Chapter 11. Fingerprint Recognition

Fingerprint recognition is becoming a familiar part of business due to the need to reliably identify people in a convenient and low-cost manner. A good optical print reader cost around US$100, and only requires the user to press a finger tip against a plastic plate above a CCD.

But this book is about the wonderful webcam, so is it possible to build a recognizer using a camera as an input device? The answer, at least for my dusty old webcam, is 'not directly'. It proved incapable of accurately focusing on a finger held close to its lens, and couldn't pick up the fingerprint's dark lines (ridges).

My somewhat unsatisfactory solution is to use pencil graphite and sticky tape to transfer an impression of my fingertip onto paper (e.g. as explained at http://www.wikihow.com/Take-Fingerprints). I photocopied the paper, enlarging the image, so my webcam could adequately focus on it.

My application for converting a fingerprint into a *template* is shown in Figure 1.

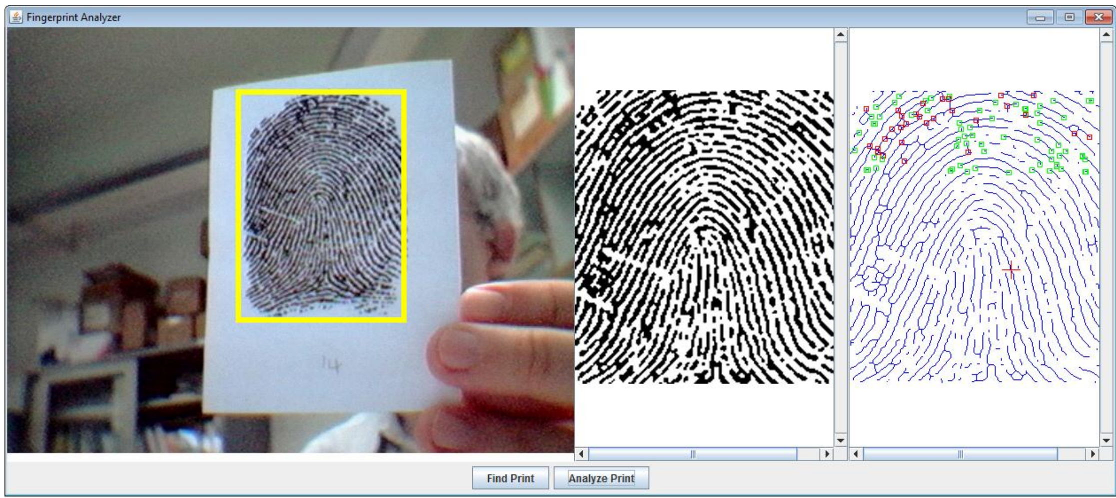

Figure 1. The Templater Application.

As I'll explain shortly, a fingerprint template is a collection of numerical data about the position and orientation of certain types of ridges in a print.

The Templater application consists of three panel: the left one shows the current webcam image, and a yellow border around the identified fingertip, and two other panels for the extracted fingerprint image, and a drawing of the template data.

There's a separate Matcher application (see Figure 2) for comparing a test fingerprint template with other templates to determine the closest match.

 (c) Andrew Davison 2013

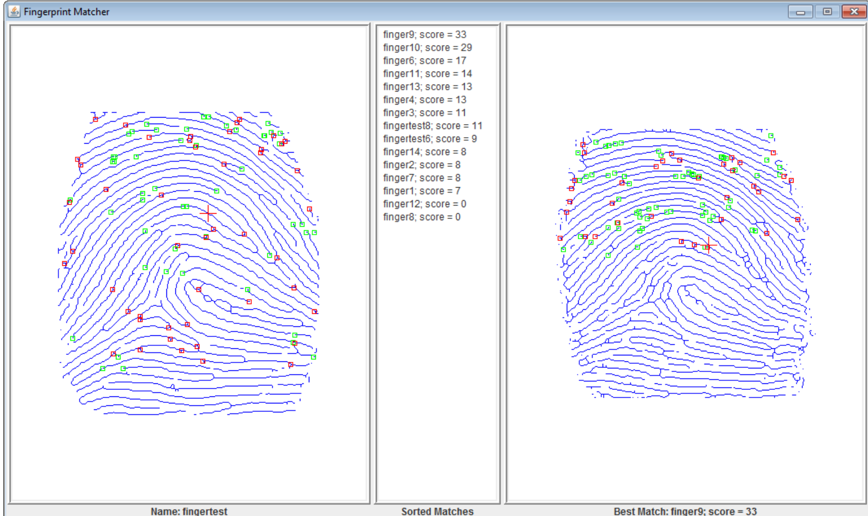

Figure 2. The Matcher Application.

The test template is drawn in the left-hand panel, the matching scores are presented in the text area, and the best matching template rendered in the panel on the right.

The Templater and Matcher applications are shown as flow diagrams in Figure 3.

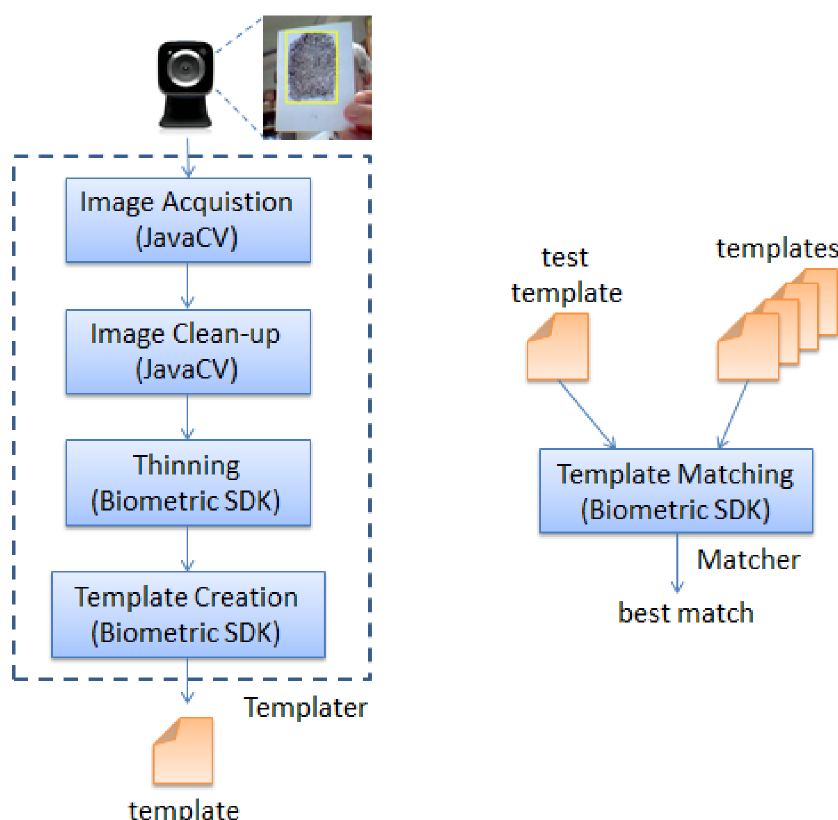

Figure 3. The Main Processing Stages of Templater and Matcher.

Templater utilizes JavaCV to improve the quality of the webcam image, and to narrow in on the fingerprint. The techniques utilized include equalization, adaptive thresholding, and masking to eliminate noise around the print.

The fingerprint processing operations (i.e. for thinning, template creation, and matching) come from Scott Johnston's Biometric SDK, Version 1.3 (http://sourceforge.net/projects/biometricsdk/).

Before going into the details of the two applications, I'll start by reviewing a few basic ideas in fingerprint classification, especially the use of *minutiae*.

1. Classifying a Fingerprint

The standard fingerprint classification scheme is based on high-level features, visible to the naked eye, such as loops, whorls, and arches (see Figure 4).

Figure 4. Some Common Fingerprint Patterns.

However, modern techniques focus on finding *minutiae* (points where ridge lines branch or end). A single fingerprint may have over 100 such identification points. Some of the major kinds of minutiae are listed in Figure 5.

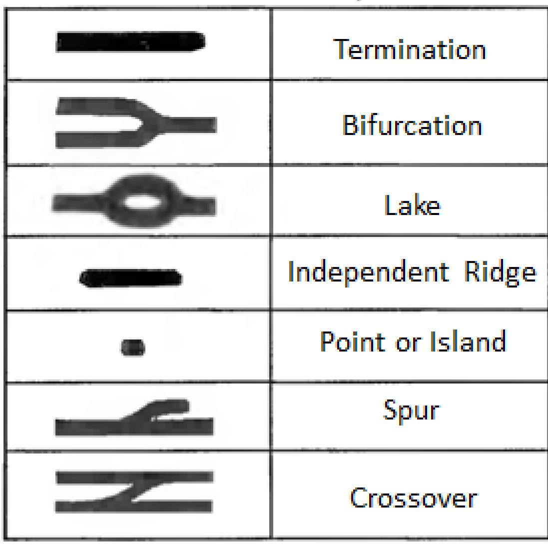

Figure 5. Some Major Minutiae.

(c) Andrew Davison 2013

Most detection systems concentrate on finding *termination* and *bifurcation* minutiae, recording their position and orientation relative to a *core* point. A core is either the northern-most loop or whorl on the finger tip, or a place where ridges drastically change direction around a small region. The relationship between a single bifurcation point and a core is illustrated in Figure 6.

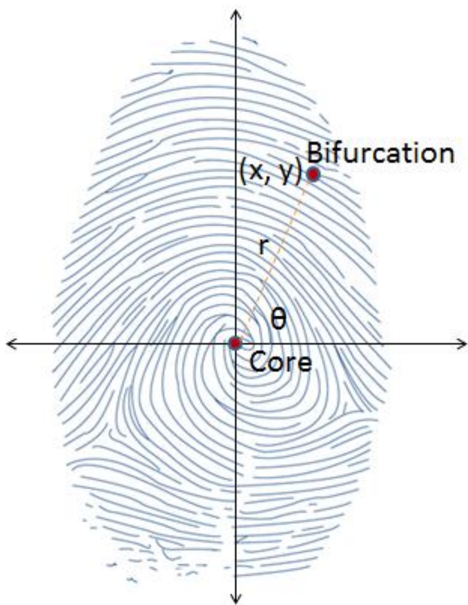

Figure 6. Measuring a Bifurcation Relative to a Core.

The bifurcation's Cartesian and polar coordinates are specified relative to the core.

Minutiae extraction represents a fingerprint's ridges as thin lines so it's easy to calculate positions and angles. Consequently, template building (i.e. the collection of minutiae for ridge endings and bifurcations) is preceded by image enhancement and thinning (as in Figure 7).

Figure 7. Enhancing and Thinning a Webcam Image,

(c) Andrew Davison 2013

Minutiae-based matching involves the comparison of existing templates and a test template by pairing up their minutiae types, and scoring their 'closeness'. The matching involves rotating points about their cores to better align the prints.

The standard work on fingerprint recognition is *Handbook of Fingerprint Recognition* by Davide Maltoni, *et al.*, Springer, 2003.

2. The Templater Application

The class diagrams for my Templater application are shown in Figure 8.

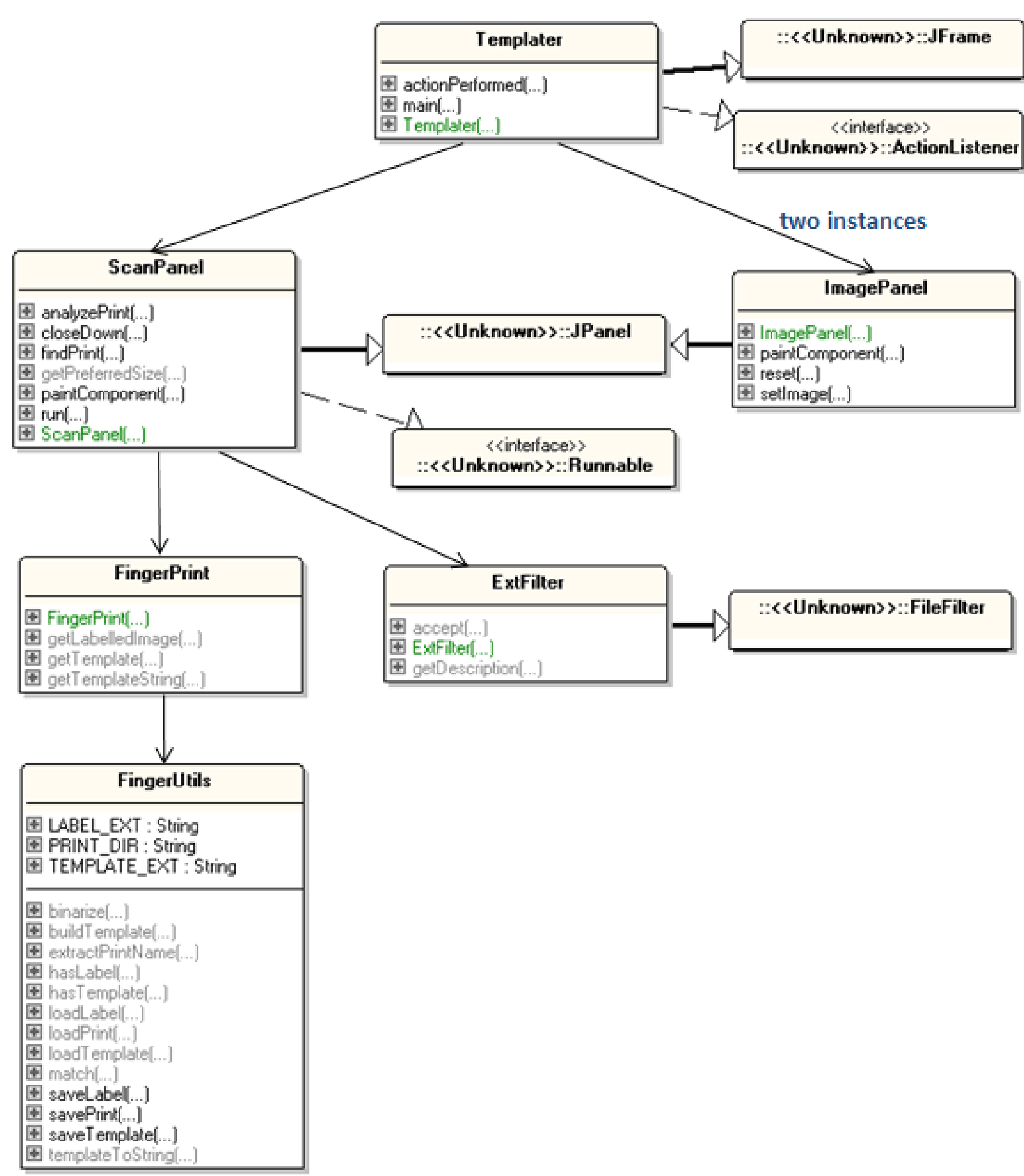

Figure 8. Class Diagrams for the Templater Application.

Templater creates the three-panel window shown in Figure 1, using an instance of ScanPanel and two ImagePanels. The GUI has two buttons, which trigger calls to methods in ScanPanel.

ScanPanel uses JavaCV's FrameGrabber to grab the current image from the webcam, and enhances the fingerprint in that image using JavaCV operations. The extracted print is displayed by Templater's middle ImagePanel, and the print's bounded box is drawn on the webcam picture in the left-hand panel.

The hard work of print thinning and template creation is performed by the FingerPrint object, which calls static methods in FingerUtils. These methods are essentially reformatted versions of methods from the CFingerPrint class in the Biometric SDK .

FingerPrint calculates a template, and also draws an image combining the thinned fingerprint and the template data. Both the template and labeled image are saved to files for later use by the Matcher application, and the labeled image is rendered in the right-hand ImagePanel of Templater.

2.1. Finding a Print

ScanPanel.findPrint() is called when the user presses the "Find Print" button in the Templater GUI. The method performs multiple JavaCV operations, which can be broadly grouped as:

- converting the webcam image to an equalized grayscale;
- creating a fingerprint-sized blob through erosion;
- sharpening the image, adding a lot of noise in the process;
- highlighting the fingerprint in the noisy image by using the blob as a mask;
- calculating the bounded box around the fingerprint with contours;
- cropping the image, to extract the print.

The code for findPrint():

```
// globals
private static final double CROP_FRAC =  0.75;
   // for cropping the top/bottom of the fingerprint image

private static final double X_LEN =  323.0;
   // x- length of final fingerprint (same as in the Biometric SDK)


private IplImage snapIm = null;  // current webcam snap

private Polygon gridPoly;     // fingerprint's bounded box
private boolean foundOutline = false;

private BufferedImage fpImage;  // extracted fingerprint image
private ImagePanel fpPanel;     // where fingerprint is displayed


public void findPrint()
{
  fpImage = null;
  fpPanel.reset();
```

```
foundOutline = false;     // not found an outline yet
skelPanel.reset();

if (snapIm == null)
  return;

// convert to grayscale and equalize
IplImage grayImg = IplImage.create(
                       cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvCvtColor(snapIm, grayImg, CV_BGR2GRAY);
cvEqualizeHist(grayImg, grayImg);

// blur fingerprint into a black blob
IplImage blobImg = IplImage.create(
                       cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvErode(grayImg, blobImg, null, 5);
     // convert print (and other areas) into grayish blobs

// change blobs to black using thresholding
cvThreshold(blobImg, blobImg, 150, 255, CV_THRESH_BINARY);

/* sharpen fingerprint (which also adds a lot of
   general noise to the image) */
IplImage threshImg = IplImage.create(
                       cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvAdaptiveThreshold(grayImg, threshImg, 255,
        CV_ADAPTIVE_THRESH_MEAN_C,  CV_THRESH_BINARY,
        5, 2);   // block size and offset

/* remove the noise surrounding the fingerprint
   by using the black fingerprint blob as a mask
   to protect the fingerprint */
cvMax(threshImg, blobImg, threshImg);
    // remove threshImg areas that are white in blobImg

// more noise reduction to improve inside the fingerprint
cvSmooth(threshImg, threshImg, CV_MEDIAN, 3);
cvEqualizeHist(threshImg, threshImg);

IplImage largeFPImg = null;

/* find a bounded box near the center of the image,
   which should be the outline of the fingerprint  */
IplImage boxImg = IplImage.create(
                       cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvNot(blobImg, boxImg);
             // so fingerprint is white on black background
  CvRect centerBox = boxNearCenter(boxImg);

if (centerBox == null) {
  System.out.println("No center box found in blob image");
}
else {
  // calculate the bounded box around the selected contour
  int x = centerBox.x();
  int y = centerBox.y();
  int w = centerBox.width();
  int h = centerBox.height();

  // store box's outline in polygon for later drawing
  synchronized(gridPoly) {
```

```
      gridPoly.reset();  // add points in clockwise order
      gridPoly.addPoint(x, y);
      gridPoly.addPoint(x+w, y);
      gridPoly.addPoint(x+w, y+h);
      gridPoly.addPoint(x, y+h);
    }
    foundOutline = true;

    // crop top and bottom of fingerprint
    int hFrac = (int)(h * CROP_FRAC);
    int yFrac = y + (h-hFrac)/2;
    IplImage fpImg = cvCreateImage(
                         cvSize(w, hFrac), IPL_DEPTH_8U, 1);
    cvSetImageROI(threshImg, cvRect(x, yFrac, w, hFrac) );
    cvCopy(threshImg, fpImg);
    cvResetImageROI(threshImg);

    // scale the image so it's x- dimension == X_LEN
    double scale = X_LEN / fpImg.width();
    largeFPImg = cvCreateImage(
          cvSize((int)(fpImg.width()*scale),
                 (int)(fpImg.height()*scale)), IPL_DEPTH_8U, 1);
    if (scale > 1)   // enlarge
      cvResize(fpImg, largeFPImg, CV_INTER_CUBIC);
    else  // shrink
      cvResize(fpImg, largeFPImg, CV_INTER_AREA);

    fpImage = largeFPImg.getBufferedImage();
                           // IplImage --> BufferedImage
    fpPanel.setImage(fpImage);   // display fingerprint in ImagePanel
  }
}  // end of findPrint()
```

The findprint() code fragment:

```
IplImage grayImg = IplImage.create(
                      cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvCvtColor(snapIm, grayImg, CV_BGR2GRAY);
cvEqualizeHist(grayImg, grayImg);
```

converts the poorly lit colored webcam picture into an equalized grayscale, as in Figure 9.

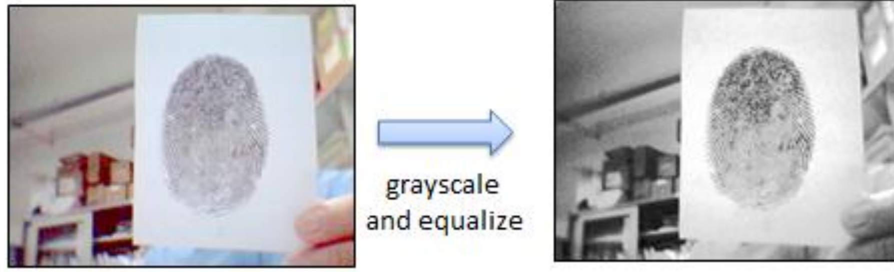

Figure 9. Webcam Image to Equalized Grayscale.

I need to create a fingerprint 'mask' for a later processing stage. Erosion and thresholding are utilized to change the image's gray areas into black blobs:

```
IplImage blobImg = IplImage.create(
                        cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvErode(grayImg, blobImg, null, 5);
cvThreshold(blobImg, blobImg, 150, 255, CV_THRESH_BINARY);
```

The transformation is shown in Figure 10.

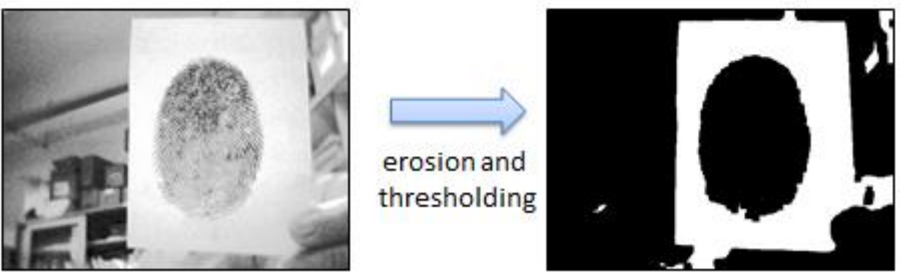

Figure 10. Blob Creation.

Since the fingerprint is printed on white paper, the resulting image will include a black fingerprint blob surrounded by white.

The tricky part is deciding on the amount of erosion, since too much will cause the blob to expand through the white border and blend with the background blobs. But, if the erosion is too little, then the blob won't cover enough of the fingerprint region.

findPrint() continues by sharpening the grayscale image (producing the top-left image in Figure 11), which makes the print's ridges clearer but adds a lot of noise around the fingerprint. This is where the blob image becomes useful since its white border can be used to remove that noise. The code:

```
IplImage threshImg = IplImage.create(
                            cvGetSize(snapIm), IPL_DEPTH_8U, 1);
cvAdaptiveThreshold(grayImg, threshImg, 255,
        CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY, 5, 2);

cvMax(threshImg, blobImg, threshImg);
             // remove threshImg areas that are white in blobImg

cvSmooth(threshImg, threshImg, CV_MEDIAN, 3);
cvEqualizeHist(threshImg, threshImg);
```

Note that the cleaned image is stored back in threshImg.

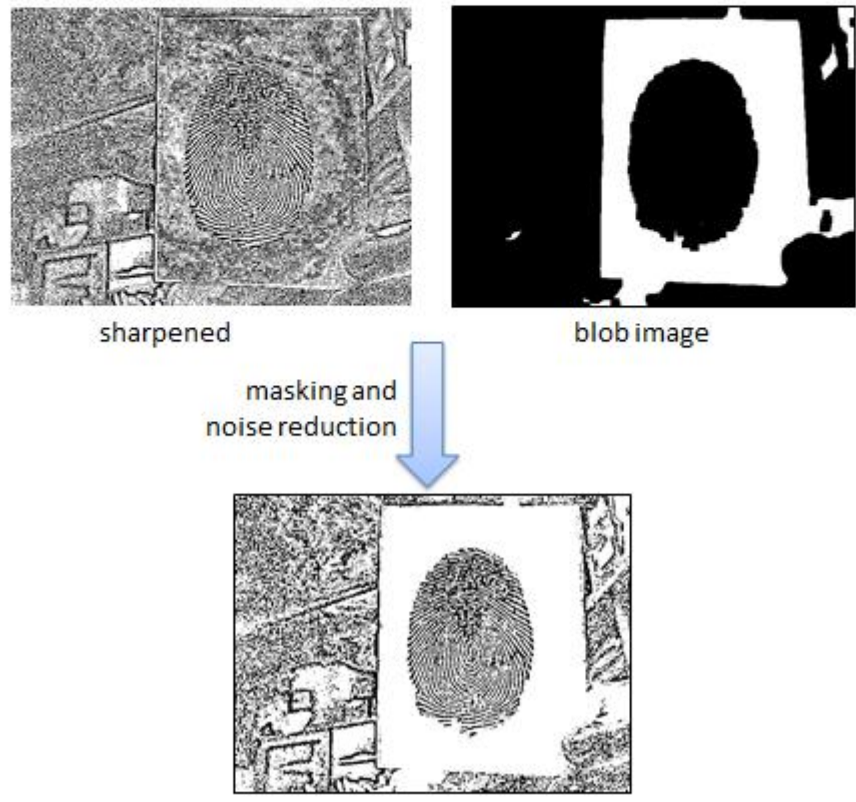

Figure 11. Sharpening and Cleaning.

findPrint() calls boxNearCenter() to calculate a bounded box around the fingerprint. boxNearCenter() builds a list of all the contours in the image, and looks for a convex polygon matching certain criteria. I assume that the bounded box for the print's size is within a certain range, and that it's near to the center of the image. In other words, I'm hoping that the user is holding the fingerprint directly in front of the webcam.

The code for boxNearCenter():

```
// globals
private static final float SMALLEST_BOX =  1000.0f;
     // ignore contour boxes smaller than SMALLEST_BOX pixels

private static final double BOX_FRAC =  0.8;
     // for the size of largest possible contour box


private CvRect boxNearCenter(IplImage boxImg)
{
  int maxBox = (int)Math.round(
                 (boxImg.width() * boxImg.height())*BOX_FRAC);
     // this stops the entire image being selected as an outline

  // generate all the contours in the image
  CvSeq contours = new CvSeq(null);
  CvMemStorage storage = CvMemStorage.create();
  cvFindContours(boxImg, storage, contours,
                   Loader.sizeof(CvContour.class),
```

```
                    CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);
  // center of the image
  int xCenter = boxImg.width()/2;
  int yCenter = boxImg.height()/2;
  int minDist2 = xCenter*xCenter + yCenter*yCenter;
                            // squared distance from center
  CvRect centerBox = null;
  /* find convex box contour nearest the center, whose box area
     is the biggest within the range SMALLEST_BOX -- maxBox */
  while (contours != null && !contours.isNull()) {
    if (contours.elem_size() > 0) {
      CvSeq quad = cvApproxPoly(contours,
           Loader.sizeof(CvContour.class), storage,
           CV_POLY_APPROX_DP, cvContourPerimeter(contours)*0.02, 0);
      CvSeq convexHull =
                   cvConvexHull2(quad, storage, CV_CLOCKWISE, 1);
      if (convexHull != null) {
        CvRect boundBox = cvBoundingRect(convexHull, 0);
        int area = boundBox.width()*boundBox.height();
        if ((area > SMALLEST_BOX) && (area < maxBox)) {
          int dist2 = distApart2(xCenter, yCenter, boundBox);
          if (minDist2 > dist2) {
            // nearer center than previous best match?
            minDist2 = dist2;
            centerBox = boundBox;
          }
        }
      }
    }
    contours = contours.h_next();
  }
  return centerBox;
}  // end of boxNearCenter()


private int distApart2(int xc, int yc, CvRect box)
// squared distance between (xc,yc) and the center of the box
{
  int xBox = box.x() + box.width()/2;
  int yBox = box.y() + box.height()/2;
  return ((xc - xBox)*(xc -xBox) + (yc - yBox)*(yc - yBox));
}  // end of distApart2()
```

It's possible that the wrong bounded box will be chosen by boxNearCenter() (i.e. one that doesn't surround the fingerprint), and so the box is shown to the user before any further processing is carried out. findPrint() uses the box to initialize a Polygon object called gridPoly, which is employed by ScanPanel's rendering thread to draw the box on top of the webcam image (as seen in Figures 1 and 12).

 (c) Andrew Davison 2013

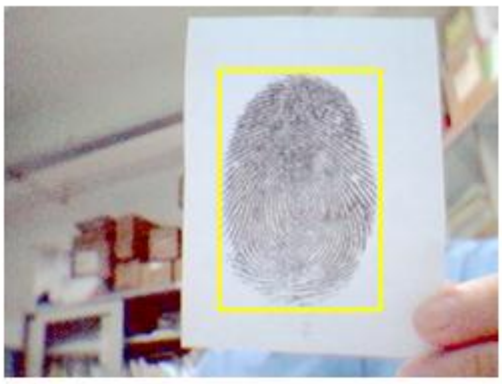

Figure 12. The Bounded Box Made Visible.

findPrint() employs the bounded box to crop the image. The cropping removes a little extra from the top and bottom of the fingerprint, and also scales the resulting image, producing something like Figure 13.

Figure 13. The Cropped and Scaled Fingerprint.

This image is displayed in the middle panel of Templater (see Figure 1).

The user now decides whether to move onto the next stages in fingerprint recognition, namely thinning and template construction (as shown in Figure 3). The user does this by pressing the "Analyze Print" button in the Templater GUI.

2.2. Analyzing the Fingerprint

The "Analyze Print" button triggers a call to ScanPanel.analyzePrint(), which farms out the analysis to a FingerPrint object.

```
// globals
private BufferedImage fpImage;  // the extracted fingerprint image
private int fileCount = 0;

private ImagePanel skelPanel;
        // where the labeled fingerprint is displayed
```

```
public void analyzePrint()
{
  if (fpImage == null)
    return;

  // get user to select a PNG filename for fingerprint image
  String printName = "finger" + fileCount + "???";
  fileCount++;
  JFileChooser jfc = new JFileChooser(FingerUtils.PRINT_DIR);
  jfc.setAcceptAllFileFilterUsed(false);
  jfc.addChoosableFileFilter(new ExtFilter("png"));
  jfc.setSelectedFile(new File(printName + ".png"));

  int userSelection = jfc.showSaveDialog(this);
  if (userSelection == JFileChooser.CANCEL_OPTION)
    return;

  // extract print name from the selected filename
  if (userSelection == JFileChooser.APPROVE_OPTION) {
    String fnm = jfc.getSelectedFile().getName();
    printName  = FingerUtils.extractPrintName(fnm);
  }

  if (printName == null)
    return;

  /* create a FingerPrint object which contains template and
     labeled image info*/
  FingerPrint fp = new FingerPrint(printName, fpImage);

  // display labeled fingerprint image in skelPanel ImagePanel
  BufferedImage labelledImage = fp.getLabelledImage();
  if (labelledImage != null)
    skelPanel.setImage(labelledImage);
} // end of analyzePrint()
```

analyzePrint()'s main task is to create a JFileChooser dialog so the user can enter a filename for the extracted fingerprint. This name (without the ".png" extension) is utilized in the FingerPrint object which also creates and saves a template file and a labeled image (e.g. like the one in Figure 14).

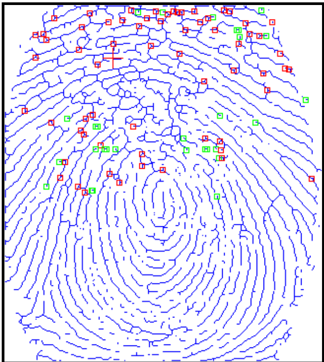

Figure 14. A Labeled Fingerprint.

The labeling highlights the termination and bifurcation ridges in the print with green and red boxes. Also, the core point is shown as a red "+" symbol. This labeled image is displayed in the right-hand ImagePanel of Templater.

3. The FingerPrint Class

A FingerPrint object holds four data items:

- a fingerprint name;
- the fingerprint image extracted from the webcam picture;
- a fingerprint template, which contains minutiae data on ridge endings, bifurcations, and the print's core;
- a labeled fingerprint image which shows the fingerprint and template data.

The object stores the fingerprint image, template and labeled image in local files so they can later be accessed by the Matcher application. The fingerprint name is used as the basis for the filenames. For example, if the name is "XXX" then the fingerprint picture will be stored in XXX.png, the template in XXXTemplate.txt, and the labeled image in XXXLabelled.png.

The hard work of generating the template and labeled image is carried out by static methods from the FingerUtils class, which are closely based on methods from the Biometric SDK.

The FingerPrint constructor:

```
// globals
private String printName;
private double[] template;
private BufferedImage labelIm;


public FingerPrint(String pName, BufferedImage im)
{
  printName = pName;
  if (im == null)
    return;

  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  FingerUtils.savePrint(printName, im);   // save fingerprint

  // create the template and labeled image
  byte[][] skel = FingerUtils.binarize(im);
  template = FingerUtils.buildTemplate(skel, imWidth, imHeight);
  labelIm = labelImage(skel, template, imWidth, imHeight);

  // save the template and labeled image in files
  System.out.println();
  FingerUtils.saveTemplate(printName, template);
  FingerUtils.saveLabel(printName, labelIm);
}  // end of FingerPrint()
```


(c) Andrew Davison 2013

The template and labeled image utilize a 2D byte array called skel, which holds 1's and 0's representing a binary version of the fingerprint image. The array is created by calling the static method FingerUtils.binarize():

```
// in FingerUtils class

public static byte[][] binarize(BufferedImage im)
// from CFingerPrint, by Scott Johnston
{
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  byte[][] skel = new byte[imWidth][imHeight];

  for (int i = 0; i < imWidth; i++) {
    for (int j = 0; j < imHeight; j++) {
      Color c = new Color(im.getRGB(i, j));
      if ((c.getBlue() < 128) && (c.getRed() < 128) &&
          (c.getGreen() < 128))
        skel[i][j] = 1;     // if color not white
      else
        skel[i][j] = 0;     // if color is white
    }
  }

  // set edges to 0
  for (int i = 0; i < imWidth; i++) {
    skel[i][0] = 0;
    skel[i][imHeight-1] = 0;
  }
  for (int j = 0; j < imHeight; j++) {
    skel[0][j] = 0;
    skel[imWidth-1][j] = 0;
  }

  return skel;
}  // end of binarize()
```

3.1. Building the Template

FingerUtils.buildTemplate() returns a template as an array of doubles, whose format is based on the old National Institute of Standards and Technology (NIST) ISO standard for fingerprint biometric data (http://www.nist.gov/itl/iad/ig/ansi_standard.cfm).

The zeroth element in the array is its size, and is followed by groups of six values, each one representing a minutiae point. The format is (x, y, radius, degree, number-of-ends, resultant-degree). The first four data items correspond to the coordinates shown in Figure 6 (i.e. radius is 'r' and degree is θ, the angle between the core and the minutiae point). The number-of-ends field is used to distinguish between termination and bifurcation ridges: termination ridges have a single 'end', while bifurcations are assigned 3. The resultant-degree value records the approximate angle that a termination point makes with the x-axis.

The core point is stored in the first group of six elements, but only the (x, y) fields have values, the others are set to 0.

Before the minutiae information is collected, the binarized fingerprint (stored in skel[][]) is thinned.The Biometric SDK contains two thinning methods, which implement the Hilditch and Hit-and-Miss algorithms for skeletonization. Both approaches use a sliding window which examines the eight pixels surrounding the pixel under consideration (i.e. pixel p1 in Figure 15).

p9	p2	p3
p8	p1	p4
p7	p6	p5

Figure 15. The Sliding Window for Skeletonization.

The window moves through all the pixels in the image, and if certain conditions are met in the surrounding pixels then p1 is set to 0, and so removed from the image. Details on Hilditch can be found at http://jeff.cs.mcgill.ca/~godfried/teaching/projects97/azar/skeleton.html, while Hit-And-Miss is explained at http://fourier.eng.hmc.edu/e161/lectures/morphology/node4.html.

buildTemplate() finds the core of the fingerprint by calling getOrigin() which scans the skel[][] array looking for a point of maximum ridge line curvature (gradient change). The gradients represent the steepness (greatest rate of increase) and direction of that change, and are represented by arrows in Figure 16.

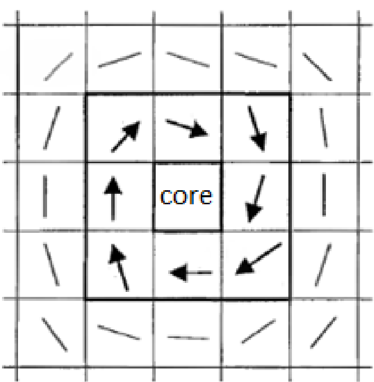

Figure 16. Finding a Core Using Gradient Changes.

These gradients are also known as directional fields, and the most common approach for calculating them is the Poincare Index, explained in "Fingerprint Classification by Directional Fields" by Sen Wang et al. (available at http://aya.technion.ac.il/projects/2005winter/Fingerprint1.pdf), and section 3.6 of *Handbook of Fingerprint Recognition* by Davide Maltoni.

(c) Andrew Davison 2013

3.2. Building the Labeled Image

A typical labeled fingerprint is shown in Figure 14 – the thinned fingerprint, drawn in blue, is overlaid with green and red boxes representing the template's information on termination and bifurcation ridges. The core point is shown as a red "+" symbol.

The image is generated by the labelImage() method in the Fingerprint class:

```
// globals
private static final int LINE_LEN = 5;


private BufferedImage labelImage(byte[][] skel, double[] tmplt,
                                       int imWidth, int imHeight)
{
  double xCore = tmplt[1];   // the core's (x, y)
  double yCore = tmplt[2];

  // draw the finger print
  BufferedImage im = new BufferedImage(imWidth, imHeight,
                                   BufferedImage.TYPE_INT_RGB);
  for (int i = 0; i < imWidth; i++) {
    for (int j = 0; j < imHeight; j++) {
      if (skel[i][j] == 1)
        im.setRGB(i, j, Color.BLUE.getRGB());   // for ridge points
      else
        im.setRGB(i, j, Color.WHITE.getRGB());   // for background
    }
  }

  /* draw a red box for each bifurcation (ridge split)
     and a green box for each ridge end */
  Graphics2D g2d = im.createGraphics();
  for (int i = 7; i < tmplt[0]; i = i+6) {
    if (tmplt[i+4] > 1) {
      // examine "number-of-ends" field for each template entry
      g2d.setColor(Color.RED);   // ridge bifurcation
      g2d.drawRect((int)tmplt[i] + (int)xCore-3,
             (int)tmplt[i+1] + (int)yCore-2, LINE_LEN, LINE_LEN);
    }
    else if (tmplt[i+4] == 1) {
      g2d.setColor(Color.GREEN);  // ridge end
      g2d.drawRect((int)tmplt[i] + (int)xCore-3,
             (int)tmplt[i+1] + (int)yCore-2, LINE_LEN, LINE_LEN);
    }
  }

  // draw the print's center (core)
  g2d.setColor(Color.RED);
  int len = 2*LINE_LEN;
  g2d.drawLine((int)xCore-len, (int)yCore,
               (int)xCore+len, (int)yCore);   // x-axis
  g2d.drawLine((int)xCore, (int)yCore-len,
               (int)xCore, (int)yCore+len);   // y-axis

  return im;
}  // end of labelImage()
```

 (c) Andrew Davison 2013

The tmplt[] array holds the template data: the size of the array's data is stored in tmplt[0], and the core's (x, y) coordinate in tmplt[1] and tmplt[2],. The minutiae on terminations and bifurcations starts at tmplt[7], with each point representing by six elements in the array.

4. Matching Templates

Figures 2 and 3 show the Matcher application which is used to compare a test template with other templates. It displays the labeled images for the test and template with the highest matching score.

Matcher is passed the fingerprint test name, which allows it to load the correct test template separate from the other templates for comparison purposes. Each template is managed by its own MatchInfo object, and an array of them is sorted into descending order by match score. At that point the GUI can show the labeled images and the match scores. All of this is implemented in the Matcher constructor:

```
public Matcher(String pName)
{
  super("Fingerprint Matcher");

  // check if printname has template and labeled image
  if (!FingerUtils.hasTemplate(pName)) {
    System.out.println("No template info found for " + pName);
    System.exit(0);
  }
  if (!FingerUtils.hasLabel(pName)) {
    System.out.println("No label info found for " + pName);
    System.exit(0);
  }

  // get names of other fingerprints
  ArrayList<String> prints = collectPrints(pName);
  if (prints.size() == 0) {
    System.out.println("No other prints found");
    System.exit(0);
  }

  // build match info for the supplied print name
  MatchInfo testFinger = new MatchInfo(pName);

  // build match info for other prints, and their match scores
  MatchInfo[] matches = new MatchInfo[prints.size()];
  for (int i=0; i < prints.size(); i++) {
    matches[i] = new MatchInfo( prints.get(i));
    matches[i].score(testFinger);
  }

  Arrays.sort(matches);
      // sort into descending order by match scores

  makeGUI(testFinger, matches);   // display the results in a GUI

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();
  setResizable(false);
```

 (c) Andrew Davison 2013

```
  setLocationRelativeTo(null);
  setVisible(true);
} // end of Matcher()
```

Each MatchInfo object loads the template for its fingerprint name, and calculates its score against the test template (which is stored in the testFinger MatchInfo object).

The MatchInfo.score() method relies on FingerUtils.match(), which comes from the Biometric SDK:

```
// in the MatchInfo class
// globals
private String printName;
private double[] template;
private int score;
    // result when this object's template and another are matched


public void score(MatchInfo mi)
{
  double[] tmplt = mi.getTemplate();
  if ((tmplt == null) || (template == null)) {
    System.out.println("Could not match templates for " +
                       mi.getPrintName() + " and " + printName);
    score = 0;
  }
  else
    score = FingerUtils.match(tmplt, template, 65, false);
}  // end of score()
```

MatchInfo implements the Comparable interface so that the array of MatchInfo objects can be sorted into descending order by Arrays.sort():

```
public class MatchInfo implements Comparable<MatchInfo>
{
  private int score;

  // other methods not shown ...

  public int compareTo(MatchInfo mi)
  {  return mi.getScore() - score; }

}  // end of MatchInfo class
```

FingerUtils.match() compares two templates by pairing up their minutiae. The relevant call in the code above is:

```
score = FingerUtils.match(tmplt, template, 65, false);
```

A point's (x, y) Cartesian coordinates (and (radius, degree) polar coordinates) are specified relative to the template's core, so the matching involves point rotation around the core. A rotation is applied to all the points in one template, then matched against the other template's points. A count is made of the number of 'close' matches,

and if this exceeds a threshold, then the resulting score is returned. The threshold is specified in the call to match() (65% in the example above), which is the number of matches divided by the total number of minutiae.

It's also possible to set an isFastMatch boolean (the last argument of match()), to make match() return as soon as it finds a score that exceeds the threshold. Alternatively, match() will try a range of rotations, and return the best overall score at the end.

5. Testing the Fingerprint Recognizer

I used the Templater application to build templates for 14 different fingerprints, and generated templates for three test prints which were extra versions of three of those originals. I checked the test prints using the Matcher application to see if it could successfully link them to the originals.

Test 1. 'fingertest1' is another version of 'finger9', which came in second place when Matcher calculated its score (see Figure 17).

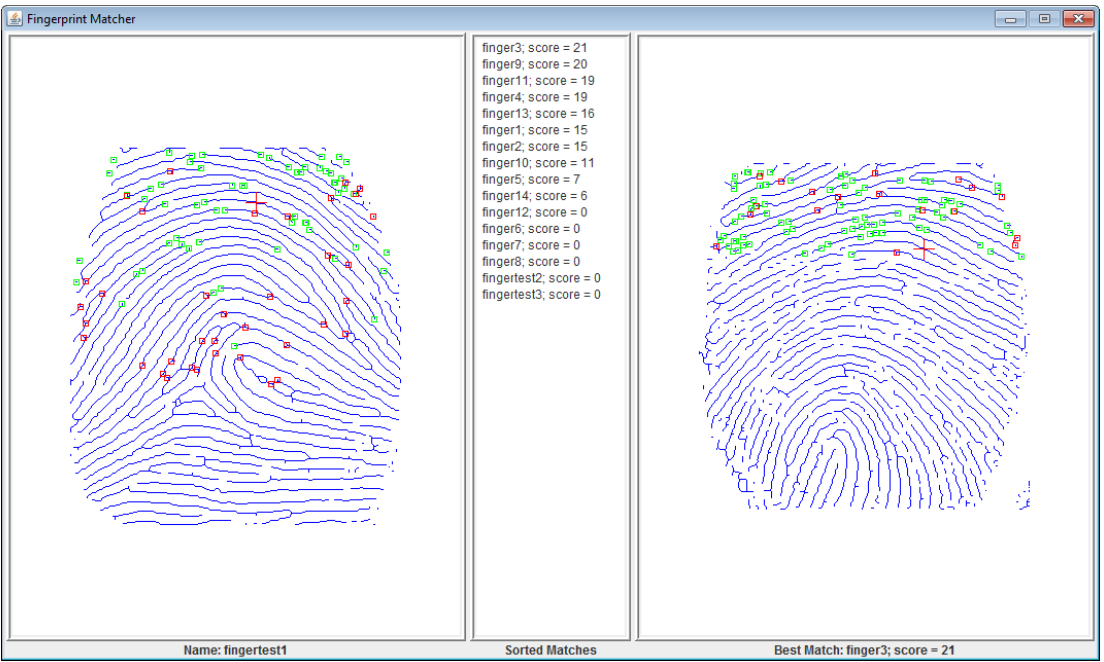

Figure 17. Matches for fingertest1 (finger9 should be first, but is second).

The highest match is a score of 21, while the 'finger9''s score is 20, so it was close.

Test 2. 'fingertest2' is another version of 'finger8', which came in fourth place when Matcher calculated its score (see Figure 18).

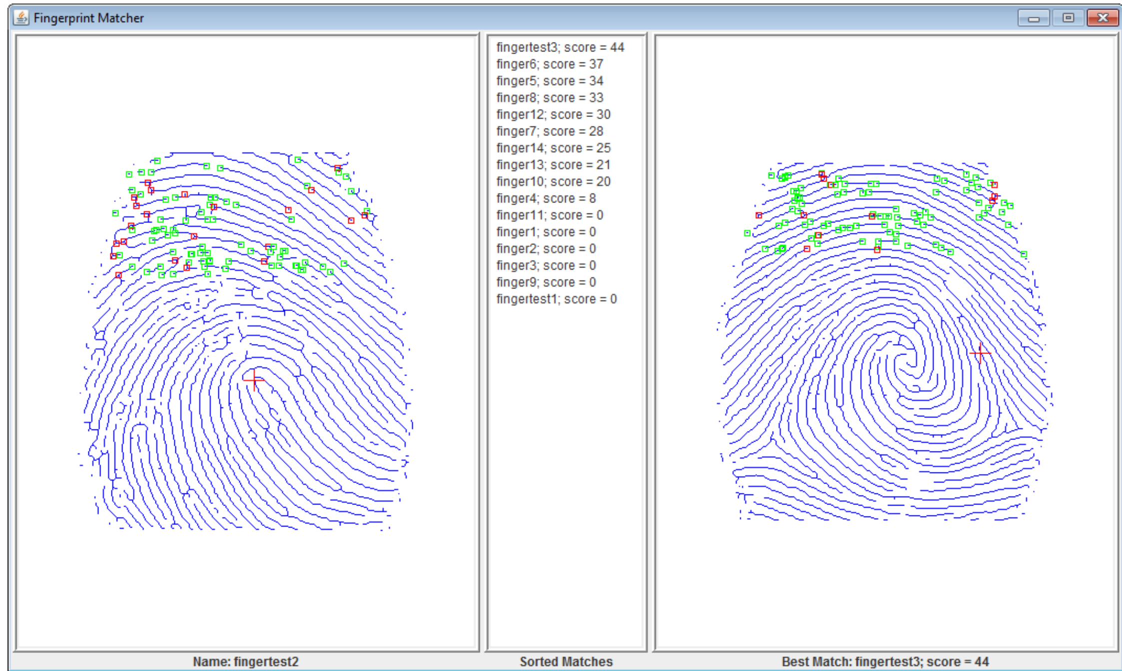

Figure 18. Matches for fingertest2 (finger8 should be first, but is fourth).

The highest match is a score of 44, while the 'finger8''s score is 33.

Test 2. 'fingertest3' is another version of 'finger6', which came in first place when Matcher calculated its score (see Figure 19).

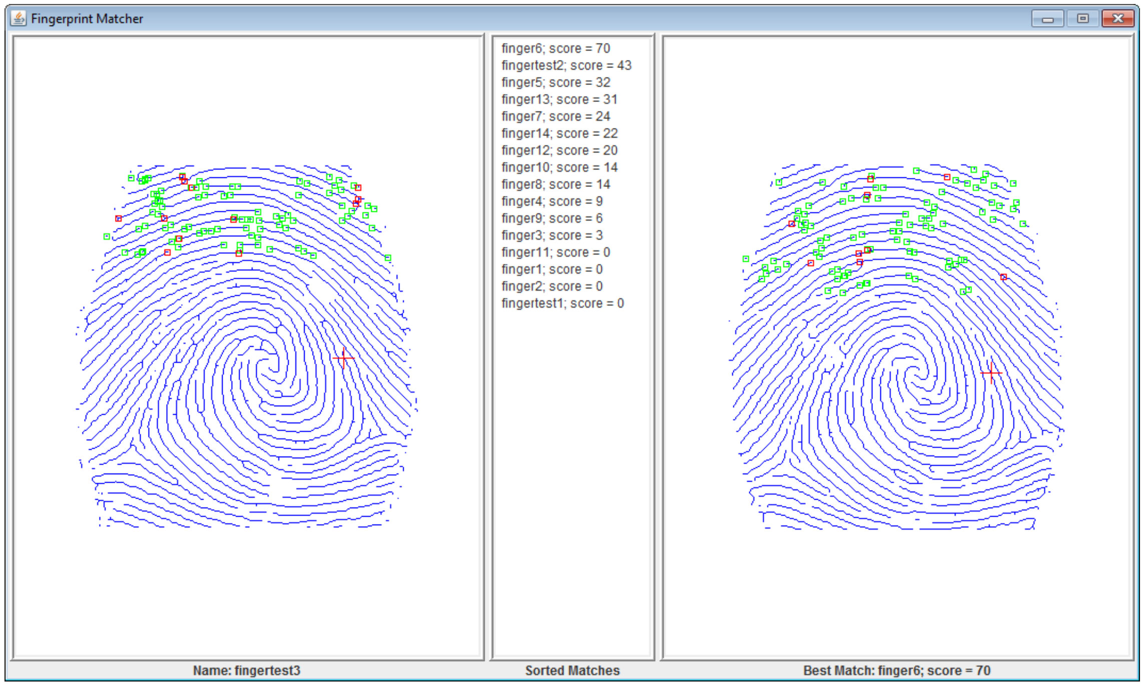

Figure 19. Matches for fingertest3 (finger6 is correctly first).

The match score is 70, the only one of the tests which exceeded the 65% threshold set by Matcher.

Based on these results, it's clear that the quality of the fingerprint analysis is quite low, mainly because of the poor standard of the input images. In particular, image blurring and poor lighting lead to the appearance of gaps in ridge lines, which causes the generation of spurious termination points. Dark areas in an image often lead to the creation of phony ridge lines that became extra bifurcations. For these reasons, a strong lighting source would help improve image quality, but a better webcam would be the best solution.

Another factor is that the core calculation is often inaccurate, which has a severe effect on the other template data since minutiae are positioned relative to the core.

(c) Andrew Davison 2013