCabe at NIST, 100 Bureau Drive, Stop 8940, Gaithersburg, MD 20899-8940.

[31] l. Hong, Y. Wan, and A. Jain, "Fingerprint Image Enhancement: Algorithm and Performance Evaluation," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 777-789, August 1998.

[32] GNU project - free UNIX-like utilities. Learn more at http://www.gnu.org.

[33] Cygwin tools - free GNU utility port for Win32 machines. Learn more at http://www.cygwin.com/.

[34] Linux - a freely available clone of the UNIX operating system. Learn more at http://www.linux.org.

[35] Independent JPEG Group (IJG) - learn more at http://www.ijg.org.

[36] "The Science of Fingerprints," Rev. 12-84, U.S. Department of Justice, Federal Bureau of Investigation. Available from U.S. Government Printing Office, Washington D.C. 20402.

[37] "Electronic Fingerprint Transmission Specification," CJIS-RS-0010 (V7). Available from Criminal Justice Information Services Division, Federal Bureau of Investigation, 935 Pennsylvania Avenue, NW, Washington D.C. 20535.

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

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

[40] 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-l:13-17, 1969.

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

[42] D.F. Specht, "Enhancements to Probabilistic Neural Networks," In *International Joint Conference on Neural Networks,* pages 1-761 - 1-768, June 1992.

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

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

[45] P.J. Grother, G.T. Candela, and J.L. Blue, "Fast Implementations of Nearest Neighbor Classifiers," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 1995.

[46] 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.

[47] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbau S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. C translation by J. Demmel and Xiaoye Li, LAPACK Users Guide, SIAM, Philadelphia, 1992.

[48] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, Parallel Distributed Processing, Volume 1: Foundations, edited by D.E. Rumelhart, J.L. McClelland, et al., MIT Press, Cambridge, pp. 318-362, 1986.

[49] C.L. Wilson, J.L. Blue, O.M. Omidvar, "The Effect of Training Dynamics on Neural Network Performance," Technical Report NISTIR 5696, August 1995.

[50] C.L. Wilson, J.L. Blue, and O.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.

[51] C.L. Wilson, J.L. Blue, and O.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.

[52] J. L. Blue and P. J. Grother, "Training Feed Forward Networks Using Conjugate Gradients," Technical Report NISTIR 4776, February 1992, and in Conference on Character Recognition and Digitizer Technologies, Vol. 1661, pp. 179-190, SPIE, San Jose, February 1992.

[53] D. Liu and J. Nocedal, "On the Limited Memory BFGS Method for Large Scale Optimization," Mathematical Programming B, Vol. 45, 503-528, 1989.

[54] O.M. Omidvar and C.L. Wilson, "Information Content in Neural Net Optimization," Technical Report NISTIR 4766, February 1992, and in Journal of Connection Science, 6:91-103, 1993.

[55] Home Office Automatic Fingerprint Recognition System (HOAFRS), License 16-93-0026, Science and Technology Group, Home Office, London, 1993.

[56] C. Wilson, M. Garris, C. Watson, A, Hicklin, "Studies of Fingerprint Matching Using the NIST Verification Test Bed (VTB)," Technical Report NISTIR 7020, July 2003. http://www.itl.nist.gov/iad/894.03/pact/pact.html.

[57] E. Tabassi, C. Wilson, C. Watson, "Fingerprint Image Quality," Technical Report 7151, August 2004. Appendices for NISTIR 7151 can be found at http://fingerprint.nist.gov/NBIS.

[58] C. Wilson, A. Hicklin, B. Ulery, H. Korves, M. Zoepfl, P. Grother, R. Michaels, C. Watson, S. Otto, M. Bone, "Fingerprint Vendor Technology Evaluation (FpVTE) 2003," Technical Report NISTIR 7123, June 2004. http://fpvte.nist.gov/

[59] C. Watson, C. Wilson, M. Indovina, R. Snelick, K. Marshall, "Studies of One-to-One Matching with Vendor SDK Matchers," Technical Report NISTIR 7119, July 2004.

[60] C. Wilson, M. Garris, C. Watson, "Matcher Performance for the US-VISIT IDENT System Using Flat Fingerprints", Technical Report NISTIR 7110, May 2004. http://www.itl.nist.gov/iad/894.03/pact/pact.html.

[61] C. Watson, "*NIST Special Database 29*: Plain and Rolled Images from Paired Fingerprint Cards," CD-ROM & documentation NISTIR 6801, November 2001.

[62] C. Watson, "*NIST Special Database 30*: Dual Resolution Images from Paired Fingerprint Cards," CD-ROM & documentation NISTIR 6800, November 2001.

[63] "WSQ Gray-scale Fingerprint Image Compression Specification," Criminal Justice Information Services, FBI, December 1997.

[64] S. Wood, C. Wilson, "Studies of Plain-to-Rolled Fingerprint Matching Using the NIST Algorithmic Test Bed (ATB)," Technical Report NISTIR 7112, April 2004. http://www.itl.nist.gov/iad/894.03/pact/pact.html.

APPENDIX A. MLP TRAINING OUTPUT

Explanation of the output produced during MLP training

When the program `mlp` does a training run, it writes output to the standard error and writes the same output to the **short_outfile** specified in the specfile. The purpose of this appendix is to explain the meaning of this output. `Mlp` produces similar output for a testing run except that the "training progress" part is missing.

Pattern-Weights

As a preliminary, it will be helpful to discuss the "pattern-weights" which `mlp` uses, since they are used in the calculations of many of the values shown in the output. The pattern-weights are "prior" weights, one for each pattern;[12] they remain constant during a training (or testing) run, although it is possible to do a training "meta-run" that is a sequence of training runs and to change the pattern-weights between the runs. The setting of the pattern-weights is controlled by the **priors** value set in the specfile and may be affected by provided data files, as follows (in all cases, the division by *N* is merely a normalization that slightly reduces the amount of calculation needed later):

allsame: if **priors** is **allsame**, then each pattern-weight is set to (1/*N*), where *N* is the number of patterns.

class: a file of given class-weights must be supplied; each given class-weight is divided by the actual class-weight of the input data set and the new class-weights are normalized so their sum is 1.0. Then each pattern-weight is set to the new class-weight of the class of the corresponding pattern, divided by *N* (number of patterns). The end result is that if the actual distribution of the data set does not equal that of the given class-weights, class-weights are adjusted so the final results approximate what the scores would be if the distribution were the same as the given class-weights. If the user is only concerned about the unadjusted score for the given data, set the given class-weights equal to the actual class-weights.

pattern: a file of (original) pattern-weights must be supplied; each of them is divided by *N* to produce the corresponding pattern-weight.

both: files of class-weights and (original) pattern-weights must both be supplied; each pattern-weight is then set to the class-weight (class-weights are adjusted as discussed in the **class** portion of this list) of the class of the corresponding pattern, times the corresponding (original) pattern-weight, divided by *N*.

The pattern-weights are used in the calculation of the error value that `mlp` attempts to minimize during training. When the training patterns are sent through the network, each pattern produces an error contribution, which is multiplied by the pattern-weight for that pattern before being added to an error accumulator (Section A.1.1.2.2). The pattern-weights are also involved in the calculations of several other quantities besides the error value; all these uses are described below. References [49] discuss the use of class-based prior weights (Section 5.4, pages 10-11) which correspond to the class setting of **priors**.

[12] A pattern is a feature-vector/class or feature-vector/target-vector pair

Explanation of Output

A.1.1.1 Header

The first part of the output is a "header" showing the specfile parameter values. Here is the header of the **short_outfile** `/NBIS/Test/pcasys/execs/mlp/mlp_dir/trn1.err` produced by the first training run of a sequence of runs used to train the fingerprint classifier:

```
Classifier MLP
 Training run
 Patterns file: fv1-9mlp.kls; using all 24300 patterns
 Final pattern-wts: made from provided class-wts and pattern-wts,
  files priors and patwts
 Error function: sum of squares
 Reg. factor: 2.000e+00
 Activation fns. on hidden, output nodes: sinusoid, sinusoid
 Nos. of input, hidden, output nodes: 128, 128, 6
 Boltzmann pruning, thresh. exp(-w^2/T), T 1.000e-05
 Will use SCG
 Initial network weights: random, seed 12347
 Final network weights will be written as file trn1.wts
 Stopping criteria (max. no. of iterations 50):
  (RMS err) <= 0.000e+00 OR
  (RMS g) <= 0.000e+00 * (RMS w) OR
  (RMS err) > 9.900e-01 * (RMS err 10 iters ago) OR
  (OK - NG count) < (count 10 iters ago) + 1. (OK level: 0.000)
 Long outfile: trn1l.err

 Given and Actual Prior Weights
   A => 0.036583 0.038025
   L => 0.338497 0.319506
   R => 0.316920 0.306584
   S => 0.000000 0.005597
   T => 0.029482 0.030123
   W => 0.278518 0.300165
 Given/Actual = New Prior Weights
   A -> 0.193897
   L -> 0.213518
   R -> 0.208333
   S -> 0.000000
   T -> 0.197247
   W -> 0.187005

 SCG: doing <= 50 iterations; 17286 variables.
```

A.1.1.2 Training Progress

The next part of the output lists a running update on the training progress. The first few lines of training progress reported are:

```
pruned    80     6    86   C  1.67872e+05  H  2.40068e+04  R   85.70  M  -0.00  T  0.0841
     Iter   Err (   Ep    Ew)     OK    UNK     NG      OK   UNK    NG
        0 0.474 (0.240 0.289)   6564      0  17736 =  27.0   0.0  73.0 %
   0.0    0  4 19  0  0 70
 pruned   108     3   111   C  1.75555e+05  H  2.54052e+04  R  85.53  M  -0.00  T  0.0836
 pruned   124     5   129   C  1.84026e+05  H  2.58204e+04  R  85.97  M  -0.00  T  0.0824
 pruned   129     6   135   C  2.20275e+05  H  2.72642e+04  R  87.62  M  -0.00  T  0.0814
 pruned   138     3   141   C  1.73226e+05  H  2.76075e+04  R  84.06  M  -0.00  T  0.0803
 pruned   138     5   143   C  1.78328e+05  H  2.99593e+04  R  83.20  M  -0.00  T  0.0762
 pruned   152     4   156   C  1.74579e+05  H  3.03576e+04  R  82.61  M  -0.00  T  0.0745
 pruned   167     5   172   C  1.81337e+05  H  3.14710e+04  R  82.65  M  -0.00  T  0.0681
 pruned   149     7   156   C  1.89832e+05  H  3.95510e+04  R  79.17  M  -0.00  T  0.0536
 pruned   178     7   185   C  1.78410e+05  H  3.90489e+04  R  78.11  M  -0.00  T  0.0526
 pruned   184     7   191   C  2.19716e+05  H  3.99658e+04  R  81.81  M  -0.00  T  0.0490
       10 0.328 (0.103 0.220)  19634      0   4666 =  80.8   0.0  19.2 %
   0.0    2 90 99  0  1 68
```

The line

```
     Iter   Err (   Ep    Ew)     OK    UNK     NG      OK   UNK    NG
```

comprises column headers that pertain to those subsequent lines that begin with an integer ("first progress lines"); each first progress line is followed by a "second progress line," and there are "pruning lines" if Boltzmann pruning is used. These three types of lines are discussed below, second progress lines first because some of the calculations used to produce them are later used to make the first progress lines.

A.1.1.2.1 Second progress lines

These are the lines that begin with fractional numbers; the first of them in the above example is

```
0.0    0  4 19  0  0 70
```

Ignoring for a moment the first value in such a line, the remaining values are the "percentages" right by class, which `mlp` calculates as follows. It maintains three pattern-weight-accumulators for each class:

$a_i^{(r)}$ = right pattern-weight-accumulator for correct class i

$a_i^{(w)}$ = wrong pattern-weight-accumulator for correct class i

$a_i^{(u)}$ = unknown (rejected) pattern-weight-accumulator for correct class i

When `mlp` sends a training pattern through the network the result is an output activation for each class; the hypothetical class is, of course, whichever class receives the highest activation. If the highest activation equals or exceeds the rejection threshold **oklvl** set in the specfile, then `mlp` accepts its result for this pattern, and adds its pattern-weight (Section 0) to either $a_i^{(r)}$ or $a_i^{(w)}$ (where i is the correct class of the pattern) according to whether the network classified the pattern rightly or wrongly. Otherwise, (i.e. if the highest activation is less then **oklvl**) `mlp` adds the pattern-weight to $a_i^{(u)}$. These accumulators reach their final values after all of the training

patterns are sent through the network. `Mlp` then defines the right "percentage" of correct class *i* to be

$$\frac{100\,a_i^{(r)}}{a_i^{(r)} + a_i^{(w)} + a_i^{(u)}}$$

It shows these values, rounded to integers, in the second progress lines, as the values after the first one. For example, the second progress line above shows that the right "percentages" of correct classes 0 and 1 are 0 and 4.[13]

If **priors** is **allsame** then the pattern-weights are all equal and so $a_i^{(r)}$, etc. are the numbers classified rightly, etc. times this single pattern-weight; the pattern-weight cancels out between the numerator and denominator of the above formula, so that the resulting value really is the percentage of the patterns of class i that the network classified rightly. If priors has a value other than **allsame** (i.e. **class**, **pattern**, or **both**) then the right "percentages" of the classes are not the simple percentages but rather are weighted quantities, which may make more sense than the simple percentages if some patterns should have more impact than others, as indicated by their larger weights.[14]

As for the first value of a second progress line, this is merely the minimum of the right "percentages" of the classes, but shown rounded to the nearest tenth rather than to the nearest integer. This minimum value shows how the network is doing on its "worst" class.[15]

A.1.1.2.2 First progress lines

These are the lines that begin with an integer. The column headings, which pertain to these lines, and the first of these lines in the example, are:

```
Iter   Err (   Ep    Ew)     OK    UNK     NG      OK   UNK    NG
   0 0.474 (0.240 0.289)   6564      0  17736 =  27.0   0.0   73.0 %
```

The values in a first progress line have the following meanings:

Iter: Training iteration number, numbering starting at 0. A first progress line (and second progress line) are produced every **nfreq**'th iteration (set in the specfile).

Err, Ep, Ew: The calculations leading to these values are as follows.

[13] In this case the classes "index numbers" are 0 through 5 and the classes are fingerprint types Arch (A), Left Loop (L), Right Loop (R), Tented Arch (T), Scar (S), and Whorl (W). In this discussion, "class *i*" merely means the class whose index number, number starting at 0, is *i*. Note also that although the software uses class index numbers that start at 0, the class index numbers it writes to **long_outfile** start at 1.

[14] In particular, if the training patterns set is such that the proportions of the patterns belonging to the various classes are not approximately equal to the natural frequencies of the classes, then it may be a good idea to use class-weights (**priors** set to **class**, and class-weights provided in a file) to compensate for the erroneous distribution. See [49].

[15] When `mlp` uses hybrid SCG/LBFGS training rather than only SCG (it does this only if pruning is not specified) it switches from SCG to LBFGS when the minimum reaches or exceeds a specified threshold, **scg_earlystop_pct**.

N = number of patterns

n = number of classes

a_{ij} = activation produced by pattern i at output node j (i.e. class j)

t_{ij} = target value for a_{ij}

$w_i^{(pat)}$ = pattern-weight of pattern i (Page 48)

$E_i^{(pat,mse)} = \sum_{j=0}^{n-1} \left(a_{ij} - t_{ij}\right)^2$, error contribution for pattern i if **errfunc** is **mse**

$E_1^{(mse)} = \frac{1}{2n} \sum_{i=0}^{N-1} w_i^{(pat)} E_i^{(pat,mse)}$

$E_i^{(pat,type1)} = 1 - \frac{1}{1 + \sum_{j \neq k} \exp\left(-\alpha\left(a_{ik} - a_{ij}\right)\right)}$, where k is correct class of pattern i,

error contribution for pattern i if **errfunc** is **type_1** (α is **alpha**)

$E_1^{(type1)} = \frac{1}{n} \sum_{i=0}^{N-1} w_i^{(pat)} E_i^{(pat,type1)}$

$E_i^{(pat,possum)} = \sum_{j=0}^{n-1} \left(10\left|a_{ij} - t_{ij}\right| + 1\right)\left|a_{ij} - t_{ij}\right|$, error contribution for pattern i if

errfunc is **pos_sum**

$E_1^{(type1)} = \frac{1}{n} \sum_{i=0}^{N-1} w_i^{(pat)} E_i^{(pat,possum)}$

E_1 = $E_1^{(mse)}$, $E_1^{(mse)}$, or $E_1^{(mse)}$, according to **errfunc**

Ep = E_1 if **errfunc** is **pos_sum**, $\sqrt{2E_1}$ otherwise

$s^{(wsq)}$ = half of mean squared network weight

Ew = $\sqrt{2s^{(wsq)}}$

E = $E_1 + \textbf{regfac} \times s^{(wsq)}$

Err = $\sqrt{2E}$

`Mlp` prints the **Err**, **Ep** and **Ew** values as defined above. Note that the value `mlp` attempts to minimize is E, but presumably the same effect would be had by attempting to minimize **Err**, since it is an increasing function of E.

OK, UNK, NG, OK, UNK, NG: "Numbers" of patterns **OK** (classified correctly), **UNK**nown (rejected), and wro**NG** or No Good (classified incorrectly), then the corresponding "percentages." `Mlp` calculates these values as follows. It adds up the by-class accumulators $a_i^{(r)}$, $a_i^{(w)}$, and $a_i^{(u)}$ defined earlier to make overall accumulators, where n is the number of classes:

$$a^{(r)} = \sum_{i=0}^{n-1} a_i^{(r)}$$

$$a^{(w)} = \sum_{i=0}^{n-1} a_i^{(w)}$$

$$a^{(u)} = \sum_{i=0}^{n-1} a_i^{(u)}$$

It computes "numbers" right, wrong, and unknown -- the first **OK**, **NG**, and **UNK** values of a first progress line -- as follows, where N is the number of patterns and square brackets denote rounding to an integer:

$$a^{(rwu)} = a^{(r)} + a^{(w)} + a^{(u)}$$

$$n^{(r)} = \left[Na^{(r)} / a^{(rwu)} \right] = \text{"number" right}$$

$$n^{(w)} = \left[Na^{(w)} / a^{(rwu)} \right] = \text{"number" wrong}$$

$$n^{(u)} = N - n^{(r)} - n^{(w)} = \text{"number" unknown}$$

From these "numbers," `mlp` computes corresponding "percentages" -- the second **OK**, **NG**, and **UNK** values -- as follows:

$$p^{(r)} = \left[100\, n^{(r)} / N \right]$$

$$p^{(w)} = \left[100\, n^{(w)} / N \right]$$

$$p^{(u)} = \left[100\, n^{(u)} / N \right]$$

If **priors** is **allsame** then since the pattern-weights are all equal, cancellation of the single pattern-weight occurs between the numerators and denominators of the formulas above for $n^{(r)}$ and $n^{(w)}$, so that they really are the numbers of patterns classified rightly and wrongly. Then it is obvious that $n^{(u)}$ really is the number unknown and that $p^{(r)}$, etc. really are the percentages classified rightly, etc.

A.1.1.2.3 Pruning lines (optional)

These lines, which begin with "pruned," appear if Boltzmann pruning is specified (**boltzmann** set to **abs_prune** or **square_prune** in specfile, and a **temperature** set). The first pruning line of the example is

```
pruned    80     6    86   C  1.67872e+05  H  2.40068e+04  R   85.70  M  -0.00  T  0.0841
```

Regardless of **nfreq**, `mlp` writes a pruning line every time it performs pruning. The first three values of a pruning line are the numbers of network weights that `mlp` pruned (temporarily set to zero) in the first weights layer, in the second layer, and in both layers together. The remaining values announced by the letters **C**, **H**, **R**, and **M**, are calculated as follows (the value announced by **T** actually is not calculated correctly, and should be ignored):

$n^{(wts)}$ = number of network weights (both layers)

$n^{(pruned)}$ = number of weights pruned

$n^{(unpruned)} = n^{(wts)} - n^{(pruned)}$

$w^{(\max)}, w^{(\min)}$ = maximum & minimum absolute values of unpruned weights

$\mathbf{C} = n^{(unpruned)}\left(\left(\log w^{(\max)} - \log w^{(\min)}\right)/\left(\log 2\right) + 1\right)$ = capacity

$s^{(\log abs)}$ = sum of logarithms of absolute values of unpruned weights

$s^{(w12)} = s^{(\log abs)}/\left(\log 2\right) + n^{(unpruned)}\left(1 - \left(\log w^{(\min)}\right)/\left(\log 2\right)\right)$

$\mathbf{H} = \mathbf{C} - s^{(w12)}$ = entropy

$\mathbf{R} = 100 \times s^{(w12)}\, s^{(w12)}/\mathbf{C}$

M = mean of unpruned weights

A.1.1.3 Confusion Matrices and Miscellaneous Information (Optional)

If **do_confuse** is set to true in the specfile, the next part of the output consists of two "confusion matrices" and some miscellaneous information:

```
oklvl 0.00
 # Highest two outputs (mean) 0.784 0.145; mean diff 0.639
   key  name
    A   A
    L   L
    R   R
    S   S
    T   T
    W   W
 #  key:      A   L   R   S   T   W
 #  row: correct, column: actual
 #       A: 333 315 267   0   0   9
 #       L:  12 7522  86   0   0 144
 #       R:  21 148 7128   0   0 153
 #       S:   0   0   0   0   0   0
 #       T:  60 346 323   0   0   3
 #       W:   2 798 509   0   0 5985
 #  unknown
 #    *    0   0   0   0   0   0

 percent of true IDs correctly identified (rows)
          36  97  96   0   0  82
percent of predicted IDs correctly identified (cols)
          78  82  86   0   0  95

 #  mean highest activation level
 #  row: correct, column: actual
 #  key:      A   L   R   S   T   W
 #       A:  35  43  43   0   0  38
 #       L:  32  83  41   0   0  48
 #       R:  32  43  83   0   0  49
 #       S:  88 4666 4042   0   0 317
 #       T:  33  49  48   0   0  38
 #       W:  29  61  58   0   0  85

 #  unknown
 #    *    0   0   0   0   0   0

 Histogram of errors, from 2^(-10) to 1
  15899   5322  10477  14278  15596  22398  16728  16005  13376   9364   6357
   10.9    3.7    7.2    9.8   10.7   15.4   11.5   11.0    9.2    6.4    4.4%
```

The first line of this optional section of the output shows the value of the rejection threshold **oklvl** set in the specfile (this was already shown in the header). The next line shows the mean values, over the training patterns as sent through the network at the end of training, of the highest and second-highest output node values, and the mean difference of these values. Next is a table showing the short classname ("key") and long classname ("name") of each class. In this example the keys and names are the same, but in general the names can be quite long whereas the keys must be no longer than two characters in length: the short keys are used to label the confusion matrices.

Next are the confusion matrices of "numbers" and of "mean highest activation level." `Mlp` has the following accumulators:

$a_{ij}^{(patwts)}$ = pattern-weight accumulator for correct class *i* and hypothetical class *j*

$a_{ij}^{(highac)}$ = high-activation accumulator for correct class *i* and hypothetical class *j*

$a_{ij}^{(highac,u)}$ = high-activation unknown accumulator for correct class *i*

If a pattern sent through the network produces a highest activation that meets or exceeds **oklvl** (so that `mlp` accepts its result for this pattern), then `mlp` adds its pattern-weight to $a_{ij}^{(patwts)}$ and adds the highest activation to $a_{ij}^{(highac)}$; where *i* and *j* are the correct class and hypothetical class of the pattern. Otherwise, i.e. if `mlp` finds the pattern to be unknown (rejects the result), it adds its pattern-weight to $a_{ij}^{(u)}$ (Section A.1.1.2.1) and adds the highest activation to $a_{ij}^{(highac,u)}$, where *i* is the correct class of the pattern. After it has processed all the patterns, `mlp` calculates the confusion matrix of "numbers" and its "unknown" line; some additional information concerning the rows and columns of that matrix; and the confusion matrix of "mean highest activation level" and its "unknown" line, as follows.

First define some notation:

$N_i^{(pat)}$	=	number of patterns of correct class i
$n_{ij}^{(confuse)}$	=	value in row i and column j of first confusion matrix (of "numbers")
$n_i^{(confuse,u)}$	=	i^{th} value of "unknown" line at bottom of first confusion matrix
$p_i^{(r,row)}$	=	i^{th} value of "percent of true IDs correctly identified (rows)" line
$p_j^{(r,col)}$	=	j^{th} value of "percent of predicted IDs correctly identified (cols)" line
$h_{ij}^{(confuse)}$	=	value in row i and column j of second confusion matrix
$h_i^{(confuse,u)}$	=	j^{th} value of "unknown" line at bottom of second confusion matrix

`Mlp` calculates the values as follows, where $a_i^{(r)}$, $a_i^{(w)}$, $a_i^{(u)}$ and are as defined in Section A.1.1.2.1 and square brackets again denote rounding to an integer:[16]

$$n_{ij}^{(confuse)} = \left[\frac{N_i^{(pats)} a_{ij}^{(patwts)}}{a_i^{(u)} + \sum_{j=0}^{n-1} a_{ij}^{(patwts)}} \right]$$

$$n_i^{(confuse,u)} = \left[\frac{N_i^{(pats)} a_i^{(u)}}{a_i^{(r)} + a_i^{(w)} + a_i^{(u)}} \right]$$

$$p_i^{(r,row)} = \left[\frac{100\, n_{ii}^{(confuse)}}{N_i^{(pats)} - n_i^{(confuse,u)}} \right]$$

$$p_j^{(r,col)} = \left[\frac{100\, n_{jj}^{(confuse)}}{\sum_{i=0}^{n-1} n_{ij}^{(confuse)}} \right]$$

$$h_{ij}^{(confuse)} = \left[\frac{100\, a_{ij}^{(highac)}}{n_{ij}^{(confuse)}} \right]$$

$$h_i^{(confuse,u)} = \left[\frac{100\, a_i^{(highac,u)}}{n_i^{(confuse,u)}} \right]$$

If **priors** is **allsame**, the pattern-weights are all equal, and cancellation of the single pattern-weight between numerator and denominator causes $n_{ij}^{(confuse)}$ above to be the number of patterns of correct class i and hypothetical class j; similarly, $n_i^{(confuse,u)}$ really is the number of patterns of

[16] The denominators of the expression shown here for $n_{ij}^{(confuse)}$ and $n_i^{(confuse,u)}$ are equal, but these expressions show what the software actually calculates.

correct class *i* that were unknown; $p_i^{(r,row)}$ and $p_j^{(r,col)}$ really are the percentages that the on-diagonal (correctly classified) numbers in the matrix comprise of their rows and columns respectively; $h_{ij}^{(confuse)}$ really is the mean highest activation level (multiplied by 100 and rounded to an integer) of the patterns of correct class *i* and hypothetical class *j*; and $h_i^{(confuse,u)}$ really is the mean highest activation level of the patterns of correct class *i* that were unknown. If **priors** has one of its other values, the printed values are weighted versions of these quantities.

The final part of this optional section of the output is a histogram of errors. This pertains to the absolute errors between output activations and target activations, across all output nodes (6 nodes in this example) and all training patterns (24,300 patterns in this example), when the patterns are sent through the trained network. Of the resulting set of absolute error values (243,000 values in this example), this histogram shows the number (first line) and percentage (second line) of these values that fall into each of the 11 intervals $(-\infty, 2^{-10}]$, $(2^{-10}, 2^{-9}]$, …, $(2^{-1}, 1]$.

A.1.1.4 Final Progress Lines

The next part of the output consists of a repeat of the column-headers line, final first-progress-line, and final second-progress-line of the training progress part of the output, but with an **F** prepended to the final first-progress-line:

```
    Iter   Err (   Ep    Ew)     OK    UNK     NG      OK   UNK    NG
 F    50 0.098 (0.081 0.040)  21211      0   3089 =  87.3   0.0   12.7 %
  0.0   36 97 96  0  0 82
```

A.1.1.5 Correct-vs.-Rejected Table (Optional)

If **do_cvr** is set to **true** in the specfile, the next part of the output is a correct-vs.-rejected table; the first and last few lines of this table, from the example output, are:

```
          thresh     right   unknown     wrong   correct  rejected
 1tr    0.000000     21211         0      3089     87.29      0.00
 2tr    0.050000     21211         0      3089     87.29      0.00
 3tr    0.100000     21211         0      3089     87.29      0.00
 4tr    0.150000     21211         0      3089     87.29      0.00
 5tr    0.200000     21211         0      3089     87.29      0.00

      ...

48tr    0.975000      3777     20521         2     99.95     84.45
49tr    0.980000      3230     21068         2     99.94     86.70
50tr    0.985000      2691     21607         2     99.93     88.92
51tr    0.990000      2141     22158         1     99.95     91.19
52tr    0.995000      1509     22791         0    100.00     93.79
```

`Mlp` produces these table values as follows. It has a fixed array of rejection-threshold values, which have been set in an unequally-spaced pattern that works well, and it uses three pattern-

t_k = k^{th} threshold

$a_k^{(cvr,r)}$ = right pattern-weight-accumulator for k^{th} threshold

$a_k^{(cvr,w)}$ = wrong pattern-weight-accumulator for k^{th} threshold

$a_k^{(cvr,u)}$ = unknown pattern-weight-accumulator for k^{th} threshold

weight-accumulators for each threshold:

As `mlp` sends each pattern through the finished network,[17] it loops over the thresholds t_k: for each k, it compares the highest network activation produced for the pattern with t_k to decide whether the pattern would be accepted or rejected. If accepted, it adds the pattern-weight of that pattern either to $a_k^{(cvr,r)}$ or to $a_k^{(cvr,w)}$ according to whether it classified the pattern rightly or

$$a^{(cvr,rwu)} = a_k^{(cvr,r)} + a_k^{(cvr,w)} + a_k^{(cvr,u)}$$

$$n^{(cvr,r)} = \left[N a_k^{(cvr,r)} \Big/ a_k^{(cvr,rwu)} \right] = \text{“number right”}$$

$$n^{(cvr,w)} = \left[N a_k^{(cvr,w)} \Big/ a_k^{(cvr,rwu)} \right] = \text{“number wrong”}$$

$$n^{(cvr,u)} = N - n^{(cvr,r)} - n^{(cvr,w)} = \text{“number unknown” (rejected)}$$

$$p^{(cvr,corr)} = 100\, n^{(cvr,r)} \Big/ \left(n^{(cvr,r)} + n^{(cvr,w)} \right) = \text{“percentage correct”}$$

$$p^{(cvr,rej)} = 100\, n^{(cvr,u)} \Big/ N = \text{“percentage rejected”}$$

wrongly; if rejected, it adds the pattern-weight to $a_k^{(cvr,u)}$. After all the patterns have been through the network, `mlp` finishes the table as follows. For each threshold t_k it calculates the following values:

`Mlp` then writes a line of the table. The values of the line are the threshold index k plus 1 with "tr"[18] appended, t_k ("thresh"), $n^{(cvr,r)}$ ("right"), $n^{(cvr,u)}$ ("unknown"), $n^{(cvr,w)}$ ("wrong"), $p^{(cvr,corr)}$ ("correct"), and $p^{(cvr,rej)}$ ("rejected"). If **priors** is **allsame** then, since all pattern-weights are the same, cancellation of the single pattern-weight occurs between numerator and denominator in the above expressions for $n^{(cvr,r)}$ and $n^{(cvr,w)}$, so they really are the number of patterns classified rightly and wrongly if threshold t_k is used. Also, it is obvious that $n^{(cvr,u)}$ really is the number of patterns unknown for this threshold, $p^{(cvr,corr)}$ really is the percentage of the patterns accepted at this threshold that were classified correctly, and $p^{(cvr,rej)}$ really is the percentage of the N patterns that were rejected at this threshold. If **priors** has one of its other values, then the tabulated values are weighted versions of these quantities.

[17] If **do_cvr** is **true** then `mlp` calculates a correct vs. reject table, but only for the final state of the network in the training run.

[18] for “training”; the correct vs. reject table for a test run uses “ts”

A.1.1.6 Final Information

The final part of the output shows miscellaneous information:

```
Iter 50; ierr 1 : iteration limit
 Used 51 iterations; 154 function calls; Err 0.098; |g|/|w| 1.603e-04
 Rms change in weights  0.289

 User+system time used:  3607.3 (s)  1:00:07.3 (h:m:s)
 Wrote weights as file trn1.wts
```

The first line here shows what iteration the training run ended on, and the value and meaning of the return code *ierr*, which indicates why `mlp` stopped its training run: in the example, the specified maximum number of iterations (**niter_max**), 50, had been used. This training run was actually the first run of a sequence; its initial network weights were random, but each subsequent run used the final weights of the preceding run as its initial weights. The only parameter varied from one run to the next was the regularization factor **regfac**, which was decreased at each step: successive regularization. Each run was limited to 50 iterations, and it was assumed that this small iteration limit would be reached before any of the other stopping conditions were satisfied. When sinusoid activation functions are used, as in this case, best training requires that successive regularization be used. If sigmoid functions are used, it is just as well to do only one training run, and in that case one should probably set the iteration limit to a large number so that training will be stopped by one of the other conditions, such as an error goal (**egoal**).

The next line shows: how many iterations `mlp` used (counting the 0'th iteration; yes, this is stupid after it already said what iteration it stopped on); how many calls of the error function it made; the final error value; and the final size of the error gradient vector (square root of sum of squares), normalized by dividing it by the final size of the weights. The next line shows the root-mean-square of the change in weights, between their initial values and their final values. The next line shows the combined user and system time used by the training run.[19] The final line merely reports the name of the file to which `mlp` wrote the final weights.

[19] Setting the initial network weights, reading the patterns file, and other (minor) setup work, are not timed.

APPENDIX B. REQUEST NBIS EXPORT CONTROL SOURCE CODE CD-ROM

NFSEG and BOZORTH3 are software packages that are subject to U.S. export control laws. It is our understanding that this software falls within ECCN 3D980, which covers software associated with the development, production or use of certain equipment controlled in accordance with U.S. concerns about crime control practices in specific countries. Therefore, the source code, documentation, and testing utilities for NFSEG and BOZORTH3 are only distributed via CD-ROM to qualifying requesters.

To request a CD-ROM copy of these export controlled packages, please send email to nbis_ec@nist.gov with the following information:

Email Header -

Subject : NBIS EXPORT CONTROL SOURCE CODE CD-ROM

Email Text Body -

Name : (Your Name)
Organization : (Organization Name or Affiliation)
Mailing Address : (Complete Postal Mailing Address)
Phone Number : (Working Phone Number)

For new releases information, please visit http://fingerprint.nist.gov/NBIS/index.html.

APPENDIX C. REFERENCE MANUAL

This appendix contains manual pages describing the invocation and use of the utilities provided in this software distribution. The utilities are listed in alphabetical order. Those belonging to the NFSEG package are designated as belonging to command set (1A), PCASYS as (1B), MINDTCT as (1C), NFIQ as (1D), BOZORTH3 as (1E), AN2K as (1F), IMGTOOLS as (1G), and IJG utilities are designated as (1H).

The manual pages listed in this section are in directory /NBIS/Main/man. To view a man, type:

```
% man -M <install_dir>/man <executable>
```

where the text <install_dir> is replaced by your specific installation directory path and <executable> is replaced by the name of the utility of interest.

NAME

an2k2iaf – Modifies minutiae and fingerprint image records in an ANSI/NIST 2000 file to meet FBI/IAFIS specifications.

SYNOPSIS

an2k2iaf <*file in*> <*file out*>

DESCRIPTION

An2k2iaf parses a standard compliant ANSI/NIST-ITL 1-2000 file and, if necessary, converts specific records and fields to meet FBI/IAFIS specifications. Please note that this utility does not exhaustively validate the output to ensure compliant FBI/IAFIS transactions, rather it focuses on the format of minutiae and image records.

Minutiae fields:
When a Type-9 record is encountered in the input file, this utility checks to see which fields are populated. If the NIST-assigned fields 5-12 are populated, but the FBI/IAFIS-assigned fields 13-23 are empty, then the FBI/IAFIS fields are populated by translating the data recorded in the NIST fields, and the NIST fields are removed.

Image records:
FBI/IAFIS specifications (EFTS V7) require binary field images; therefore, this utility looks for tagged field fingerprint records and converts them appropriately. If a Type-13 or Type-14 record is encountered, it is inspected to determine if the image is bi-level or grayscale and to see what scan resolution and image compression was used. Records containing bi-level images scanned at 19.69 ppmm (500 ppi) and either WSQ-compressed or uncompressed are converted to Type-6 records; records containing grayscale images scanned at 19.69 ppmm and either WSQ-compressed or uncompressed are converted to Type-4 records; otherwise, the tagged field image record is ignored.

OPTIONS

<*file in*>
: the ANSI/NIST file to be converted

<*file out*>
: the resulting ANSI/NIST file

EXAMPLES

From *test/an2k/execs/an2k2iaf/an2k2iaf.src*:

```
% an2k2iaf ../../data/nist.an2 iafis.an2
```

SEE ALSO

iaf2an2k(1F), **mindtct**(1C)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

an2k2txt – Converts an ANSI/NIST 2000 file to a formatted text file.

SYNOPSIS

an2k2txt <*ansi_nist in*> <*fmttext out*>

DESCRIPTION

An2k2txt parses a standard compliant ANSI/NIST-ITL 1-2000 file and writes its contents to a new file in a textually viewable and editable format. The contents of binary image fields are stored to temporary files and externally referenced in the output file.

OPTIONS

<*ansi_nist in*>
the ANSI/NIST file to be converted

<*fmttext out*>
the output text file

OUTPUT FORMAT

Every line in the output text represents a single information item from the ANSI/NIST file. These lines are formatted as follows:

r.f.s.i [*t.n*]=*value{US}*

r.f.s.i references the information item with

- *r* the item's positional *r*ecord index in the file
- *f* the item's positional *f*ield index in the record
- *s* the item's positional *s*ubfield index in the field
- *i* the item's positional *i*tem index in the subfield

Note that all indices start at 1.

t.n references the Record Type and Field ID from the standard.

- *t* the record's *t*ype
- *n* the field's ID *n*umber

value is the textual content of the information item, unless the information item contains binary image data, in which case, the value is the name of an external file containing the binary data.

{US} is the non-printable ASCII character 0x1F. This separator character is one of 4 used in the standard. In VIM, this non-printable character may be entered using the ^v command and entering the decimal code "31". In Emacs, this non-printable character may be entered using the ^q command and entering the octal code "037".

Example Output Lines

1.5.1.1 [1.005]=19990708•

This is the information item corresponding to the Date (DAT) field in the standard. It is the 5th field in a Type-1 record, and the Type-1 record is always the first record in the ANSI/NIST file; therefore, its record index is 1, its field index is 5, its subfield index is 1, and its item index is 1. The value of this information item represents the date of July 8, 1999. The '•' at the end of the line represents the non-printable {US} character.

1.3.4.1 [1.003]=14•

This information item is part of the File Content (CNT) field. The CNT field is the 3rd field in a

Type-1 record, so this information item's record index is 1 and its field index is 3. This information item is in the 4th subfield of the CNT field, and has an item index of 1; therefore, its value 14 signifies that the 4th record (the subfield index) in this ANSI/NIST file is a Type-14 record.

4.14.1.1 [14.999]=fld_2_14.tmp•

This information item corresponds to an Image Data field of a Type-14 record. This field always has numeric ID 999 and is always the last field in the image record. This Type-14 record is the 4th record in this ANSI/NIST file, so the Image Data information item has record index 4, and it is in the 14th field (field index 14) in the record. This information item in the ANSI/NIST file contains binary pixel data, so the output value "fld_2_14.tmp" references an external filename into which **an2k2txt** stored the item's binary data.

EXAMPLES

From *test/an2k/execs/an2k2txt/an2k2txt.src*:

% an2k2txt ../../data/nist.an2 nist.fmt

SEE ALSO

an2ktool(1F), **txt2an2k**(1F)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

an2ktool – Parses, manipulates, and/or writes the results to an ANSI/NIST 2000 file in batch mode.

SYNOPSIS

an2ktool <*option*>
-print <all|*r*[.*f*[.*s*[.*i*]]]> <*file in*> [*file out*]
-delete *r*[.*f*[.*s*[.*i*]]] <*file in*> [*file out*]
-substitute *r.f.s.i* <new value> <*file in*> [*file out*]
-substitute *r*[.*f*[.*s*]] <*fmttext file*> <*file in*> [*file out*]
-insert *r.f.s.i* <new value> <*file in*> [*file out*]
-insert *r*[.*f*[.*s*]] <*fmttext file*> <*file in*> [*file out*]

DESCRIPTION

An2ktool parses a standard compliant ANSI/NIST-ITL 1-2000 file, manipulates its contents, and writes the results back out. Batch operations may be conducted at the level of record, field, subfield, or information item. Possible operations include printing, deleting, substituting, or inserting data.

OPTIONS

All switch names may be abbreviated; for example, **-print** may be written **-p**.

PRINT OPTION

-print <all|*r*[.*f*[.*s*[.*i*]]]> <*file in*> [*file out*]

Prints the contents of the specified structure {file, record, field, subfield, or information item} to either the specified output file or to standard output.

Option settings:

- all — The entire contents of the input file is printed. Any binary image fields in the file are stored to temporary files, and their file names are externally referenced in the printed output. This option setting is equivalent to running **an2k2txt** on the input file.
- *r* — The contents of the record at position *r* in the file is printed.
- *r.f* — The contents of the field at position *f* within record *r* is printed.
- *r.f.s* — The contents of the subfield at position *s* within field *f* within record *r* is printed.
- *r.f.s.i* — The contents of the information item at position *i* within subfield *s* within field *f* within record *r* is printed.

<*file in*>
The ANSI/NIST file whose content is to be printed.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

DELETE OPTION

-delete *r*[.*f*[.*s*[.*i*]]] <*file in*> [*file out*]

Deletes the contents of the specified structure {record, field, subfield, or information item} from the ANSI/NIST file, writing the results to either the specified output file or to standard output.

Option settings:

- *r* — The contents of the record at position *r* in the file is deleted.
- *r.f* — The contents of the field at position *f* within record *r* is deleted.

r.f.s The contents of the subfield at position *s* within field *f* within record *r* is deleted.

r.f.s.i The contents of the information item at position *i* within subfield *s* within field *f* within record *r* is deleted.

<file in>
The ANSI/NIST file whose content is to be modified.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

SUBSTITUTE OPTION 1

-substitute *r.f.s.i* <new value> *<file in>* [*file out*]

Substitutes the contents of the specified information item in an ANSI/NIST file with the string value provided on the command line, writing the results to either the specified output file or to standard output.

r.f.s.i The position indices of the information item to be substituted.

<new value>
The new string value.

<file in>
The ANSI/NIST file whose content is to be modified.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

SUBSTITUTE OPTION 2

-substitute *r*[.*f*[.*s*]] *<fmttext file> <file in>* [*file out*]

Substitutes the contents of the specified structure {record, field, or subfield} in an ANSI/NIST file with the contents of a formatted text file consistent in format to those produced by **an2k2txt**. The results are written to either the specified output file or to standard output.

Option settings:

r The contents of the record at position *r* in the file is substituted.

r.f The contents of the field at position *f* within record *r* is substituted.

r.f.s The contents of the subfield at position *s* within field *f* within record *r* is substituted.

r.f.s.i The contents of the information item at position *i* within subfield *s* within field *f* within record *r* is substituted.

<fmttext file>
The formatted text file containing the new values.

<file in>
The ANSI/NIST file whose content is to be modified.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

INSERT OPTION 1

-insert *r.f.s.i* <new value> *<file in>* [*file out*]

Inserts an information item at the specified position within an ANSI/NIST file, assigning the new item the string value provided on the command line. The results are written to either the specified output file or to standard output.

r.f.s.i The position indices where the new information item is to be inserted.

<new value>
The new information item's string value.

<file in>
The ANSI/NIST file whose content is to be modified.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

INSERT OPTION 2

-insert *r*[.*f*[.*s*]] *<fmttext file> <file in>* [*file out*]

Inserts a structure {record, field, or subfield} at the specified position within an ANSI/NIST file. The new structure is assigned the contents of a formatted text file consistent in format to those produced by **an2k2txt**. The results are written to either the specified output file or to standard output.

Option settings:

r A record at position *r* is inserted.

r.f A field at position *f* within record *r* is inserted.

r.f.s A subfield at position *s* within field *f* within record *r* is inserted.

r.f.s.i An information item at position *i* within subfield *s* within field *f* within record *r* is inserted.

<fmttext file>
The formatted text file containing the new values.

<file in>
The ANSI/NIST file whose content is to be modified.

[*file out*]
The optional output file. If omitted, results are printed to standard output.

EXAMPLES

From *test/an2k/execs/an2ktool/an2ktool.src*:

% an2ktool -d 2.12.1.4 ../../data/nist.an2 delete.an2
deletes the information item recording the first minutia's type.

% an2ktool -i 2.12.1.4 A delete.an2 insert.an2
inserts an information item setting the first minutia's type to "A".

% an2ktool -s 2.12.1.4 A ../../data/nist.an2 subitem.an2
replaces the information item recording the first minutia's type with the value "A".

% an2ktool -s 2.12.1 subfld.fmt ../../data/nist.an2 subfld.an2
replaces the subfield containing all the attributes related to the first minutia with the contents of the formatted text file *subfld.fmt*.

SEE ALSO

an2k2iaf(1F), **an2k2txt**(1F), **dpyan2k**(1F), **iaf2an2k**(1F), **txt2an2k**(1F)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

asc2bin – converts a PCASYS data file from ascii to binary form.

SYNOPSIS

asc2bin <*ascii_data_in*> <*binary_data_out*>

DESCRIPTION

Asc2bin reads a PCASYS ascii data file of any type, or the standard input, and produces a corresponding PCASYS binary data file.

OPTIONS

<*ascii_data_in*>

Ascii data file to be read.

<*binary_data_out*>

Binary data file to be written. If this file already exists, asc2bin overwrites it.

EXAMPLE(S)

From *test/pcasys/execs/asc2bin/asc2bin.src*:

% asc2bin ../../data/oas/fv1.cls fv1.bin

Converts the class file from ascii to binary data.

SEE ALSO

bin2asc (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

bin2asc – converts a PCASYS data file from binary to ascii form.

SYNOPSIS

bin2asc <*binary_data_in*> <*ascii_data_out*>

DESCRIPTION

Bin2asc reads a PCASYS binary data file of any type, and produces a corresponding PCASYS ascii data file or writes the ascii data to the standard output.

OPTIONS

<*binary_data_in*>

Binary data file to be read.

<*ascii_data_out*>

Ascii data file to be written. If the file already exists, bin2asc overwrites the file.

EXAMPLE(S)

From *test/pcasys/execs/bin2asc/bin2asc.src*:

% bin2asc ../asc2bin/fv1.bin fv1.cls

Converts the class file from binary to ascii data.

SEE ALSO

asc2bin (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

chgdesc – changes the description string of a PCASYS data file.

SYNOPSIS

chgdesc <*datafile*> <*new_desc*>

DESCRIPTION

Chgdesc replaces the existing description string of a PCASYS data file with the provided string.

The new description can be of any length, but it must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To delete a file's description string, i.e. to replace it with the empty string, use '' (quoted empty string) as the string argument.

The command makes, and then removes, a temporary version of the data file, in the same directory as the original file; its filename is the original filename with _chgdesc<pid> appended, where <pid> is the process id of the chgdesc command instance. (In the unlikely event that a file of this name already exists, chgdesc exits with an error message, rather than replacing that file.) If the original data file is large, then make sure, before running chgdesc, that the disk partition where the original file lives has enough room for the temporary copy.

OPTIONS

<*datafile*>
: Data file whose description field is to be changed.

<*new_desc*>
: The new description string.

EXAMPLE(S)

From *test/pcasys/execs/chgdesc/chgdesc.src*:

% chgdesc fv1.acl 'fv1.cls'
Puts the string fv1.cls in the description field of the file fv1.acl.

SEE ALSO

datainfo (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

cjpeg – compress an image file to a JPEG file

SYNOPSIS

cjpeg [*options*] [*filename*]

DESCRIPTION

cjpeg compresses the named image file, or the standard input if no file is named, and produces a JPEG/JFIF file on the standard output. The currently supported input file formats are: PPM (PBMPLUS color format), PGM (PBMPLUS gray-scale format), BMP, Targa, and RLE (Utah Raster Toolkit format). (RLE is supported only if the URT library is available.)

OPTIONS

All switch names may be abbreviated; for example, **–grayscale** may be written **–gray** or **–gr**. Most of the "basic" switches can be abbreviated to as little as one letter. Upper and lower case are equivalent (thus **–BMP** is the same as **–bmp**). British spellings are also accepted (e.g., **–greyscale**), though for brevity these are not mentioned below.

The basic switches are:

–quality *N*
: Scale quantization tables to adjust image quality. Quality is 0 (worst) to 100 (best); default is 75. (See below for more info.)

–grayscale
: Create monochrome JPEG file from color input. Be sure to use this switch when compressing a grayscale BMP file, because **cjpeg** isn't bright enough to notice whether a BMP file uses only shades of gray. By saying **–grayscale**, you'll get a smaller JPEG file that takes less time to process.

–optimize
: Perform optimization of entropy encoding parameters. Without this, default encoding parameters are used. **–optimize** usually makes the JPEG file a little smaller, but **cjpeg** runs somewhat slower and needs much more memory. Image quality and speed of decompression are unaffected by **–optimize**.

–progressive
: Create progressive JPEG file (see below).

–targa Input file is Targa format. Targa files that contain an "identification" field will not be automatically recognized by **cjpeg**; for such files you must specify **–targa** to make **cjpeg** treat the input as Targa format. For most Targa files, you won't need this switch.

The **–quality** switch lets you trade off compressed file size against quality of the reconstructed image: the higher the quality setting, the larger the JPEG file, and the closer the output image will be to the original input. Normally you want to use the lowest quality setting (smallest file) that decompresses into something visually indistinguishable from the original image. For this purpose the quality setting should be between 50 and 95; the default of 75 is often about right. If you see defects at **–quality** 75, then go up 5 or 10 counts at a time until you are happy with the output image. (The optimal setting will vary from one image to another.)

–quality 100 will generate a quantization table of all 1's, minimizing loss in the quantization step (but there is still information loss in subsampling, as well as roundoff error). This setting is mainly of interest for experimental purposes. Quality values above about 95 are **not** recommended for normal use; the compressed file size goes up dramatically for hardly any gain in output image quality.

In the other direction, quality values below 50 will produce very small files of low image quality. Settings around 5 to 10 might be useful in preparing an index of a large image library, for example. Try **–quality** 2 (or so) for some amusing Cubist effects. (Note: quality values below about 25 generate 2-byte quantization tables, which are considered optional in the JPEG standard. **cjpeg** emits a warning message when you give such a quality value, because some other JPEG programs may be unable to decode the resulting file. Use

−baseline if you need to ensure compatibility at low quality values.)

The **−progressive** switch creates a "progressive JPEG" file. In this type of JPEG file, the data is stored in multiple scans of increasing quality. If the file is being transmitted over a slow communications link, the decoder can use the first scan to display a low-quality image very quickly, and can then improve the display with each subsequent scan. The final image is exactly equivalent to a standard JPEG file of the same quality setting, and the total file size is about the same --- often a little smaller. **Caution:** progressive JPEG is not yet widely implemented, so many decoders will be unable to view a progressive JPEG file at all.

Switches for advanced users:

−dct int
Use integer DCT method (default).

−dct fast
Use fast integer DCT (less accurate).

−dct float
Use floating-point DCT method. The float method is very slightly more accurate than the int method, but is much slower unless your machine has very fast floating-point hardware. Also note that results of the floating-point method may vary slightly across machines, while the integer methods should give the same results everywhere. The fast integer method is much less accurate than the other two.

−restart *N*
Emit a JPEG restart marker every N MCU rows, or every N MCU blocks if "B" is attached to the number. **−restart 0** (the default) means no restart markers.

−smooth *N*
Smooth the input image to eliminate dithering noise. N, ranging from 1 to 100, indicates the strength of smoothing. 0 (the default) means no smoothing.

−maxmemory *N*
Set limit for amount of memory to use in processing large images. Value is in thousands of bytes, or millions of bytes if "M" is attached to the number. For example, **−max 4m** selects 4000000 bytes. If more space is needed, temporary files will be used.

−outfile *name*
Send output image to the named file, not to standard output.

−verbose
Enable debug printout. More **−v**'s give more output. Also, version information is printed at startup.

−debug
Same as **−verbose**.

The **−restart** option inserts extra markers that allow a JPEG decoder to resynchronize after a transmission error. Without restart markers, any damage to a compressed file will usually ruin the image from the point of the error to the end of the image; with restart markers, the damage is usually confined to the portion of the image up to the next restart marker. Of course, the restart markers occupy extra space. We recommend **−restart 1** for images that will be transmitted across unreliable networks such as Usenet.

The **−smooth** option filters the input to eliminate fine-scale noise. This is often useful when converting dithered images to JPEG: a moderate smoothing factor of 10 to 50 gets rid of dithering patterns in the input file, resulting in a smaller JPEG file and a better-looking image. Too large a smoothing factor will visibly blur the image, however.

Switches for wizards:

−baseline
Force baseline-compatible quantization tables to be generated. This clamps quantization values to 8 bits even at low quality settings. (This switch is poorly named, since it does not ensure that the

output is actually baseline JPEG. For example, you can use **–baseline** and **–progressive** together.)

–qtables *file*
Use the quantization tables given in the specified text file.

–qslots *N[,...]*
Select which quantization table to use for each color component.

–sample *HxV[,...]*
Set JPEG sampling factors for each color component.

–scans *file*
Use the scan script given in the specified text file.

The "wizard" switches are intended for experimentation with JPEG. If you don't know what you are doing, **don't use them**. These switches are documented further in the file wizard.doc.

EXAMPLES

This example compresses the PPM file foo.ppm with a quality factor of 60 and saves the output as foo.jpg:

cjpeg –quality *60 foo.ppm* **>** *foo.jpg*

HINTS

Color GIF files are not the ideal input for JPEG; JPEG is really intended for compressing full-color (24-bit) images. In particular, don't try to convert cartoons, line drawings, and other images that have only a few distinct colors. GIF works great on these, JPEG does not. If you want to convert a GIF to JPEG, you should experiment with **cjpeg**'s **–quality** and **–smooth** options to get a satisfactory conversion. **–smooth 10** or so is often helpful.

Avoid running an image through a series of JPEG compression/decompression cycles. Image quality loss will accumulate; after ten or so cycles the image may be noticeably worse than it was after one cycle. It's best to use a lossless format while manipulating an image, then convert to JPEG format when you are ready to file the image away.

The **–optimize** option to **cjpeg** is worth using when you are making a "final" version for posting or archiving. It's also a win when you are using low quality settings to make very small JPEG files; the percentage improvement is often a lot more than it is on larger files. (At present, **–optimize** mode is always selected when generating progressive JPEG files.)

ENVIRONMENT

JPEGMEM
If this environment variable is set, its value is the default memory limit. The value is specified as described for the **–maxmemory** switch. **JPEGMEM** overrides the default value specified when the program was compiled, and itself is overridden by an explicit **–maxmemory**.

SEE ALSO

djpeg(1H), **jpegtran**(1H), **rdjpgcom**(1H), **wrjpgcom**(1H)
ppm(5), **pgm**(5)
Wallace, Gregory K. "The JPEG Still Picture Compression Standard", Communications of the ACM, April 1991 (vol. 34, no. 4), pp. 30-44.

AUTHOR

Independent JPEG Group

BUGS

Arithmetic coding is not supported for legal reasons.

GIF input files are no longer supported, to avoid the Unisys LZW patent. Use a Unisys-licensed program if you need to read a GIF file. (Conversion of GIF files to JPEG is usually a bad idea anyway.)

Not all variants of BMP and Targa file formats are supported.

The **–targa** switch is not a bug, it's a feature. (It would be a bug if the Targa format designers had not been clueless.)

Still not as fast as we'd like.

NAME

cjpegb – compresses a grayscale or color (RGB) image using *lossy* Baseline JPEG (JPEGB).

SYNOPSIS

cjpegb *<q=20=95> <outext> <image file>*
[**-raw_in** *w*,*h*,*d*,[*ppi*]
[**-nonintrlv**]]
[*comment file*]

DESCRIPTION

Cjpegb takes as input a file containing an uncompressed grayscale or color (RGB) image. Two possible input file formats are accepted, NIST IHead files and raw pixmap files. If a raw pixmap file is to be compressed, then its image attributes must be provided on the command line as well. Once read into memory, the grayscale or color pixmap is then *lossy* compressed to a specified level of reconstruction quality using the Independent JPEG Group's (IJG) library for Baseline JPEG (JPEGB). The JPEGB results are then written to an output file.

Note that **cjpegb** calls the IJG library in a default color mode where one of the compression steps includes a colorspace conversion from RGB to YCbCr, and then the Cb & Cr component planes are downsampled by a factor of 2 in both dimensions. Due to this colorspace conversion, **cjpegb** should only be used to compress RGB color images.

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. By default, **cjpegb** assumes interleaved color pixels. (See INTERLEAVE OPTIONS below.) Regarding color pixmaps, the NIST IHead file format only supports interleaved RGB images.

OPTIONS

All switch names may be abbreviated; for example, **-raw_in** may be written **-r**.

<q=20-95>
specifies the level of quality in the reconstructed image as a result of lossy compression. The integer quality value may range between 20 and 95. The lower the quality value, the more drastic the compression.

<outext>
the extension of the compressed output file. To construct the output filename, **cjpegb** takes the input filename and replaces its extension with the one specified here.

<image file>
the input file, either an IHead file or raw pixmap file, containing the grayscale or color (RGB) image to be compressed.

-raw_in *w*,*h*,*d*,[*ppi*]
the attributes of the input image. This option must be included on the command line if the input is a raw pixmap file.

- *w* the pixel width of the pixmap
- *h* the pixel height of the pixmap
- *d* the pixel depth of the pixmap
- *ppi* the optional scan resolution of the image in integer units of pixels per inch.

-nonintrlv
specifies that the color components in an *input* raw pixmap file image are non-interleaved and stored in separate component planes. (See INTERLEAVE OPTIONS below).

comment file
an optional user-supplied ASCII comment file. (See COMMENT OPTIONS below.)

INTERLEAVE OPTIONS

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. Color components are interleaved when a pixel's (R)ed, (G)reen, and (B)lue components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **cjpegb** assumes interleaved color components, and note that all color IHead images must be interleaved.

COMMENT OPTIONS

Upon successful compression, this utility generates and inserts in the compressed output file a specially formatted comment block, called a NISTCOM. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. The utility **rdjpgcom** scans a JPEG compressed file for any and all comment blocks. Once found, the contents of each comment block is printed to standard output. Using this utility, the NISTCOM provides easy access to relevant image attributes. The following is an example NISTCOM generated by **cjpegb**:

NIST_COM 12
PIX_WIDTH 768
PIX_HEIGHT 1024
PIX_DEPTH 24
PPI -1
LOSSY 1
COLORSPACE YCbCr
NUM_COMPONENTS 3
HV_FACTORS 2,2:1,1:1,1
INTERLEAVE 1
COMPRESSION JPEGB
JPEGB_QUALITY 50

Cjpegb also accepts an optional comment file on the command line. If provided, the contents of this file are also inserted into the compressed output file. If the comment file is a NISTCOM attribute list, then its contents are merged with the NISTCOM internally generated by **cjpegb** and a single NISTCOM is written to the compressed output file. Note that **cjpegb** gives precedence to internally generated attribute values. If the user provides a non-NISTCOM comment file, then the contents of file are stored to a separate comment block in the output file. Using these comment options enables the user to store application-specific information in a JPEG file.

EXAMPLES

From *test/imgtools/execs/cjpegb/cjpegb.src*:

% cjpegb 50 jpb face08.raw -r 768,1024,8
compresses a grayscale face image in a raw pixmap file.

% cjpegb 50 jpb face24.raw -r 768,1024,24
compresses a color face image in a raw pixmap file.

SEE ALSO

cjpeg(1H), **cjpegl**(1G), **djpegb**(1G), **dpyimage**(1G), **intr2not**(1G), **jpegtran**(1H), **not2intr**(1G), **rdjpgcom**(1H), **wrjpgcom**(1H)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

cjpegl – compresses a grayscale or color image using Lossless JPEG (JPEGL).

SYNOPSIS

cjpegl <*outext*> <*image file*>
[**-raw_in** *w*,*h*,*d*,[*ppi*]
[**-nonintrlv**]
[**-YCbCr** *H0*,*V0*:*H1*,*V1*:*H2*,*V2*]]
[*comment file*]

DESCRIPTION

Cjpegl takes as input a file containing an uncompressed grayscale or color image. Two possible input file formats are accepted, NIST IHead files and raw pixmap files. If a raw pixmap file is to be compressed, then its image attributes must be provided on the command line as well. Once read into memory, the grayscale or color pixmap is then compressed using Lossless JPEG (JPEGL). The JPEGL results are then written to an output file.

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. By default, **cjpegl** assumes interleaved color pixels. In fact **cjpegl**'s internal encoder requires non-interleaved components planes; therefore, interleaved pixmaps are automatically converted prior to encoding and results are stored accordingly. (See INTERLEAVE OPTIONS below.)

Cjpegl also supports the compression of raw pixmap files containing YCbCr images with potentially downsampled component planes. By default, this utility assumes no downsampling of component planes. (See YCbCr OPTIONS below.) Regarding color pixmaps, the NIST IHead file format only supports interleaved RGB images.

OPTIONS

All switch names may be abbreviated; for example, **-raw_in** may be written **-r**.

<*outext*>
the extension of the compressed output file. To construct the output filename, **cjpegl** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input file, either an IHead file or raw pixmap file, containing the grayscale or color image to be compressed.

-raw_in *w*,*h*,*d*,[*ppi*]
the attributes of the input image. This option must be included on the command line if the input is a raw pixmap file.

w	the pixel width of the pixmap
h	the pixel height of the pixmap
d	the pixel depth of the pixmap
ppi	the optional scan resolution of the image in integer units of pixels per inch.

-nonintrlv
specifies that the color components in an *input* raw pixmap file image are non-interleaved and stored in separate component planes. (See INTERLEAVE OPTIONS below.)

-YCbCr *H0*,*V0*:*H1*,*V1*:*H2*,*V2*
denotes an *input* raw pixmap file containing a YCbCr colorspace image and whether any component planes have been *previously* downsampled. H,V factors all set to 1 represent no downsampling. (See YCbCr Options below.)

comment file
an optional user-supplied ASCII comment file. (See COMMENT OPTIONS below.)

INTERLEAVE OPTIONS

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. Color components are interleaved when a pixel's (R)ed, (G)reen, and (B)lue components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **cjpegl** assumes interleaved color components, and all color IHead images must be interleaved. Note that **cjpegl**'s internal encoder requires non-interleaved component planes; therefore, interleaved pixmaps are automatically converted prior to encoding and results are stored accordingly.

YCbCr OPTIONS

Cjpegl compresses color images with 3 components per pixel, including RGB and YCbCr colorspaces. A common compression technique for YCbCr images is to downsample the Cb & Cr component planes. **Cjpegl** supports a limited range of YCbCr downsampling schemes that are represented by a list of component plane factors. These factors together represent downsampling ratios relative to each other. The comma-separated list of factor pairs correspond to the Y, Cb, and Cr component planes respectively. The first value in a factor pair represents the downsampling of that particular component plane in the X-dimension, while the second represents the Y-dimension. Compression ratios for a particular component plane are calculated by dividing the maximum component factors in the list by the current component's factors. These integer factors are limited between 1 and 4. H,V factors all set to 1 represent no downsampling. For complete details, **cjpegl** implements the downsampling and interleaving schemes described in the following reference:

> W.B. Pennebaker and J.L. Mitchell, "JPEG: Still Image Compression Standard," Appendix A - "ISO DIS 10918-1 Requirements and Guidelines," Van Nostrand Reinhold, NY, 1993, pp. A1-A4.

For example the option specification:

> -YCbCr 4,4:2,2:1,1

represents a YCbCr image with non-downsampled Y component plane (4,4 are the largest X and Y factors listed); the Cb component plane is downsampled in X and Y by a factor of 2 (maximum factors 4 divided by current factors 2); and the Cr component plane is downsampled in X and Y by a factor of 4 (maximum factors 4 divided by current factors 1). Note that downsampling component planes is a form of *lossy* compression, so while **cjpegl** enables the image byte stream associated with an input YCbCr image to be reconstructed perfectly, if any of its component planes were previously downsampled, then image loss has already taken place. The utility **rgb2ycc** converts an RGB image to the YCbCr colorspace, and it will conduct component plane downsampling as specified. Note that IHead images can only have RGB color components, so YCbCr options only pertain to raw pixmap files.

COMMENT OPTIONS

Upon successful compression, this utility generates and inserts in the compressed output file a specially formatted comment block, called a NISTCOM. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. The utility **rdjpgcom** scans a JPEG compressed file for any and all comment blocks. Once found, the contents of each comment block is printed to standard output. Using this utility, the NISTCOM provides easy access to relevant image attributes. The following is an example NISTCOM generated by **cjpegl**:

NIST_COM 11
PIX_WIDTH 768
PIX_HEIGHT 1024
PIX_DEPTH 24
PPI -1
LOSSY 0
NUM_COMPONENTS 3
HV_FACTORS 1,1:1,1:1,1
INTERLEAVE 0
COMPRESSION JPEGL
JPEGL_PREDICT 4

Cjpegl also accepts an optional comment file on the command line. If provided, the contents of this file are also inserted into the compressed output file. If the comment file is a NISTCOM attribute list, then its contents are merged with the NISTCOM internally generated by **cjpegl** and a single NISTCOM is written to the compressed output file. Note that **cjpegl** gives precedence to internally generated attribute values. If the user provides a non-NISTCOM comment file, then the contents of file are stored to a separate comment block in the output file. Using these comment options enables the user to store application-specific information in a JPEG file.

EXAMPLES

From *test/imgtools/execs/cjpegl/cjpegl.src*:

% cjpegl jpl face.raw -r 768,1024,24
compresses a color face image in a raw pixmap file.

SEE ALSO

cjpegb(1G), **djpegl**(1G), **dpyimage**(1G), **intr2not**(1G), **not2intr**(1G), **rdjpgcom**(1H), **rgb2ycc**(1G), **wrjpgcom**(1H)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

cmbmcs – combines PCASYS mean/covariance data file pairs.

SYNOPSIS

cmbmcs *<meanfile_in[meanfile_in...]> <covfile_in[covfile_in...]> <meanfile_out> <meanfile_out_desc> <covfile_out> <covfile_out_desc> <ascii_outfiles>*

DESCRIPTION

Cmbmcs combines pairs of PCASYS mean vector and covariance matrix data files, to produce a mean/covariance pair that is approximately the same as would have resulted if all the vectors that were used to make the input means and covariances had been given to the meancov function in one large set. So, if the sample covariance matrix needs to be made from a large set of vectors, and several processors are available, it may be possible to save run time by, first, running several simultaneous instances of meancov, each on a subset of the vectors, and second, running cmbmcs to combine the means and covariances made by the meancov instances. (Even if only the covariance is ultimately needed, i.e. not the mean, it is necessary for the meancov instances to compute the means and for cmbmcs to use them, to compute the overall covariance.)

OPTIONS

<meanfile_in[meanfile_in...]>

Input files each containing a mean vector. These files must be in PCASYS "matrix" format, each with first dimension 1 and all having the same second dimension. (Usually the output of **meancov**.)

<covfile_in[covfile_in...]>

Input files each containing a covariance matrix. These files must be in PCASYS "covariance" format. The i'th input covariance file goes with the i'th input mean file. These covariances must all have the same order, which must be the second dimension of the input mean vector files. (Usually the output of **meancov**.)

<meanfile_out>

Mean file to be written, in PCASYS "matrix" format.

<meanfile_out_desc>

A string to be written into the mean output files description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quotes, i.e. single–quoted empty string). To let cmbmcs make a description (stating that this is a mean vector made by cmbmcs and listing the names of the input files), use – (hyphen).

<covfile_out>

Covariance file to be produced, in PCASYS "covariance" format. Its "number of vectors" field will be set to the sum of the values of that field across the input covariances.

<covfile_out_desc>

Description string for output covariance file or - to let cmbmcs make the description (as for output mean file description).

<ascii_outfiles>

If y, makes ascii output files; if n, binary. Binary is recommended, unless the output files must be portable across different byte orders or floating_point formats.

EXAMPLE(S)

From *test/pcasys/execs/cmbmcs/cmbmcs.src*:

% cmbmcs ../meancov/fv1.men ../meancov/fv2.men ../meancov/fv1.cov ../meancov/fv2.cov fv_tst.men - fv_tst.cov - n

Combines the mean/covariance files for fv1 and fv2 into a mean/covariance set of files.

SEE ALSO

meancov (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

cwsq – WSQ compresses grayscale fingerprint images.

SYNOPSIS

cwsq <*r bitrate*> <*outext*> <*image file*>
[**-raw_in** *w*,*h*,*d*,[*ppi*]] [*comment file*]

DESCRIPTION

Cwsq takes as input a file containing an uncompressed grayscale fingerprint image. Two possible input file formats are accepted, NIST IHead files and raw pixmap files. If a raw pixmap file is to be compressed, then its image attributes must be provided on the command line as well. Once read into memory, the pixmap is *lossy* compressed using Wavelet Scalar Quantization as described in the FBI's Criminal Justice Information Services (CJIS) document, "WSQ Gray-scale Fingerprint Compressions Specification," Dec. 1997. The results are then written to an output file in a format dictated by this document. This is the only fingerprint compression format accepted by the FBI IAFIS system.

OPTIONS

All switch names may be abbreviated; for example, **-raw_in** may be written **-r**.

<*r bitrate*>
determines the amount of lossy compression.
Suggested settings:
r bitrate = 2.25 yields around 5:1 compression
r bitrate = 0.75 yields around 15:1 compression

<*outext*>
the extension of the compressed output file. To construct the output filename, **cwsq** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input file, either an IHead file or raw pixmap file, containing the fingerprint image to be compressed.

-raw_in *w*,*h*,*d*,[*ppi*]
the attributes of the input image. This option must be included on the command line if the input is a raw pixmap file.

w	the pixel width of the pixmap
h	the pixel height of the pixmap
d	the pixel depth of the pixmap
ppi	the optional scan resolution of the image in integer units of pixels per inch.

comment file
an optional user-supplied ASCII comment file. (See COMMENT OPTIONS below.)

COMMENT OPTIONS

Upon successful compression, this utility generates and inserts in the compressed output file a specially formatted comment block, called a NISTCOM. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. The utility **rdwsqcom** scans a WSQ compressed file for any and all comment blocks. Once found, the contents of each comment block is printed to standard output. Using this utility, the NISTCOM provides easy access to relevant image attributes. The following is an example NISTCOM generated by **cwsq**:

```
NIST_COM 9
PIX_WIDTH 500
PIX_HEIGHT 500
```

PIX_DEPTH 8
PPI 500
LOSSY 1
COLORSPACE GRAY
COMPRESSION WSQ
WSQ_BITRATE 0.750000

Cwsq also accepts an optional comment file on the command line. If provided, the contents of this file are also inserted into the compressed output file. If the comment file is a NISTCOM attribute list, then its contents are merged with the NISTCOM internally generated by **cwsq** and a single NISTCOM is written to the compressed output file. Note that **cwsq** gives precedence to internally generated attribute values. If the user provides a non-NISTCOM comment file, then the contents of file are stored to a separate comment block in the output file. Using these comment options enables the user to store application-specific information in a WSQ file.

EXAMPLES

From *test/imgtools/execs/cwsq/cwsq.src*:

% cwsq .75 wsq finger.raw -r 500,500,8,500
compresses a raw fingerprint pixmap.

SEE ALSO

dpyimage(1G), **dwsq**(1G), **rdwsqcom**(1G), **wrwsqcom**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

datainfo – shows the header information of a PCASYS data file.

SYNOPSIS

datainfo <*datafile*>

DESCRIPTION

Datainfo reads the header information from a PCASYS data file and writes a short report of this information to the standard output.

The file must be in one of the official PCASYS data file formats, which are "matrix" (matrix of floating point numbers), "covariance" (covariance matrix of floating point numbers, with only the nonstrict lower triangle stored since a covariance matrix is symmetric), and "classes" (classes, represented as unsigned characters, which are thought of as being integers in the range 0 through 255). PCASYS data files come in these three types, and can be either "ascii" or "binary", so there are really 6 types in all.

Datainfo reports the description string of the data file, its type (matrix, etc.), whether it is ascii or binary, and then some final information specific to the file type: if matrix, the two dimensions; if covariance, the order (of the symmetric covariance matrix, i.e. the number that both dimensions equal) and the number of input vectors used to make the covariance; if classes, the number of elements.

OPTIONS

<*datafile*>

Data file whose header information is to be reported.

EXAMPLE(S)

From *test/pcasys/execs/datainfo/datainfo.src*:

% datainfo ../../data/oas/fv1-9.cls >& fv1-9cls.dat

% datainfo ../../data/oas/fv1.oas >& fv1oas.dat

% datainfo ../meancov/fv1-9.cov >& fv1-9cov.dat

Prints out the file header information for the various types of files, class (fv1-9.cls), orientation arrays (fv1.oas), and covariance (fv1-9.cov).

SEE ALSO

chgdesc (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

diffbyts – takes two binary data files and compares them byte for byte, compiling a cumulative histogram of differences.

SYNOPSIS

diffbyts <*file1*> <*file2*>

DESCRIPTION

Diffbyts takes as input two binary files of equal length and compares the contents between the two files byte for byte. The differences between corresponding pairs of bytes are accumulated into a histogram, where each bin in the histogram represents the integer difference between the byte pairs. Therefore, the first bin in the histogram contains the count of all those byte pairs that are exactly the same (a difference of 0); the next bin contains the count of all those byte pairs that are different by exactly 1; and so on.

Upon completion, this utility prints a formatted report of the accumulated histogram to standard output. Each difference bin in the histogram is listed on separate line in the report and formatted at follows:

d[*b*] = *c* : *p*

where

b is the current bin's byte pair difference.

c is the number of corresponding byte pairs with difference equal to *b*.

p is the cumulative percentage of corresponding pairs of bytes counted in all the bins up to and including the current bin.

OPTIONS

<*file1*> a binary file to be compared byte for byte to *file2*.

<*file2*> a binary file to be compared byte for byte to *file1*.

EXAMPLES

From *test/imgtools/execs/diffbyts/diffbyts.src*:

% diffbyts ../../data/finger/gray/raw/finger.raw ../dwsq/finger.raw > finger.hst
compiles and stores a byte-difference histogram reporting the amount of image degradation due to lossy WSQ compression.

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

djpeg – decompress a JPEG file to an image file

SYNOPSIS

djpeg [*options*] [*filename*]

DESCRIPTION

djpeg decompresses the named JPEG file, or the standard input if no file is named, and produces an image file on the standard output. PBMPLUS (PPM/PGM), BMP, GIF, Targa, or RLE (Utah Raster Toolkit) output format can be selected. (RLE is supported only if the URT library is available.)

OPTIONS

All switch names may be abbreviated; for example, **–grayscale** may be written **–gray** or **–gr**. Most of the "basic" switches can be abbreviated to as little as one letter. Upper and lower case are equivalent (thus **–BMP** is the same as **–bmp**). British spellings are also accepted (e.g., **–greyscale**), though for brevity these are not mentioned below.

The basic switches are:

–colors *N*

Reduce image to at most N colors. This reduces the number of colors used in the output image, so that it can be displayed on a colormapped display or stored in a colormapped file format. For example, if you have an 8-bit display, you'd need to reduce to 256 or fewer colors.

–quantize *N*

Same as **–colors**. **–colors** is the recommended name, **–quantize** is provided only for backwards compatibility.

–fast Select recommended processing options for fast, low quality output. (The default options are chosen for highest quality output.) Currently, this is equivalent to **–dct fast –nosmooth –onepass –dither ordered**.

–grayscale

Force gray-scale output even if JPEG file is color. Useful for viewing on monochrome displays; also, **djpeg** runs noticeably faster in this mode.

–scale *M/N*

Scale the output image by a factor M/N. Currently the scale factor must be 1/1, 1/2, 1/4, or 1/8. Scaling is handy if the image is larger than your screen; also, **djpeg** runs much faster when scaling down the output.

–bmp Select BMP output format (Windows flavor). 8-bit colormapped format is emitted if **–colors** or **–grayscale** is specified, or if the JPEG file is gray-scale; otherwise, 24-bit full-color format is emitted.

–gif Select GIF output format. Since GIF does not support more than 256 colors, **–colors 256** is assumed (unless you specify a smaller number of colors).

–os2 Select BMP output format (OS/2 1.x flavor). 8-bit colormapped format is emitted if **–colors** or **–grayscale** is specified, or if the JPEG file is gray-scale; otherwise, 24-bit full-color format is emitted.

–pnm Select PBMPLUS (PPM/PGM) output format (this is the default format). PGM is emitted if the JPEG file is gray-scale or if **–grayscale** is specified; otherwise PPM is emitted.

–rle Select RLE output format. (Requires URT library.)

–targa Select Targa output format. Gray-scale format is emitted if the JPEG file is gray-scale or if **–grayscale** is specified; otherwise, colormapped format is emitted if **–colors** is specified; otherwise, 24-bit full-color format is emitted.

Switches for advanced users:

−dct int
Use integer DCT method (default).

−dct fast
Use fast integer DCT (less accurate).

−dct float
Use floating-point DCT method. The float method is very slightly more accurate than the int method, but is much slower unless your machine has very fast floating-point hardware. Also note that results of the floating-point method may vary slightly across machines, while the integer methods should give the same results everywhere. The fast integer method is much less accurate than the other two.

−dither fs
Use Floyd-Steinberg dithering in color quantization.

−dither ordered
Use ordered dithering in color quantization.

−dither none
Do not use dithering in color quantization. By default, Floyd-Steinberg dithering is applied when quantizing colors; this is slow but usually produces the best results. Ordered dither is a compromise between speed and quality; no dithering is fast but usually looks awful. Note that these switches have no effect unless color quantization is being done. Ordered dither is only available in **−onepass** mode.

−map *file*
Quantize to the colors used in the specified image file. This is useful for producing multiple files with identical color maps, or for forcing a predefined set of colors to be used. The *file* must be a GIF or PPM file. This option overrides **−colors** and **−onepass**.

−nosmooth
Use a faster, lower-quality upsampling routine.

−onepass
Use one-pass instead of two-pass color quantization. The one-pass method is faster and needs less memory, but it produces a lower-quality image. **−onepass** is ignored unless you also say **−colors** *N*. Also, the one-pass method is always used for gray-scale output (the two-pass method is no improvement then).

−maxmemory *N*
Set limit for amount of memory to use in processing large images. Value is in thousands of bytes, or millions of bytes if "M" is attached to the number. For example, **−max 4m** selects 4000000 bytes. If more space is needed, temporary files will be used.

−outfile *name*
Send output image to the named file, not to standard output.

−verbose
Enable debug printout. More **−v**'s give more output. Also, version information is printed at startup.

−debug
Same as **−verbose**.

EXAMPLES

This example decompresses the JPEG file foo.jpg, quantizes it to 256 colors, and saves the output in 8-bit BMP format in foo.bmp:

djpeg −colors 256 −bmp *foo.jpg* **>** *foo.bmp*

HINTS

To get a quick preview of an image, use the **–grayscale** and/or **–scale** switches. **–grayscale –scale 1/8** is the fastest case.

Several options are available that trade off image quality to gain speed. **–fast** turns on the recommended settings.

–dct fast and/or **–nosmooth** gain speed at a small sacrifice in quality. When producing a color-quantized image, **–onepass –dither ordered** is fast but much lower quality than the default behavior. **–dither none** may give acceptable results in two-pass mode, but is seldom tolerable in one-pass mode.

If you are fortunate enough to have very fast floating point hardware, **–dct float** may be even faster than **–dct fast**. But on most machines **–dct float** is slower than **–dct int**; in this case it is not worth using, because its theoretical accuracy advantage is too small to be significant in practice.

ENVIRONMENT

JPEGMEM

If this environment variable is set, its value is the default memory limit. The value is specified as described for the **–maxmemory** switch. **JPEGMEM** overrides the default value specified when the program was compiled, and itself is overridden by an explicit **–maxmemory**.

SEE ALSO

cjpeg(1H), **jpegtran**(1H), **rdjpgcom**(1H), **wrjpgcom**(1H)
ppm(5), **pgm**(5)
Wallace, Gregory K. "The JPEG Still Picture Compression Standard", Communications of the ACM, April 1991 (vol. 34, no. 4), pp. 30-44.

AUTHOR

Independent JPEG Group

BUGS

Arithmetic coding is not supported for legal reasons.

To avoid the Unisys LZW patent, **djpeg** produces uncompressed GIF files. These are larger than they should be, but are readable by standard GIF decoders.

Still not as fast as we'd like.

NAME

djpegb – decompresses a Baseline JPEG (JPEGB) grayscale or color image.

SYNOPSIS

djpegb <*outext*> <*image file*>
[**-raw_out** [**-nonintrlv**]]

DESCRIPTION

Djpegb takes as input a file containing a Baseline JPEG (JPEGB) compressed grayscale or color image. Once read into memory, the lossy-compressed pixmap is decoded and reconstructed using the Independent JPEG Group's (IJG) library for Baseline JPEG.

Upon completion, two different output image file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag). In addition, a specially formatted text file, called a NISTCOM, is created with extension ".ncm". The NISTCOM file contains relevant image attributes associated with the decoded and reconstructed output image. (See NISTCOM OUTPUT below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
: the extension of the decompressed output file. To construct the output filename, **djpegb** takes the input filename and replaces its extension with the one specified here.

<*image file*>
: the input JPEGB file to be decompressed.

-raw_out
: specifies that the decoded and reconstructed image should be stored to a raw pixmap file.

-nonintrlv
: specifies that the color components in the reconstructed image should be organized into separate component planes. The **-raw_out** flag must be used with this option, because the IHead format only supports interleaved color pixels. (See INTERLEAVE OPTIONS below.)

INTERLEAVE OPTIONS

For example, given an RGB image, its color components may be interleaved or non-interleaved. Color components are interleaved when a pixel's (R)ed, (G)reen, and (B)lue components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **djpegb** uses interleaved color component pixels in the reconstructed output image. Note that all color IHead images must be interleaved.

NISTCOM OUTPUT

Upon successful completion, **djpegb**, creates a specially formatted text file called a NISTCOM file. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. These attributes are collected and merged from two different sources to represent the history and condition of the resulting reconstructed image. The first source is from an optional NISTCOM comment block inside the JPEGB-encoded input file. This comment block can be used to hold user-supplied attributes. The JPEGB encoder, **cjpegb**, by convention inserts one of these comment blocks in each compressed output file it creates. (The utility **rdjpgcom** can be used to scan a JPEG file for any and all comment blocks.) The second source of attributes comes from the decompression process itself. In general, attribute values from this second source are given precedence over those from the first.

The NISTCOM output filename is constructed by combining the basename of the input JPEGB file with the extension ".ncm". By creating the NISTCOM file, relevant attributes associated with the decoded and reconstructed image are retained and easily accessed. This is especially useful when dealing with raw pixmap files and creating image archives. The following is an example NISTCOM generated by **djpegb**:

NIST_COM 10
PIX_WIDTH 768
PIX_HEIGHT 1024
PIX_DEPTH 24
PPI -1
LOSSY 1
COLORSPACE RGB
NUM_COMPONENTS 3
HV_FACTORS 1,1:1,1:1,1
INTERLEAVE 1

EXAMPLES

From *test/imgtools/execs/djpegb/djpegb.src*:

% djpegb raw face24.jpb -r
decompresses a JPEGB-encoded RGB face image and stores the reconstructed image to a raw pixmap file. Note the NISTCOM file, **face24.ncm**, is also created.

SEE ALSO

cjpegb(1G), **dpyimage**(1G), **intr2not**(1G), **jpegtran**(1H), **not2intr**(1G), **rdjpgcom**(1H), **wrjpgcom**(1H)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

djpegl – decompresses a Lossless JPEG (JPEGL) grayscale or color image.

SYNOPSIS

djpegl <*outext*> <*image file*>
[**-raw_out** [**-nonintrlv**]]

DESCRIPTION

Djpegl takes as input a file containing a Lossless JPEG (JPEGL) compressed grayscale or color image. Once read into memory, the compressed pixmap is decoded and reconstructed to its original condition prior to compression.

Upon completion, two different output image file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag). In addition, a specially formatted text file, called a NISTCOM, is created with extension ".ncm". The NISTCOM file contains relevant image attributes associated with the decoded and reconstructed output image. (See NISTCOM OUTPUT below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
the extension of the decompressed output file. To construct the output filename, **djpegl** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input JPEGL file to be decompressed.

-raw_out
specifies that the decoded and reconstructed image should be stored to a raw pixmap file.

-nonintrlv
specifies that the color components in the reconstructed image should be organized into separate component planes. The **-raw_out** flag must be used with this option, because the IHead format only supports interleaved color pixels. (See INTERLEAVE OPTIONS below.)

INTERLEAVE OPTIONS

For example, given an RGB image, its color components may be interleaved or non-interleaved. Color components are interleaved when a pixel's (R)ed, (G)reen, and (B)lue components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **djpegl** uses interleaved color component pixels in the reconstructed output image. Note that all color IHead images must be interleaved.

NISTCOM OUTPUT

Upon successful completion, **djpegl**, creates a specially formatted text file called a NISTCOM file. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. These attributes are collected and merged from two different sources to represent the history and condition of the resulting reconstructed image. The first source is from an optional NISTCOM comment block inside the JPEGL-encoded input file. This comment block can be used to hold user-supplied attributes. The JPEGL encoder, **cjpegl**, by convention inserts one of these comment blocks in each compressed output file it creates. (The utility **rdjpgcom** can be used to scan a JPEG file for any and all comment blocks.) The second source of attributes comes from the decompression process itself. In general, attribute values from this second source are given precedence over those from the first.

The NISTCOM output filename is constructed by combining the basename of the input JPEGL file with the extension ".ncm". By creating the NISTCOM file, relevant attributes associated with the decoded and reconstructed image are retained and easily accessed. This is especially useful when dealing with raw pixmap files and creating image archives. The following is an example NISTCOM generated by **djpegl**:

NIST_COM 9
PIX_WIDTH 768
PIX_HEIGHT 1024
PIX_DEPTH 24
PPI -1
LOSSY 0
NUM_COMPONENTS 3
HV_FACTORS 1,1:1,1:1,1
INTERLEAVE 1

EXAMPLES

From *test/imgtools/execs/djpegl/djpegl.src*:

% djpegl raw finger.jpl -r
decompresses a JPEGL-encoded grayscale fingerprint image and stores the reconstructed image to a raw pixmap file. Note the NISTCOM file, **finger.ncm**, is also created.

% djpegl raw face.jpl -r
decompresses a JPEGL-encoded RGB face image and stores the reconstructed image to a raw pixmap file. Note the NISTCOM file, **face.ncm**, is also created.

SEE ALSO

cjpegl(1G), **dpyimage**(1G), **intr2not**(1G), **not2intr**(1G), **rdjpgcom**(1H), **wrjpgcom**(1H)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

djpeglsd – decompresses a grayscale image that was compressed using the old Lossless JPEG compression distributed with Special Databases 4, 9, 10, and 18. **Cjpegl** should be used in the future to Lossless JPEG (JPEGL) compress images.

SYNOPSIS

djpeglsd <*outext*> <*image file*>
[**-sd #**] [**-raw_out**]

DESCRIPTION

Djpeglsd takes as input a file containing a grayscale image that was compressed with the old Lossless JPEG (JPEGLSD). Specifically the version that is included on the CDROMs with Special Databases 4, 9, 10, and 18. Once read into memory, the compressed pixmap is decoded and reconstructed to its original condition prior to compression. **Cjpegl** should be used for any future Lossless JPEG compression of images.

Upon completion, two different output image file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag). In addition, a specially formatted text file, called a NIST-COM, is created with extension ".ncm". The NISTCOM file contains relevant image attributes associated with the decoded and reconstructed output image. If given a special database number, **djpeglsd** will put all the important class, sex, age, and file history information, that may exist for that database, in the NISTCOM file. (See NISTCOM OUTPUT below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
the extension of the decompressed output file. To construct the output filename, **djpeglsd** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input JPEGLSD file to be decompressed.

-sd # Specify that the input image is from NIST Special Database #.

-raw_out
specifies that the decoded and reconstructed image should be stored to a raw pixmap file.

NISTCOM OUTPUT

Upon successful completion, **djpeglsd**, creates a specially formatted text file called a NISTCOM file. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. These attributes are collected from information about the decompressed image. Detailed attributes are collected if the *-sd #* flag is used.

The NISTCOM output filename is constructed by combining the basename of the input JPEGLSD file with the extension ".ncm". By creating the NISTCOM file, relevant attributes associated with the decoded and reconstructed image are retained and easily accessed. This is especially useful when dealing with raw pixmap files and creating image archives. The following are example NISTCOMs generated by **djpeglsd** for SD9 and SD18 images (The highlighted items are attributes specific to that database.):

NIST_COM 12
SD_ID 9
HISTORY f0000771.pct ac/dm_fpw:20 tape9.n1125012.01 4096x1536
FING_CLASS W
SEX f

SCAN_TYPE i
PIX_WIDTH 832
PIX_HEIGHT 768
PIX_DEPTH 8
PPI 500
LOSSY 0
COLORSPACE GRAY

NIST_COM 12
SD_ID 18
HISTORY f00117_1.pct
SEX m
AGE 26
FACE_POS f
PIX_WIDTH 592
PIX_HEIGHT 448
PIX_DEPTH 8
PPI 500
LOSSY 0
COLORSPACE GRAY

EXAMPLES

From *test/imgtools/execs/djpeglsd/djpeglsd.src*:

% djpeglsd raw sd04.old -sd 4 -r
% djpeglsd raw sd09.old -sd 9 -r
% djpeglsd raw sd10.old -sd 10 -r
% djpeglsd raw sd18.old -sd 18 -r
decompresses JPEGLSD-encoded images from the Special Databases and stores the reconstructed images to a raw pixmap files. Note the NISTCOM files, **sd04.ncm**, **sd09.ncm**, **sd10.ncm**, and **sd18.ncm**, are also created.

SEE ALSO

cjpegl(1G), **djpegl**(1G), **dpyimage**(1G),

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

dpyan2k – display image and minutiae contents of an ANSI/NIST file.

SYNOPSIS

dpyan2k [*options*] <*ANSI_NIST ...*>

-a *n*
-v
-x
-b *n*
-i
-n
-p *n*
-W *n*
-H *n*
-X *n*
-Y *n*
-T *title*
-d *display*

DESCRIPTION

Dpyan2k displays in a sequence of X11 windows all the image records and overlays any corresponding minutiae from Type-9 records contained in an ANSI/NIST-ITL 1-2000 file.

If multiple input files are specified, *dpyan2k* reads each ANSI/NIST file into memory and displays its contents, one file at a time. Multiple image records within an ANSI/NIST file are displayed simultaneously by forking background window processes, one for each image record.

If an image is too large to be displayed on the screen, the upper left hand corner will be displayed and the rest of the image can be moved into view by holding down a mouse button, moving in the direction desired, and then releasing the button. Button presses when another button(s) is already down and button releases when another button(s) is still down are ignored.

Users may remove a displayed image window by striking any key within that window. Once all windows associated with a particular ANSI/NIST file have been removed, the utility proceeds to display the contents of the next ANSI/NIST file listed on the command line.

OPTIONS

-a *n* sets drag accelerator to *n* — changes in pointer position will result in *n* shifts in the displayed image [1].

-v turns on verbose output.

-x turns on debug mode, causing a core dump when an X11 error occurs.

-b *n* sets border width to *n* pixels [4].

-i directs the utility to use the FBI/IAFIS fields 13-23 in a Type-9 record when overlaying minutiae on an image.

-n directs the utility to use the NIST fields 5-12 in a Type-9 record when overlaying minutiae on an image. This is the default setting.

-p sets the pixel width of overlayed minutia points [3].

-W *n* displays image in a window of width *n* pixels.

-H *n* displays image in a window of height *n* pixels.

-X *n* positions image window with top-left corner *n* pixels to the right of the display's top-left corner [0].

-Y *n* positions image window with top-left corner *n* pixels below the display's top-left corner [0].

-T *title* sets all image window names to *title*.

-d *display*
connects to an alternate X11 display.

<ANSI_NIST ...>
one or more ANSI/NIST files with images and possbily minutiae to be displayed.

EXAMPLES

From *test/an2k/execs/dpyan2k/dpyan2k.src*:

% dpyan2k ../../data/nist.an2
displays image records and overlays minutia using NIST Type-9 fields.

% dpyan2k -i ../../data/iafis.an2
displays image records and overlays minutia using FBI/IAFIS Type-9 fields.

SEE ALSO

an2ktool(1F), **dpyimage**(1G), **mindtct**(1C)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

dpyimage – displays the image contents of Baseline JPEG, Lossless JPEG, WSQ, IHead, and raw pixmap files.

SYNOPSIS

dpyimage [*options*] *image-file ...*

-r *w,h,d,wp*
-A
-s *n*
-a *n*
-v
-x
-b *n*
-N *n*
-O
-k
-W *n*
-H *n*
-X *n*
-Y *n*
-n
-T *title*
-t
-D *dir*
-d *display*

DESCRIPTION

Dpyimage reads various image file formats, decompresses and reconstructs pixmaps as needed, and displays image contents in an X11 window. Supported file formats include Baseline JPEG (lossy), Lossless JPEG, WSQ (lossy), NIST IHead, and raw pixmap files. Raw pixmaps containing either grayscale or interleaved RGB color pixels are supported. This utility automatically differentiates between these different formats.

If only one file (or the *-n* option) is specified on the command line, the image or images are simply read from disk and then displayed. If multiple files are specified, **dpyimage** attempts to minimize the display waiting time by forking a background process to pre-read images from disk. By default, the child transfers images to the parent via a pipe. This always allows at least one image to be read in from disk while the user is viewing the current image. Since a process writing on a pipe is blocked (until a read on the other end of the pipe) after transferring four kilobytes, the child will only be one image ahead of the parent except when handling smaller images.

If the *-t* option appears on the command line, the processes use temporary files as the means of exchanging image data. Therefore, the child is not constrained on the number of images it may pre-read for the parent. However, the filesystem on which the directory for temporary files resides must have enough space for copies of all images in uncompressed state or an error may occur. This is the suggested mode for viewing compressed images for which decompression takes considerably longer than disk I/O.

If the image is too large to be displayed on the screen, the upper lefthand corner will be displayed and the rest of the image can be moved into view by holding down a mouse button, moving in the direction desired, and then releasing the button. Button presses when another button(s) is already down and button releases when another button(s) is still down are ignored.

Users may exit from the program by striking keys 'x' or 'X'. Advancing to the next image is accomplished by any other keystroke.

OPTIONS

-r *w*,*h*,*d*,*wp*
raw pixmap attributes:
w - pixel width,
h - pixel height,
d - pixel depth,
wp - white pixel value
bi-level wp=0|1
grayscale wp=0|255
RGB wp=0 (value ignored)

-A automatically advances through images.

-s *n* in automatic mode, sleeps *n* seconds before advancing to the next image [2].

-a *n* sets drag accelerator to *n* — changes in pointer position will result in *n* shifts in the displayed image [1].

-v turns on verbose output.

-x turns on debug mode, causing a core dump when an X11 error occurs.

-b *n* sets border width to *n* pixels [4].

-N *n* the child I/O process is niced to level *n*.

-O overrides the redirect on windows (no window manager).

-k informs utility that there is no keyboard input.

-W *n* displays image in a window of width *n* pixels.

-H *n* displays image in a window of height *n* pixels.

-X *n* positions image window with top-left corner *n* pixels to the right of the display's top-left corner [0].

-Y *n* positions image window with top-left corner *n* pixels below the display's top-left corner [0].

-n does not fork to display multiple images.

-T *title* sets window name to *title* [*file*]. **-t uses temporary files to transfer multiple images to parent [via pipe].**

-D *directory*
creates temporary files in *directory* [/tmp].

-d *display*
connects to alternate display.

image-file ...
one or more image files whose pixmaps are to be displayed.

ENVIRONMENT

If the environment variable **TMPDIR** is set and the **-D** option is not set on the command line, **dpyimage** uses this directory as the location for temporary files.

EXAMPLES

From *test/imgtools/execs/dpyimage/dpyimage.src*:

% dpyimage -r 500,500,8,255 ../../data/finger/gray/raw/finger.raw
displays a fingerprint image from a raw pixmap file.

% dpyimage ../../data/finger/gray/jpegl/finger.jpl
displays a reconstructed fingerprint image from a Lossless JPEG file.

% dpyimage ../../data/finger/gray/wsq/finger.wsq
displays a reconstructed fingerprint image from a WSQ file.

% dpyimage ../../data/face/gray/jpegb/face.jpb
displays a reconstructed grayscale face image from a Baseline JPEG file.

% dpyimage -r 768,1024,24,0 ../../data/face/rgb/raw/intrlv/face.raw
displays a color face image from a raw pixmap file.

% dpyimage ../../data/face/rgb/jpegb/face.jpb
displays a reconstructed color face image from a Baseline JPEG file.

% dpyimage ../../data/face/rgb/jpegl/face.jpl
displays a reconstructed color face image from a Lossless JPEG file.

SEE ALSO

an2ktool(1F), **cjpegb**(1G), **cjpegl**(1G), **cwsq**(1G), **djpegb**(1G), **djpegl**(1G), **dpyan2k**(1F), **dwsq**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

dwsq – decompresses a WSQ-encoded grayscale fingerprint image.

SYNOPSIS

dwsq <*outext*> <*image file*> [**-raw_out**]

DESCRIPTION

Dwsq takes as input a file containing a WSQ-compressed grayscale fingerprint image. Once read into memory, the lossy-compressed pixmap is decoded and reconstructed using Wavelet Scalar Quantization as described in the FBI's Criminal Justice Information Services (CJIS) document, "WSQ Gray-scale Fingerprint Compressions Specification," Dec. 1997. This is the only fingerprint compression format accepted by the FBI IAFIS system.

Upon completion, two different output image file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag). In addition, a specially formatted text file, called a NISTCOM, is created with extension ".ncm". The NISTCOM file contains relevant image attributes associated with the decoded and reconstructed output image. (See NISTCOM OUTPUT below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
: the extension of the decompressed output file. To construct the output filename, **dwsq** takes the input filename and replaces its extension with the one specified here.

<*image file*>
: the input WSQ file to be decompressed.

-raw_out
: specifies that the decoded and reconstructed image should be stored to a raw pixmap file.

NISTCOM OUTPUT

Upon successful completion, **dwsq**, creates a specially formatted text file called a NISTCOM file. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. These attributes are collected and merged from two different sources to represent the history and condition of the resulting reconstructed image. The first source is from an optional NISTCOM comment block inside the WSQ-encoded input file. This comment block can be used to hold user-supplied attributes. The WSQ encoder, **cwsq**, by convention inserts one of these comment blocks in each compressed output file it creates. (The utility **rdwsqcom** can be used to scan a WSQ file for any and all comment blocks.) The second source of attributes comes from the decompression process itself. In general, attribute values from this second source are given precedence over those from the first.

The NISTCOM output filename is constructed by combining the basename of the input WSQ file with the extension ".ncm". By creating the NISTCOM file, relevant attributes associated with the decoded and reconstructed image are retained and easily accessed. This is especially useful when dealing with raw pixmap files and creating image archives. The following is an example NISTCOM generated by **dwsq**:

```
NIST_COM 7
PIX_WIDTH 500
PIX_HEIGHT 500
PIX_DEPTH 8
PPI 500
LOSSY 1
COLORSPACE GRAY
```

EXAMPLES

From *test/imgtools/execs/dwsq/dwsq.src*:

% dwsq raw finger.wsq -r
decompresses a WSQ-encoded fingerprint image and stores the reconstructed image to a raw pixmap file. Note the NISTCOM file, **finger.ncm**, is also created.

SEE ALSO

cwsq(1G), **dpyimage**(1G), **rdwsqcom**(1G), **wrwsqcom**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

dwsq14 – decompresses a WSQ14-encoded grayscale fingerprint image from NIST Special Database 14.

SYNOPSIS

dwsq14 <*outext*> <*image file*> [**-raw_out**]

DESCRIPTION

Dwsq14 takes as input a file containing a WSQ14-compressed grayscale fingerprint image from NIST Special Database 14. The version of WSQ14 used to compress the images on this database is not certified and produces loss that may be more than permitted by a certified WSQ compression algorithm. Once read into memory, the lossy-compressed pixmap is decoded and reconstructed using Wavelet Scalar Quantization as described in the FBI's Criminal Justice Information Services (CJIS) document, "WSQ Gray-scale Fingerprint Compressions Specification," Dec. 1997. This is the only fingerprint compression format accepted by the FBI IAFIS system.

Upon completion, two different output image file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag). In addition, a specially formatted text file, called a NISTCOM, is created with extension ".ncm". The NISTCOM file contains relevant image attributes associated with the decoded and reconstructed output image. (See NISTCOM OUTPUT below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
: the extension of the decompressed output file. To construct the output filename, **dwsq14** takes the input filename and replaces its extension with the one specified here.

<*image file*>
: the input WSQ14 file to be decompressed.

-raw_out
: specifies that the decoded and reconstructed image should be stored to a raw pixmap file.

NISTCOM OUTPUT

Upon successful completion, **dwsq14**, creates a specially formatted text file called a NISTCOM file. A NISTCOM is a text-based attribute list comprised of (name, value) pairs, one pair per text line. The first line of a NISTCOM always has name = "NIST_COM" and its value is always the total number of attributes included in the list. These attributes are collected from two sources and merged to represent the history and condition of the resulting reconstructed image. The two sources for the attributes are the IHEAD header of the compressed image (specific information about the fingerprint itself is contained here) and the decompression process. In general, attribute values from the second source are given precedence over those from the first.

The NISTCOM output filename is constructed by combining the basename of the input WSQ14 file with the extension ".ncm". By creating the NISTCOM file, relevant attributes associated with the decoded and reconstructed image are retained and easily accessed. This is especially useful when dealing with raw pixmap files and creating image archives. The following is an example NISTCOM generated by **dwsq14** (highlighted items specific to SD14):

NIST_COM 12
PPI 500
SD_ID 14
HISTORY f0000001.wsq 20 tape3.t1116010.01 4096x1536
FING_CLASS R
SEX m
SCAN_TYPE i

PIX_WIDTH 832
PIX_HEIGHT 768
PIX_DEPTH 8
LOSSY 1
COLORSPACE GRAY

EXAMPLES

From *test/imgtools/execs/dwsq14/dwsq14.src*:

% dwsq14 raw sd14.old -r
decompresses a WSQ14-encoded fingerprint image and stores the reconstructed image to a raw pixmap file. Note the NISTCOM file, **sd14.ncm**, is also created.

SEE ALSO

cwsq(1G), **dwsq**(1G), **dpyimage**(1G),

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

eva_evt – finds a desired number of eigenvalues and eigenvectors.

SYNOPSIS

eva_evt *<covfile_in> <num_eva_evt_wanted> <evafile> <eva_desc> <evtfile> <evt_desc> <ascii_outfiles>*

DESCRIPTION

eva_evt finds a desired number of eigenvalues, and corresponding eigenvectors, of a covariance matrix (or really, of any symmetric positive definite real matrix). Uses CLAPACK routines, which were originally in Fortran but were converted into C using f2c.

ARGUMENTS

<covfile_in>

The covariance matrix (really, symmetric positive definite real matrix) some of whose eigenvalues and corresponding eivenvectors are to be found. Must be a PCASYS "covariance" file. (Usually the output of **meancov**.)

<num_eva_evt_wanted>

Specifies how many eigenvalues and eigenvectors to return for the given covariance matrix.

<evafile>

File to be written containing the eigenvalues that are found; will be a PCASYS "matrix" file, with first dimension equal to 1 and second dimension equal to the number of eigenvalues found. The eigenvalues will be stored in decreasing order.

<eva_desc>

A string to be written into the eigenvalues output file as its description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quotes, i.e. single-quoted empty string). To let eva_evt make a description (stating that this is an eigenvalues file made by eva_evt, and showing the covariance file and number of eigenvalues), use - (hyphen).

<evtfile>

File to be written containing the eigenvectors that are found; will be a PCASYS "matrix" file. The i'th row of this matrix will be the eigenvector corresponding to the i'th entry in the eigenvalues output file.

<evt_desc>

Description string for eigenvectors output file, or - to let eva_evt make the description. As per the *eva_desc*.

<ascii_outfiles>

If y, makes ascii output files; if n, binary. Binary is recommended, unless the output files must be portable across different byte orders or floating−point formats.

EXAMPLE(S)

From *test/pcasys/execs/eva_evt/eva_evt.src*:

% eva_evt ../meancov/fv1-9.cov 128 fv1-9.eva - fv1-9.evt - n

Computes the eigen-values and eigen-vectors for fv1-9.cov and sorts in decreasing order, then returns the top 128 from that list.

SEE ALSO

asc2bin (1B), bin2asc (1B), lintran (1B), meancov (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

fing2pat – Takes a list of fingerprint images and creates a feature vector file that can be used in MLP training for the NFIQ algorithm.

SYNOPSIS

fing2pat *[-z <znormfile>] <image&class list> <binpats out>*

DESCRIPTION

Fing2pat Takes a list of grayscale fingerprint images, and for each image, computes a feature vector for use by the NFIQ algorithm for MLP training. **Fing2pat** gives the user the ability to create feature vector files that can be used to retrain the weights used by the MLP classifier in determining a fingerprint's image quality value. These could then be used by **nfiq** instead of the default weights.

OPTIONS

[-z <znormfile>]
: if set the feature vectors are ZNormalized based on the statistics provided in the znormfile.

<image&class list>
: file list of the fingerprint images and their corresponding class to be used for MLP training.

<binpats out>
: file where the feature vectors are written

EXAMPLES

From *test/nfiq/execs/fing2pat/fing2pat.src*:

fing2pat -z ../../../../nfiq/znorm.dat imgcls.lst norm.bin

NOTE: The classes assigned in the file imgcls.lst are not REAL they are just made up for this example.

SEE ALSO

znormdat(1D), **znormpat**(1D), **nfiq**(1D), **mlp**(1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

fixwts – M-weighted robust weight filter from network activations.

SYNOPSIS

fixwts <*long_error_file*> <*output_pat_wts*>

DESCRIPTION

Fixwts creates a set of unequal pattern weights. The unequal pattern weights make the least squares function more efficient for non-standard distributions of the output errors by giving less weight to outliers in the distribution.
[ref. D.F.Andrews, *Technometrics*, 16 (1974) 523.]

OPTIONS

<*long_error_file*>

The long error output file from an MLP training run. (Usually the output of a training run for **mlp**.)

<*output_pat_wts*>

The pattern weights to be used in the next MLP training run. It would replace "fg_pat_wts" in the example directory test/pcasys/execs/mlp/mlp_dir.

EXAMPLE(S)

From *test/pcasys/execs/fixwts/fixwts.src*:

% fixwts ../mlp/mlp_dir/trn01l.err fixwts.out
Takes the long error file (trn01l.err) from a mlp training run and computes a set of robust weights (fixwts.out) to use in the next training run.

SEE ALSO

mlp (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

iaf2an2k – Takes an ANSI/NIST file conforming to FBI/IAFIS specifications and modifies minutiae and fingerprint image records in support of the new 2000 standard.

SYNOPSIS

iaf2an2k <*file in*> <*file out*>

DESCRIPTION

Iaf2an2k parses an ANSI/NIST file conforming to the FBI/IAFIS (EFTS V7) specifications and, if necessary, converts specific records and fields to take advantage of the ANSI/NIST-ITL 1-2000 standard. This utility focuses on the format of minutiae and image records.

Minutiae fields:
When a Type-9 record is encountered in the input file, this utility checks to see which fields are populated. If the NIST-assigned fields 5-12 are empty, but the FBI/IAFIS-assigned fields 13-23 are populated, then the NIST fields are populated by translating the data recorded in the FBI/IAFIS fields, and the FBI/IAFIS fields are removed.

Image records:
FBI/IAFIS specifications (EFTS V7) require binary field images, but the ANSI/NIST 2000 standard introduces tagged field image records. To support these new image records, this utility looks for binary field fingerprint records and converts them appropriately. If a Type-4 or Type-6 record is encountered, it is inspected to determine the impression type of the fingerprint. Latent fingerprints are converted to Type-13 records, while all others are converted to Type-14 records.

OPTIONS

<*file in*>
: the ANSI/NIST file to be converted

<*file out*>
: the resulting ANSI/NIST file

EXAMPLES

From *test/an2k/execs/iafan2k/iafan2k.src*:

% iafan2k ../../data/iafis.an2 nist.an2

SEE ALSO

an2k2iaf(1F)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

intr2not – converts an image comprised of pixels with interleaved color components to an image with non-interleaved component planes.

SYNOPSIS

intr2not <*outext*> <*image file*>
[**-raw_in** *w*,*h*,*d*,[*ppi*]
[**-YCbCr** *H0*,*V0*:*H1*,*V1*:*H2*,*V2*]]

DESCRIPTION

Intr2not takes as input an uncompressed image of pixels with interleaved color components and converts the image into non-interleaved component planes. This utility requires there be three color components in the input image. Two input file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_in** option). The non-interleaved results are stored to a raw pixmap file regardless of the input file format, because the IHead format only supports interleaved pixels.

For example, the pixels of an RGB color image are interleaved when a pixel's R, G, and B components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utility **not2intr** converts non-interleaved to interleaved color components.

It is possible that the component planes of an input YCbCr image have been previously downsampled. If so, the **-YCbCr** flag must be included on the command line, listing the appropriate component plane downsampling factors. By default, this utility assumes no downsampling. (See YCbCr OPTIONS below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_in** may be written **-r**.

<*outext*>
the extension of the output file. To construct the output filename, **intr2not** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input file, either an IHead file or raw pixmap file, containing the color image to be converted.

-raw_in *w*,*h*,*d*,[*ppi*]
the attributes of the input image. This option must be included on the command line if the input is a raw pixmap file.

w	the pixel width of the pixmap
h	the pixel height of the pixmap
d	the pixel depth of the pixmap
ppi	the optional scan resolution of the image in integer units of pixels per inch.

-YCbCr *H0*,*V0*:*H1*,*V1*:*H2*,*V2*
indicates that a YCbCr color image is being input whose component planes have been previously downsampled. (See YCbCr Options below.)

YCbCr OPTIONS

A common compression technique for YCbCr images is to downsample the Cb & Cr component planes. **Intr2not** can handle a limited range of YCbCr downsampling schemes that are represented by a list of component plane factors. These factors together represent downsampling ratios relative to each other. The comma-separated list of factor pairs correspond to the Y, Cb, and Cr component planes respectively. The

first value in a factor pair represents the downsampling of that particular component plane in the X-dimension, while the second represents the Y-dimension. Compression ratios for a particular component plane are calculated by dividing the maximum component factors in the list by the current component's factors. These integer factors are limited between 1 and 4. H,V factors all set to 1 represent no downsampling. For complete details, **intr2not** implements the downsampling and interleaving schemes described in the following reference:

> W.B. Pennebaker and J.L. Mitchell, "JPEG: Still Image Compression Standard," Appendix A - "ISO DIS 10918-1 Requirements and Guidelines," Van Nostrand Reinhold, NY, 1993, pp. A1-A4.

For example the option specification:

> -YCbCr 4,4:2,2:1,1

indicates that there has been no downsampling of the Y component plane (4,4 are the largest X and Y factors listed); the Cb component plane has been downsampled in X and Y by a factor of 2 (maximum factors 4 divided by current factors 2); and the Cr component plane has been downsampled in X and Y by a factor of 4 (maximum factors 4 divided by current factors 1). Note that downsampling component planes is a form of *lossy* compression. The utility **rgb2ycc** converts RGB pixmaps to the YCbCr colorspace, and it conducts downsampling of the resulting YCbCr component planes upon request.

EXAMPLES

From *test/imgtools/execs/intr2not/intr2not.src*:

> **% intr2not nin face.raw -r 768,1024,24**
> converts the interleaved RGB pixels of a face image in a raw pixmap file into separate color component planes.

SEE ALSO

not2intr(1G), **rgb2ycc**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

jpegtran – lossless transformation of JPEG files

SYNOPSIS

jpegtran [*options*] [*filename*]

DESCRIPTION

jpegtran performs various useful transformations of JPEG files. It can translate the coded representation from one variant of JPEG to another, for example from baseline JPEG to progressive JPEG or vice versa. It can also perform some rearrangements of the image data, for example turning an image from landscape to portrait format by rotation.

jpegtran works by rearranging the compressed data (DCT coefficients), without ever fully decoding the image. Therefore, its transformations are lossless: there is no image degradation at all, which would not be true if you used **djpeg** followed by **cjpeg** to accomplish the same conversion. But by the same token, **jpegtran** cannot perform lossy operations such as changing the image quality.

jpegtran reads the named JPEG/JFIF file, or the standard input if no file is named, and produces a JPEG/JFIF file on the standard output.

OPTIONS

All switch names may be abbreviated; for example, **–optimize** may be written **–opt** or **–o**. Upper and lower case are equivalent. British spellings are also accepted (e.g., **–optimise**), though for brevity these are not mentioned below.

To specify the coded JPEG representation used in the output file, **jpegtran** accepts a subset of the switches recognized by **cjpeg**:

–optimize
Perform optimization of entropy encoding parameters.

–progressive
Create progressive JPEG file.

–restart *N*
Emit a JPEG restart marker every N MCU rows, or every N MCU blocks if "B" is attached to the number.

–scans *file*
Use the scan script given in the specified text file.

See **cjpeg**(1) for more details about these switches. If you specify none of these switches, you get a plain baseline-JPEG output file. The quality setting and so forth are determined by the input file.

The image can be losslessly transformed by giving one of these switches:

–flip horizontal
Mirror image horizontally (left-right).

–flip vertical
Mirror image vertically (top-bottom).

–rotate 90
Rotate image 90 degrees clockwise.

–rotate 180
Rotate image 180 degrees.

–rotate 270
Rotate image 270 degrees clockwise (or 90 ccw).

–transpose
Transpose image (across UL-to-LR axis).

−transverse
Transverse transpose (across UR-to-LL axis).

The transpose transformation has no restrictions regarding image dimensions. The other transformations operate rather oddly if the image dimensions are not a multiple of the iMCU size (usually 8 or 16 pixels), because they can only transform complete blocks of DCT coefficient data in the desired way.

jpegtran's default behavior when transforming an odd-size image is designed to preserve exact reversibility and mathematical consistency of the transformation set. As stated, transpose is able to flip the entire image area. Horizontal mirroring leaves any partial iMCU column at the right edge untouched, but is able to flip all rows of the image. Similarly, vertical mirroring leaves any partial iMCU row at the bottom edge untouched, but is able to flip all columns. The other transforms can be built up as sequences of transpose and flip operations; for consistency, their actions on edge pixels are defined to be the same as the end result of the corresponding transpose-and-flip sequence.

For practical use, you may prefer to discard any untransformable edge pixels rather than having a strange-looking strip along the right and/or bottom edges of a transformed image. To do this, add the **−trim** switch:

−trim Drop non-transformable edge blocks.

Obviously, a transformation with **−trim** is not reversible, so strictly speaking **jpegtran** with this switch is not lossless. Also, the expected mathematical equivalences between the transformations no longer hold. For example, **−rot 270 -trim** trims only the bottom edge, but **−rot 90 -trim** followed by **−rot 180 -trim** trims both edges.

Another not-strictly-lossless transformation switch is:

−grayscale
Force grayscale output.

This option discards the chrominance channels if the input image is YCbCr (ie, a standard color JPEG), resulting in a grayscale JPEG file. The luminance channel is preserved exactly, so this is a better method of reducing to grayscale than decompression, conversion, and recompression. This switch is particularly handy for fixing a monochrome picture that was mistakenly encoded as a color JPEG. (In such a case, the space savings from getting rid of the near-empty chroma channels won't be large; but the decoding time for a grayscale JPEG is substantially less than that for a color JPEG.)

jpegtran also recognizes these switches that control what to do with "extra" markers, such as comment blocks:

−copy none
Copy no extra markers from source file. This setting suppresses all comments and other excess baggage present in the source file.

−copy comments
Copy only comment markers. This setting copies comments from the source file, but discards any other inessential data.

−copy all
Copy all extra markers. This setting preserves miscellaneous markers found in the source file, such as JFIF thumbnails and Photoshop settings. In some files these extra markers can be sizable.

The default behavior is **−copy comments**. (Note: in IJG releases v6 and v6a, **jpegtran** always did the equivalent of **−copy none**.)

Additional switches recognized by jpegtran are:

−maxmemory *N*
Set limit for amount of memory to use in processing large images. Value is in thousands of bytes, or millions of bytes if "M" is attached to the number. For example, **−max 4m** selects 4000000 bytes. If more space is needed, temporary files will be used.

−outfile *name*
Send output image to the named file, not to standard output.

−verbose
Enable debug printout. More **−v**'s give more output. Also, version information is printed at startup.

−debug
Same as **−verbose**.

EXAMPLES

This example converts a baseline JPEG file to progressive form:

jpegtran −progressive *foo.jpg* **>** *fooprog.jpg*

This example rotates an image 90 degrees clockwise, discarding any unrotatable edge pixels:

jpegtran −rot 90 -trim *foo.jpg* **>** *foo90.jpg*

ENVIRONMENT

JPEGMEM
If this environment variable is set, its value is the default memory limit. The value is specified as described for the **−maxmemory** switch. **JPEGMEM** overrides the default value specified when the program was compiled, and itself is overridden by an explicit **−maxmemory**.

SEE ALSO

cjpeg(1H), **djpeg**(1H), **rdjpgcom**(1H), **wrjpgcom**(1H)
Wallace, Gregory K. "The JPEG Still Picture Compression Standard", Communications of the ACM, April 1991 (vol. 34, no. 4), pp. 30-44.

AUTHOR

Independent JPEG Group

BUGS

Arithmetic coding is not supported for legal reasons.

The transform options can't transform odd-size images perfectly. Use **−trim** if you don't like the results without it.

The entire image is read into memory and then written out again, even in cases where this isn't really necessary. Expect swapping on large images, especially when using the more complex transform options.

NAME

kltran – runs a Karhunen-Loeve transform on a set of vectors.

SYNOPSIS

kltran *<vecsfile_in[vecsfile_in...]> <mean file> <tranmat_file> <nrows_use> <vecsfile_out> <vecsfile_out_desc> <ascii_outfile> <message_freq>*

DESCRIPTION

Kltran runs a Karhunen-Loeve transform on a set of vectors and reduces the dimensionality of the feature vectors using the given basis set *tranmat_file*.

If several processors are available, it may be possible to save time, when transforming a large set of vectors. First, run several simultaneous instances of kltran, each instance transforming a subset of the vectors. Then, use stackms to combine the resulting output files, in the sense of stacking together the matrices. See the stackms man page.

OPTIONS

<vecsfile_in[vecsfile_in...]>

Input data file(s) in PCASYS "matrix" format, each consisting of a block of the vectors that are to be transformed. The input vectors are the rows. All input vectors must have the same number of elements, so the second dimensions of these files (if more than one file) must all be equal. (Usually the output of the **mkoas** command.)

<mean file>

Input mean vector that gets subtracted from all the input feature vectors before using the transform matrix. (Usually the output of **meancov** command.)

<tranmat_file>

A PCASYS "matrix" file containing a transform matrix, some of whose rows are to be used (see next argument). The second dimension of the transform matrix must equal the second dimension of the file(s) of input vectors. (Usually the output of the **eva_evt** or **mktran** commands.)

<nrows_use>

How many (first) rows of the transform matrix are to be used. This is how many elements each output vector will have.

<vecsfile_out>

The output vectors, stacked together as a PCASYS "matrix" file, each vector being one row of the matrix.

<vecsfile_out_desc>

A string to be written into the output file as its description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quotes, i.e. single–quoted empty string). To let kltran make a description (indicating that kltran was used, and listing the names of the file(s) of input vectors and of the transform matrix file), use – (hyphen).

<ascii_outfile>

If y, makes an ascii output file; if n, binary. Binary is recommended, unless the output file must be portable across different byte orders or floating–point formats.

<message_freq>

If a positive integer, then every this many vectors through each input file kltran writes a progress message to the standard output. If zero, no messages.

EXAMPLE(S)

From *test/pcasys/execs/kltran/kltran.src*:

% kltran ../mkoas/sv10.oas ../meancov/fv1-9.men ../eva_evt/fv1-9.evt 128 sv10.kls - n 100
Does transformation using a eigen-vector set made by the **eva_evt** command.

SEE ALSO

lintran (1B), asc2bin (1B), bin2asc (1B), eva_evt (1B), mkoas (1B), mktran (1B), stackms (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

lintran – runs a linear transform on a set of vectors.

SYNOPSIS

lintran *<vecsfile_in[vecsfile_in...]> <tranmat_file> <nrows_use> <vecsfile_out> <vecsfile_out_desc> <ascii_outfile> <message_freq>*

DESCRIPTION

Lintran runs a linear transform on a set of vectors and reduces the dimensionality of the feature vectors using the given basis set *tranmat_file*.

If several processors are available, it may be possible to save time, when transforming a large set of vectors. First, run several simultaneous instances of lintran, each instance transforming a subset of the vectors. Then, use stackms to combine the resulting output files, in the sense of stacking together the matrices. See the stackms man page.

OPTIONS

<vecsfile_in[vecsfile_in...]>

Input data file(s) in PCASYS "matrix" format, each consisting of a block of the vectors that are to be transformed. The input vectors are the rows. All input vectors must have the same number of elements, so the second dimensions of these files (if more than one file) must all be equal. (Usually the output of the **mkoas** command.)

<tranmat_file>

A PCASYS "matrix" file containing a transform matrix, some of whose rows are to be used (see next argument). The second dimension of the transform matrix must equal the second dimension of the file(s) of input vectors. (Usually the output of the **eva_evt** or **mktran** commands.)

<nrows_use>

How many (first) rows of the transform matrix are to be used. This is how many elements each output vector will have.

<vecsfile_out>

The output vectors, stacked together as a PCASYS "matrix" file, each vector being one row of the matrix.

<vecsfile_out_desc>

A string to be written into the output file as its description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quotes, i.e. single–quoted empty string). To let lintran make a description (indicating that lintran was used, and listing the names of the file(s) of input vectors and of the transform matrix file), use – (hyphen).

<ascii_outfile>

If y, makes an ascii output file; if n, binary. Binary is recommended, unless the output file must be portable across different byte orders or floating–point formats.

<message_freq>

If a positive integer, then every this many vectors through each input file lintran writes a progress message to the standard output. If zero, no messages.

EXAMPLE(S)

From *test/pcasys/execs/lintran/lintran.src*:

% lintran ../mkoas/sv10.oas ../eva_evt/fv1-9.evt 128 sv10mlp.kls - n 100
Does transformation using a eigen-vector set made by the **eva_evt** command. Used by MLP classifier.

% lintran ../mkoas/sv10.oas ../mktran/fv1-9.opt 64 sv10pnn.kls - n 100
Does transformation using a set of eigen-vectors that were adjusted using the **optrws** and **mktran** commands. Used by the PNN classifier.

SEE ALSO

asc2bin (1B), bin2asc (1B), eva_evt (1B), mkoas (1B), mktran (1B), stackms (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

meancov – computes mean vector and covariance matrix for a set of feature vectors.

SYNOPSIS

meancov *<vecsfile_in[vecsfile_in...]> <meanfile_out> <meanfile_out_desc> <covfile_out> <covfile_out_desc> <ascii_outfiles> <message_freq>*

DESCRIPTION

Meancov computes sample mean vector and sample covariance matrix of a set of feature vectors.

If several processors are available, it may be possible to save time, when computing the mean and covariance of a large set of feature vectors. First, run several simultaneous instances of meancov, each instance computing the mean and covariance of a subset of the vectors. Then, use cmbmcs to combine the resulting output files. See the cmbmcs man page. **Note:** If using cmbmcs, the subset mean vectors made by the meancov instances must be saved for later use by cmbmcs even if, ultimately, all that is wanted is the overall covariance matrix. Construction of the overall covariance requires the subset means, as well as the subset covariances.

OPTIONS

<vecsfile_in[vecsfile_in...]>

Input data file(s) in PCASYS "matrix" format, each consisting of a block of the vectors that are to be used, i.e. the vectors are the rows of the matrix (matrices). Of course, all input matrices must have the same second dimension, which is the dimension of the constituent vectors. (Usually the output of **mkoas**.)

<meanfile_out>

Mean file to be written, in PCASYS "matrix" format, with first dimension set to 1 and with second dimension set to the dimension of the input vectors.

<meanfile_out_desc>

A string to be written into the mean output file as its description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use ’’ (two single quotes, i.e. single–quoted empty string). To let meancov make a description (stating that this is a mean vector made by meancov and listing the names of the input files), use – (hyphen).

<covfile_out>

Covariance file to be written. Meancov saves memory and cycles by allocating a buffer only large enough for the nonstrict lower triangle of the symmetric covariance matrix and computing only those elements, and it saves disk space by storing the covariance in PCASYS "covariance" format, which stores only the nonstrict lower triangle. The order of the covariance is the dimension of the input vectors.

<covfile_out_desc>

Description string for covariance file or – to let meancov make the description, same as for the mean file description argument.

<ascii_outfiles>

If y, makes ascii output files; if n, binary. Binary is recommended, unless the output files must be portable across different byte orders or floating–point formats.

<message_freq>

If a positive integer, then every this many vectors through each input file, during the accumulation phase, meancov writes a progress message to the standard output, and it also writes a few other progress messages. If 0, no messages.

EXAMPLE(S)

From *test/pcasys/execs/meancov/meancov.src*:

% meancov ../../data/oas/fv[1-9].oas fv1-9.men - fv1-9.cov - n 100
Compute the mean and covariance matrices for a set of feature vectors.

SEE ALSO

cmbmcs (1B), mkoas (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

mindtct – detects minutiae from a fingerprint image that is either an ANSI/NIST 2000 formatted file or a WSQ compressed file.

SYNOPSIS

mindtct *[-b] [-m1] <finger_img_in> <oroot>*

DESCRIPTION

Mindtct takes either a WSQ compressed image file or parses a standard compliant ANSI/NIST-ITL 1-2000 file searching for the first occurrence of a grayscale fingerprint image record. The fingerprint image is processed and minutiae are automatically detected.

If the input file was in ANSI/NIST 2000 format the minutiae results are formatted and stored using the NIST fields 5-12 in a Type-9 record. Upon successful completion, the input ANSI/NIST record sequence is augmented with two new records, the Type-9 minutiae record and a tagged field image record containing the results of image binarization. This augmented record sequence is then written to the output file <oroot>.mdt. The minutiae values are also written to a text file <oroot>.xyt in "x y theta quality" format. This <oroot>.xyt file is the format used by the bozorth3 matcher.

If the input image is a WSQ compressed file the minutiae are only written to the text file <oroot>.xyt. In addition a file called <oroot>.brw is created which is a raw pixel file of the binarized image created by mindtct when detecting the minutiae points.

The default is minutiae points computed based on the pixel origin being at the bottom left of the image and directions are pointing out and away from the ridge ending or bifurcation valley.

Mindtct also generates the following text files in the current working directory: *<oroot>.dm*, *<oroot>.hcm*, *<oroot>.lcm*, *<oroot>.lfm*, *<oroot>.qm*, and *<oroot>.min*. These files are described below.

OPTIONS

[-b] perform image enhancement on low contrast images. Only affects low contrast images, others are unchanged.

[-m1] write the minutiae points (in files <oroot>.{mdt,xyt,min}) according to ANSI INCITS 378-2004. This format has the pixel origin at the top left of the image and directions are pointing up the ridge ending or bifurcation valley. The default for **mindtct** has the pixel origin at the bottom left of the image and directions are pointing out and away from the ridge ending or bifurcation valley. NOTE: If this flag is used when extracting the mintuiae points it must also be used with the **bozorth3** matcher.

<finger_img_in>
the fingerprint file to be processed

<oroot>
the root name for the output files (<oroot>.???)

TEXT OUTPUT FILES

<oroot>.dm

The *Direction Map* represents the direction of ridge flow within the fingerprint image. The map contains a grid of integer directions, where each cell in the grid represents an 8x8 pixel neighborhood in the image. Ridge flow angles are quantized into 16 integer bi-directional units equally spaced on a semicircle. Starting with vertical direction 0, direction units increase clockwise and represent incremental jumps of 11.25 degrees, stopping at direction 15 which is 11.25 degrees shy of vertical. Using this scheme, direction 8 is horizontal. A value of -1 in this map represents a neighborhood where no valid ridge flow was determined.

<oroot>.hcm

The *High-Curvature Map* represents areas in the image having high-curvature ridge flow. This is especially true of core and delta regions in the fingerprint image, but high-curvature is not limited to just these cases. This is a bi-level map with same dimension as the Direction Map. Cell values of 1 represent 8x8 pixel neighborhoods in the fingerprint image that are located within a high-curvature region, otherwise cell values are set to 0.

<oroot>.lcm

The *Low-Contrast Map* represents areas in the image having low-contrast. The regions of low contrast most commonly represent the background in the fingerprint image. This is a bi-level map with same dimension as the Direction Map. Cell values of 1 represent 8x8 pixel neighborhoods in the fingerprint image that are located within a low-contrast region, otherwise cell values are set to 0.

<oroot>.lfm

The *Low-Flow Map* represents areas in the image having non-determinable ridge flow. Ridge flow is determined using a set of discrete cosine wave forms computed for a predetermined range of frequencies. These wave forms are applied at 16 incremental orientations. At times none of the wave forms at none of the orientations resonate sufficiently high within the region in the image to satisfactorily determine a dominant directional frequency. This is a bi-level map with same dimension as the Direction Map. Cell values of 1 represent 8x8 pixel neighborhoods in the fingerprint image that are located within a region where a dominant directional frequency could *not* be determined, otherwise cell values are set to 0. The Direction Map also records cells with non-determinable ridge flow. The difference is that the Low-Flow Map records *all* cells with non-determinable ridge flow, while the Direction Map records only those that remain non-determinable after extensive *interpolation* and *smoothing* of neighboring ridge flow directions.

<oroot>.qm

The *Quality Map* represents regions in the image having varying levels of quality. The maps above are combined heuristically to form 5 discrete levels of quality. This map has the same dimension as the Direction Map, with each value in the map representing an 8x8 pixel neighborhood in the fingerprint image. A cell value of 4 represents highest quality, while a cell value of 0 represent lowest possible quality.

<oroot>.xyt

This text file reports the minutiae detection results. This reports only the x,y coordinates, theta, and quality of the minutie points for the image. Each line in this file contains the space delimited information for one minutiae point. The <oroot>.xyt is the minutiae format used by the **bozorth3** matching algorithm.

<oroot>.min

This text file reports the minutiae detection results. The majority of the results listed in this text file are also encoded and stored in a Type-9 record in the output ANSI/NIST file. The first non-empty line in the text file lists the number of minutiae that were detected in the fingerprint image. Following this, the attributes associated with each detected minutia are recorded, one line of text per minutia. Each minutia line has the same format. Fields are separated by a ':', subfields are separated by a ';', and items within subfields are separated by a ','. A minutia line may be represented as:

MN : *MX*, *MY* : *DIR* : *REL* : *TYP* : *FTYP* : *FN* : *NX1*, *NY1*; *RC1* : ...

where:

MN is the integer identifier of the detected minutia.

MX is the x-pixel coordinate of the detected minutia.

MY is the y-pixel coordinate of the detected minutia.

DIR is the direction of the detected minutia. Minutia direction is represented similar to ridge flow direction, only minutia direction is uni-directional starting at vertical pointing up with unit 0 and increasing clockwise in increments of 11.25 degrees completing a full circle. Using this scheme, the angle of a detected minutia is quantized into the range 0 to 31 with 8 representing horizontal to the right, 16 representing vertical pointing down, and 24 representing horizontal to the left.

REL is the reliability measure assigned to the detected minutia. This measure is computed by looking up the quality level associated with the position of the minutia from the Quality Map. The quality level is then heuristically combined with simple neighborhood pixel statistics surrounding the minutia point. The results is a floating point value in the range 0.0 to 1.0, with 0.0 representing lowest minutia quality and 1.0 representing highest minutia quality.

TYP is the type of the detected minutia.
bifurcation = "BIF"
ridge ending = "RIG"

FTYP is the type of feature detected.
appearing = "APP"
disappearing = "DIS"
(This attribute is primarily useful for purposes internal to the minutia detection algorithm.)

FN is the integer identifier of the type of feature detected. (This attribute is primarily useful for purposes internal to the minutia detection algorithm.)

NX1 is the x-pixel coordinate of the first neighboring minutia.

NY1 is the y-pixel coordinate of the first neighboring minutia.

RC1 is the ridge count calculated between the detected minutia and its first neighbor.

... for each additional neighbor ridge count computed, the pixel coordinate of the neighbor and the ridge count to that neighbor are reported.

EXAMPLES

From *test/mindtct/execs/mindtct/mindtct.src*:

```
% mindtct ../../data/g001t2u.eft g001t2u
```

SEE ALSO

an2k2txt(1F), **an2ktool**(1F), **dpyan2k**(1F), **bozorth3**(1E)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

mkoas – makes orientation arrays from fingerprint images.

SYNOPSIS

mkoas <*prsfile*>

DESCRIPTION

Mkoas makes orientation arrays (oas), for a set of fingerprint image files. An oa can be thought of as a 28 (height) by 30 (width) array of real two-dimensional orientation vectors, each of which represents the local average ridge/valley orientation at one point of an equally-spaced rectangular grid; but, sometimes it is more convenient to think of an oa as a single 1680-dimensional real vector (1680 = 28 x 30 x 2). To make an oa from a fingerprint, mkoas uses the same sequence of preprocessing/feature-extraction routines that is used by the classifier demos pcasys and pcasysx. Mkoas causes each oa to be one row of the PCASYS "matrix" file that is its output.

Mkoas sets the values of its parameters as follows. First, it reads the default oas-production parms file *pcasys/parms/oas.prs*; then, it reads the file of default values of additional mkoas parms, *pcasys/parms/mkoas.prs*; finally, it reads the required user parms file, which is the argument (*prsfile*). Each time a parms file is read, its values override those set by previously read parms file(s), if any. See PARAMETER FILES, below, for a description of the difference between *oas.prs* and *mkoas.prs*.

Since the oas of a large set of fingerprints can turn out to be quite a large amount of data, it may be that the entire set of oas that are to be produced cannot exist as a single file, because of disk space limitations. If so, one should run several instances of mkoas, each producing a matrix file that is a subset of the required oas. To estimate output file size as a function of number of oas, note that each oa consists of 1680 single-precision floating-point numbers, and therefore it takes 1680 x 4 = 6720 bytes. Allow slightly more space, for header data contained in a matrix file.

OPTIONS

<*prsfile*>

A file containing parameters. To find out what the available parameters are, and as examples of the format of parameters files, consult the default files *pcasys/parms/oas.prs* and *pcasys/parms/mkoas.prs*. Each parameter is specified by having its name and value on a line; a pound sign indicates that the rest of its line is a comment.

PARAMETER FILES

pcasys/parms/oas.prs

Contains default values of the parameters that affect the making of orientation arrays (oas): these are the parms of the segmentor (sgmnt), the image enhancer (enhnc), the ridge-valley orientation finder (rors), the registration program (r92a), and the registration-implementing pixelwise orientations reaverager (rgar). The values used for these parms when making the oas used in optimizing the classifier should also be used when running the finished classifier.

Default settings in *pcasys/parms/oas.prs*

Used in the segmentation routine:

sgmnt_fac_n *5*

How many threshold-making factors to try.

sgmnt_min_fg *2000*

Minimum allowed number of foreground (true) pixels.

sgmnt_max_fg *8000*

Maximum allowed number of foreground (true) pixels.

sgmnt_nerode *3*

Do this many erosions in foreground cleanup.

sgmnt_rsblobs *1*
If 1, remove small blobs in foreground cleanup.

sgmnt_fill *1*
If 1, fill holes in rows, columns in foreground cleanup.

sgmnt_min_n *25*
Cutting angle becomes zero if any foreground edge has fewer than this many pixels.

sgmnt_hist_thresh *20*
Threshold that tilted-rows-histogram must meet to find top-location for cutting.

sgmnt_origras_wmax *2000*
Maximum allowed width of original raster.

sgmnt_origras_hmax *2000*
Maximum allowed height of original raster.

sgmnt_fac_min *0.75*
Minimum threshold-making factor value.

sgmnt_fac_del *0.05*
Delta of threshold-making factor value.

sgmnt_slope_thresh *0.90*
If any of the three edges has slope differing by more than this from the average of the slopes, then cutting angle is set to zero.

Used in the FFT image enhancer:

enhnc_rr1 *150*
High-frequency elements of FFT whose filter plane value is less than this value are discarded.

enhnc_rr2 *449*
Low-frequency elements of FFT whose filter plane number is greater than this value are discarded.

enhnc_pow *0.3*
Power spectrum is raised to this power before it is multiplied by the FFT output.

Used in the ridge-valley orientation finder:

rors_slit_range_thresh *10*
If the difference between the maximum and minimum slit-sums at a pixel is less than this, then this pixel makes no contribution to the histogram used to make the local average orientation.

Used in the r92a wrapper for r92 registration program:

r92a_discard_thresh *0.01*
If squared-length of a local-average orientation vector is less than this, then conversion of this vector to an angle for use by r92 just produces the special value 100., which means an undefined angle.

Used in the registering pixelwise-orientations-reaverager:

rgar_std_corepixel_x *245*
X coordinate of standard (median) core position.

rgar_std_corepixel_y *189*
Y coordinate of standard (median) core position. This is the standard registration point, to which the particular core point gets translated to implement registration.

pcasys/parms/mkoas.prs
Contains default values of additional parameters needed by mkoas, besides those appearing in *pcasys/parms/oas.prs*. Parameters without defaults values must appear in the users *prsfile*.

Default settings in *pcasys/parms/mkoas.prs*

ascii_oas *n*
Ascii (y) or binary (n) output?

update_freq *1*
Frequency of progress messages.

clobber_oas_file *n*
Overwrite an oas_file if it already exists?

proc_images_list *(no default, user must set)*
The list of fingerprint images to make orientation arrays from.

oas_file *(no default, user must set)*
The output file that is to be produced containing orientation arrays.

EXAMPLE(S)

From *test/pcasys/execs/mkoas/mkoas.src*:

% mkoas sv10.prs
Creates a set of orientation arrays based on the file list given in the parameters file *sv10.prs*.

SEE ALSO

bin2asc (1B), asc2bin (1B), chgdesc (1B), stackms (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

mktran – makes transform matrix from regional weights and eigenvectors.

SYNOPSIS

mktran <*regwts_file*> <*eigvecs_file*> <*n_eigvecs_use*> <*tranmat_file*> <*tranmat_file_desc*> <*ascii_outfile*>

DESCRIPTION

Mktran takes a matrix of regional weights, and a set of eigenvectors, and makes a transform matrix from the regional weights and the specified number of (first) eigenvectors. The resulting matrix is suitable for transforming an orientation array into a low–dimensional feature vector.

OPTIONS

<*regwts_file*>

Regional weights file in PCASYS "matrix" format. The dimensions must be 14x15, because that is the pattern of 2x2–vector blocks of orientation vectors. (Usually the output of **optrws**.)

<*eigvecs_file*>

Eigenvectors file in PCASYS "matrix" format. The first dimension is the number of eigenvectors contained in the file; the second dimension must be 1680, which is the dimension of an orientation array when it is thought of as a single vector. (Usually the output of **eva_evt**.)

<*n_eigvecs_use*>

The number (first) eigenvectors to be used. This will be the first dimension of the resulting transform matrix.

<*tranmat_file*>

Transform file to be made, in PCASYS "matrix" format. First dimension will be *n_eigvecs_use* and second dimension will be 1680.

<*tranmat_file_desc*>

A string to be written into the transform matrix output file as its description string. This string can be of any length, but must not contain embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quotes, i.e. single–quoted empty string). To let mktran make a description, use – (hyphen).

<*ascii_outfile*>

If y, makes an ascii output file; if n, binary. Binary is recommended, unless the output file must be portable across different byte orders or floating–point formats.

EXAMPLE(S)

From *test/pcasys/execs/mktran/mktran.src*:

% mktran ../optrws/optrws.bin ../eva_evt/fv1-9.evt 64 fv1-9.opt - n

Uses a set of optimized regional weights (*optrws.bin*) to adjust a eigen-vector basis set (*fv1-9.evt*) and create a new transformation matrix (*fv1-9.opt*) that is used to reduce the dimensionality of the feature vectors.

SEE ALSO

eva_evt (1B), lintran (1B), optrws (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

mlp – Does training and testing runs using a 3-layer feed-forward linear perceptron Neural Network.

SYNOPSIS

mlp *[-c] [specfile]*

DESCRIPTION

Mlp trains a 3-layer feed-forward linear perceptron using novel methods of machine learning that help control the learning dynamics of the network. As a result, the derived minima are superior, the decision surfaces of the trained network are well-formed, the information content of confidence values is increased, and generalization is enhanced. The theory behind the machine learning techniques used in this program is discussed in the following reference:

[C. L. Wilson, J. L. Blue, O. M. Omidvar, "The Effect of Training Dynamics on Neural Network Performance," NIST Internal Report 5696, August 1995.]

Machine learning is controlled through a batch-oriented iterative process of training the MLP on a set of prototype feature vectors, and then evaluating the progress made by running the MLP (in its current state) on a separate set of testing feature vectors. Training on the first set of patterns then resumes for a predetermined number of passes through the training data, and then the MLP is tested again on the evaluation set. This process of training and then testing continues until the MLP has been determined to have satisfactorily converged.

The MLP neural network is suitable for use as a classifier or as a function-approximator. The network has an input layer, a hidden layer, and an output layer, each layer comprising a set of nodes. The input nodes are feed-forwardly connected to the hidden nodes, and the hidden nodes to the output nodes, by connections whose weights (strengths) are trainable. The activation function used for the hidden nodes can be chosen to be sinusoid, sigmoid (logistic), or linear, as can the activation function for the output nodes. Training (optimization) of the weights is done using either a Scaled Conjugate Gradient (SCG) algorithm [1], or by starting out with SCG and then switching to a Limited Memory Broyden Fletcher Goldfarb Shanno (LBFGS) algorithm [2]. Boltzmann pruning [3], i.e. dynamic removal of connections, can be performed during training if desired. Prior weights can be attached to the patterns (feature vectors) in various ways.

[1] J. L. Blue and P. J. Grother, "Training Feed Forward Networks Using Conjugate Gradients," NIST Internal Report 4776, February 1992, and in Conference on Character Recognition and Digitizer Technologies, Vol. 1661, pp. 179-190, SPIE, San Jose, February 1992.

[2] D. Liu and J. Nocedal, "On the Limited Memory BFGS Method for Large Scale Optimization," IMathematical Programming B, Vol. 45, 503-528, 1989.

[3] O. M. Omidvar and C. L. Wilson, "Information Content in Neural Net Optimization," NIST Internal Report 4766, February 1992, and in *Journal of Connection Science*, 6:91-103, 1993.

Training and Testing Runs

When mlp is invoked, it performs a sequence of runs. Each run does either training, or testing:

training run: A set of patterns is used to train (optimize) the weights of the network. Each pattern consists of a feature vector, along with either a class or a target vector. A feature vector is a tuple of floating-point numbers, which typically has been extracted from some natural object such as a handwritten character. A class denotes the actual class to which the object belongs, for example the character which a handwritten mark is an instance of. The network can be trained to become a classifier: it trains using a set of feature vectors extracted from objects of known classes. Or, more generally, the network can be trained to learn, again from example input-output pairs, a function whose output is a vector of floating-point numbers, rather than a class; if this is done, the network is a sort of interpolator or function-fitter. A training run finishes by writing the final values of the network weights as a file. It also produces a summary file showing various information about the run, and optionally produces a longer file that shows the results the final (trained) network produced for each individual pattern.

testing run: A set of patterns is sent through a network, after the network weights are read from a file. The output values, i.e. the hypothetical classes (for a classifier network) or the produced output vectors (for a fitter network), are compared with target classes or vectors, and the resulting error rate is computed. The program can produce a table showing the correct classification rate as a function of the rejection rate.

OPTIONS

[-c] Only do error checking on the specfile parameters and print any warnings or errors that occur in the specfile format.

[specfile]

Specfile to be used by mlp. The default is a specfile named "spec" located in the current working directory.

This is a file produced by the user, which sets the parameters (henceforth "parms") of the run(s) that mlp is to perform. It consists of one or more blocks, each of which sets the parms for one run. Each block is separated from the next one by the word "newrun" or "NEWRUN". Parms are set using name-value pairs, with the name and value separated by non-newline white space characters (blanks or tabs). Each name-value pair is separated from the next pair by newline(s) or semicolon(s). Since each parm value is labeled by its parm name, the name-value pairs can occur in any order. Comments are allowed; they are delimited the same way as in C language programs, with /* and */. Extraneous white space characters are ignored.

When mlp is run, it first scans the entire specfile, to find and report any (fatal) errors (e.g. omitting to set a necessary parm, or using an illegal parm name or value) and also any conditions in the specfile which, although not fatally erroneous, are worthy of warnings (e.g. setting a superfluous parm). Mlp writes any applicable warning or error messages; then, if there are no errors in the specfile, it starts to perform the first run. Warnings do not prevent mlp from starting to run. The motivation for having mlp check the entire specfile before it starts to perform even the first run, is that this will prevent an mlp instance that runs a multi-run specfile from failing, perhaps many hours, or days, after it was started, because of an error in a block far into the specfile: such errors will be detected up front and presumably fixed by the user, because that is the only way to cause mlp to get past its checking phase. To cause mlp only to check the specfile without running it, use the -c option.

The following listing describes all the parms that can be set in a specfile. There are four types of parms: string (value is a filename), integer, floating-point, and switch (value must be one of a set of defined names, or may be specified as a code number). A block of the specfile, which sets the parms for one run, often can omit to set the values of several of the parms, either because the parm is unneeded (e.g., a training "stopping condition" when the run is a test run; or, **temperature** when boltzmann is *no_prune*), or because it is an architecture parm (**purpose, ninps, nhids, nouts, acfunc_hids, or acfunc_outs**), whose value will be read from **wts_infile**. The descriptions below indicate which of the parms are needed only for training runs (in particular, those described as stopping conditions). Architecture parms should be set in a specfile block only if its run is to be a training run that generates random initial network weights: a training run that reads initial weights from a file (typically, final weights produced by a previous training session), or a test run (must read the network weights from a file), does not need to set any of the architecture parms in its specfile block, because their values are stored in the weights file that it will read. (Architecture parms are ones whose values it would not make sense to change between training runs of a single network that together comprise a training "meta-run", nor between a training run for a network and a test run of the finished network.) Setting unneeded parms in a specfile block will result in warning messages when mlp is run, but not fatal errors; the unneeded values will be ignored.

If a parm-name/parm-value pair occurring in a specfile has just its value deleted, i.e. leaving just a parm name, then the name is ignored by mlp; this is a way to temporarily unset a parm while leaving its name visible for possible future use.

String Parms (Filename)

short_outfile

This file will contain summary information about the run, including a history of the training process if a training run. The set of information to be written is controlled, to some extent, by the switch parms **do_confuse** and **do_cvr**.

long_outfile

This optionally produced file will have two lines of header information followed by a line for each pattern. The line will show: the sequence number of the pattern; the correct class of the pattern (as a number in the range 1 through **nouts**); whether the hypothetical class the network produced for this pattern was right (R) or wrong (W); the hypothetical class (number); and the **nouts** output-node activations the network produced for the pattern. (See the switch parm **show_acs_times_1000** below, which controls the formatting of the activations.) In a testing run, mlp produces this file for the result of running the patterns through the network whose weights are read from **wts_infile**; in a training run, mlp produces this file only for the final network weights resulting from the training session. This is often a large file; to save disk space by not producing it, just leave the parm unset.

patterns_infile

This file contains patterns upon which mlp is to train or test a network. A pattern is either a feature-vector and an associated class, or a feature-vector and an associated target-vector. The file must be in one of the two supported patterns-file formats, i.e. ASCII and (FORTRAN-style) binary; the switch parm **patsfile_ascii_or_binary** must be set to tell mlp which of these formats is being used.

wts_infile

This optional file contains a set of network weights. Mlp can read such a file at the start of a training run - e.g., final weights from a preceding training run, if one is training a network using a sequence of runs with different parameter settings (e.g., decreasing values of **regfac**) - or, in a testing run, it can read the final weights resulting from a training run. This parm should be left unset if random initial weights are to be generated for a training run (see the integer parm **seed**).

wts_outfile

This file is produced only for a training run; it contains the final network weights resulting from the run.

lcn_scn_infile

Each line of this optional file should consist of a long class-name (as shown at the top of **patterns_infile**) and a corresponding short class-name (1 or 2 characters), with the two names separated by white space; the lines can be in any order. This file is required only for a run that requires short class-names, i.e. only if **purpose** is *classifier* and (1) **priors** is *class* or *both* (these settings of **priors** require class-weights to be read from **class_wts_infile**, and that type of file can be read only if the short class-names are known) or (2) **do_confuse** is *true* (proper output of confusion matrices requires the short class-names, which are used as labels).

class_wts_infile

This optional file contains class-weights, i.e. a "prior weight" for each class. (See switch **parm** priors to find out how mlp can use these weights.) Each line should consist of a short class-name (as shown in **lcn_scn_infile**) and the weight for the class, separated by

white space; the order of the lines does not matter.

pattern_wts_infile

This optional file contains pattern-weights, i.e. a "prior weight" for each pattern. (See switch parm **priors** to find out how mlp can use these weights.) The file should be just a sequence of floating-point numbers (ascii) separated from each other by white space, with the numbers in the same order as the patterns they are to be associated with.

Integer Parms

npats

Number of (first) patterns from **patterns_infile** to use.

ninps, nhids, nouts

Specify the number of input, hidden, and output nodes in the network. If **ninps** is smaller than the number of components in the feature-vectors of the patterns, then the first **ninps** components of each feature-vector are used. If the network is a *classifier* (see **purpose**), then **nouts** is the number of classes, since there is one output node for each class. If the network is a *fitter*, then **ninps** and **nouts** are the dimensionalities of the input and output real vector spaces. These are architecture parms, so they should be left unset for a run that is to read a network weights file.

seed

For the UNI random number generator, if initial weights for a training run are to be randomly generated. Its values must be positive. Random weights are generated only if **wts_infile** is not set. (Of course, the **seed** value can be reused to generate identical initial weights in different training runs; or, it can be varied in order to do several training runs using the same values for the other parameters. It is often advisable to try several seeds, since any particular **seed** may produce atypically bad results (training may fail). However, the effect of varying the **seed** is minimal if Boltzmann pruning is used.)

niter_max

A stopping condition: maximum number of iterations a training run will be allowed to use.

nfreq

At every nfreq'th iteration during a training run, the **errdel** and **nokdel** stopping conditions are checked and a pair of status lines is written to the standard error output and to **short_outfile**.

nokdel

A stopping condition: stop if the number of iterations used so far is at least kmin and, for each of the most recent NNOT (defined in *src/lib/mlp/optchk.c*) sequences of **nfreq** iterations, the number right and the number right minus number wrong have both failed to increase by at least **nokdel** during the sequence.

lbfgs_mem

This value is used for the m argument of the LBFGS optimizer (if that optimizer is used, i.e. only if there is no Boltzmann pruning). This is the number of corrections used in the bfgs update. Values less than 3 are not recommended; large values will result in excessive

computing time, as well as increased memory usage. Values in the range 3 through 7 are recommended; value must be positive.

Floating-Point Parms

regfac

Regularization factor. The error value that a training run attempts to minimize, contains a term consisting of regfac times half the average of the squares of the network weights. (The use of a regularization factor often improves the generalization performance of a neural network, by keeping the size of the weights under control.) This parm must always be set, even for test runs (since they also compute the error value, which always uses **regfac**); however, its effect can be nullified by just setting it to 0.

alpha

A parm required by the **type_1** error function.

temperature

For Boltzmann pruning: see the switch parm **boltzmann**. A higher temperature causes more severe pruning.

egoal

A stopping condition: stop when error becomes less than or equal to **egoal**.

gwgoal

A stopping condition: stop when | **g** | / | **w** | becomes less than or equal to **gwgoal**, where **w** is the vector of network weights and **g** is the gradient vector of the error with respect to **w**.

errdel

A stopping condition: stop if the number of iterations used so far is at least kmin and the error has not decreased by at least a factor of **errdel** over the most recent block of **nfreq** iterations.

oklvl

The value of the highest network output activation produced when the network is run on a pattern (the position of this highest activation among the output nodes is the hypothetical class) can be thought of as a measure of confidence. This confidence value is compared with the threshold **oklvl**, in order to decide whether to classify the pattern as belonging to the hypothetical class, or to reject it, i.e. to consider its class to be unknown because of insufficient confidence that the hypothetical class is the correct class. The numbers and percentages of the patterns that *mlp* reports as *correct*, *wrong*, and *unknown*, are affected by **oklvl**: a high value of **oklvl** generally increases the number of unknowns (a bad thing) but also increases the percentage of the accepted patterns that are classified correctly (a good thing). If no rejection is desired, set **oklvl** to 0. (*Mlp* uses the single **oklvl** value specified for a run; but if the switch parm **do_cvr** is set to *true*, then *mlp* also makes a full *correct vs. rejected* table for the network (for the finished network if a training run). This table shows the (number correct) / (number accepted) and (number unknown) / (total number) percentages for each of several standard **oklvl** values.)

trgoff

This number sets how mildly the target values for network output activations vary

between their "low" and "high" values. If **trgoff** is 0 (least mild, i.e. most extreme, effect), then the low target value is 0 and the high, 1; if **trgoff** is 1 (most mild effect), then low and high targets are both (1 / **nouts**); if **trgoff** has an intermediate value between 0 and 1, then the low and high targets have intermediately mild values accordingly.

scg_earlystop_pct

This is a percentage that controls how soon a hybrid SCG/LBFGS training run (hybrid training can be used only if there is to be no Boltzmann pruning) switches from SCG to LBFGS. The switch is done the first time a check (checking every nfreq'th iteration) of the network results finds that every class-subset of the patterns has at least **scg_earlystop_pct** percent of its patterns classified correctly. A suggested value for this parm is 60.0.

lbfgs_gtol

This value is used for the gtol argument of the LBFGS optimizer. It controls the accuracy of the line search routine mcsrch. If the function and gradient evaluations are inexpensive with respect to the cost of the iteration (which is sometimes the case when solving very large problems) it may be advantageous to set **lbfgs_gtol** to a small value. A typical small value is 0.1. **Lbfgs_gtol** must be greater than 1.e-04.

Switch Parms

Each of these parms has a small set of allowed values; the value is specified as a string, or less verbosely, as a code number (shown in parentheses after string form):

train_or_test

train *0*

Train a network, i.e. optimize its weights in the sense of minimizing an error function, using a training set of patterns.

test *1*

Test a network, i.e. read in its weights and other parms from a file, run it on a test set of patterns, and measure the quality of the resulting performance.

purpose

Which of two possible kinds of engine the network is to be. This is an architecture parm, so it should be left unset for a run that is to read a network weights file. The allowed values are:

classifier *0*

The network is to be trained to map any feature vector to one of a small number of classes. It is to be trained using a set of feature vectors and their associated correct classes.

fitter *1*

The network is to be trained to approximate an unknown function that maps any input real vector to an output real vector. It is to be trained using a set of input-vector/output-vector pairs of the function. **NOTE: this is not currently supported.**

errfunc

Type of error function to use (always with the addition of a regularization term, consisting of **regfac** times half the average of the squares of the network weights).

mse *0*

Mean-squared-error between output activations and target values, or its equivalent computed using classes instead of target vectors. This is the recommended error function.

type_1 *1*

Type 1 error function; requires floating-point parm **alpha** be set. (Not recommended.)

pos_sum *2*

Positive sum error function. (Not recommended.)

boltzmann

Controls whether Boltzmann pruning of network weights is to be done and, if so, the type of threshold to use:

no_prune *0*

Do no Boltzmann pruning.

abs_prune *2*

Do Boltzmann pruning using threshold exp(- |**w**| / **T**), where **w** is a network weight being considered for possible pruning and **T** is the Boltzmann **temperature**.

square_prune *3*

Do Boltzmann pruning using threshold exp(- **w^2** / **T**), where **w** and **T** are as above.

acfunc_hids, acfunc_outs

The types of *activation functions* to be used on the hidden nodes and on the output nodes (separately settable for each layer). These are architecture parms, so they should be left unset for a run that is to read a network weights file. The allowed values are:

sinusoid *0*

f(x) = 0.5 * (1 + sin(0.5 * x))

sigmoid *1*

f(x) = 1 / (1 + exp(-x)) (Also called logistic function.)

linear *2*

f(x) = 0.25 * x

priors

What kind of prior weighting to use to set the final pattern-weights, which control the relative amounts of impact the various patterns have when doing the computations. These final pattern-weights remain fixed for the duration of a training run, but of course they can be changed between training runs.

allsame *0*

Set each final pattern-weight to (1 / **npats**). (The simplest thing to do; appropriate if the set of patterns has a natural distribution.)

class *1*

Set each final pattern-weight to the class-weight of the class of the pattern concerned divided by **npats**. The class-weights are derived by dividing the given-class-weights, read from the **class_wts_infile**, by the derived-class-weights, computed for the current data set and the normalize them to sum to 1.0. (Appropriate if the frequencies of the several classes, in the set of patterns, are not approximately equal to the natural frequencies (prior probabilities), so as to compensate for that situation.)

pattern *2*

Set the final pattern-weights to values read from **pattern_wts_infile** divided by **npats**. (Appropriate if none of the other settings of priors does satisfactory calculations (one can do whatever calculations one desires), or if one wants to dynamically change these weights between sessions of training.)

both *3*

Set each final pattern-weight to the class-weight of the class of the pattern concerned, times the provided pattern-weight, and divided by **npats**; compute the class-weights as previously described in **class priors** and read pattern-weights from file **pattern_wts_infile**. (Appropriate if one wants to both adjust for unnatural frequencies, and dynamically change the pattern weights.)

patsfile_ascii_or_binary

Tells mlp which of two supported formats to expect for the patterns file that it will read at the start of a run. (If much compute time is being spent reading ascii patsfiles, it may be worthwhile to convert them to binary format: that causes faster reading, and the binary-format files are considerably smaller.)

ascii *0*

patterns_infile is in ascii format.

binary *1*

patterns_infile is in binary (FORTRAN-style binary) format.

do_confuse

true *1*

Compute the confusion matrices and miscellaneous information and include them in **short_outfile**.

false *0*

Do not compute the confusion matrices and miscellaneous information.

show_acs_times_1000

This parm need be set only if the run is to produce a **long_outfile**.

true *1*

Before recording the network output activations in **long_outfile**, multiply them by 1000 and round to integers.

false *0*

Record the activations as their original floating-point values.

do_cvr *(See the notes on* **oklvl***.)*

true *1*

Produce a correct-vs.-rejected table and include it in **short_outfile**.

false *0*
Do not produce a correct-vs.-rejected table.

EXAMPLE(S)

From *test/pcasys/execs/mlp/mlp.src*:

% mlp
Runs mlp assuming the default specfile ("spec") in the local directory.

% mlp myspecfile
Runs mlp using the specfile "myspecfile".

SEE ALSO

fixwts (1B), mlpfeats (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

mlpfeats – converts PCASYS formatted feature files into the format compatible with the mlp command line function.

SYNOPSIS

mlpfeats <*feats_file*> <*class_file*> <*mlp_feats_file*>

DESCRIPTION

Mlpfeats is used to convert a PCASYS formatted feature file from the lintran function into the format used by the mlp command. PCASYS keeps the feature file and class file in separate files and mlp stores them in a single file with different header information format. At this point it was decided to just convert the format from PCASYS features to mlp features file format, any future release may try and resolve this problem into a single format.

OPTIONS

<*feats_file*>

The PCASYS formatted features file. (Usually the output of **lintran**.)

<*class_file*>

The PCASYS formatted class file.

<*mlp_feats_file*>

The MLP features file to be written, containing the PCASYS formatted feature and class files merged into a single MLP formatted file.

EXAMPLE(S)

From *test/pcasys/execs/mlpfeats/mlpfeats.src*:

% mlpfeats ../lintran/fv1-9mlp.kls ../../data/oas/fv1-9.cls fv1-9mlp.kls

Converts the feature file from the PCASYS file format to the mlp data file format (combines class and features into a single file.) It is unfortunate that two file formats exist but for now it is easier to keep both formats around.

SEE ALSO

lintran (1B), mlp (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

nfiq – evaluates a fingerprint image and returns a quality value for the image on a scale of 1 (highest quality) to 5 (lowest quality).

SYNOPSIS

nfiq <*fingimage in*> *[-d,-v]*

DESCRIPTION

Nfiq takes as input a file containing a fingerprint image. The image file can be in ANSI/NIST or NIST IHEAD format or compressed with WSQ, baseline JPEG, or lossless JPEG compression. **nfiq** uses the image maps generated by **mindtct** to create a feature vector for the image which is passed to an multi-layer perceptron (MLP) neural network. The MLP NN returns an activation value that is used to rank the images quality on a scale of 1 (highest quality) to 5 (lowest quality).

OPTIONS

-b *default*

prints the image quality value (1-5) to standard output.

-v *verbose*

prints all the feature vectors, image quality value (1-5) and neural network activiation values (0.0-1.0) to standard output.

<*fingimage in*>

the input fingerprint image to be evaluated.

EXAMPLES

From *test/nfiq/execs/nfiq/nfiq.src*:

% nfiq ../../data/a011_02_02.wsq -d >& nfiq.log
prints the image quality value (1-5) to standard output.

% nfiq ../../data/a011_02_02.wsq -v >& nfiq.log
prints all the feature vectors, image quality value (1-5) and neural network activiation values (0.0-1.0) to standard output.

SEE ALSO

mindtct(1C), **bozorth3**(1E)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

not2intr – converts an image comprised of non-interleaved color component planes to an image with interleaved color component pixels.

SYNOPSIS

not2intr <*outext*> <*image file*> <*w*,*h*,*d*,[*ppi*]>
[**-raw_out**]
[**-YCbCr** *H0*,*V0*:*H1*,*V1*:*H2*,*V2*]

DESCRIPTION

Not2intr takes as input a raw pixmap file containing an uncompressed image comprised of non-interleaved color component planes and converts the image to interleaved color component pixels. This utility requires there be three color components in the input image. Two output file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** option).

For example, the pixels of an RGB color image are interleaved when a pixel's R, G, and B components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utility **intr2not** converts interleaved to non-interleaved color components.

It is possible that the component planes of an input YCbCr image have been previously downsampled. If so, the **-YCbCr** flag must be included on the command line, listing the appropriate component plane downsampling factors. By default, this utility assumes no downsampling. YCbCr image results should always be explicitly stored in a raw pixmap file, because the IHead format only supports RGB pixels. (See YCbCr OPTIONS below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
the extension of the output file. To construct the output filename, **not2intr** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input raw pixmap file containing the color image to be converted.

<*w*,*h*,*d*,[*ppi*]>
the attributes of the input image in the raw pixmap file.

w	the pixel width of the pixmap
h	the pixel height of the pixmap
d	the pixel depth of the pixmap
ppi	the optional scan resolution of the image in integer units of pixels per inch.

-raw_out
specifies that the results should be stored to a raw pixmap file.

-YCbCr *H0*,*V0*:*H1*,*V1*:*H2*,*V2*
indicates that a YCbCr color image is being input whose component planes have been previously downsampled. The **-raw_out** flag should always be used in conjunction with this option. (See YCbCr Options below.)

YCbCr OPTIONS

A common compression technique for YCbCr images is to downsample the Cb & Cr component planes. **Not2intr** can handle a limited range of YCbCr downsampling schemes that are represented by a list of component plane factors. These factors together represent downsampling ratios relative to each other. The comma-separated list of factor pairs correspond to the Y, Cb, and Cr component planes respectively. The first value in a factor pair represents the downsampling of that particular component plane in the X-dimension, while the second represents the Y-dimension. Compression ratios for a particular component plane are calculated by dividing the maximum component factors in the list by the current component's factors. These integer factors are limited between 1 and 4. H,V factors all set to 1 represent no downsampling. For complete details, **not2intr** implements the downsampling and interleaving schemes described in the following reference:

> W.B. Pennebaker and J.L. Mitchell, "JPEG: Still Image Compression Standard," Appendix A - "ISO DIS 10918-1 Requirements and Guidelines," Van Nostrand Reinhold, NY, 1993, pp. A1-A4.

For example the option specification:

> -YCbCr 4,4:2,2:1,1

indicates that there has been no downsampling of the Y component plane (4,4 are the largest X and Y factors listed); the Cb component plane has been downsampled in X and Y by a factor of 2 (maximum factors 4 divided by current factors 2); and the Cr component plane has been downsampled in X and Y by a factor of 4 (maximum factors 4 divided by current factors 1). Note that downsampling component planes is a form of *lossy* compression. The utility **rgb2ycc** converts RGB pixmaps to the YCbCr colorspace, and it conducts downsampling of the resulting YCbCr component planes upon request.

EXAMPLES

From *test/imgtools/execs/not2intr/not2intr.src*:

> **% not2intr raw face.nin 768,1024,24 -r**
> converts the non-interleaved RGB face image in a raw pixmap file into interleaved color pixels.

SEE ALSO

intr2not(1G), **rgb2ycc**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

oas2pics – makes pictures of orientation arrays.

SYNOPSIS

oas2pics *<oasfile> <i_start> <i_finish outpics_dir> <verbose>*

DESCRIPTION

Oas2pics reads a specified segment of orientation arrays (oas) from a file, and makes IHead raster images depicting the oas. This can be useful for testing whether oas are reasonable, and to find out about their characteristics.

OPTIONS

<oasfile>

A PCASYS "matrix" file containing orientation arrays, with each row being one oa. First dimension is number of oas in the file, and second dimension must be 1680 (the dimensionality of one oa). (Usually the output of **mkoas**.)

<i_start> <i_finish>

The program makes pictures of the segment consisting of oas i_start through i_finish, numbering starting at 1.

<outpics_dir>

The program makes image files in this directory. (If the directory does not already exist, the program makes it.) The files will have names i.pct where i goes from i_start through i_finish.

<verbose>

If y, the program writes a progress message to stdout for each oa it is making a picture of.

EXAMPLE(S)

From *test/pcasys/execs/asc2bin/asc2bin.src*:

% oas2pics ../../data/oas/fv1.oas 1 2 oaspics y

Makes a set of images files (IHEAD format) so the user can see how the orientation arrays look and compare to the actual fingerprint image if desired. The files can be converted to JPEG format using the **cjegb** command.

SEE ALSO

mkoas (1B), dpyimage (1G), cjpegb (1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

optosf – optimizes the overall smoothing factor for the PNN classifier.

SYNOPSIS

optosf <*prsfile*>

DESCRIPTION

Optosf optimizes the overall smoothing factor (osf) for the Probabilistic Neural Network (PNN) classifier.

The regional weights are optimized using the **optrws** command. To save optrws runtime, it is suggested that optrws be run using the K-L feature vectors of only a reasonably small set of fingerprints, perhaps a small subset of the full prototype set that will be used in the finished classifier. But then, after the full prototype set of feature vectors is made by transforming previously made orientation arrays using the transform matrix incorporating the optimized regional weights, one can expect that the classifier that uses these feature vectors will be slightly more accurate if it uses an overall smoothing factor slightly larger than 1, to compensate for the fact that the prototype set is larger than it was during optimization of the regional weights. During optimization of the regional weights, no explicit overall smoothing factor is used, since any effect such a factor would have had could equally well be produced by just using different values of the regional weights; so, optrws in effect fixes the overall smoothing factor at 1.

The optosf command is provided to optimize the overall smoothing factor for best results on the full set of prototypes. It optimizes osf by attempting to find a minimum (or at least a local minimum) of an activation error rate that results when a set of finished feature vectors is classified by PNN. The set of prototypes used by the PNN, during this optimization, is a superset of the set on which the activation error rate is computed: both sets start at the beginning of the provided data, but they are of different lengths. Whichever fingerprint the classifier is running on is temporarily left out of the prototypes set, i.e. a leave-one-out method is used in order to simulate a realistic situation.

The optimization method used is a very simple one, consisting of taking steps of an initial size, then halving the stepsize and reversing direction if the error rates ceases to decrease, etc. This method, while obviously not sufficient for the general problem of minimizing a real function of one real variable, appears to be sufficient for this particular problem, since the activation error rate as a function of the osf seems to always have a unimodal form.

OPTIONS

<*prsfile*>

A file specifying values of some or all of the parameters. Parameters not specified in this file assume default values. To find out what the parameters are, and as an example of the format of a parameters file, see the file *pcasys/parms/optosf.prs* in the PARAMETER FILES section below. The user's *prsfile* must specify values for those parameters that *optosf.prs* indicates have no defaults; it can also specify default-overriding values for one or more of the parameters that have defaults.

PARAMETER FILES

pcasys/parms/optosf.prs

Contains default values of the parameters for optosf (optimize overall smoothing factor command). Parameters with no defaults must be set in the users *prsfile*.

Default settings in *pcasys/parms/optosf.prs*

n_feats_use *64*

How many (first) features of the feature vectors to use.

osf_init *.1*

Initial value for osf (overall smoothing factor).

osf_initstep *.2*

Initial step size for osf.

osf_stepthr *.01*
Program stops when step size becomes <= this value.

tablesize *1000*
Size of the table used to avoid redundant computing.

verbose *y*
If y, write progress messages to stdout.

outfile_desc -
A – (hyphen) means let optosf make the description; otherwise, value is the description.

fvs_file *(no default, user must set)*
The file containing the prototype feature vectors, each vector stored as one row of the matrix.

classes_file *(no default, user must set)*
The file containing the classes of the prototype feature vectors.

n_fvs_use_as_protos_set *(no default, user must set)*
The number of first feature vectors from fvs_file to use as the PNN prototypes when optimizing osf.

n_fvs_use_as_tuning_set *(no default, user must set)*
The number of first feature vectors from fvs_file to run the PNN on to optimize osf.

outfile *(no default, user must set)*
The results output file.

EXAMPLE(S)

From *test/pcasys/execs/optosf/optosf.src*:

% optosf optosf.prs
Optimize the overall smoothing factor based on the parameters set in the file *optosf.prs*.

SEE ALSO

optrws (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

optrws – optimizes the regional weights for PNN classifier.

SYNOPSIS

optrws <*prsfile*>

DESCRIPTION

Optrws optimizes the regional weights, each of which goes with one 2x2-vector block of the orientation array. Since that array has 28 x 30 vectors, there are (28 / 2) x (30 / 2) = 210 regional weights.

Optimization consists of attempting to approximate the minimum, or at least a local minimum, of an "activation error rate" of the Probabilistic Neural Network (PNN) classifier when it is run on a set of fingerprints, using the same set as the prototypes for the PNN but leaving out of the prototype set, each time, the particular fingerprint that the network is being run on. The program first finds a reasonable value to use as an initial value for all 210 weights. Then, it uses a very simple form of a gradient descent algorithm to finish optimizing the weights. Each iteration consists of, first, estimating the gradient of the error at the current "basepoint", and second, approximately finding the minimum (or at least a local minimum) of the error along the straight line pointing in the anti-estimated-gradient direction. (Estimating the gradient seems to be sufficient, and calculating it from its definition may actually require more computation then estimating.) Because the activation error is apparently such a well-behaved function of distance along this line, for this particular problem, it seems sufficient to use a very simple algorithm for the line search: this consists of taking large equal-sized steps in the anti-gradient direction until the error ceases to decrease, then halving the stepsize and going in the opposite direction along the line until the error again ceases to decrease, etc., with the process stopping when the step size becomes smaller than a threshold. This finds a local minimum, and it appears that this function generally is unimodal along the line, so that this local minimum will be the minimum along the line.

The 0th basepoint is (irw,irw,...,irw), where irw is the initial value decided upon for all regional weights. The 1st basepoint is the result of the line search that follows the gradient-estimation at the 0th basepoint; etc. Stopping of the program is controlled by specifying the number of line searches that are to be done. If this parameter is set to 1, then the program only gets as far as basepoint 1. Since optrws records each basepoint, the program can be manually stopped if it turns out that it is taking too much time, without the run being a total waste of cycles.

At each of the basepoints, optrws produces the following files (in a directory specified as one of the parameters). The basepoint, as a "matrix" file of dimensions 14 x 15 (these dimensions correspond to the geometric interpretation of the basepoint as a set of regional weights); these files have names bspt_0.bin, bspt_1.bin, etc. or bspt_0.asc, bspt_1.asc, etc. The estimated gradient of the activation error rate at the basepoint, also as a "matrix" file of dimensions 14 x 15; these files have names egrad_0.bin, egrad_1.bin, etc. or egrad_0.asc, egrad_1.asc, etc. And, the activation error rate at the basepoint, as a text file; these files have names acerr_0.txt, acerr_1.txt, etc. As optrws is running (which make take several hours), these intermediate results files may be examined to find out what kind of progress the optimization is making. The acerr files obviously can be examined to find out if the reported error rate is still decreasing or has leveled off. Also, the rwpics command (see rwpics man page) can be used to make, from a set of bspt files, a set of grayscale IHead images depicting these regional weights sets in their proper geometric layout. Rwpics can also make two other kinds of pictures: grayscale pictures of a set of estimated gradients (egrad files), and grayscale-binary (i.e. 0 and 255 pixels) pictures showing the signs of the elements of estimated gradients. (The blocks whose estimated gradient elements, i.e. estimated partials, are negative, are ones whose weights will be increasing as optrws takes steps in the anti-estimated-gradient direction.) For the "optimal" set of regional weights, just use the final bspt file produced before the optrws run stops by itself (because of doing the specified number of iterations) or, if optimization appears not to be making much more progress, kill the optrws process and use the last bspt file produced. Or, it could also be interesting to do testing using various basepoints, to find out whether the decreases in the activation error during optimization correspond to error decreases on a test set, i.e. to find out whether even small improvements in the weights in the sense of training error rate, are actually significant in the sense of generalizing to other data. (The weights seem to generalize well, not too suprisingly since there are only 210 of them, hardly a large enough number

for them to be capable of becoming very specifically tuned to the training data in such a way as to have little generalization value.)

The parameters of optrws are specified by parameter files. The program first reads *pcasys/parms/optrws.prs*, which contains default values of some of its parameters; then it reads the user-provided parameters file whose name is given as the argument. Consult optrws.prs to find out what the parameters are, and as an example of the format of a parameters file. Optrws.prs specifies default values for the parameters that have defaults, and it also has a comment concerning each parameter that has no default value. The user parameters file must specify a value for each parameter that does not have a default, and it also can specify default-overriding values for one or more of the other parameters.

Optrws can start several simultaneous instances of another program, optrwsgw, each time it needs to estimate the gradient, if desired. This can reduce the time needed for optimization, if there are several processors available. To use this feature, set acerror_stepped_points_nprocs in your parameters file to a value > 1 (probably should be <= number of processors available). If the operating system on your computer does not implement fork() and execl(), then the Makefile for optrws should be modified by appending -DNO_FORK_AND_EXECL to the definition of CFLAGS, so that a different subset of the code will be compiled and the linker will thereby find no unresolved references.

OPTIONS

<prsfile>

A file specifying values of some or all of the parameters. Parameters not specified in this file assume default values.

PARAMETER FILES

pcasys/parms/optrws.prs

Contains default values for some of the optrws parameters. The remaining parameters, with no default values must be specified in the user *prsfile*.

Default settings in *pcasys/parms/optrws.prs*

n_feats_use *64*
How many (first) features of each K-L feature vector to use.

irw_init *0.1*
Initial value for irw.

irw_initstep *1.0*
Initial step size for irw.

irw_stepthr *.01*
Optimization of irw stops when step size becomes smaller than this threshold.

grad_est_stepsize *.001*
Step size for secant-estimation of gradient.

n_linesearches *2*
Number of (estimate gradient, line search) iterations to do.

linesearch_initstep *.1*
Initial step size for line search.

linesearch_stepthr *.01*
Line search stops when its step size becomes smaller than this threshold.

tablesize *1000*
Size of a table used to hold pairs of values corresponding to previous computations of the error, either as a function of irw or as a function of distance along downhill-pointing line. Lookup in this table saves some cycles by avoiding repeated calculations.

acerror_stepped_points_nprocs *1*
How many processes to use when computing the activation error at the points stepped to from a basepoint, in order to compute the approximate gradient by secant method. If 1, optrws computes the error at all stepped points itself. If > 1, optrws starts this many child processes, each of which computes the error at an interval of the stepped points.

verbose *y*
If y, write progress messages to standard output.

ascii_outfiles *n*
Whether outputfiles are to be ascii (y) or binary (n).

klfvs_file *(no default, user must set)*
File containing K-L feature vectors to be used as prototypes set, and also as "tuning" set, for the optimization. Usually the output on the lintran function.

classes_file *(no default, user must set)*
File containing the classes that go with the feature vectors of *klfvs_file*. Must be a pcasys "classes" formatted file.

n_klfvs_use *(no default, user must set)*
How many of the K-L feature vectors to use (off the top).

eigvecs_file *(no default, user must set)*
File containing the eigenvectors.

outfiles_dir *(no default, user must set)*
The directory in which optrws is to produce its output files.

EXAMPLE(S)

From *test/pcasys/execs/optrws/optrws.src*:

% optrws optrws.prs
Optimizes the regional weights for a set of feature vectors based on the parameters set in the file *optrws.prs*.

SEE ALSO

rwpics (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

pcasys – fingerprint classifier demo, non-graphical version

SYNOPSIS

pcasys *[prsfile]*

DESCRIPTION

Pcasys is the non-graphical fingerprint classifier demo program. It reads a sequence of image files, each depicting one box as scanned from a fingerprint card, and classifies each fingerprint, using a Multi-Layer Perceptron (MLP) or Probabilistic (PNN) Neural Network, to one of six pattern–level classes: Arch, Left loop, Right loop, Scar, Tented arch, and Whorl. The type of classifier MLP or PNN is chosen in the parameters file *pcasys/parms/pcasys.prs*. Pcasys optionally makes an output file, containing a results line for each fingerprint and a summary at the end showing the error rate and the "confusion matrix", and it optionally writes progress messages to the standard output.

The graphical version, pcasysx, is recommended as being much more interesting than this version. However, pcasys is suitable if (1) the X Window System, which pcasysx requires, is not installed, or (2) greatest classification speed is desired. (The graphical displays take a significant amount of additional time.)

Pcasys will look in the default location *pcasys/parms* for the default parameter files it needs. There are prototype and weights files in *pcasys/weights/{mlp|pnn}* that are needed by the MLP and PNN classifiers. Also, the current default location location for the 2700 sample fingerprint images is */NBIS//Test/pcasys/data/images*.

OPTIONS

[prsfile]

A file containing one or more parameter values that are to override the default values. To find out what the parameters are, and as examples of the format of a parameters file, consult the default parameters files that pcasys reads, namely *pcasys/parms/oas.prs* and *pcasys/parms/pcasys.prs* which are described in the section PARAMETER FILES. Each line, in the parms file consists of a parameter name followed by a value; a pound sign indicates that the rest of its line is a comment. If pcasys is run with no argument, i.e. not specifying a user parameters file, then it uses the default values of all parameters.

PARAMETER FILES

pcasys/parms/oas.prs

Contains default values of the parameters that affect the making of orientation arrays (oas): these are the parms of the segmentor (sgmnt), the image enhancer (enhnc), the ridge-valley orientation finder (rors), the registration program (r92a), and the registration-implementing pixelwise orientations reaverager (rgar). The values used for these parms when making the oas used in optimizing the classifier should also be used when running the finished classifier. **See the *mkoas* man page for more information about the parameters in this file.**

pcasys/parms/pcasys.prs

Contains default values of the remaining parameters of pcasys. Also look at *pcasys.mlp* and *pcasys.pnn* for examples on using each classifier.

Default settings in *pcasys/parms/pcasys.prs*

network_type *2*

Set classifier as (1) PNN (Probabilistic Neural Net) or (2) MLP (Multi-layer Perceptron).

trnsfrm_nrows_use *128*

How many (first) rows of the transform matrix to use, and hence, how many features to make for the feature vector of each incoming fingerprint, and also how many (first) features to use of each prototype feature vector when running the classifier:

trnsfrm_matrix_file *pcasys/weights/mlp/mlp_tmat.bin*
File used by the demo to transform the orientation array of an incoming fingerprint into the low-dimensional feature vector that will be sent to the classifier.

cls_str *ALRSTW*
Class string used in graphics mode to display the ouput activations. Should be same size as number of outputs (ie. pnn_nclasses or number outputs in mlp_wts file). Must be some combination of "ALRSTW". For PNN, these must be the same classes as used in the prototype files and be in the same order as when the prototype were optimized.

pnn (Probabilistic Neural Net) parameters:

pnn_nprotos_use *24300*
How many first feature vectors to use, from the set of prototypes. The value 24300 corresponds to the entire provided set, corresponding to volumes 1 - 9 "f" rollings of Special Database 14.

pnn_nclasses *6*
How many different classes there are. For the fingerprint pattern-level classification problem, there are 6: A, L, R, S, T, and W.

pnn_osf *1.368750*
Overall smoothing factor for the PNN. May be optimized using optosf.

pnn_protos_fvs_file *pcasys/weights/pnn/profvs.bin*
Prototype feature vectors file.

pnn_protos_classes_file *pcasys/weights/pnn/procls.asc*
Prototype classes file.

MLP (Multi-layer Perceptron) network parameters:

mlp_wts_file *pcasys/weights/mlp/mlp_wts.bin*
MLP weights file.

Parameters used by the pseudoridge tracer:

pseudo_slthresh0 *0.0*
If squared-length of an orientation vector (in the fine grid used by pseudo) is < this value, then the vector is zeroed before the (possible) application of smoothing iterations.

pseudo_slthresh1 *0.04*
If, after (possible) smoothing iterations, the squared-length of an orientation vector is < this value, then this location is marked as bad, meaning that no pseudoridge is allowed to start here and if one arrives here, tracing stops at this point.

pseudo_smooth_cwt *0.0*
Center-weight for each iteration of smoothing of the orientation grid. An iteration consists of replacing each vector with the weighted average of itself and its four neighbors, with itself getting this much weight and its neighbors equally dividing the remaining weight (sum of weights is 1).

pseudo_stepsize *1.0*
Length of one step in the production of a pseudoridge, which is actually a polygon. A value of 1. corresponds to the spacing between vectors in the (finer) orientation array used by pseudo.

pseudo_max_tilt *45*
Max allowed tilt of a candidate concave-upward's vertex (point of sharpest turning) from a horizontal that corresponds to exact uprightness. In degrees.

pseudo_min_side_turn *70*
Minimum cumulative turn that each side of concave-upward must have. In degrees.

Limits for the block of starting positions in pseudoridge tracing:

pseudo_initi_s *11*
Small limit, vertical. (TOP)

pseudo_initi_e *46*
Large limit, vertical. (BOTTOM)

pseudo_initj_s *11*
Small limit, horizontal. (LEFT)

pseudo_initj_e *50*
Large limit, horizontal. (RIGHT)

pseudo_maxsteps_eachdir *200*
Maximum number of steps that tracer ever takes in either of the two directions from starting point. (Controls the amount of memory needed to store a pseudoridge, and more importantly, such a limit is needed to prevent possible infinitely looping pseudridges in some whorls.)

pseudo_nsmooth *3*
How many iterations of smoothing.

pseudo_maxturn *40*
Maximum turn that is allowed to occur in a single step (in degrees). An attempted turn sharper than this causes tracing to stop.

Used by the combine routine.

combine_clash_confidence *.9*
This is the confidence value combine assigns if pseudo finds a concave-upward (causing hyp class to be whorl) but PNN thinks the print is not a whorl:

PCASYS I/O parameters.

demo_images_list *pcasys/parms/first20.txt*
List of fingerprint images to run the demo on. The default list here lists the first 20 fingerprints of the provided demo set, which consists of the 2700 fingerprints of volume 10 "s" rollings of NIST Special Database 14. *pcasys/parms/all.txt* list all 2700 files.

outfile *pcasys.out*
Output file to be produced. If no output file is wanted, set this to /dev/null.

clobber_outfile *n*
If n, then if outfile already exists, exit with an error message. If y, then overwrite outfile if it already exists.

verbose *y*
If y, then write progress messages to stdout.

EXAMPLE(S)

From *test/pcasys/execs/pcasys/pcasys.src*:

% pcasys
Runs the pcasys demo using the default settings found in
pcasys/parms/pcasys.prs.

% pcasys myprsfile
Runs the pcasys demo using parameters set in *myprsfile* to change the value of the default settings.

SEE ALSO

pcasysx (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

pcasysx – fingerprint classifier demo, graphical version

SYNOPSIS

pcasysx *[prsfile]*

DESCRIPTION

Pcasysx is the graphical fingerprint classifier demo program. It reads a sequence of image files, each depicting one box as scanned from a fingerprint card, and classifies each fingerprint, using a Multi-Layer Perceptron (MLP) or Probabilistic (PNN) Neural Network, to one of six pattern–level classes: Arch, Left loop, Right loop, Scar, Tented arch, and Whorl. The type of classifier MLP or PNN is chosen in the parameters file *pcasys/parms/pcasys.prs*. Additional parameters specific to pcasysx are in *pcasys/parms/pcasysx.prs*.

Pcasysx produces screen graphics illustrating the results of the processing phases (requires X Windows). It optionally makes an output file, containing a results line for each fingerprint and a summary at the end showing the error rate and the "confusion matrix", and it optionally writes progress messages to the standard output.

Pcasysx will look in the default location *pcasys/parms* for the default parameter files it needs. There are prototype and weights files in *pcasys/weights/{mlp|pnn}* that are needed by the MLP and PNN classifiers and images needed for the graphics display in *pcasys/images*. Also, the current default location for the 2700 sample fingerprint images is */NBIS/Test/pcasys/data/images*.

OPTIONS

[prsfile]

A file containing one or more parameter values that are to override the default values. To find out what the parameters are, and as examples of the format of a parameters file, consult the default parameters files that pcasysx reads, namely *pcasys/parms/oas.prs*, *pcasys/parms/pcasys.prs*, and *pcasys/parms/pcasysx.prs*, which are described in the section PARAMETER FILES. Each line, in the parms file consists of a parameter name followed by a value; a pound sign indicates that the rest of its line is a comment. If pcasysx is run with no argument, i.e. not specifying a user parameters file, then it uses the default values of all parameters.

PARAMETER FILES

pcasys/parms/oas.prs

Contains default values of the parameters that affect the making of orientation arrays (oas): these are the parms of the segmentor (sgmnt), the image enhancer (enhnc), the ridge-valley orientation finder (rors), the registration program (r92a), and the registration-implementing pixelwise orientations reaverager (rgar). The values used for these parms when making the oas used in optimizing the classifier should also be used when running the finished classifier. **See the *mkoas* man page for more information about the parameters in this file.**

pcasys/parms/pcasys.prs

Contains default values of the parameters for **pcasysx**. Also look at *pcasys.mlp* and *pcasys.pnn* for examples on using each classifier. **See the *pcasys* man page for more information about the parameters in this file.**

pcasys/parms/pcasysx.prs

Contains default values of parameters, in addition to *pcasys/parms/pcasys.prs*, that are specific to **pcasysx**.

Default settings in *pcasys/parms/pcasysx.prs*

Parameters for the graphical demo, pcasysx, that control sleeping (pausing) after displaying various intermediate results. Value -1 is also allowed: that means wait for user to type enter key before continuing.

sleeps_titlepage *0*
after title page

sleeps_sgmntwork *1*
intermediate results of segmentor

sleeps_segras *0*
segmented image

sleeps_enhnc *1*
enhanced image

sleeps_core_medcore *3*
ridge-orientation bars, core, standard core

sleeps_regbars *2*
registered ridge-orientation bars

sleeps_featvec *1*
bar graph of feature vector input to PNN

sleeps_normacs *2*
bar graph of normalized PNN outputs

sleeps_foundconup *1*
found a concave-upward pseudoridge (conup)

sleeps_noconup *0*
all pseudoridges, if no conup is found

sleeps_lastdisp *2*
results display for the fingerprint

Mouse control parameter:

warp_mouse *n*
If y (yes), then warp the mouse pointer into graphical window so its colormap takes effect. If n (no), no warping.

PCASYS I/O parameters.

outfile *pcasysx.out*
Output file to be produced. If no output file is wanted, set this to /dev/null.

EXAMPLE(S)

From *test/pcasys/execs/pcasys/pcasys.src*:

% pcasysx Runs the pcasysx demo using the default settings found in *pcasys/parms/pcasysx.prs*.

% pcasysx myprsfile
Runs the pcasysx demo using parameters set in *myprsfile* to change the value of the default settings.

SEE ALSO

pcasys (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

rdjpgcom – display text comments from a JPEG file

SYNOPSIS

rdjpgcom [**–verbose**] [*filename*]

DESCRIPTION

rdjpgcom reads the named JPEG/JFIF file, or the standard input if no file is named, and prints any text comments found in the file on the standard output.

The JPEG standard allows "comment" (COM) blocks to occur within a JPEG file. Although the standard doesn't actually define what COM blocks are for, they are widely used to hold user-supplied text strings. This lets you add annotations, titles, index terms, etc to your JPEG files, and later retrieve them as text. COM blocks do not interfere with the image stored in the JPEG file. The maximum size of a COM block is 64K, but you can have as many of them as you like in one JPEG file.

OPTIONS

–verbose

Causes **rdjpgcom** to also display the JPEG image dimensions.

Switch names may be abbreviated, and are not case sensitive.

HINTS

rdjpgcom does not depend on the IJG JPEG library. Its source code is intended as an illustration of the minimum amount of code required to parse a JPEG file header correctly.

In **–verbose** mode, **rdjpgcom** will also attempt to print the contents of any "APP12" markers as text. Some digital cameras produce APP12 markers containing useful textual information. If you like, you can modify the source code to print other APPn marker types as well.

SEE ALSO

cjpeg(1H), **djpeg**(1H), **jpegtran**(1H), **wrjpgcom**(1H)

AUTHOR

Independent JPEG Group

NAME

rdwsqcom – scans a WSQ-encoded image file for any and all comment blocks, printing their contents to standard output.

SYNOPSIS

rdwsqcom <*image file*>

DESCRIPTION

Rdwsqcom takes as input a file containing a WSQ-compressed image, and *without* decoding and reconstructing the image, the utility scans the file for any and all comment blocks. As a comment block is encountered, its contents is printed to standard output. Comments can be written to a WSQ file by using the **wrwsqcom** command.

OPTIONS

<*image file*>

the input WSQ file to be scanned.

EXAMPLES

From *test/imgtools/execs/rdwsqcom/rdwsqcom.src*:

% rdwsqcom finger.wsq > finger.com

prints any comments stored in the WSQ fingerprint file to an output file.

SEE ALSO

cwsq(1G), **wrwsqcom**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

rgb2ycc – converts a color RGB image to the YCbCr colorspace and downsamples component planes as specified.

SYNOPSIS

rgb2ycc <*outext*> <*image file*>
[**-raw_in** *w*,*h*,*d*,[*ppi*]
[**-nonintrlv**]]
[**-YCbCr** *H0*,*V0*:*H1*,*V1*:*H2*,*V2*]

DESCRIPTION

Rgb2ycc takes as input a file containing an uncompressed color RGB image. Two possible input file formats are accepted, NIST IHead files and raw pixmap files. If a raw pixmap file is to be converted, then its image attributes must be provided on the command line as well. Once read into memory, the RGB pixmap is converted to the YCbCr colorspace. The results are always written to a raw pixmap file because the NIST IHead file format only supports interleaved RGB images.

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. By default, **rgb2ycc** assumes interleaved color pixels. (See INTERLEAVE OPTIONS below.)

If requested, **rgb2ycc** also conducts downsampling of the YCbCr component planes. By default, this utility does no downsampling. Regardless of downsampling, the conversion from RGB to YCbCr and back to RGB will not result in the same exact image. Some pixels values will be slightly perturbed due to the round-off of the floating point transformations that are applied. (See YCbCr OPTIONS below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_in** may be written **-r**.

<*outext*>
the extension of the YCbCr output file. To construct the output filename, **rgb2ycc** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input file, either an IHead file or raw pixmap file, containing the color RGB image to be converted.

-raw_in *w*,*h*,*d*,[*ppi*]
the attributes of the input image. This option must be included on the command line if the input is a raw pixmap file.

- *w* the pixel width of the pixmap
- *h* the pixel height of the pixmap
- *d* the pixel depth of the pixmap
- *ppi* the optional scan resolution of the image in integer units of pixels per inch.

-nonintrlv
specifies that the color components in an *input* raw pixmap file image are non-interleaved and stored in separate component planes. (See INTERLEAVE OPTIONS below.)

-YCbCr *H0*,*V0*:*H1*,*V1*:*H2*,*V2*
this option, if provided on the command line, directs **rgb2ycc** to conduct downsampling of the YCbCr component planes. If all the H,V factors are set to 1 then no downsampling is done; this is equivalent to ommiting the option. (See YCbCr Options below.)

INTERLEAVE OPTIONS

The color components of RGB pixels in a raw pixmap file may be interleaved or non-interleaved. Color components are interleaved when a pixel's (R)ed, (G)reen, and (B)lue components are sequentially adjacent in the image byte stream, ie. RGBRGBRGB... . If the color components are non-interleaved, then all (R)ed components in the image are sequentially adjacent in the image byte stream, followed by all (G)reen components, and then lastly followed by all (B)lue components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **rgb2ycc** assumes interleaved color components, and note that all color IHead images must be interleaved.

YCbCr OPTIONS

Rgb2ycc converts color RGB images to the YCbCr colorspace. A common compression technique for YCbCr images is to downsample the Cb & Cr component planes. **Rgb2ycc** conducts a limited range of YCbCr downsampling schemes that are represented by a list of component plane factors. These factors together represent downsampling ratios relative to each other. The comma-separated list of factor pairs correspond to the Y, Cb, and Cr component planes respectively. The first value in a factor pair represents the downsampling of that particular component plane in the X-dimension, while the second represents the Y-dimension. Compression ratios for a particular component plane are calculated by dividing the maximum component factors in the list by the current component's factors. These integer factors are limited between 1 and 4. H,V factors all set to 1 represent no downsampling. For complete details, **rgb2ycc** implements the downsampling and interleaving schemes described in the following reference:

> W.B. Pennebaker and J.L. Mitchell, "JPEG: Still Image Compression Standard," Appendix A - "ISO DIS 10918-1 Requirements and Guidelines," Van Nostrand Reinhold, NY, 1993, pp. A1-A4.

For example the option specification:

> -YCbCr 4,4:2,2:1,1

directs **rgb2ycc** to not downsample the Y component plane (4,4 are the largest X and Y factors listed); the Cb component plane will be downsampled in X and Y by a factor of 2 (maximum factors 4 divided by current factors 2); and the Cr component plane will be downsampled in X and Y by a factor of 4 (maximum factors 4 divided by current factors 1). Note that downsampling component planes is a form of *lossy* compression. The utility **ycc2rgb** takes the YCbCr results and converts them back to the RGB colorspace. If downsampling was applied to the YCbCr components, then the downsampled planes are up-sampled prior to conversion to RGB. Note that even without dowsampling, the conversion from RGB to YCbCr and back to RGB will not result in the same exact image. Some pixels values will be slightly perturbed due to the round-off of the floating point transformations that are applied.

EXAMPLES

From *test/imgtools/execs/rgb2ycc/rgb2ycc.src*:

> **% rgb2ycc ycc face.raw -r 768,1024,24 -Y 4,4:1,1:1,1**
> converts an RGB face image in a raw pixmap file to the YCbCr colorspace and downsamples the Cb and Cr component planes by a factor of 4 in both dimensions.

SEE ALSO

cjpegl(1G), **dpyimage**(1G), **intr2not**(1G), **not2intr**(1G), **ycc2rgb**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

rwpics – makes pictures of regional weights or gradients.

SYNOPSIS

rwpics <*rwfile_in[rwfile_in...]*> <*rws|eg|seg*> <*outpics_dir*>

DESCRIPTION

Rwpics reads a set of PCASYS "matrix" format files each of size 14 x 15. The input files either should be a set of points in regional weights space (in particular, basepoints produced by the optimize regional weights command optrws), or they should be a set of estimated gradients of the activation error (also produced by optrws). Makes a corresponding set of IHead format image files depicting the weights points or gradients; these image files can then be displayed using dpyimage. Rwpics can be used as a sanity check on whether optrws is working properly, and to help decide when the time has come that it is reasonable to stop an optrws run because further significant change of the weights seems unlikely.

OPTIONS

<*rwfile_in[rwfile_in...]*>

The pcasys "matrix" format file(s) to be depicted. Either they should each be a basepoint of the optrws optimization (point in regional weights space), or they should each be an estimated gradient, also produced by optrws. In either case, they must be matrices of size 14 x 15, corresponding to the weights, each of which is associated with one 2 x 2 vector block of the 28 x 30 pattern of orientation vectors. (Usually the output of **optrws**.)

<*rws|eg|seg*>

A code telling rwpics what kind of pictures to make:

If rws ("regional weights"), the program makes a grayscale picture that is reasonable if the input file represents a set of regional weights, e.g. one of the bspt files produced by an optrws run. To do so, it linearly maps absolute values of input values to gray tones, setting the mapping so that 0. maps to black and the maximum absolute value across all components of all input matrices, to white. Absolute values are the reasonable thing to depict when examining a point in regional-weights space, since the sign of a regional weights has no effect on the PNN classifier. (Optimization may sometimes cause some unimportant outer weights to be slightly negative.) The mapping is adapted to the maximum absolute value across all input files, rather than being separately adapted for each input file; this is done so that the several resulting pictures can be examined side by side with the knowledge that all gray tones are on the same scale.

If eg ("estimated gradient"), the program makes a grayscale picture that is reasonable if the input file represents an estimated gradient of the error function, e.g. one of the egrad files produced by an optrws run. To do so, it affinely maps input values to gray tones, setting the mapping so that the minimum input value across all input files is mapped to white and the maximum input value, to black.

If seg ("sign of estimated gradient"), the program makes a grayscale-binary (ie. 0 and 255 pixel values) picture that is reasonable if the input file represents an estimated gradient of the error function. To do so, it maps negative values to white (255) and nonnegative values to black (0). This is interesting because if the component of the estimated gradient (i.e., the estimated partial derivative) associated with a region is negative, that shows that the weight for this region should be increased (and will be increased by optrws).

<*outpics_dir*>

The directory in which the program should produce its output files, which will be raster images in the NIST IHead format. (The directory must already exist, i.e. rwpics does not produce it.) Each output file's name is produced by taking the last component of the corresponding input file and appending an underscore, then the rwsleg|seg code, then .pct (the standard IHEAD file suffix). The output image files may be examined using the dpyimage command.

EXAMPLE(S)

From *test/pcasys/execs/rwpics/rwpics.src*:

% rwpics ../optrws/optrws.bin rws rwpics
Produces an image of the optimized regional weights, which can be converted to JPEG format using the **cjpegb** command.

% rwpics ../optrws/optdir/egrad_1.bin eg rwpics
Produces an image of the estimated gradient, which can be converted to JPEG format using the **cjpegb** command.

% rwpics ../optrws/optdir/egrad_1.bin seg rwpics
Produces an image of the sign of estimated gradient, which can be converted to JPEG format using the **cjpegb** command.

SEE ALSO

dpyimage (1B), optrws (1B), cjpegb (1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

sd_rfmt – takes images from NIST Special Databases 4, 9, 10, 14, and 18 and reformats the compressed data to work with the decompressors **djpegl** and **dwsq**.

SYNOPSIS

sd_rfmt <*SD #*> <*image file*>
SD list = {4,9,10,14,18}

DESCRIPTION

Sd_rfmt reformats images compressed with the old JPEGLSD and WSQ14 compression, on NIST Special Databases 4, 9, 10, 18 (JPEGLSD) and 14 (WSQ14), so the images can be decompressed with the new commands **djpegl** and **dwsq**.

When JPEGLSD was used to compress images on NIST Special Databases (4, 9, 10, 18) the NIST IHEAD header was used to store the data for the JPEGLSD compression. In the new versions **cjpegl** and **djpegl** the full JPEG format is used not the IHEAD header.

The WSQ14 compression used on SD14 has problems with ordering of the data in the compressed file. **Sd_rfmt** simply reorders the data to comply with the format specified in the FBI's Criminal Justice Information Services (CJIS) document, "WSQ Gray-scale Fingerprint Compressions Specification," Dec. 1997. This is the only fingerprint compression format accepted by the FBI IAFIS system. **NOTE: The method for selecting the quantization amount was refined after the release of SD14 so the data loss in the reconstructed image may be more than seen when using the new versions cwsq and dwsq.**

OPTIONS

<*SD #*>
Specify that the input image is from NIST Special Database #.

<*outext*>
the extension of the reformatted output file. [a]To construct the output filename, **sd_rfmt** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the compressed input file to be reformatted.

EXAMPLES

From *test/imgtools/execs/sd_rfmt/sd_rfmt.src*:

% sd_rfmt 7 jpl sd09.old
% sd_rfmt 10 jpl sd10.old
% sd_rfmt 14 wsq sd14.old
% sd_rfmt 18 jpl sd18.old
Convert the special database images to the correct formatted compressed files (JPEGL and WSQ). User could use **rdjpgcom** and **rdwsqcom** to read the NISTCOM comment that is written in the reformatted output file. After reformatting the new file can be decompressed with **djpegl** or **dwsq**.

SEE ALSO

djpegl(1G), **dwsq**(1G), **dpyimage**(1G), **rdjpgcom**(1G), **rdwsqcom**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

stackms – stacks PCASYS formatted matrix files together.

SYNOPSIS

stackms *<matrixfile_in[matrixfile_in...]> <matrixfile_out> <matrixfile_out_desc> <ascii_outfile messages>*

DESCRIPTION

Stackms stacks together several PCASYS "matrix" files: produces a file whose matrix has, as its rows, the rows of all the input matrices. All input matrices must have the same second dimension. (The standard "cat" (catenate files) command is insufficient for proper stacking of matrix files, since the files contain header information and not just the rows of entries.)

OPTIONS

<matrixfile_in[matrixfile_in...]>

The matrix files to be read. (Can be ascii or binary, and they don't all have to have the same ascii/binary setting.) All must have the same second dimension.

<matrixfile_out>

The matrix file to be written. Its rows will be the rows of all the input matrices.

<matrixfile_out_desc>

A string to be written into the output matrix file as its description string; must contain no embedded newline characters. If it contains spaces, tabs, or shell metacharacters that are not to be expanded, then it should be quoted. To leave the description empty, use '' (two single quote marks, i.e. a single–quoted empty string). To let stackms make the description (stating that the file was made by stackms, and listing the names of the input files), use – (hyphen).

<ascii_outfile>

If y, stackms makes an ascii output file; if n, it makes a binary output file. Binary is recommended, unless the output file must be portable across different byte orders or floating–point formats.

<messages>

If y, stackms writes a progress message to the standard output each time it is about to start reading a new input file; if n, no messages.

EXAMPLE(S)

From *test/pcasys/execs/stackms/stackms.src*:

% stackms ../meancov/fv1.men ../meancov/fv2.men tst.men - n y

Combines the mean files *fv1.men* and *fv2.men* into a single file. This is only and example, the **cmbmcs** should be used to combine mean files. The mean files were used to preserve disk space. The **stackms** in practice is used to combine feature vector files (ie. where one wants to stack the data in the files).

SEE ALSO

asc2bin (1B), bin2asc (1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

txt2an2k – Converts a formatted text file into an ANSI/NIST 2000 file.

SYNOPSIS

txt2an2k <*fmttext in*> <*ansi_nist out*>

DESCRIPTION

Txt2an2k parses a specially formatted text file representing the contents of an ANSI/NIST-ITL 1-2000 file, and writes its contents to a new file in the standard compliant format. This utility when used in conjunction with **an2k2txt** enables changes to be interactively made to an ANSI/NIST file using a simple text editor.

OPTIONS

<*fmttext in*>
the formatted text file to be converted

<*ansi_nist out*>
the output ANSI/NIST file

INPUT FORMAT

Every line in the input text represents a single information item for the ANSI/NIST file. These lines are formatted as follows:

r.f.s.i [*t.n*]=*value{US}*

r.f.s.i references the information item with

- *r* the item's positional *r*ecord index in the file
- *f* the item's positional *f*ield index in the record
- *s* the item's positional *s*ubfield index in the field
- *i* the item's positional *i*tem index in the subfield

Note that all indices start at 1.

t.n references the Record Type and Field ID from the standard.

- *t* the record's *t*ype
- *n* the field's ID *n*umber

value is the textual content of the information item, unless the information item contains binary image data, in which case, the value is the name of an external file containing the binary data.

{US} is the non-printable ASCII character 0x1F. This separator character is one of 4 used in the standard. In VIM, this non-printable character may be entered using the ^v command and entering the decimal code "31". In Emacs, this non-printable character may be entered using the ^q command and entering the octal code "037".

Example Input Lines

1.5.1.1 [1.005]=19990708•

This is the information item corresponding to the Date (DAT) field in the standard. It is the 5th field in a Type-1 record, and the Type-1 record is always the first record in the ANSI/NIST file; therefore, its record index is 1, its field index is 5, its subfield index is 1, and its item index is 1. The value of this information item represents the date of July 8, 1999. The '•' at the end of the line represents the non-printable {US} character.

1.3.4.1 [1.003]=14•

This information item is part of the File Content (CNT) field. The CNT field is the 3rd field in a

Type-1 record, so this information item's record index is 1 and its field index is 3. This information item is in the 4th subfield of the CNT field, and has an item index of 1; therefore, its value 14 signifies that the 4th record (the subfield index) in this ANSI/NIST file is a Type-14 record.

4.14.1.1 [14.999]=fld_2_14.tmp•

This information item corresponds to an Image Data field of a Type-14 record. This field always has numeric ID 999 and is always the last field in the image record. This Type-14 record is the 4th record in this ANSI/NIST file, so the Image Data information item has record index 4, and it is in the 14th field (field index 14) in the record. This information item in the ANSI/NIST file contains binary pixel data, so the input value "fld_2_14.tmp" references an external filename from which **txt2an2k** reads the item's binary data.

EXAMPLES

From *test/an2k/execs/txt2an2k/txt2an2k.src*:

% txt2an2k ../an2k2txt/nist.fmt nist.an2

SEE ALSO

an2k2txt(1F), **an2ktool**(1F)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

wrjpgcom – insert text comments into a JPEG file

SYNOPSIS

wrjpgcom [**–replace**] [**–comment** *text*] [**–cfile** *name*] [*filename*]

DESCRIPTION

wrjpgcom reads the named JPEG/JFIF file, or the standard input if no file is named, and generates a new JPEG/JFIF file on standard output. A comment block is added to the file.

The JPEG standard allows "comment" (COM) blocks to occur within a JPEG file. Although the standard doesn't actually define what COM blocks are for, they are widely used to hold user-supplied text strings. This lets you add annotations, titles, index terms, etc to your JPEG files, and later retrieve them as text. COM blocks do not interfere with the image stored in the JPEG file. The maximum size of a COM block is 64K, but you can have as many of them as you like in one JPEG file.

wrjpgcom adds a COM block, containing text you provide, to a JPEG file. Ordinarily, the COM block is added after any existing COM blocks; but you can delete the old COM blocks if you wish.

OPTIONS

Switch names may be abbreviated, and are not case sensitive.

–replace
Delete any existing COM blocks from the file.

–comment *text*
Supply text for new COM block on command line.

–cfile *name*
Read text for new COM block from named file.

If you have only one line of comment text to add, you can provide it on the command line with **–comment**. The comment text must be surrounded with quotes so that it is treated as a single argument. Longer comments can be read from a text file.

If you give neither **–comment** nor **–cfile**, then **wrjpgcom** will read the comment text from standard input. (In this case an input image file name MUST be supplied, so that the source JPEG file comes from somewhere else.) You can enter multiple lines, up to 64KB worth. Type an end-of-file indicator (usually control-D) to terminate the comment text entry.

wrjpgcom will not add a COM block if the provided comment string is empty. Therefore **–replace –comment ""** can be used to delete all COM blocks from a file.

EXAMPLES

Add a short comment to in.jpg, producing out.jpg:

wrjpgcom –c *"View of my back yard" in.jpg* **>** *out.jpg*

Attach a long comment previously stored in comment.txt:

wrjpgcom *in.jpg* **<** *comment.txt* **>** *out.jpg*

or equivalently

wrjpgcom -cfile *comment.txt* **<** *in.jpg* **>** *out.jpg*

SEE ALSO

cjpeg(1H), **djpeg**(1H), **jpegtran**(1H), **rdjpgcom**(1H)

AUTHOR

Independent JPEG Group

NAME

wrwsqcom – inserts a specified comment block into a WSQ-encoded image file.

SYNOPSIS

wrwsqcom <*image file*> **<-f** *comment file* | **-t** *comment text*>

DESCRIPTION

Wrwsqcom takes as input a file containing a WSQ-compressed image, and inserts a user-specified comment block into the file. The comment may be represented by the contents of a file or it may be represented as a string on the command line. Comments can be read from a WSQ file by using the **rdwsqcom** command.

OPTIONS

<*image file*>
: the input WSQ file to modified.

-f *comment file*
: specifies that the comment text is contained in the following file.

-t *comment text*
: specifies that the comment text is the following string on the command line.

EXAMPLES

From *test/imgtools/execs/wrwsqcom/wrwsqcom.src*:

% wrwsqcom finger.wsq -f comment.txt
inserts the contents of the file **comment.txt** into the WSQ fingerprint file.

SEE ALSO

cwsq(1G), **rdwsqcom**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

ycc2rgb – converts a color YCbCr image to the RGB colorspace and accounts for downsampling of the YCbCr component planes.

SYNOPSIS

ycc2rgb <*outext*> <*image file*> <*w*,*h*,*d*,[*ppi*]>
[**-raw_out**]
[**-nonintrlv**]
[**-YCbCr** *H0*,*V0*:*H1*,*V1*:*H2*,*V2*]

DESCRIPTION

Ycc2rgb takes as input a raw pixmap file containing an uncompressed color YCbCr image and converts its pixels to the RGB colorspace. Two output file formats are possible, a NIST IHead file (the default) or a raw pixmap file (specified by the **-raw_out** flag).

The color components of YCbCr pixels in a raw pixmap file may be interleaved or non-interleaved. By default, **ycc2rgb** assumes interleaved color pixels. Note that interleaved input produces interleaved output, and likewise, non-interleaved input produces non-interleaved output. Results from non-interleaved input should be explicitly stored in a raw pixmap file, because the IHead format only supports interleaved pixels. (See INTERLEAVE OPTIONS below.)

It is possible that the component planes of the input YCbCr image have been previously downsampled. If so, the **-YCbCr** flag must be included on the command line listing the appropriate component plane downsampling factors. If downsampling is specified, then **ycc2rgb** conducts appropriate upsampling of the YCbCr component planes prior to RGB conversion. By default, this utility assumes no downsampling. Regardless of downsampling, the conversion from RGB to YCbCr and back to RGB will not result in the same exact image. Some pixels values will be slightly perturbed due to the round-off of the floating point transformations that are applied. (See YCbCr OPTIONS below.)

OPTIONS

All switch names may be abbreviated; for example, **-raw_out** may be written **-r**.

<*outext*>
the extension of the RGB output file. To construct the output filename, **ycc2rgb** takes the input filename and replaces its extension with the one specified here.

<*image file*>
the input raw pixmap file containing the color YCbCr image to be converted.

<*w*,*h*,*d*,[*ppi*]>
the attributes of the input image in the raw pixmap file.

- *w* the pixel width of the pixmap
- *h* the pixel height of the pixmap
- *d* the pixel depth of the pixmap
- *ppi* the optional scan resolution of the image in integer units of pixels per inch.

-raw_out
specifies that the results should be stored to a raw pixmap file.

-nonintrlv
specifies that the color components in the *input* raw pixmap file image are non-interleaved and stored in separate component planes. The **-raw_out** flag should always be used in conjunction with this option. (See INTERLEAVE OPTIONS below.)

-YCbCr *H0*,*V0*:*H1*,*V1*:*H2*,*V2*

this option, if provided on the command line, indicates that the YCbCr component planes of the input image have been previously downsampled. **Ycc2rgb** uses the listed factors to conduct upsampling of the YCbCr component planes prior to RGB pixel conversion. If all the H,V factors are set to 1 then no upsampling is required; this is equivalent to omitting the option. (See YCbCr Options below.)

INTERLEAVE OPTIONS

The color components of YCbCr pixels in a raw pixmap file may be interleaved or non-interleaved. Color components are interleaved when a pixel's Y, Cb, and Cr components are sequentially adjacent in the image byte stream, ie. YCbCrYCbCrYCbCr... . If the color components are non-interleaved, then all Y components in the image are sequentially adjacent in the image byte stream, followed by all Cb components, and then lastly followed by all Cr components. Each complete sequence of color components is called a *plane*. The utilities **intr2not** and **not2intr** convert between interleaved and non-interleaved color components. By default, **ycc2rgb** assumes interleaved color pixels. Note that interleaved input produces interleaved output, and likewise, non-interleaved input produces non-interleaved output. Results from non-interleaved input should be *explicitly* stored in a raw pixmap file, because the IHead format only supports interleaved pixels.

YCbCr OPTIONS

Ycc2rgb converts color YCbCr images to the RGB colorspace. A common compression technique for YCbCr images is to downsample the Cb & Cr component planes. **Ycc2rgb** can handle a limited range of YCbCr downsampling schemes that are represented by a list of component plane factors. These factors together represent downsampling ratios relative to each other. The comma-separated list of factor pairs correspond to the Y, Cb, and Cr component planes respectively. The first value in a factor pair represents the downsampling of that particular component plane in the X-dimension, while the second represents the Y-dimension. Compression ratios for a particular component plane are calculated by dividing the maximum component factors in the list by the current component's factors. These integer factors are limited between 1 and 4. H,V factors all set to 1 represent no downsampling. For complete details, **ycc2rgb** implements the downsampling and interleaving schemes described in the following reference:

> W.B. Pennebaker and J.L. Mitchell, "JPEG: Still Image Compression Standard," Appendix A - "ISO DIS 10918-1 Requirements and Guidelines," Van Nostrand Reinhold, NY, 1993, pp. A1-A4.

For example the option specification:

> -YCbCr 4,4:2,2:1,1

indicates that there has been no downsampling of the Y component plane (4,4 are the largest X and Y factors listed); the Cb component plane has been downsampled in X and Y by a factor of 2 (maximum factors 4 divided by current factors 2); and the Cr component plane has been downsampled in X and Y by a factor of 4 (maximum factors 4 divided by current factors 1). The utility **rgb2ycc** converts RGB pixmaps to the YCbCr colorspace, and it conducts downsampling of the resulting YCbCr component planes upon request. Note that downsampling component planes is a form of *lossy* compression. If downsampling was applied to an input image, **ycc2rgb** takes the downsamples planes and upsamples them prior to RGB conversion. Note that even without downsampling, the conversion from RGB to YCbCr and back to RGB will not result in the same exact image. Some pixels values will be slightly perturbed due to the round-off of the floating point transformations that are applied.

EXAMPLES

From *test/imgtools/execs/ycc2rgb/ycc2rgb.src*:

> **% ycc2rgb raw face.ycc 768,1024,24 -r -Y 4,4:1,1:1,1**
> converts a YCbCr face image with downsampled Cb and Cr component planes to the RGB colorspace, storing the results to a raw pixmap file.

SEE ALSO

intr2not(1G), **not2intr**(1G), **rgb2ycc**(1G)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

nzormdat – Takes a patterns file of (un-normalized) feature vectors and computes and writes the statistics (mean and stddev) to support ZNormalization.

SYNOPSIS

znormdat <*binpats in*>

DESCRIPTION

Znormdat takes a patterns file of (un-normalized) feature vectors and computes their mean and stddev. Prints these statistics to stdout which can be stored and used as input to **znormpat** to perform ZNormalization on the feature vector file.

OPTIONS

<*binpats in*>

input patterns file containing un-normalized feature vectors

EXAMPLES

From *test/nfiq/execs/znormdat/znormdat.src*:

znormdat ../fing2pat/nonorm.bin > znorm.dat

NOTE: This is just an example of how to generate the global mean and standard deviation statistics used to z-normalize the pattern feature vectors. There are only 8 feature vectors in the file ../fing2pat/nonorm.bin and approximately 240,000 were used to generate the statistics in ../../../../nfiq/znorm.dat

SEE ALSO

feats2pat(1D), **znormpat**(1D), **nfiq**(1D), **mlp**(1B)

AUTHOR

NIST/ITL/DIV894/Image Group

NAME

nzormpat – Takes a patterns file of (un-normalized) feature vectors and ZNormalizes its feature vectors based on global statistics provided.

SYNOPSIS

znormpat <*znormfile*> <*binpats in*> <*binpats out*>

DESCRIPTION

Znormpat takes a patterns file of (un-normalized) feature vectors and performs ZNormalization based on the global statistics provided in <znormfile>. The znormalized feature vectors are written to the <binpats out> file. The statistics in <znormfile> are computed using **znormdat**.

OPTIONS

<*znormfile*>
: file containing the global statistics used to perform ZNormalization

<*binpats in*>
: input patterns file containing un-normalized feature vectors

<*binpats out*>
: ouptut patterns file containing ZNormalized feature vectors

EXAMPLES

From *test/nfiq/execs/znormpat/znormpat.src*:

znormpat ../../../../nfiq/znorm.dat ../fing2pat/nonorm.bin norm.bin

SEE ALSO

feats2pat(1D), **znormdat**(1D), **nfiq**(1D), **mlp**(1B)

AUTHOR

NIST/ITL/DIV894/Image Group