Evaluation of transformation methods to detect structures in fingerprints

Sebastian Boßung
bossung@gmx.de

July 11, 2002

Abstract

Three transformation methods for detecting structures of fingerprints are evaluated: The Covariance transformation can be used to find the center-point of arch-type fingerprint images. The variance transformation provides means to localize regions of special interest that can be further searched for details. It too allows center-localization in arches. It is also investigated whether it is possible to determine the type of a fingerprint image by means of its Fourier transformation. This performed very poorly.

Contents

1 Introduction **2**
- 1.1 Context . . . 2
- 1.2 Classes of Fingerprints . . . 2
- 1.3 Orientation Fields . . . 3
- 1.4 Region of Interest . . . 4

2 The Covariance Transformation **5**
- 2.1 Original Idea and Application . . . 5
- 2.2 Details on Connected Components . . . 6
- 2.3 Investigation . . . 6
- 2.4 Room for Improvements . . . 8

3 The Variance Transformation **9**
- 3.1 Definition . . . 9
- 3.2 Detecting areas of special interest . . . 9
- 3.3 Minimal-Phase System . . . 10
  - 3.3.1 Example . . . 10
  - 3.3.2 The General Case . . . 10
- 3.4 Quality of Components . . . 11
- 3.5 Investigation . . . 12
- 3.6 Improvements over the Covariance Transformation . . . 12
- 3.7 Practical use . . . 13

4 The Fourier Transformation **14**
- 4.1 Definition . . . 14
- 4.2 Classification Criteria . . . 14
- 4.3 Implementing a Test . . . 16
- 4.4 Empirical Analysis . . . 16
- 4.5 Conclusion . . . 18

A The Code **19**
- A.1 C-Code for Variance and Covariance Transformation . . . 19
  - A.1.1 covar.c . . . 19
  - A.1.2 types2.h . . . 27
- A.2 MATLAB File visualizing the Fourier Transformation . . . 28
- A.3 C-Code the Fourier Transformation Test . . . 28
  - A.3.1 classfft.c . . . 28
- A.4 Tool Files . . . 31
  - A.4.1 utils2.c . . . 31

Chapter 1

Introduction

1.1 Context

This thesis fits into a project on the topic of robust recognition of fingerprints. The algorithms to detect singular points (see section 1.2), orientations fields (see section 1.3), and regions of interest (see section 1.4) were taken from this project. They are not discussed here.

1.2 Classes of Fingerprints

At the tip of each finger humans have a characteristic pattern called the fingerprint. This pattern is believed to uniquely identify a person. Using this property brings to mind a variety of applications that involve matching a fingprint image agains a database and similar things.

Generally speaking, fingerprints are pattern formed by ridges and furrows. Most of these patterns contain certain points of special interest called singular points. There are two kinds of points: cores and deltas. Both are points at which ridges terminate. The difference is that in the local region of cores the ridges form a semi-circular pattern, while around deltas the pattern is of hyperbolic shape.

Fingerprints are commonly divided into five classes. The usual classes are (see figure 1.1 for an example of each class):

1. *Arches:* In an arch the ridges form a bump. Notice that arch-type fingerprint images contain neither cores nor deltas.
2. *Left Loops:* The majority of the ridges comes from the right (from the right when looking at the finger, keep in mind that the image is flipped) and turn around near the center of the image to move out of the image again on the right side. Left loops generally have one core (the turning point) and one delta to the left of the core.
3. *Right Loops:* Analogous to left loops but flipped over.
4. *Whorls:* The general pattern in whorls are concentric circles. Inside the innermost circle one usually finds two cores (not very obvious in the example image). There are also deltas to the left and right of the concentric-circle pattern.
5. *Tented Arches:* Tented arches are similar to plain arches. However, the bump is steeper, which leads to a delta underneath the bump.

It is possible to classify a fingerprint image based upon the singular points found by searching the ridges (see [5] for details on the process). Tracing the whole image is avoidable by using the variance transformation described in chapter 3.

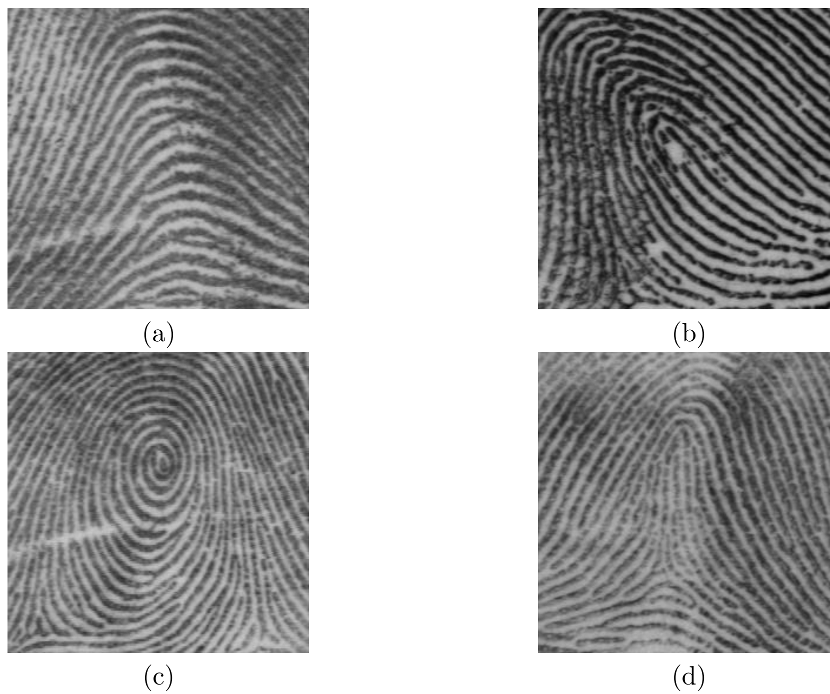

Figure 1.1: Classes of fingerprints: (a) Arch, (b) Loop, (c) Whorl, (d) Tented Arch

1.3 Orientation Fields

Many algorithms in fingerprint processing are based on the orientation field of the image. The orientation field is a matrix that contains directions represented as angles. As fingerprints are composed of ridges and furrows, one can determine the tangential direction at each point along a ridge. In pratice this direction is not calculated at every point, because this would mean extrodinary computational cost. Rather the image is divided into squares and an average tangential direction is computed for each square. An edge length of 17 for the squares has turned out to be suitable when dealing with images of size 512×512.

The local orientations are computed for each square and stored in a matrix of appropriate dimensions. The orientations themselves are represented as angles ϕ where $\phi \in [\pi/2 \ldots 3\pi/2]$ (also see figure 1.2).

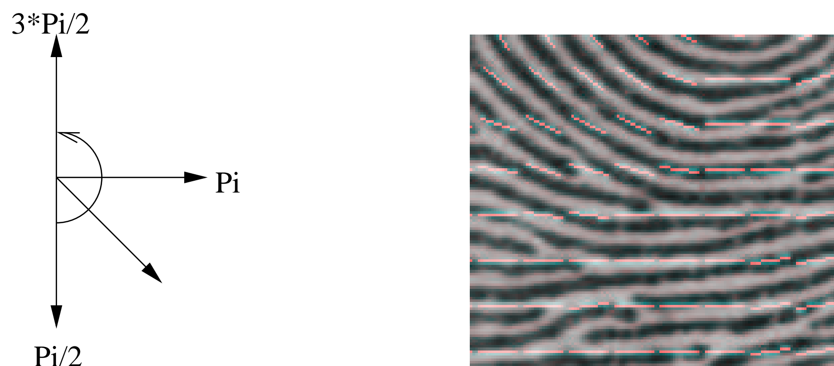

Figure 1.2: Local orientations in a fingerprint

1.4 Region of Interest

Many fingerprint images contain regions that either do not belong to the fingerprint at all (the white regions along the edge of the image) or are of very bad quality. Including these regions in further processing is not desireable as it is not possible to specify a local orientation inside them. Thus the algorithm constructing the orientation field will yield incorrect results, which in turn confuse detection methods that are based on the orientations field, i.e. the ones using variance and covariance transformation (see sections 2.3 and 3.5 on why this is a problem).

It is necessary to determine which regions of the image belong to a fingerprint imprint and are of sufficient quality to make further processing possible. All points that meet these criteria are considered to be inside the region of interest. In this particular project the region of interest is made up of blocks of size 17 just as the orientation field.

The region of interest is stored in a matrix with values 0 if the corresponding square is inside the region of interest, and 255 if it is not.

Chapter 2

The Covariance Transformation

2.1 Original Idea and Application

The covariance transformation is used in [3] to determine the center of an arch-type fingerprint image. As arches do not contain singular points, this is necessary in order to find the region of the image that is characteristic and thus needed for recognition.

The covariance transformation is based on the orientation field (see section 1.3) of the fingerprint image. It is computed by the following steps:

1. For each point (x, y) of the orientation field matrix take the submatrix of dimensions (q x q) centered at (x, y). q is a parameter to the algorithm. In this implementation $q = 3$ was used[1].

2. Compute the submatrix' covariance-matrix. The covariance-matrix is defined as[2]:

$$C := \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})(x_i - \bar{x})^T \tag{2.1}$$

   Where x_i are the columns of the orientation matrix and $\bar{x}$ is the vector consisting of the column's means.

3. Next the maximum eigenvalue λ of C is determined. This is done using a modified Jacobi method (for details on the Jacobi method see [2], for the corresponding C-code see A.1). Using the Jacobi method is possible because covariance matrices are always symmetric and the values of the orientation field are all real yielding a real covariance matrix. The Jacobi method usually gives both eigenvalues and eigenvectors. Here only the eigenvalues are required.

4. The value of the covariance transformation at (x, y) is the maximum eigenvalue of the corresponding covariance-matrix C: $c_{xy} = \lambda$.

In this implementation a region of interest as computed in earlier processing steps is used (see section 1.4 for details). Points outside the region of interest are marked yellow in figure 2.1 and are not considered to be part of a valid fingerprint image. The covariance transformation is computed only if all points of the submatrix from step (1.) are inside the region of interest. Neglecting the region of interest would take orientations into account that do not correspond to any rigde in the fingerprint (as an example consider the lower left corner in figure 2.1).

As proposed in [3] the covariance transformation can then be used to locate the center point of an arch-type fingerprint image. To do this, the matrix C is thresholded with a threshold level

[1]This value can be changed by defining `LOCALSIZE` in `covar.c` to something other than 3. Please note that the value has to be odd.

[2]You can also think of the covariance matrix as: $c_{ij} = E((x_i - \mu_i)(x_j - \mu_j))$ where x_i is the i-th column interpreted as stochastic variable and μ_i is the mean of the i-th column.

of $max(C)/6$ (where $max(C)$ is the maximum element of C). The value of 6 was determined empirically.

Next, the thresholded matrix is searched for connected components. The center of the fingerprint image is considered to be at the geometric center of the largest connected component. See figure 2.1 for an example of the center detection.

2.2 Details on Connected Components

Usually connectedness is interpreted in either a four or an eight-connected way. Both types are implemented[3]. The four-connected components were found to perform better. In the eight-connected mode two components lying close to each other are able to diagonally merge and connect to one single component. This behavior is discouraged by using the four-connected mode.

Another problem is the tendency of components to degenerate to long lines of width one in some images. These lines are disallowed as they are only one-dimensional and therefore do not give a representation of the two-dimensional variance of the orientation field. In addition to being part of the component in a four or eight-connected sense, a point also has to meet one of the following criteria, which prevent the formation of lines:

1. It has more than two neighbours in the eight-connected sense.
2. If it only has two neighbours (in the eight-connected sense) these neighbours have to be adjacent.

These criteria are implemented in the function `valid_neighbourhood` in `covar.c` (see A.1 for code). Together with four-connectedness they best prevent the formation of long thin lines.

2.3 Investigation

The covariance transformation was tested on 25 arch-type fingerprint images of decent quality[4]. It found a point close to what humans would classify as the center on most of them. Exact numbers are hard to give, because the precise location of the center in an arch-type fingerprint is disputable.

Unusual patterns in the fingerprint, such as scars, prove to be a problem for the covariance transformation. In the region of scars the local orientations change rapidly yielding a high value in the covariance feature image. In the worst case the direction of the scar is orthogonal to the local orientation of the ridges. The change in orientation around the scar is generally greater than the change anywhere else in the image. Thus the biggest - and often only - connected component is detected at the scar, its center is returned as the (false) center of the fingerprint.

Low-quality images tend to have regions where the ridges and furrows are not clearly distinguishable. Usually these regions will be disallowed in the region of interest algorithm. However, in some cases they are not, which confuses the algorithm constructing the orientation field into yielding wrong values for these regions. The covariance feature image can then assume high values, depending on the (mis-)detected orientations. While being a problem of the orientation field algorithm, this still is fatal for the covariance transformation and its center location.

The covariance transformation also suffers from the problem discussed in section 3.3 below concerning almost vertical directions in the orientation field. This is what stops covariance transformation from being applied to non-arch fingerprints in a sensible manner. To fix this the covariance matrix would have to be reconsidered in a similar way the variance is in the next chapter. This does not seem worthwhile, since the variance transformation can substitute the covariance transformation in every way.

[3]The type of connectedness used can be specified by defining `CONNECTEDNESS` in `covar.c` to be `4` or `8`.

[4]Decent means the fingerprint covers most of the image, there are no foreign artefacts (such as handwriting or print) in the image, and most of the image is inside the region of interest (especially the center).

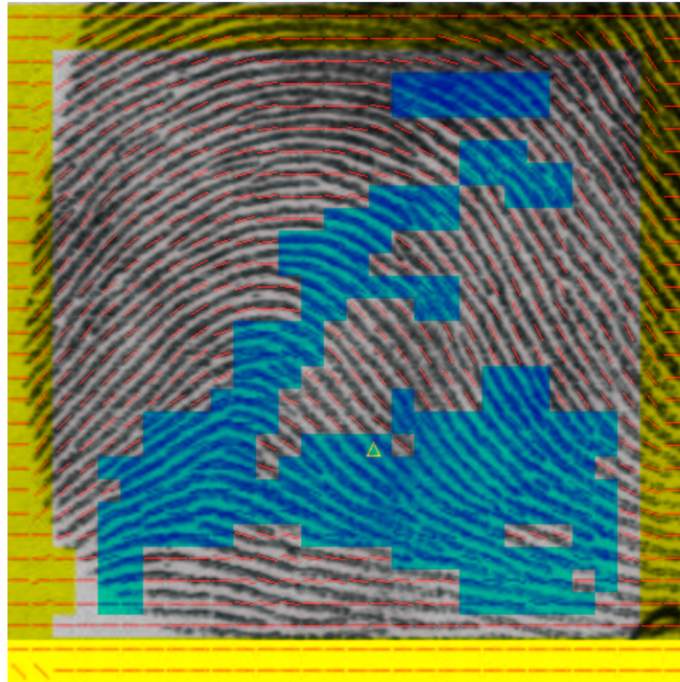

Figure 2.1: Example of the covariance transformation. Yellow: not inside the region of interest, red lines: local orientations as in the orientation matrix, blue: connected components, yellow triangle: center of the largest connected component.

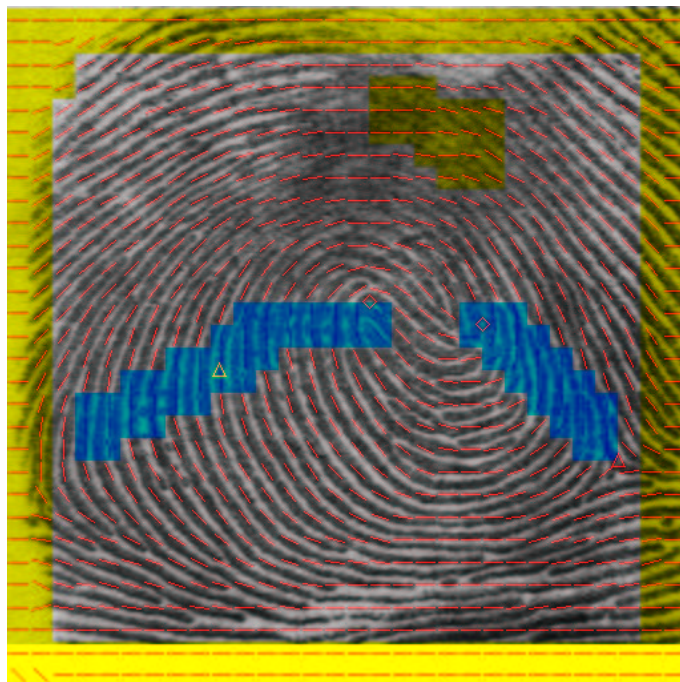

Figure 2.2: Example of a covariance misdetection of a non-arch. High covariances in the orientation field are detected in places where there are none.

2.4 Room for Improvements

1. Locating the center point (or other points of interest) only works for arch-type input images.
2. Computing the covariance transformation is quite time-consuming.
3. The covariance should be more robust against unusual regions - such as scars - in fingerprints. However, this seems hard to achieve, if one does not know beforehand that the presented image is an arch. The problem with scars is not unique to the covariance transformation. The type of these images might be misdetected by common algorithms - based on searching for singular points - as well.

Chapter 3

The Variance Transformation

3.1 Definition

Just as the covariance transformation the variance transformation is based on the orientation field of the fingerprint image. It evolved from the covariance transformation and was developed with the weaknesses of the latter in mind.

The computation of the variance transformation is simpler and less costly than that of the covariance transformation as it involves less computation in the following steps:

1. For each point (x, y) of the orientation field take the submatrix of dimensions (q x q) centered at (x, y). q is a parameter to the algorithm. In this implementation $q = 3$ again was used[1].
2. Although not absolutely necessary for the variance transformation to work, this implementation copies the elements of the submatrix into an array of size q^2. This facilitates further computations.
3. Determine the minimal-phase system for the orientations in the array. See section 3.3 on why this is necessary and how it is done.
4. Compute the ordinary variance of the array elements as:

$$\sigma^2 = \sum_{i=1}^{q^2} x_i^2 - \mu^2 \tag{3.1}$$

   Where x_i are the elements of the array and μ is the average of the array.
5. The value of the variance transformation at (x, y) is σ^2. Store this value in a matrix: $(V)_{xy} = \sigma^2$. V is called the variance feature image and has the same dimesions as the orientation field matrix.

3.2 Detecting areas of special interest

The variance transformation can be used to detect areas of special interest[2] as well as center points of arches. This works well on any decent quality image. In addition to center location in arches the variance transformation will also identify regions that contain singular points (see section 1.2) in non-arch fingerprints, because the change of the directions in the orientations field is high in

[1] This value can be changed by defining `LOCALSIZE` in `covar.c` to something other than 3. Please note that the value has to be odd.

[2] Note that "area of special interest" is not the same thing as the region of interest described in section 1.4. Rather the term referrs to the areas of the fingerprint image that are likely to contain points which can be useful to classify or to detect the fingerprint.

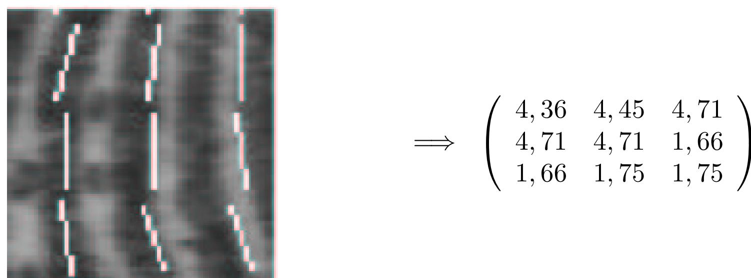

Figure 3.1: Example of a critical area in the orientation field.

the neighbourhood of these points. These regions can be further searched for singular points using classical algorithms.

The process of selecting the appropriate areas is almost identical to that used with the covariance transformation for finding the center of an arch:

1. Find the maximum element v_{max} in the variance transformation matrix V.
2. Threshold the matrix V with a threshold value of $v_{max}/5.2$. The value of 5.2 was determined empirically. Depending on the source of the images there might be room for improvements here.
3. Search for connected components in the thresholded matrix as described in section 2.2.

3.3 Minimal-Phase System

3.3.1 Example

Consider the detail of a fingerprint image shown in figure 3.1. The corresponding part of the orientation field matrix is:

$$A = \begin{pmatrix} 4,36 & 4,45 & 4,71 \\ 4,71 & 4,71 & 1,66 \\ 1,66 & 1,75 & 1,75 \end{pmatrix} \tag{3.2}$$

The variance of the system A is 2.32. This is quite a lot, considering that in reality the orientations differ very little, as all are close to vertical. The problem is, that all orientations are from $[\pi/2 \ldots 3\pi/2]$. This forces similar orientations to differ by almost π if both are close to the ends of the range. However, one can add π to an orientation without changing it, because orientations do not have a direction. Adding π to some orientations might decrease the difference between the smallest and largest element (this difference is also called the phase). The resulting system might then have a smaller variance.

Thus the values in A are equivalent to (add π to those orientations < 4):

$$A' = \begin{pmatrix} 4,36 & 4,45 & 4,71 \\ 4,71 & 4,71 & 4,80 \\ 4,80 & 4,89 & 4,89 \end{pmatrix} \tag{3.3}$$

The variance of A' is only 0.034. Unfortunately, some values are outside the allowed range for orientations. This can be fixed by subtracting an offset $f = \pi - a'_{min}$, where a'_{min} is the minimal element of A'. However, this is not necessary when using the minimal phase system A' in the variance transformation, because the variance is invariant to the offset.

3.3.2 The General Case

When computing the variance of the local orientations, one is interested in how much the directions actually differ. Unfortunately the direction itself is not the same as the angle it is represented by.

Figure 3.2: Example of the quality computation of a 2×2 square. Ideal case on the left, discrete case on the right. Note that the centers are different (circles). In the discrete case the coordinates of each tile is also shown (solid dots). The maximum radius is marked with an arrow. The quality in the ideal case is $Q_{ideal} = \frac{4}{(\sqrt{2})^2} = 2 < \pi$. The discrete case however has a quality of $Q_{discrete} = \frac{4}{(\sqrt{2}/2)^2} = 8 > \pi$.

The difference becomes clearer when considering that every direction d_i can be represented by an angle $\alpha_i + n_i\pi$, where $n_i \in \mathbb{N}$ and α_i is unique to the direction. The problem arising from this is that the variance in the local neighbourhood depends on the value of each n_i. As a result, the same system of directions can cause the variance transformation to have different values depending on the angles that represent the system. See the above example for further illustration.

Depending on the choice of n_i for each angle the variance will vary. To find the system of angles with the minimum phase the following algorithm is suitable (it assumes that the angles are stored in a vector a as is the case when computing the variance transformation):

1. Sort the vector elements to be in ascending order. Let ϕ be the initial phase ($\phi = a_n - a_1$) of the system, let $x = -1$.
2. For each element a_i check if $(a_i + \pi) - a_{i+1} < \phi$. (This condition checks whether flipping all direction up to the i-th over by 180° will decrease the phase ϕ of the system.)
3. If so, let $\phi = a_{i+1} - a_i$. Set $x = i$. Go back to (2.)
4. If no further improvement can be made, that is condition (2.) is not true, add π to all a_i, $i = 1 \ldots x$.

For cosmetic reasons one can then sort the vector again and subtract $f = \pi - a_1$ from all elements (f as above). This yields a system with $a_i \in [\pi/2 \ldots 3\pi/2]$. In the context of the variance transformation this is not necessary.

3.4 Quality of Components

When taking a closer look at the connected components yielded by the variance transformation, one discovers that the components containing multiple singular points usually have a long, stretched shape. The ratio of a component's area A to the square of its radius r (here called quality Q of the component) is used in an attempt to measure this:

$$Q = \frac{A}{r^2} \tag{3.4}$$

Note that a circle would have a quality of π. In an ideal world this would be the maximum quality any component could have. Unfortunately in the discretized world of the orientation field (where the connected components are made up of squares with size 17×17) much higher values for Q are possible. For small components this is especially problematic (see figure 3.2 for an example), whereas on larger components the quality approaches the value of the continuous case. This makes is impossible to compare qualities between components of different sizes. Therefore the concept of qualities is of little practical use.

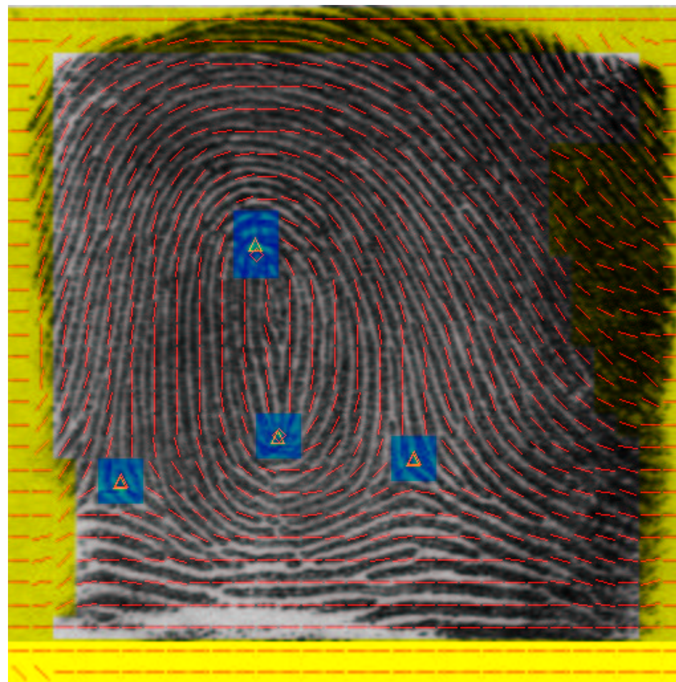

Figure 3.3: Example for the variance transformation. Yellow: not inside the region of interest, red lines: local orientations as in the orientation matrix, blue: connected components, yellow triangle: center of the largest connected component.

3.5 Investigation

The variance transformation reliably detects connected components such that each singular point is inside a component. The only exception to this are singular points that are only one square away from the edge of the region of interest. No connected component is found for them, because there are no 3×3-neighbourhoods around the singular point.

Singular points are usually located very close to the center of the connected component they are situated in. However, by means of the variance transformation there seems to be no way of knowing if the connected component contains a singular point at all (i.e. in arches it does not). As it is also not possible to find the singular point's type, one cannot determine the class of the fingerprint by the variance transformation alone.

One remaining problem is, that there might be more than one singular point in a connected component. A way to detect the number of points in a component using only the variance transformation was not found. If the image is of good quality and is not an arch, the size of a connected component is related to the number of singular points inside of it. As this is not a general rule applicable to all input images, no algorithm can be based upon it.

Nevertheless the process of detecting the singular points is facilitated, because only a small portion of the image needs to be checked. For example a typical whorl (see figure 3.3) with two cores and two deltas will exhibit connected components of a total size of about 24 while the size of the complete image is $30 \times 30 = 900$.

3.6 Improvements over the Covariance Transformation

1. Depending on the input image the variance transformation is three to seven times faster than the covariance transformation (the broad range of three to seven is due to the uncertain number of iterations the Jacobi method has to go through when computing the maximum eigenvalue in the covariance transformation).

2. It is possible to isolate regions with singular points in non-arch type images. This makes it possible to treat all images with the same algorithm, whereas the covariance transformation only works on arches. When using the variance transformation, it is not necessary to detect the fingerprint's class beforehand.

3.7 Practical use

Commonly fingerprint classification is carried out in the following steps:

1. Find local orientations
2. Detect region of interest
3. Search the whole image for singular points
4. Based on the detected singular points the image can be classified.

Inserting an additional step using the variance transformation provides means to locate the center point of arches. It also gives an estimate on the location of the singular points, that can be used in further processing:

1. Find local orientations
2. Detect region of interest
3. Compute the variance transformation to preselect interesting areas.
4. Search the connected components for singular points
5. Based on the detected singular points the image can be classified.

After the third step one can draw conclusions based on the variance transformation using the following rules:

1. If the largest connected compontent detected by the variance transformation covers more than 5% of the image, one can assume that the image is an arch. The center of the arch is the center of the connected component. No further processing is required.

   This rule was determined empirically. It is based on the fact that during evaluation of the variance transformation no arch was encountered that violated it. Should there be an arch that does, it will be dealt with by rule 3. (An example for this could be an image where the region of interest is very small, in other words, large portions of the image do not belong to a fingerprint).

2. Otherwise all components of sizes up to 9 will only contain one singular point, which will be very close to the center of the component. Its type can be detected by means of the Pointcaré index (see [5]) if this is desired.

   This rule is an empirical one as well. If one finds that there are counter-examples in a particular system, it might be necessary to decrease the block size in the orientation field algorithm (see section 1.4). A block size of 17 with an image size of 512×512 lead to at most one singular point in components of sizes up to 9.

3. Larger components might contain more than one singular point and will have to be searched by classical means (again see [5] for more information on this topic). Arches that did not conform to rule (1.) will be detected in this step.

Chapter 4

The Fourier Transformation

4.1 Definition

The Fourier transformation $F(\omega_1, \omega_2)$ of an image $f(m, n)$ is defined in the usual way (see for example [1] pp. 19 for details):

$$F(\omega_1, \omega_2) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m, n) e^{-j(m\omega_1 + n\omega_2)} \tag{4.1}$$

The resulting frequency-domain image is then shifted, such that the origin ($\omega_1 = 0, \omega_2 = 0$) is at the center of the image. When referring to "the Fourier transformation", the shifted version is meant.

4.2 Classification Criteria

Fingerprints are mainly composed of ridges and furrows. When moving along the image the gray values are repeated in an almost periodic pattern. This is due to the ridge-to-ridge distance varying very little over the area of the fingerprint. The periodicity of the pattern suggests to look at the Fourier transformation of the image. As the frequency is almost the same along any direction, one does expect the frequency domain image to strongly resemble a circle. When looking at the example in figure 4.1 this is indeed the case.

When considering the loops, arches and whorls more closely, one can imagine that the frequence domain image of a whorl will be a full circle. This is because the whorl is almost indifferent to rotation. A loop on the other hand might have some frequencies missing, since there is an area (in the lower left or right corner, depending on what type of loop one is dealing with) resulting in a Fourier transformation similar to a circle except for a pie-slice missing in the left or right corner. An arch will have a slice missing on the left as well as the right.

This idea is presented in [4]. Fitz and Green express it a little differently:

1. *Arches* will have the majority of frequencies between 45° and 135° (range[1] 90°).
2. *Whorls* between 0° and 180° (range 180°).
3. *Loops* between 30° and 120° (range 90°).

See figure 4.1 for examples.

When looking at a larger number of fingerprints, it seems as if the range for both arches and loops is closer to 120° than it is to 90°.

[1] In this context the term range referrs to the difference of the angles inbetween which the majority of frequencies lies.

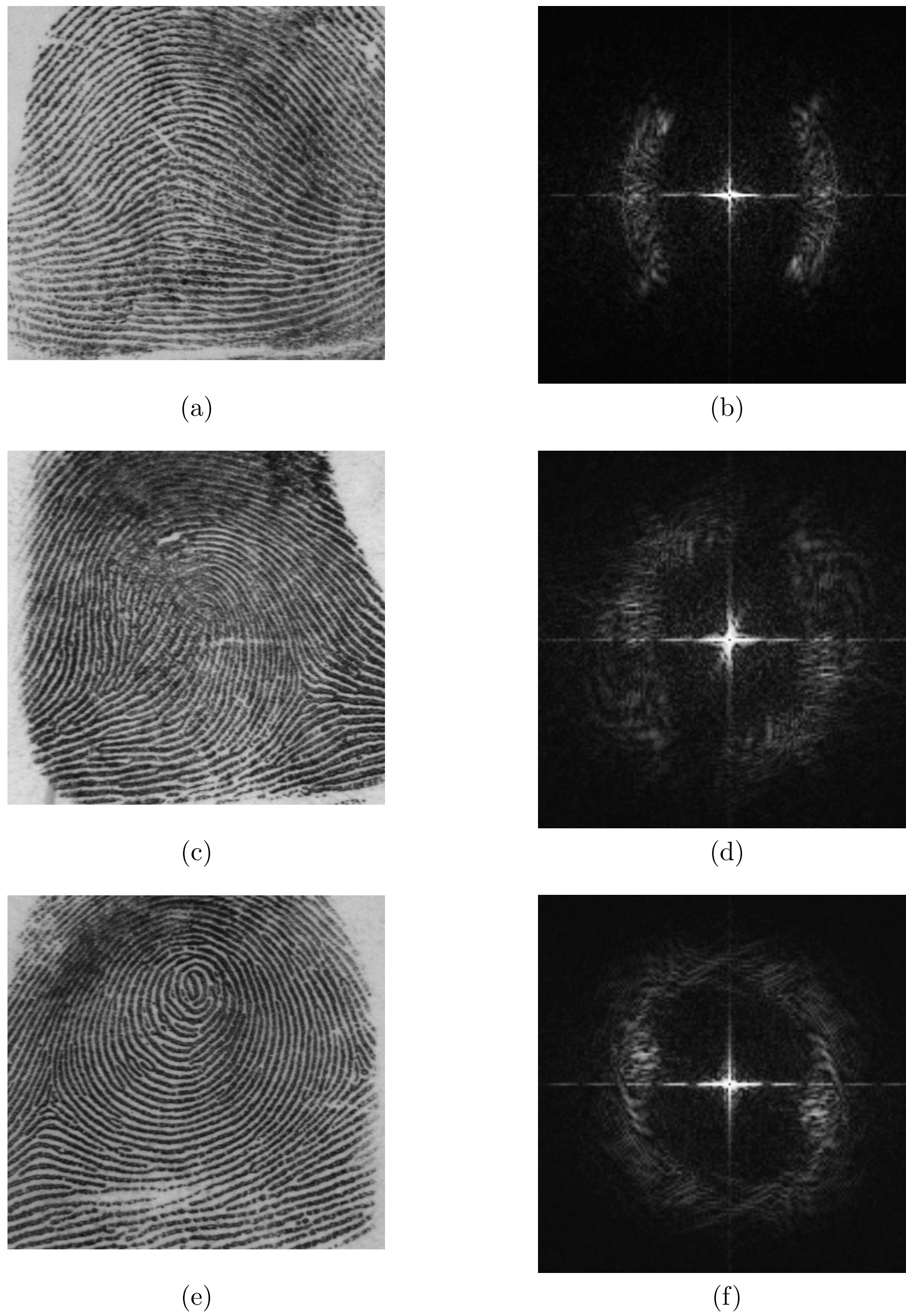

Figure 4.1: Fingerprint images and their Fourier transformations. The Fourier transform images have been cropped to show only the center square with an edge length of 32% of the original. Contrast and brightness have also been enhanced for better visibility.

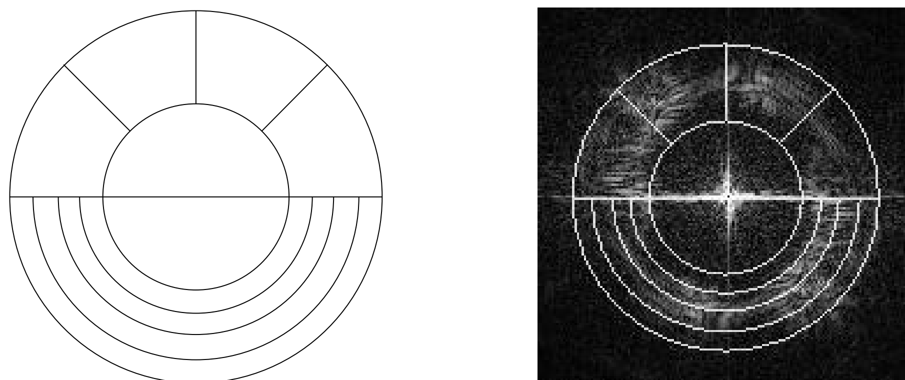

Figure 4.2: Sectors and rings separately (left) and on top of a Fourier transform (right).

4.3 Implementing a Test

To test for the aforementioned criteria, the upper part of the Fourier transformation is divided into sectors (pie-slices), as is illustrated in figure 4.2. Since the Fourier transformation is symmetric with respect to the ω_1-axis, one only needs to consider its upper half.

The algorithm goes through the image point-wise (for the C-Code see appendix A.3.1). The algorithm uses a vector `sector_sum`, where each vector element `sector_sum[i]` is the integral of the Fourier transform in the i-th sector. The vector is initialized to 0. At each point (ω_1, ω_2) it does the following steps:

1. Determine the sector i the point (ω_1, ω_2) is in.
2. Increment the integral `sector_sum[i]` by the value of the Fourier transform $F(\omega_1, \omega_2)$.

Using the values in `sector_sum` the approximate range of the fingerprint can be determined. This becomes more accurate the more sectors one uses[2].

To find the range of the Fourier transformation the mean m of the vector `sector_sum` is computed. The range r' in sectors is then defined as the number of sectors who's integral is greater than $0.45m$. The range r' can also be expressed as an angular range r using $r = 180r'/\texttt{NUM_SECTORS}$, `NUM_SECTORS` as defined in `classfft.c`.

In analogy to the sectors rings are also implemented. This was done to investigate, whether there are any patterns in the radial direction.

4.4 Empirical Analysis

A database of 82 images was used to investigate the possibility to deduce the fingerprint class from the Fourier transformation. According to the properties discussed in section 4.2 arches and loops have the same range but are oriented differently. Hence, arches should be symmetic with respect to the vertical axis, while loops should be look aproximately like arches but rotated around the center. Unfortunately there is a significant number of unsymmetric arches as well as symmetric loops. This makes it impossible to find out whether one is confronted with a loop or an arch.

Again, according to the rules from section 4.2 whorls should have a range of about 180° whereas loops and arches only have ranges of 90° to 120°. The possibility to separate whorls from non-whorls by looking at the range was tested. To this end the algorithm from section 4.3 was used. A Fourier transformation with the majority of frequencies in a range of more than 160° was classified as an whorl.

Among the 82 images used were 24 whorls, seven of which had a range of less than 160° and were therefore not classified as a whorl. On the other hand there were many non-whorl fingerprints that

[2]The number of sector can be adjusted in `classfft.c` by setting `NUM_SECTORS` to the desired value.

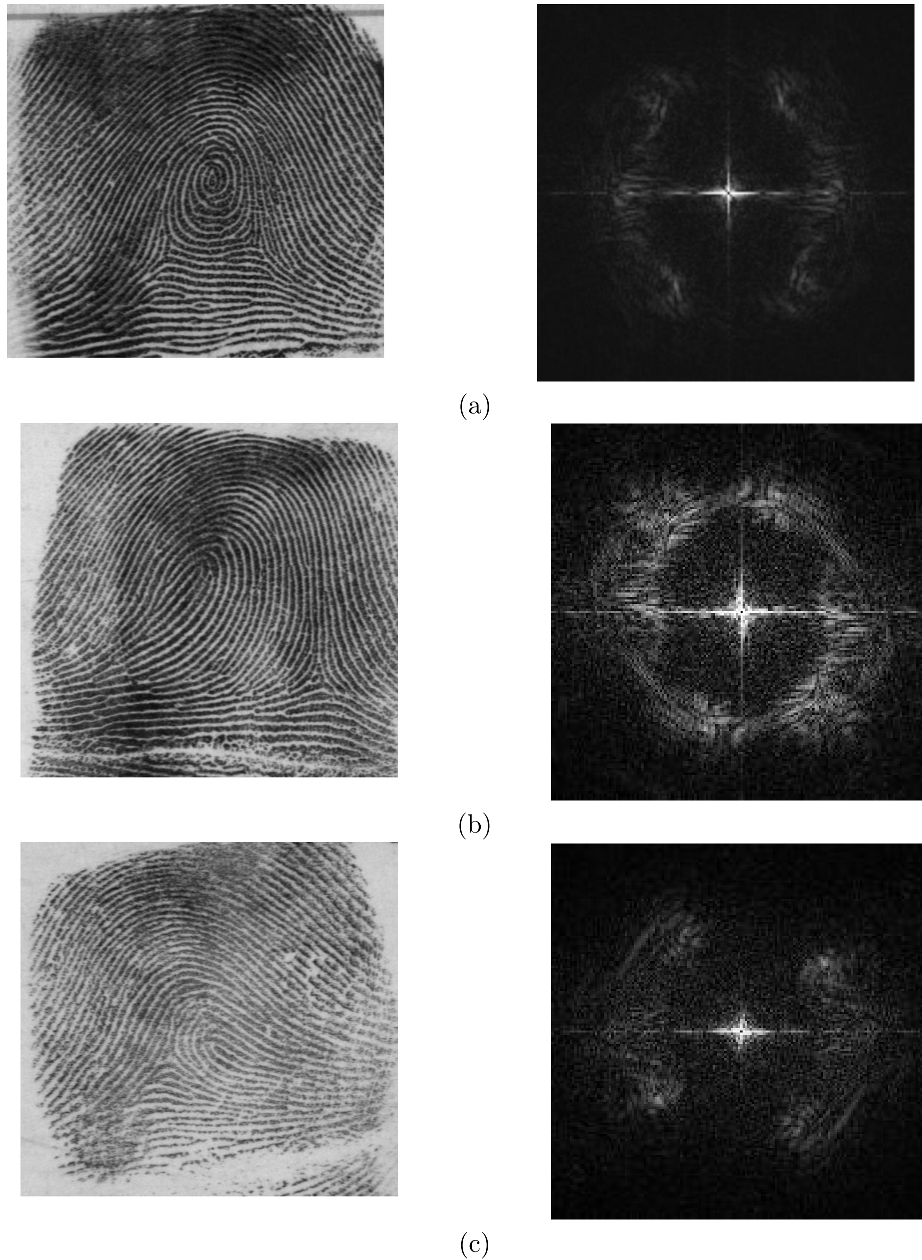

Figure 4.3: Three examples of misclassification when using the Fourier transformation: (a) Whorl and its Fourier transformation having a range of about $135°$. (b) Loop with its Fourier transformation showing a full circle. (c) Loop that is classified as an arch.

have a Fourier transformation with a range much greater than 120°. These were often misclassified as whorl. In total out of the 82 images 18 were misclassified when the task was only to separate whorls from non-whorls. This corresponds to a failure rate of about 22%. Fitz and Green ([4]) get a classification rate of 85% for this test.

4.5 Conclusion

Based on the testing that was carried out and described in section 4.4 one can say that it is impossible to classify fingerprints using three categories (arch, loop, and whorl) by means of the Fourier transformation and the rules proposed by Fitz and Green. When simplifying the problem to telling non-whorls from whorls there is still a high failure rate. Since even the simple case does not work, the proposed Fourier-based method is useless when trying to classify fingerprints.

The Fourier transformation has a property that could be useful in fingerprint detection: it is invariant to translation. It might be possible to carry out the whole recognition process in the frequency domain using a system of sectors and rings (see figure 4.2). As this is beyond the scope of classification, the idea was not pursued further.

Appendix A

The Code

Please note that this implemenation is part of a bigger project. That is why not all files needed to compile and run the code are here. This refers especially to the computation of the orientation field and the determination of the region of interest. The matrix types used are straightforeward, i.e.:

```
typedef struct floatMatrix {
  float **data;  // Pixel values
  int nr, nc;  // Rows and columns
  int oi, oj;  // Origin, not used here
} *FloatMatrix;
```

A.1 C-Code for Variance and Covariance Transformation

A.1.1 covar.c

The file `covar.c` has implementations of covariance and variance transformation. `types.h` contains matrix and vector definitions analogous to the one above. In `util.c` functionality to handle matricies and vectors is implemented. `visual.h` contains nothing but the declaration of a matrix used for visualization of the connected components.

```
#include <math.h>
#include <stdio.h>
#include "types.h"
#include "util.h"
#include "types2.h"
#include "utils2.h"
#include "defines2.h"
#include "visual.h"

#define VERBOSE 0 // how much to talk to stdout. Meaningful values are 0..6
#define CONNECTEDNESS 4 // what connectedness to use (4 or 8).
#define MAX_COMPONENTS 30 // how many components to find at most.
#define MIN_COMPONENT_SIZE 4 // min. size of a component to be considered
#define THRESHOLD_FRACTION 5.2 // max/THRESHOLD_FRACTION is the
                              //threshold for the variance transformation
const int LOCALSIZE = 3; // size of the local covar. matricies. Has to be ODD!
/** return the covariance matrix of <code>matrix</code>.
*/
FloatMatrix covariance_matrix(FloatMatrix A, int x, int y, int size) {
  int i, j, n;
  float delta;
  FloatMatrix covar = newFloatMatrix(size, size);
  FloatVector means = newFloatVector(size);
  FloatMatrix matrix = float_submatrix(A, x, y, size);
```

```
  if ((matrix == 0) || (means == 0)) printf("out of memory in covariance_matrix.\n");
  // compute the means of each column:
  for(i = 0; i < size; i++) {
    means->data[i] = 0;
    for(j = 0; j < size; j++) means->data[i] += matrix->data[j][i];
    means->data[i] = means->data[i] / size;
  }
  // initialize covariance matrix with zeros:
  for(i = 0; i < size; i++)
    for(j = 0; j < size; j++)
      covar->data[i][j] = 0;
  // outer loop is the matrix summation for calculation of the covariance matrix
  for(n = 0; n < size; n++) {
    // inner two loops add element-wise:
    for(i = 0; i < size; i++) {
      // covariance matricies are symmeric (only compute upper triangle)!
      for(j = i; j < size; j++) {
        delta = (matrix->data[n][j] - means->data[j])*(matrix->data[n][i] - means->data[j] );
        covar->data[i][j] += delta;
        // fill in lower triangle, unless we are on the diagonal
        if (i != j) covar->data[j][i] += delta;
      }
    }
  }
  // divide the covar-matrix element-wise by MATRIX_SIZE - 1
  for(i = 0; i < size; i++)
    for(j = 0; j < size; j++)
      covar->data[i][j] = covar->data[i][j] / (size - 1);
  if (means != 0) freeFloatVector(means);
  if (matrix != 0) freeFloatMatrix(matrix);
  return covar;
}
/** Computes the maximum eigenvalue of a REAL, SYMMETRIC matrix.
* Real and symmetric are important, but not checked!
* The upper triangle is DESTROYED. If you don't want this, use:
* <code>max_eigenvalue(float_submatrix(matrix, matrix->nr, matrix->nc));</code>
* Uses a modified Jacobi method to yield only the eigenvalues.
* See: Numerical recipes in C (pp.360) */
float max_eigenvalue(FloatMatrix matrix) {
  int size = matrix->nr;
  FloatVector d = newFloatVector(size);
  int iters, i, j, row, col;
  float tresh, theta, tau, t, sum, s, h, g, c;
  if (d == 0) printf("out of memory in max_eigenvalue.\n");
  // initialize d (which will contain the approximations for the eigenvalues):
  for(i = 0; i < size; i++) d->data[i] = matrix->data[i][i];
  // main loop. Do at most 50 iterations (sweeps)
  for(iters = 0; iters <= 100; iters++) {
    // first sum the upper-diag. elements (if zero, we are done)
    sum = 0;
    for(row = 0; row < size; row++)
      for(col = row + 1; col < size; col++)
        sum += (float)fabs(matrix->data[row][col]);
    if (sum == 0) {
      float max = 0;
      for(i = 0; i < size; i++)
        if (d->data[i] > max) max = d->data[i];
      // free allocated memory:
      if (d != 0) freeFloatVector(d);
      return max;
    }
    // on the first 3 sweeps use a non-zero threshold:
    if (iters < 4) tresh = (float)(0.2 * sum / (size * size));
    else tresh = 0;

    for(row = 0; row < size; row++) {
      for(col = row + 1; col < size; col++) {
        g = (float)(100 * fabs(matrix->data[row][col]));
```

```
      // If off-diagonal element is small after 4 sweeps, skip the rotation:
      if (iters > 4 && fabs(d->data[row]) + g == fabs(d->data[row]) &&
        fabs(d->data[col]) + g == fabs(d->data[col]))
        matrix->data[row][col] = 0;
      else if (fabs(matrix->data[row][col]) > tresh) {
        h = d->data[col] - d->data[row];
        if (fabs(h) + g == fabs(h)) t = (matrix->data[row][col]) / h;
        else {
          theta = (float)(0.5 * h / (matrix->data[row][col]));
          t = (float)(1.0/(fabs(theta)+sqrt(1+theta*theta)));
          if (theta < 0) t = -t;
        }
        c = (float)(1/sqrt(1+t*t));
        s = t * c;
        tau = s/(1+c);
        h = t* matrix->data[row][col];
        d->data[row] -= h;
        d->data[col] += h;
        matrix->data[row][col] = 0;
        for(j = 0; j < row; j++) {
          g = matrix->data[j][row];
          h = matrix->data[j][col];
          matrix->data[j][row] = g - s*(h+g*tau);
          matrix->data[j][col] = h + s*(g-h*tau);
        }
        for(j = row + 1; j < col; j++) {
          g = matrix->data[row][j];
          h = matrix->data[j][col];
          matrix->data[row][j] = g - s*(h+g*tau);
          matrix->data[j][col] = h + s*(g-h*tau);
        }
        for(j = col + 1; j < size; j++) {
          g = matrix->data[row][j];
          h = matrix->data[col][j];
          matrix->data[row][j] = g - s*(h+g*tau);
          matrix->data[col][j] = h + s*(g-h*tau);
        }
      }
    }
  }
}
printf("Too many iterations to compute eigenvalues. Returning 0.");
if (d != 0) freeFloatVector(d);
return 0;
}
/** return 1 if all points of the area are marked 0 in the region of interest,
*/
int valid_area(int row, int col, int delta, UcharMatrix regionOfInterest) {
  int i, j;
  for (i = row - delta; i <= row + delta; i++)
    for (j = col - delta; j <= col + delta; j++)
      if (regionOfInterest->data[i][j] != 0) return 0;
  return 1;
}
/* Computes the covariance feature image as proposed in Jain et al. IEEE
Transactions Vol. 21 No. 4. Of each local neighborhood (size can be
configured by <code>LOCALSIZE</code>) the covariance matrix
is computed. Its max. eigenvalue is set as the value of the feature
image at the center of the component. The resulting matrix is of the
same size as the <code>orientation</code>. <code>regionOfInterest</code>
is used to determine which points are relavant at all. */
FloatMatrix covariance_feature_image(FloatMatrix orientation, UcharMatrix regionOfInterest) {
  FloatMatrix feature = newFloatMatrix(orientation->nr, orientation->nc);
  FloatMatrix lro = subtract_elementwise(orientation, PI/2);
  FloatMatrix covar;
  int i, j, delta;
  float eig;
```

```c
  if (feature == 0) printf("out of memory in covariance_feature_image.\n");
  // initialize with zeros. This could be optimized, b/c a lot of the
  // zeros will later be overwritten by the feature image values.
  for (i = 0; i < feature->nr; i++)
    for (j = 0; j < feature->nc; j++)
      feature->data[i][j] = 0;
  // ceil(LOCALSIZE/2), this is the distance edge-center in the local matricies
  delta = (LOCALSIZE/2);
  // compute the feature image values:
  for (i = delta; i < feature->nr - delta; i++)
    for (j = delta; j < feature->nc - delta; j++) {
      if (valid_area(i, j, delta, regionOfInterest)) {
        covar = covariance_matrix(lro, i - delta, j - delta, LOCALSIZE);
        eig = max_eigenvalue(covar);
        feature->data[i][j] = eig;
        if (covar != 0) freeFloatMatrix(covar);
      } else {
        feature->data[i][j] = 0;
      }
    }
  if (lro != 0) freeFloatMatrix(lro);
  return feature;
}
/* =========== END OF COVARIANCE TRANSFORMATION ============== */
/* =========== VARIANCE TRANSFORMATION ====================== */
/** This method takes an array of size <code>size</code> of orientations in
<code>*values</code>. The orientations are flipped (add PI) in such a
way that the difference between the smallest and largest orientation is
minimum. The smallest value is then subtracted from each orientation.
This allows the array to start at zero. */
void min_phase_system(float *values, int size) {
  int i;    // running var. through the array
  int index = -1;  // up to which index to flip
  float tmp;
  float angle, newangle, full_angle;

  if (VERBOSE > 3) { printf("before: "); print_float_array(values, size);}
  // sort the values to be of ascending size:
  sort(values, size);
  if (VERBOSE > 5) { printf("after sort: "); print_float_array(values, size); }
  // if the range is already below PI/2 we cannot make an improvement, otherwise:
  if ((values[size - 1] - values[0]) > (PI/2)) {
    angle = values[size - 1] - values[0];
    full_angle = angle;
    for (i = 0; i < size; i++) {
      if (values[i] + PI - values[i + 1] < full_angle) {
        newangle = values[i] + PI - values[i + 1];
        //printf("%i: %f\n", i, newangle);
        if (newangle <= angle) {
          index = i;
          angle = newangle;
        }
      }
    }
    // do flip up to determined index:
    for (i = 0; i <= index; i++) values[i] += PI;
    if (VERBOSE > 5) { printf("after rearranging: "); print_float_array(values, size); }
    // sort again (order might have been destroyed by the filp, this is mainly for cosmetic reasons:
    sort(values, size);
  }
  // subtract the minimum (stored in tmp):
  tmp = values[0];
  for (i = 0; i < size; i++) {
    values[i] -= tmp;
  }
  if (VERBOSE > 2) { printf("after: "); print_float_array(values, size); }
}
/* ============ */
```

```
/** Tool function to serialize a portion of a matrix into an array.
The elements of the submatrix (row-delta, col-delta)...(row+delta, col+delta) are        230
copied into the array <code>values</code>. */
void copy_to_array(FloatMatrix matrix, int row, int col, int delta, float *values) {
  int i, j;
  int k = 0;
  for (i = row − delta; i <= row + delta; i++)
    for (j = col − delta; j <= col + delta; j++) {
      values[k] = matrix−>data[i][j];
      k++;
    }
}                                                                                          240
/** Computes the variance of elements in the local neighborhood with center
(row, col) and size 2*delta + 1.
*/
float variance(FloatMatrix matrix, int row, int col, int delta) {
  int i;
  float avg = 0;
  float sum = 0;
  int size = (2 * delta + 1) * (2 * delta + 1);
  float values[size];
  if (VERBOSE > 2) { printf("(%i %i)\n", row, col);                                       250
    print_matrix(float_submatrix(matrix, row − delta, col − delta, 2*delta +1)); }
  copy_to_array(matrix, row, col, delta, values);
  min_phase_system(values, size);
  // first compute the average:
  for (i = 0; i < size; i++) avg += values[i];
  avg /= size;
  // then compute the sum of the squares:
  for (i = 0; i < size; i++) sum += values[i] * values[i];
  return sum − avg * avg;
}                                                                                          260
/** Computes the variance feature image by assigning each point of the feature the
value of the variance of the corresponding neighborhood in the orientation matrix.
<code>regionOfInterest</code> is used to find the points to be considered. */
FloatMatrix local_variance_feature_image(FloatMatrix orientation, UcharMatrix regionOfInterest) {
  FloatMatrix feature = newFloatMatrix(orientation−>nr, orientation−>nc);
  int i, j, delta;
  if (feature == 0) printf("out of memory in covariance_feature_image.\n");
  // initialize with zeros. This could be optimized, b/c a lot of the
  // zeros will later be overwritten by the feature image values.
  for (i = 0; i < feature−>nr; i++)                                                      270
    for (j = 0; j < feature−>nc; j++)
      feature−>data[i][j] = 0;
  delta = LOCALSIZE/2;
  // compute the feature image values:
  for (i = delta; i < feature−>nr − delta; i++)
    for (j = delta; j < feature−>nc − delta; j++) {
      if (valid_area(i, j, delta, regionOfInterest)) {
        feature−>data[i][j] = variance(orientation, i, j, delta);
      } else {
        feature−>data[i][j] = 0;                                                          280
      }
    }
  return feature;
}
/** */
float sum_elements(FloatMatrix m) {
  int i, j;
  float sum = 0;
  for (i = 0; i < m−>nr; i++)
    for (j = 0; j < m−>nc; j++)                                                          290
      sum += m−>data[i][j];
  return sum;
}
/** Checks if the point (row, col) is inside the matrix. That is:
row and col are > 0 AND row < matrix->nr AND col < matrix->nc */
int in_bounds(UcharMatrix matrix, int row, int col) {
```

```
  if ((row >= 0) || (col >= 0) || (row < matrix-> nr) || (col < matrix->nc)) return 1;
  else return 0;
}
/** Checks if the point (row, col) has more than two neigbours or, if                300
it only has two neighbours those are adjacent. Neighbors are determined
in a 8-connected sense!
*/
int valid_neighbourhood(UcharMatrix matrix, int row, int col) {
  int i, j;
  int n = 0;
  int last = 0;
  int connected_neighbours = 0;
  if ((CONNECTEDNESS == 8)|| (CONNECTEDNESS == 4)) {
    for (i = -1; i < 2; i++) {                                                          310
      for (j = -1; j < 2; j++) {
        if ((in_bounds(matrix, row + i, col + j)) && (i != j)) {
          if (matrix->data[row + i][col + j]) {
            n++;
            if (last) connected_neighbours = 1;
            last = 1;
          }
        } else {
          last = 0;
        }                                                                               320
      }
    }
    if (n > 2) {
      return 1;
    } else {
      if (connected_neighbours && (n == 2)) {
        return 1;
      } else {
        return 0;
      }                                                                                 330
    }
  } else {
    printf("invalid connectedness in valid_neighbourhood().\n");
    return 0;
  }
}
/** Evaluate if the offsets i and j are appropriate of the chosen connectedness:
*/
int valid_offset(int i, int j) {
  if (CONNECTEDNESS == 8) return !((i == 0) && (j == 0));                               340
  /* 4-connectedness in plain english: be on one axis but not in the center.*/
  if (CONNECTEDNESS == 4) return ((i == 0) || (j == 0)) && !((i == 0) && (j == 0));
  return 0;
}
/** Recursivly adds points to the existing component. Visited points are
marked in <code>trace</code>. Points that have only two neighbours are not
considered. The current size is stored in curr_size. */
void recursive_connected_component(UcharMatrix matrix, UcharMatrix trace,
PointVector component, int x, int y, int *curr_size) {
  uchar own_value = matrix->data[x][y];                                                 350
  int i, j;
  // we don't want long lines with width 1:
  if (!valid_neighbourhood(matrix, x, y)) return;
  // append yourself to vector:
  component->data[*curr_size].x = x;
  component->data[*curr_size].y = y;
  (*curr_size)++;
  // make sure the spot is marked in trace (it is not, iff this is the initial call)
  trace->data[x][y] = 1;
  for (i = -1; i <= 1; i++)                                                             360
    for (j = -1; j <= 1; j++) {
      // can't be outside matrix, nor in the center:
      if (in_bounds(matrix, x + i, y + j) && valid_offset(i, j)) {
        //printf("(%i, %i) ", x + i, y + j);
```

```
        if ((matrix->data[x + i][y + j] == own_value) && (trace->data[x + i][y + j] == 0)) {
          // and mark as handled in the trace matrix
          //printf("write to trace (%i, %i)\n", x + i, y + j);
          trace->data[x + i][y + j] = 1;
          recursive_connected_component(matrix, trace, component, x + i, y + j, curr_size);
        }                                                                              370
      }
    }
}
/** Compute the centroid of all the points in the <code>PointVector points</code>.
<code>size</code> is the length of the vector.
*/
FloatPoint compute_center(PointVector points, int size) {
  FloatPoint result;
  int i;
  float sum;                                                                           380
  if (size == 0) {
    printf("No connected component found!\n");
    result.x = 0; result.y = 0;
    return result;
  }
  sum = 0;
  for (i = 0; i < size; i++) sum += points->data[i].x;
  result.x = ((float)sum / size);
  sum = 0;
  for (i = 0; i < size; i++) sum += points->data[i].y;                                390
  result.y = ((float)sum / size);
  return result;
}
/** return <code>UcharMatrix</code> where (i, j) = 1 if matrix->data[i][j] > threshold
*/
UcharMatrix interesting_threshold_matrix(FloatMatrix matrix, float threshold, UcharMatrix roi) {
  UcharMatrix result = newUcharMatrix(matrix->nr, matrix->nc);
  int i, j;
  for (i = 0; i < matrix->nr; i++)
    for (j = 0; j < matrix->nc; j++) {                                                400
      if ((matrix->data[i][j] >= threshold) && (roi->data[i][j] == 0)) {
        result->data[i][j] = 1;
      } else {
        result->data[i][j] = 0;
      }
    }
  return result;
}
/** Visualizes the <code>PointVector component</code> in the variable
<code>threshold_map</code>. <code>threshold_map</code> is declared in                 410
<code>visual.h</code>. All elements from <code>component</code>
are set to <code>1</code> in the matrix. */
void draw_component(PointVector component, int size) {
  int i;
  if (threshold_map == 0) return;
  else
    for (i = 0; i < size; i++)
      threshold_map->data[component->data[i].x][component->data[i].y] = 1;
}
/** Returns an estimate for the "quality" of the component. The term quality referrs to the    420
fragmentation of the comp., where more fragmented components are considered to be of
inferior quality.
The component's quality if computed as: (maximum_radius)^2/(component size) */
float compute_quality(PointVector component, int size, FloatPoint center) {
  float max_radius = 0; // the radii are acutally radii squared, but that's messy to type
  float curr_radius;
  int i;
  float x, y;
  for (i = 0; i < size; i++) {
    x = component->data[i].x - center.x; y = component->data[i].y - center.y;         430
    curr_radius = x*x + y*y;
    if (curr_radius > max_radius) max_radius = curr_radius;
```

```
  }
  printf("  component of size %i with quality %f center at (%f, %f)\n",
    size, size / max_radius, center.x, center.y);
  return size / max_radius;
}
float weight_qualities(float* qualities, int* sizes, int array_size) {
  int i;
  float nom = 0;                                                          440
  float den = 0;
  for (i = 0; i < array_size; i++) {
    nom += qualities[i] * sizes[i];
    den += sizes[i];
  }
  return nom / den;
}
/** Find all components in the matrix <code>A</code> of points who's value
is above the <code>threshold</code>. Only consider points inside the
<code>regionOfInterest</code>. The <code>PointVector centers</code> is used    450
for the results (the centers of the connected components), the array
<code>qualities</code> holds the qualities of each component. Only components
of sizes at least <code>MIN_COMPONENT_SIZE</code> are returned.
@return int number of components found */
int find_components(FloatMatrix A, float threshold, UcharMatrix regionOfInterest,
FloatPointVector centers, float* qualities, int* sizes) {
  UcharMatrix matrix = interesting_threshold_matrix(A, threshold, regionOfInterest);
  UcharMatrix trace = newUcharMatrix(matrix->nr, matrix->nc);
  // This will be sufficient in any case (there might be room for optimizations here)
  int point_vector_size = matrix->nr * matrix->nc;                        460
  PointVector component = newPointVector(point_vector_size);
  int i, j, size;
  int component_count = 0;
  threshold_map = newUcharMatrix(matrix->nr, matrix->nc);
  zero_uchar_matrix(threshold_map);
  zero_uchar_matrix(trace);
  //printf("region of interest = \n"); print_uchar_matrix(regionOfInterest);
  if (VERBOSE) { printf("\ninteresting thresholded matrix = [t = %f]\n", threshold);
    print_binary_matrix(matrix); }
  if ((trace == 0) || (component == 0))                                   470
    printf("out of memory in center_max_connected_component.\n");
  for (i = 0; i < trace->nr; i++)
    for (j = 0; j < trace->nc; j++) {
      // element is not yet member of a component and non-zero:
      if (trace->data[i][j] == 0 && matrix->data[i][j] != 0) {
        size = 0;
        recursive_connected_component(matrix, trace, component, i, j, &size);
        if ((component != 0) && (size >= MIN_COMPONENT_SIZE)) {
          centers->data[component_count] = compute_center(component, size);
          qualities[component_count] = compute_quality(component, size,  480
            centers->data[component_count]);
          sizes[component_count] = size;
          draw_component(component, size);
          component_count++;
        }
      }
    }
  if (component != 0) freePointVector(component);
  if (matrix != 0) freeUcharMatrix(matrix);
  if (trace != 0) freeUcharMatrix(trace);                                 490
  return component_count;
}
/* Input:   a float matrix, which will be thresholded by <code>threshold</code>
The max. connected component of ONES in the thresholded matrix will be returned as
a <code>PointVector</code>. */
VarianceTransformData center_connected_components(FloatMatrix A, float threshold,
UcharMatrix regionOfInterest) {
  VarianceTransformData result;
  float qualities[MAX_COMPONENTS];
  int sizes[MAX_COMPONENTS];                                              500
```

```
  result.centers = newFloatPointVector(MAX_COMPONENTS);
  result.center_count = find_components(A, threshold, regionOfInterest,
    result.centers, qualities, sizes);
  result.quality_estimate = qualities;
  return result;
}

FloatPoint center_max_connected_component(FloatMatrix A, float threshold,
UcharMatrix regionOfInterest) {
  FloatPointVector centers = newFloatPointVector(MAX_COMPONENTS);                510
  float qualities[MAX_COMPONENTS];
  int sizes[MAX_COMPONENTS];
  int marker = 0;
  int i;
  int count = find_components(A, threshold, regionOfInterest, centers, qualities, sizes);
  for (i = 0; i < count; i++) if (sizes[i] > sizes[marker]) marker = i;
  return centers->data[marker];
}

/* ===== INTERFACE FUNCTIONS TO BE CALLED FROM OUTSIDE: ===== */                520
/** Determines the center of the fingerprint by means of the covariance feature image.
Does not work as well as the algorithm with just the variance and takes a lot more time.
*/
FloatPoint center_of_orientation_field(FloatMatrix orientation, float threshold,
UcharMatrix regionOfInterest) {
  FloatMatrix covar;
  FloatPoint result;
  float max;
  if (VERBOSE) { printf("orientation = \n"); print_matrix_esc(orientation, '.'); }
  covar = covariance_feature_image(orientation, regionOfInterest);              530
  if (VERBOSE) { printf("covariance feature image = \n"); print_matrix_esc(covar, '.'); }
  max = max_element(covar);
  result = center_max_connected_component(covar, max / 6 * threshold, regionOfInterest);
  if (covar != 0) freeFloatMatrix(covar);
  return result;
}
/** Finds the local centers of the fingerprint represented by the
<code>orientation</code> matrix by means of the variance transform.
*/
VarianceTransformData centers_by_variance(FloatMatrix orientation,              540
UcharMatrix regionOfInterest) {
  float max;
  FloatMatrix covar;
  VarianceTransformData result;
  if (VERBOSE) { printf("orientation = \n"); print_matrix_esc(orientation, '.'); }
  covar = local_variance_feature_image(orientation, regionOfInterest);
  if (VERBOSE) { printf("covariance feature image = \n"); print_matrix_esc(covar, '.'); }
  max = max_element(covar);
  result = center_connected_components(covar, max / THRESHOLD_FRACTION, regionOfInterest);
  if (covar != 0) freeFloatMatrix(covar);                                        550
  return result;
}
```

A.1.2 types2.h

The file `types2.h` defines the types `FloatPoint`, `FloatPointVector` and `VarianceTransformData` that are not used elsewhere in the project.

```
#include "types.h"
typedef struct floatpoint {
        float x;
        float y;
} FloatPoint;
typedef struct FloatPointVector {
        FloatPoint *data;
```

```
        long n;
} *FloatPointVector;
/** Data structure to hold the results of the center computations. The <code>PointVector centers</code>   10
holds all the local centers. <code>center_count</code> is the number of components found. <code>
quality_estimate</code> is a guess on how compact the components are.
*/
typedef struct varianceTransformData {
        FloatPointVector centers;
        int center_count;
        float *quality_estimate;
} VarianceTransformData;
```

A.2 MATLAB File visualizing the Fourier Transformation

The function `fpft_batch(name)` displays a plot window with the original image and five Fourier transformations binarized and thresholded at different values (this makes finding the range easier for humans). The argument is a filename in the current directory without the extension ".bmp".

```
function void = fpft_batch(name)
    subplot(2, 3, 1);
    img = imread(strcat(name, '.bmp'));
    imshow(img);
    I = abs(fftshift(fft2(img)));
    s = 't = ';
    for j = 1 : 5
        t = 12500 + 7500 * j;
        s = strcat(s, num2str(t), ', ');
        F = I .* (I > t);                                                        10
        subplot(2, 3, 1 + j);
        [x y] = size(F);
        imshow(F([floor(0.25*x):floor(0.75*x)], [floor(0.25*y):floor(0.75*y)]));
        imwrite(F, strcat(name, '_', num2str(t), '.bmp'));
    end;
    subplot(2,3,1);
    text(0, -20, s);

end;
```

A.3 C-Code the Fourier Transformation Test

A.3.1 classfft.c

The file `classfft.c` has implementations of covariance and variance transformation:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "types.h"
#include "types2.h"
#include "fft2d.h"
#include "util.h"
#include "utils2.h"
#include "define2.h"                                                             10
#include "fptypes.h"
#include "io.h"
/* ========= PARAMETERS OF THE ALGORITHM =============== */
#define NUM_SECTORS 36 // The number for sectors the fingerprint is divided into
#define NUM_RINGS 10 // The number of rings
#define MIN_RADIUS 0.5 // 0..1: how big the inner circle that is not considered is.
#define SECTOR_THRESHOLD 0.45 // 0..1: fraction of the mean of sectors a
```

```
                                // sector has to be to contribute to the range
#define FFT_THRESHOLD 0.2 // 0..1: fraction of the max value of the fft a point
                              // has to be to contribute to the sector−sum                20
/* ============================================================ */
#define NO_SECTOR −1
#define NO_RING −1
/** Computes the absolute value of the 2D-fourier transform
of the input <code>image</code>. The origin is shifted to the center
of the output after transformation.
*/
FloatMatrix compute_abs_fft2d(UcharMatrix image) {
  FloatMatrix real = newFloatMatrix(image−>nr, image−>nc);
  FloatMatrix imag = newFloatMatrix(image−>nr, image−>nc);                                30
  copyUcharToFloat(image, real);

  FFT2D(real, imag, 1);
  shiftFFT2D(real, imag);
  absFFT2D(real, imag);
  freeFloatMatrix(imag);

  return real;
}
/** Determine the sector the point (x, y) is in. The sectors are elements of a circle that       40
is bounded by the square of <code>size x size</code>. Sectors are only in the upper half of
the circle b/c the fourier transform is symmetric. The number of sectors is controlled by
<code>NUM_SECTORS</code>.
*/
int find_sector(int x, int y, int size) {
  int i; int dx, dy, radius;
  float angle; float angle_per_sector;
  dx = x − size/2; dy = y − size/2;
  radius = sqrt(dx * dx + dy * dy);
  if ((radius < MIN_RADIUS * size /2) || (radius > size/2)) return NO_SECTOR;          50
  if (dx == 0) angle = PI/2;
  else angle = atan((float) dy / dx) + PI/2;
  angle_per_sector = PI/NUM_SECTORS;
  for (i = 0; i < NUM_SECTORS; i++)
    if (((i * angle_per_sector) < angle) && (((i + 1) * angle_per_sector) > angle)) return i;
  return NO_SECTOR;
}
/** Determine the ring the point (x, y) is in. The rings are elements of a circle that
is bounded by the square of <code>size x size</code>. The number of rings is controlled by
<code>NUM_RINGS</code>.                                                                    60
*/
int find_ring(int x, int y, int size) {
  int dx, dy, radius, ring_width;
  dx = x − size/2; dy = y − size/2;
  radius = sqrt(dx * dx + dy * dy);
  if ((radius < MIN_RADIUS * size /2) || (radius > size/2)) return NO_RING;
  ring_width = (1 − MIN_RADIUS) * size / (2 * NUM_RINGS);
  radius −= MIN_RADIUS * size / 2;
  return radius / ring_width;
}                                                                                          70
/** Returns the number of sectors in the array who's
sum is > SECTOR_THRESHOLD*mean of the array*/
int range(float sector_sum[NUM_SECTORS]) {
  float threshold = SECTOR_THRESHOLD * mean(sector_sum, NUM_SECTORS);
  int i, j;
  int count = NUM_SECTORS; // count the # of sectors
  for (i = 0; i < NUM_SECTORS / 2; i++) {
    if (sector_sum[i] < threshold) count−−;
    else break;
  }                                                                                        80
  for (j = NUM_SECTORS − 1; j > NUM_SECTORS / 2; j−−) {
    if (sector_sum[j] < threshold) count−−;
    else break;
  }
  return count;
```

```
}
/** */
float centroid(float sector_sum[NUM_SECTORS]) {
  int i; float d = 0; float n = 0;
  for (i = 0; i < NUM_SECTORS; i++) {                                                    90
    d += sector_sum[i] * i;
    n += sector_sum[i];
  }
  return d / n;
}
/** Prints a visual representation of the <code>array sector_sum</code> to stdout.*/
void visualize(float sector_sum[NUM_SECTORS], int sector_count[NUM_SECTORS],
int min_size) {
  int i, j; float max, tmp; float avg = 0;
  max = max_array_element(sector_sum, NUM_SECTORS);                                      100
  avg = mean(sector_sum, NUM_SECTORS) / max * 50;
  printf("sectors:\n");
  for (i = 0; i < NUM_SECTORS; i++) {
    tmp = sector_sum[i];
    printf("    (%2i) %6.5g, %6.5g |", i, tmp, tmp/sector_count[i]*min_size);
    for (j = 1; j < tmp / max * 50; j++) {
      if (j < avg) printf("O"); else printf("=");
    } printf("\n");
  }
  printf("                    mean: |"); for (j = 1; j < avg; j++) printf("=");              110
  printf("\n");
}
/** Due to the mapping of the cartesian onto a polar coordinate system, each ring
has a slightly different area. The area is counted in <code>ring_count</code>.
This function divides each ring sum by its area. */
void normalize_ring_sum(float ring_sum[NUM_RINGS], int ring_count[NUM_RINGS]) {
  int i;
  for (i = 0; i < NUM_RINGS; i++) ring_sum[i] /= ring_count[i];
}
/** Prints a visual representation of the distribution of the sums per ring to stdout.*/     120
void visualize_rings(float ring_sum[NUM_RINGS], int ring_count[NUM_RINGS], int min_size) {
  int i, j; float max, tmp; float avg = 0;
  normalize_ring_sum(ring_sum, ring_count);
  max = max_array_element(ring_sum, NUM_RINGS);
  avg = mean(ring_sum, NUM_RINGS) / max * 50;
  printf("rings:\n");
  for (i = 0; i < NUM_RINGS; i++) {
    tmp = ring_sum[i];
    printf("    (%2i) %6.5g, %6.5g |", i, tmp, tmp * min_size);
    for (j = 1; j < tmp / max * 50; j++)                                                 130
      if (j < avg) printf("O"); else printf("=");
    printf("\n");
  }
  printf("                    mean: |"); for (j = 1; j < avg; j++) printf("=");
  printf("\n");
}
/** Finds the maximum value the fourier transform has inside any sector.*/
float find_max_in_sectors(FloatMatrix fft, int min_size) {
  float max = 0; int i, j, sector;
  for (i = 0; i < fft->nr; i++)                                                          140
    for (j = 0; j < fft->nc; j++) {
      sector = find_sector(i, j, min_size);
      if ((sector != NO_SECTOR) && (fft->data[i][j] > max)) max = fft->data[i][j];
    }
  printf("max = %f\n", max);
  return max;
}
/** Interface function to be called from outside.
Takes the image and returns a value from <code>fptypes.h</code>. Currently only
<code>TYPE_NON_WHORL</code> or <code>TYPE_WHORL</code> are returned. This is b/c        150
it turned out the be impossible to distinguish the other ones (the whorls perform poorly, too).
*/
int classify_by_fft(UcharMatrix image) {
```

```
    FloatMatrix whole_fft = compute_abs_fft2d(image);// whole fft image
    FloatMatrix fft_t, fft; // clipped fourier transform and its transpose
    int min_size, i, j, sector, ring;
    float max, sector_sum[NUM_SECTORS], ring_sum[NUM_RINGS];
    int sector_count[NUM_SECTORS], ring_count[NUM_RINGS];
    int r;
    min_size = min(image->nr, image->nc);                                          160
    fft_t = float_submatrix(whole_fft, (int)(0.34 * min_size), (int)(0.34 * min_size),
      (int)(0.32 * min_size));
    fft = transpose_float_matrix(fft_t);
    freeFloatMatrix(whole_fft);
    freeFloatMatrix(fft_t);
    min_size = (int) min_size * 0.32;
    max = find_max_in_sectors(fft, min_size);
    for (i = 0; i < NUM_SECTORS; i++) {
      sector_count[i] = 0; sector_sum[i] = 0;
      ring_count[i] = 0; ring_sum[i] = 0;                                          170
    }
    for (i = 0; i < fft->nr; i++)
      for (j = 0; j < fft->nc; j++) {
        sector = find_sector(i, j, min_size);
        if (sector != NO_SECTOR) {
          if (fft->data[i][j] > FFT_THRESHOLD * max) sector_sum[sector] += fft->data[i][j];
          sector_count[sector]++;
        }
        ring = find_ring(i, j, min_size);
        if (ring != NO_RING) {                                                     180
          if (fft->data[i][j] > FFT_THRESHOLD * max) ring_sum[ring] += fft->data[i][j];
          ring_count[ring]++;
        }
      }
    // visualize(sector_sum, sector_count, min_size);
    // visualize_rings(ring_sum, ring_count, min_size);
    for (i = 0; i < NUM_SECTORS; i++)
      sector_sum[i] = sector_sum[i] / sector_count[i] * min_size;

    r = range(sector_sum);                                                         190
    printf("range of sectors: %i (%i)\n", r, r * 180 / NUM_SECTORS);
    printf("centroid: %f\n", centroid(sector_sum) * 180 / NUM_SECTORS);
    if (r > ((float)160 / 180 * NUM_SECTORS)) {
      //printf(" => Whorl\n");
      return TYPE_WHORL;
    } else {
      //printf(" => not a Whorl\n");
      return TYPE_NON_WHORL;
    }
  }                                                                                200
```

A.4 Tool Files

A.4.1 utils2.c

The file `utils2.c` contains various tool functions needed in `covar.c` and `classfft.c`. The implementation of obvious functions is omitted.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "types.h"
#include "util.h"
#include "types2.h"
void print_matrix(FloatMatrix matrix) { /*...*/ }
void print_matrix_esc(FloatMatrix matrix, char esc) { /*...*/ }
void print_uchar_matrix(UcharMatrix matrix) { /* ... */}
void print_point_vector(PointVector vec) { /* .. */}                              10
```

```
void print_binary_matrix(UcharMatrix matrix) { /* ... */ }
void print_float_array(float a[], int size) { /*...*/ }
UcharMatrix copyUcharMatrix(UcharMatrix source) {/*...*/}
/* Returns a submatrix of <code>matrix</code> */
FloatMatrix float_submatrix(FloatMatrix matrix, int x, int y, int size) {
        int i, j;
        FloatMatrix submatrix = newFloatMatrix(size, size);
        if (submatrix == 0) printf("out of memory in float_submatrix.\n");
        for (i = 0; i < size; i++)
                for (j = 0; j < size; j++)                                      20
                        submatrix->data[i][j] = matrix->data[x + i][y + j];
        return submatrix;
}
FloatMatrix float_submatrix_rect(FloatMatrix matrix, int x, int y, int size_x, int size_y) {
/* ... */ }
FloatMatrix transpose_float_matrix(FloatMatrix matrix) { /*... */ }
UcharMatrix transpose_uchar_matrix(UcharMatrix matrix) { /* ... */ }
int min(int a, int b) { /*... */}
float max_element(FloatMatrix matrix) { /*... */ }
UcharMatrix threshold_matrix(FloatMatrix matrix, float threshold) {             30
  UcharMatrix result = newUcharMatrix(matrix->nr, matrix->nc);
  int i, j;
  for (i = 0; i < matrix->nr; i++)
    for (j = 0; j < matrix->nc; j++) {
      if (matrix->data[i][j] >= threshold) {
        result->data[i][j] = 1;
      } else {
        result->data[i][j] = 0;
      }
    }                                                                           40
  return result;
}
void element_multiply(FloatMatrix m1, UcharMatrix m2) { /*... */ }
FloatMatrix subtract_elementwise(FloatMatrix matrix, float x) { /* ... */ }
float max_array_element(float a[], int size) { /*...*/ }
/** sa show the region of interest in a RGB Image.
        param spoint is a pointer to the thresholded coherence matrix, which represents
        the region of interest.
        */
void showRegion(UcharMatrix region, RGBMatrix image, uchar regionIndicator) {   50
  int x, y;
  for(y = 0; y < region->nr; y++) {
    for(x = 0; x < region->nc; x++) {
      if (region->data[y][x] == regionIndicator) {
        image->data[y][x].r = 0;
        image->data[y][x].b = 150;
      }
    }
  }
}                                                                               60
void sort(float *values, int size) { /* ... */ }
float mean(float array[], int size) { /* .... */ }
/** Fills <code>matrix</code> with zeros. */
void zero_uchar_matrix(UcharMatrix matrix) { /* .... */ }
float average_float_array(float array[], int size) { /* ... */ }
FloatPointVector newFloatPointVector(long n) {
  FloatPointVector v;
  v = (FloatPointVector) malloc((size_t) (sizeof(struct FloatPointVector)));
  if (!v) { printf("Out of storage in newFloatPointVector.\n"); return 0; }
  v->data = (FloatPoint *) malloc(n*sizeof(FloatPoint));                        70
  if (!(v->data)) { printf("Out of storage in newFloatPointVector.\n"); return 0; }
  v->n = n;
  return v;
}
void freeFloatPointVector(FloatPointVector      v) {
  if (v != 0) { free (v->data);   free (v); }
}
```

Bibliography

[1] *Fundamentals for Digital Image Processing*, Anil K. Jain, Prentice Hall, Englewood Cliffs, New Jersey, 1989

[2] *Numerical Recipes in C*, William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, Cambridge University Press, Cambrigde, 1988, pp. 360

[3] *A Multichannel Approach to Fingerprint Classification*, Anil K. Jain, Salil Prabhakar, Lin Hong, IEEE transactions on Pattern analysis and Machine Intelligence, Vol 21, No. 4, April 1999, pp. 33

[4] *Fingerprint Classification Using a Hexagonal Fast Fourier Transform*, A.P. Fitz, R.J. Green, Pettern Recognition, Vol. 29, No. 10, pp. 1587-1597

[5] *Classification of Fingerprint Images*, Lin Hong, Anil Jain, Proceedings of 11th Scandinavian Conference on Image Analysis, June 7-11, Kangerlussuaq, Greenland, 1999. (Internet: http://web.cse.msu.edu:80/TR/MSUCPS:TR98-18)