Automatic Fingerprint Recognition

Benjamin May & Edward Wastell

March 5, 2014

Abstract

Stuff happened.

1 Introduction

Fingerprints have been used to identify people since the 19th century and have been used in criminal investigations since about that time. More recently fingerprints have been used as biometric markers used in boarder control, library stock control and computer and building access control systems. The need for a robust automatic fingerprint recognition system is obvious.

Most fingerprint recognition systems in use are based on the idea of identifying minutiae (points where a ridge ends or joins with another ridge) — in this article a system that uses the greylevel gradient to find minutiae is discussed.

2 Theory and Implementation

2.1 Point Normal Direction

The point normal function works by taking four mutually adjacent points (*i.e.* a 2×2 array of pixels) and fitting a plane to them. If the greylevel at the pixel (x_k, y_k) is denoted h_k and the level from the fitted plane is denoted p_k then the plane fitting part of the function can be seen as minimising the following expression:

$$\min_{n_1,n_2,c} \sum_k |h_k - p_k|^2 \tag{1}$$

Which in matrix form can be expresed as:

$$\left| \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{pmatrix} - \begin{pmatrix} -x_1 & -y_1 & 1 \\ -x_2 & -y_2 & 1 \\ -x_3 & -y_3 & 1 \\ -x_4 & -y_4 & 1 \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ c \end{pmatrix} \right|^2 \tag{2}$$

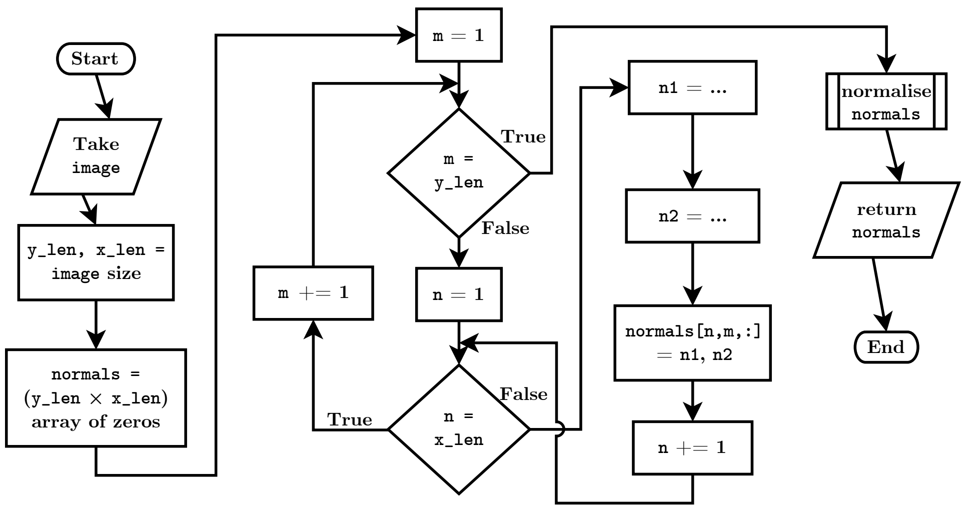

Figure 1: Flow chart of the PND function as implemented.

Where n_1, n_2 and c are the x, y and z components of the surface normal respectively. This is a simple least-squares minimisation and after some rearranging we obtain the following expressions for the optimum surface normal:

$$\begin{aligned} n_1 &= \frac{-h_1 + h_2 + h_3 - h_4}{4} \\ n_2 &= \frac{-h_1 - h_2 + h_3 + h_4}{4} \\ c &= \frac{h_1 + h_2 + h_3 + h_4}{4} \end{aligned} \tag{3}$$

Since we are only concerned with the components in the x, y–plane the function implemented here doesn't bother calculating c (figure 1). Once all 2×2 neighbourhoods have been fitted the function returns n_1 and n_2 then terminates.

As the NPD function needs a four points to fit a plane to the decision has to be made regarding how edges are handled. One possibility is to 'wrap' the edges of the image around, effectively forming a torus — this was not implemented here because it would mean that for two edges the values of n_1 and n_2 would depend on greylevels from the other two edges. The second possibility (implemented here) is to make the output array smaller by 1 pixel in both dimensions, so if the original image is $n \times m$ pixels the array of normals is $(n-1) \times (m-1)$. From figure 1 it can be seen that the PND function implemented here loses the top and right hand pixels.

2.2 Averaged Tangent Direction

The ATD function calculates the x, y–plane tangent that best fits with the surface normals generated by the PND function in a given $n \times n$ neighbourhood. This has the effect of smoothing out (plane normal angle) noise but, as with analogous smoothing operations, can obscure features with a size comparable to that of the kernel chosen. The fitting is a least squares minimisation of the row-by-row dot product:

$$\min_{u_1,u_2} \sum_k |(n_{1,k}, n_{2,k}) \cdot (u_1, u_2)|^2 \\ = U(u_1, u_2) \tag{4}$$

Where $n_{1,k}$ is the k^{th} n_1 value as defined above *e.t.c.* and (u_1, u_2) is the normalised fitted vector. If we define the following:

$$\begin{aligned} A &= \sum_k (n_{1,k})^2 \\ B &= \sum_k (n_{2,k})^2 \\ C &= \sum_k n_{1,k} n_{2,k} \end{aligned} \tag{5}$$

then we can express $U(u_1, u_2)$ as

$$U(u_1, u_2) = \\ (u_1, u_2) \begin{pmatrix} A & C \\ C & B \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \tag{6}$$

The eigenvalues of the above 2×2 matrix, λ_1 and λ_2, are the maximum and minimum values of U respectively. It is trivial to determine that the two eigenvalues are given by:

$$\begin{aligned} \lambda_1 &= \frac{A + B + \sqrt{(A-B)^2 + 4C^2}}{4} \\ \lambda_2 &= \frac{A + B - \sqrt{(A-B)^2 + 4C^2}}{4} \end{aligned} \tag{7}$$

With some further rearranging of equation (6) we find that:

$$u_1 = \sqrt{\frac{1}{1 + \left(\frac{\lambda_2 - A}{C}\right)^2}} \tag{8}$$

$$u_2 = u_1 \left(\frac{\lambda_2 - A}{C}\right) \tag{9}$$

In implementing these equations several special cases have to be taken into account. The first special case is when $C = 0$, which would require deviding by zero. To cope with this case the function is set to test if $C = 0$ and if so the function will set $(u_1, u_2) = (1, 0)$ or $(u_1, u_2) = (0, 1)$ depending on weather $A < B$ or $B > A$ respectively. The second special case is when $\lambda_1 = \lambda_2$, which the function tests for after determining if $C = 0$. In this case there is no optimum orientations thus $(u_1, u_2) = (0, 0)$.

A final problem with implementing equation (8) is the fact that u_1 can only ever be positive. This ensures that in the x–direction one cannot distinguish between the two sides edges of a ridge. To solve this the function calculates the mean of the n_1 values then checks if this mean is negative and if so it makes u_1 negative.

2.3 The Ridge Follower

The ridge following system takes a slice (of width $2\sigma + 1$ and height 1 pixel) of the image, finds the peak of the ridge and moves to it, calculates the direction of the ridge then moves forward by μ pixels before starting the process again (figure 2). In this manor the ridge follower follows the ridge, and since the function logs where it's been it's possible to determine where the minutiae are.

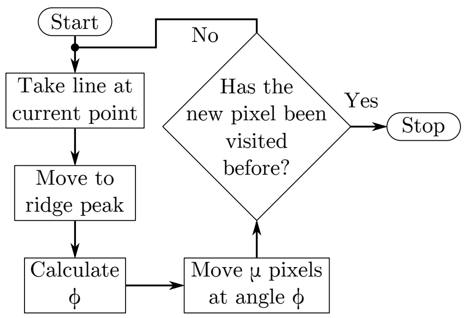

Figure 2: Simple version of the ridge following algorithm.

Taking a line at a given point is simple enough

3 Experimental Procedure

4 Results & Analysis

5 Conclusion

A Code

```
#! /usr/bin/python3

## Basic fragments of code that will probably be useful

## Uses axes, imshow, colorbar, title, savefig, close, figure
## from matplotlib.pyplot
from matplotlib.pyplot import axes, imshow, colorbar, title, \
    savefig, close, figure
def show_pic(image,plot_name='Test Image',colourmap=None):
    """Method to simply shwo the image. Useful for tests"""
    figure(facecolor='white', figsize=(5,4))

    # Remove axes from the plot
    ax = axes(frameon=False)
    ax.get_yaxis().set_visible(False)
    ax.get_xaxis().set_visible(False)
```

```python
    # Give the image a title
    title(plot_name)

    if colourmap != None:
        # Make image with a colourmap
        imshow(image, interpolation='none',cmap=colourmap)
        colorbar()
    else:
        # Make an image without a colourmap
        imshow(image, interpolation='none')


def save_pic(image, plot_name, colourmap=None):
    """A function that nicley abstracts producing an image object"""
    figure(facecolor='white', figsize=(5,4))

    # Remove axes from the plot
    ax = axes(frameon=False)
    ax.get_yaxis().set_visible(False)
    ax.get_xaxis().set_visible(False)

    # Give the image a title
    title(plot_name)

    if colourmap != None:
        # Make image with a colourmap
        imshow(image, interpolation='none',cmap=colourmap)
        colorbar()
    else:
        # Make an image without a colourmap
        imshow(image, interpolation='none')
    savefig("{:s}.pdf".format(plot_name).replace(' ', '_'))
    close()

## Uses sqrt, sum
def normalise(array):
    """Normalises a 3D array"""
    from numpy import sqrt
    yl = len(array)
    xl = len(array[0])
    for n in range(yl):
        for m in range(xl):
            y = array[n,m,0]
            x = array[n,m,1]
```

```
            if x == 0 and y == 0:
                array[n,m,:] = 0 , 0
            else:
                array[n,m,:] = array[n,m,:]*(x**2 + y**2)**(-0.5)

    return array

## Uses zeros and array from numpy
def pnd(image):
    """Returns the point normal determination"""
    from numpy import zeros, array
    # Get image dimensions
    y_pix = len(image)
    x_pix = len(image[0])

    # Make array to hold normals
    normals = zeros((y_pix-1, x_pix-1,2))

    # Loop through and calculate the normal vector
    for n in range(1, y_pix):
        for m in range(1, x_pix):
            n1 = (-image[n, m] + image[n-1, m] + image[n-1, m-1] \
                  - image[n, m-1])/4
            n2 = (-image[n, m] - image[n-1, m] + image[n-1, m-1] \
                  + image[n, m-1])/4

            normals[n-1, m-1, :] = n1, n2

    normals = normalise(normals)

    return normals

## Uses zeros, sum, sqrt from numpy
def atd(image, window=9):
    """Returns the averaged tangent diraction of a normal array"""
    from numpy import zeros, sum, sqrt, array, mean
    # Get image dimensions
    y_pix = len(image)
    x_pix = len(image[0])

    # Calculate window half-size
    k1 = int(window/2)
    if k1 == window/2:
```

```
    k2 = int(window/2) - 1
else:
    k2 = k1

# Make array to hold tangents
tangents = zeros((y_pix-1, x_pix-1,2))

# Calcualte normals
normals = pnd(image)

# Test to see if a pixel is in the image
inImageTest = lambda ky,kx : not ((ky < 0) and (kx < 0) and \
                                  (ky > y_pix - k1 - 1) and \
                                  (kx > x_pix - k1 - 1))

# Loop through and calculate ATD
for n in range(1,y_pix-(k1+1)):
    for m in range(1,x_pix-(k1+1)):
        # Extract window from image
        window = [normals[ky,kx] for ky in range(n-k1,n+k2) \
                  for kx in range(m-k1,m+k2) if inImageTest(ky,kx)]

        # Make window a numpy array for easier indexing
        window = array(window)

        # Calculate normalised ATD
        A = sum(window[:,0]**2)
        B = sum(window[:,1]**2)
        C = sum(window[:,0]*window[:,1])
        mn1 = mean(window[:,0])

        # Diagonal matrix case
        if C == 0 and A < B:
            u1 = 1
            u2 = 0

        elif C == 0 and A > B:
            u1 = 0
            u2 = 1

        # Plateau
        elif ((A+B) + sqrt((A-B)**2 + 4*C**2)) == \
               ((A+B) - sqrt((A-B)**2 + 4*C**2)):
            u1 = 0
```

```
                u2 = 0

            # Non-diagonal case
            else:
                D = (B - A)/(2*C) - sqrt(1 + ((B - A)/(2*C))**2)

                # Ensure that u1 points in the correct direction
                if mn1 > 0:
                    u1 = 1/sqrt(1 + D**2)
                else:
                    u1 = -1/sqrt(1 + D**2)

                u2 = u1*D

            tangents[n,m,:] = u1, u2

    return tangents

def OC1(A, B, C, D, E, xl, yl):
    """Calculates the curvature point via method 1 - needs ADT results"""
    # Straight line case
    if A*B == C**2:
        x = xl/2
        y = yl/2

    # Curved line case
    else:
        x = (B*D - C*E)/(A*B - C**2)
        y = (A*E  - C*D)/(A*B - C**2)

    pc = x, y

    return pc

def OC2(A, B, C, D, E, M, xl, yl):
    """Calculates  the curvature point via method 2 - needs ADT results""
    # Two points case
    if not M == 0:
        a = D**2/E + (A - B)*D - E
        b = (B - A)*M - D**2 - E**2 + 2*M*D/E
        c = (C*M/E + D)*M

        x = (-b - (b**2 - 4*a*c))/(2*a)
        y = (M - D*x)/E
```

```
    # Infinite curvature case
    else:
        x = xl/2
        y = yl/2

    pc = x, y

    return pc

## Uses sum from numpy
def OC_switch(window, threshold=0.1):
    """Decided which OC function to use"""
    from numpy import sum
    # Get the window size
    yl = len(window)
    xl = len(window)

    # Calculate vector weighting parameter
    r = [n*window[n,m,0] + m*window[n,m,1] for n in range(yl) \
         for m in range(xl)]

    # Calculate curvature paramaters
    A = sum(window[:,:,0]**2)
    B = sum(window[:,:,1]**2)
    C = sum(window[:,:,0]*window[:,:,1])
    D = sum(window[:,:,0]*r)
    E = sum(window[:,:,1]*r)
    M = sum(r**2)

    # Calculate decision eigenvalue
    l = (A + B - ((A - B)**2 + 4*C**2)**0.5)/(2*yl*xl)

    # Decide which OC to use
    if l > threshold:
        pc = OC1(A, B, C, D, E, xl, yl)
    else:
        pc = OC2(A, B, C, D, E, M, xl, yl)

    return pc
```