WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2007

Fractal analysis of fingerprints

John C. Deal
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Deal, John C., "Fractal analysis of fingerprints" (2007). *Graduate Theses, Dissertations, and Problem Reports*. 1852.
https://researchrepository.wvu.edu/etd/1852

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Fractal Analysis of Fingerprints

By

John C. Deal

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Electrical Engineering

Matthew Valenti, Ph.D., Chair
Alfred Stiller, Ph.D.
Charles Jaffé, Ph.D.
Xin Li, Ph.D.
Natalia Schmid, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2007

Keywords: Fractals, Fingerprints, Iterated Function Sequence, Forensics, Biometrics

Copyright 2007 John C. Deal

Abstract

Fractal Analysis of Fingerprints

by

John C. Deal
Master of Science in Electrical Engineering

West Virginia University

Matthew Valenti, Ph.D., Chair

Current methods for comparing fingerprints have weaknesses that have opened them to criticism. Current methods concentrate on the comparison of minutia in the print either manually or with the assistance of a computer algorithm. This causes these methods to depend highly on the presence of minutia and their relationship to one another. Absence or rotations of minutia can prevent current methods form making accurate comparisons. The goal of this process is to develop a new method for analyzing fingerprints that addresses many of the concerns with current methods.

The developed process uses an iterated function sequence (IFS) to convert the image of a fingerprint into a fractal pattern. The input for the IFS is constructed by a random walk through the image. Once a fingerprint is converted into a fractal pattern, the fractals can be used to make comparisons. Fractals are well defined mathematical objects that make them far easier to compare than fingerprints themselves. This process addresses many of the issues with current methods. This method is global in nature and thus it is not dependent on a set number of minutiae. Moreover, the rules for the random walk are constructed so as to make the fractal produced invariant of orientation of the print.

This method offers a new fast way to compare images. This method can be used to increase confidence, both in court and public opinion, in the use of fingerprints as identification. It can offer both an independent and/or supplemental method to the current ones used.

Acknowledgements

I thank Alfred Stiller, for introducing me to this area and for teaching me much more than just how to perform research on fractals. I thank Charles Jaffe, for all of his guidance with my research and the revision of my Thesis. I thank Lyn Ratcliff for his extensive help with the Windows API and the program. I would also like to thank Matthew Valenti for accepting my invitation to lead this diverse committee, and for all of his help adapting this project to the Computer Science and Electrical Engineering department. I also thank Natalia Schmid for offering insight on how this compares with other methods. I thank Xin Li for serving on this committee. I would like to thank the Computer Science and Electrical Engineering department and the Department of Chemistry for facilitating this inter-departmental thesis to be presented. Finally, I thank the National Institute of Justice for funding this research under Grant # 2003-RC-CX-K001.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

Table of Figures vi

List of Tables ix

List of Equations x

I. Introduction 1

A. Background (literature review) 2

Correlation-based Techniques 5

Minutiae-based Methods 6

Ridge Feature Based Techniques 14

B. Observations 17

C. Statement of the Problem 17

D. Goal of Research 17

E. Products of Research 18

II. Experimentation 19

A. Approach & Design of Methodology 19

B. Fractal Analysis 20

Chaos Game 20

Random Walk 22

Scale & Scale Spectrum 26

Normalization 30

C. Image Preprocessing 33

Outlining 34

Old Method for Conversion to Binary 35

New Method for Conversion to Binary 36

III. Program 37

Step 1 Preprocessing 37

Step 2 Fractal Creation 37

Diagrams 39

IV. Analysis **42**
A. Direct Comparison 44
Error Bars 44
Orientation Independence 48
Smudge Removal 49
B. Fourier Transform 57
C. Other Comparison Methods 59
V. Conclusions **62**
Goals of Research 63
Further Research and Future Applications 64
Bibliography **66**
Appendix A Software Manual **68**
General description 68
Installation 69
Getting Started 69
Working with the fingerprint 70
Global Options and Batch Processing 74
Other Tools 75
Appendix B Source Code **77**
Appendix C **100**

Table of Figures

Figure I-1. The image above shows fingerprints from the five classes..4

Figure I-2. Image showing the cores of two different classes of print...4

Figure I-3. Illustration of the different types of minutiae in fingerprints.....................................6

Figure I-4. Images that illustrate the termination/bifurcation duality principle. 7

Figure I-5. Partial image of the average orientation map for a print. 8

Figure I-6. Images showing the Poincare index 9

Figure I-7. Images showing the conversion of a grayscale image to its reduced binary form 10

Figure I-8. Images illustrating the crossing number for various minutiae. 11

Figure I-9. Image that shows some typical false minutiae and their repaired forms. 11

Figure I-10. Image of matching minutiae from two images..12

Figure I-11. Illustration of an local texture matching method...15

Figure II-1 Two examples of the chaos game. 20

Figure II-2. Two results of the chaos game...22

Figure II-3. This figure illustrated the process developed to pick the two points 24

Figure II-4. The fractal produced from the described process 25

Figure II-5. The two fractals above are from the same fingerprint but on different scales..........26

Figure II-6. Fractal with the corner probabilities labeling their respective regions.....................28

Figure II-7. Scale spectrum for a fingerprint..29

Figure II-8. Two different fingerprints for the same class of print..29

Figure II-9. Scale spectra for the two prints in Figure II-8. 30

Figure II-10. Graph showing the normalized spectrum for a fingerprint...................................32

Figure II-11. Graph showing the determinate of the probability matrix....................................32

Figure II-12. Image showing a fingerprint that has been outlined 33

Figure II-13. Image showing the 9x9 neighborhood of pixels 35

Figure III-1. Data flow diagram for program. 39

Figure III-2. General flowchart for the whole program. 40

Figure III-3. Detailed flowchart for the random walk. 41

Figure IV-1.Samples of the four databases. 43

Figure IV-2. Plot the scale spectra from two prints. 44

Figure IV-3. Graph showing uncertainty as a function of random walk length. 45

Figure IV-4. Graph showing the time necessary to perform a random walk 46

Figure IV-5. Graph showing uncertainty as a function of time to execute. 46

Figure IV-6. The graph shows the difference in two runs over the same image.. 47

Figure IV-7. The above plot shows the difference from a print and its rotated version. 48

Figure IV-8. Fingerprint image and its outlined version with masked pixels shown in green..... 50

Figure IV-9. Image of a fingerprint with a small portion blacked out 51

Figure IV-10. Plot showing the difference for the fingerprint and its smudged version. 51

Figure IV-11. The images above show the original print with 5% smudged 52

Figure IV-12. Plot showing the difference for a fingerprint and its version with 5% smudged. . 52

Figure IV-13.The above images show the original print with 10% smudged 53

Figure IV-14. Plot shows the differences for a print and its smudged version with 10% 53

Figure IV-15. The above images show the original print with 5% smudged in a new region 54

Figure IV-16. Plot shows the differences version with 5% removed from a new region. 55

Figure IV-17. Images show the original print with 2% smudged in a new 5 different regions ... 55

Figure IV-18. This plot shows the differences in scale spectra for a print and its smudged version with 2% removed from 5 different regions. 56
Figure IV-19. Plot showing the difference in two different prints. 57
Figure IV-20. Plot showing the FFT for two different fingerprints. 59
Figure IV-21. Plot showing the scale spectrum and trend lines. 60
Figure A-1. The Initial program window. 70
Figure A-2. TIFF image dialog. 71
Figure A-3. Scale Method Dialog. 72
Figure A-4. Tiff dialog upon completion. 73
Figure A-5. Tiff dialog upon completion. 74
Figure A-6. Scaling parameter to fractal tool. 76

List of Tables

Table I-1. Table showing the estimated fractal dimensions for some prints. 16

Table II-II-1. Table showing the steps to take in the chaos game 23

Table IV-1. The four FVC 2000 databases taken from Handbook...43

Table IV-2. Table showing the coefficients of the fitted polynomials. 61

List of Equations

Equation I-1. Equation for correlation comparison. 5

Equation I-2. Equation to determine the orientation image components. 8

Equation I-3. Equation for the Poincaré index. 9

Equation I-4. Equation for the crossing number. 10

Equation II-1. Equation relating the scaling parameters. 28

Equation II-2. Equation for the probability of a black pixel at a given scale. 31

Equation II-3. Equation for the probability of a white pixel at a given scale. 31

Equation II-4. Equation to verify probabilities. 31

Equation II-5. The first normalization equation. 31

Equation II-6. Scaling parameters matrix and the equation for the determinate. 31

Equation II-7. Outlining threshold equation. 34

Equation II-8. Old binary conversion decision equations. 36

Equation II-9. New binary conversion decision equations. 36

Equation IV-1. Multidimensional vector distance formula. 60

I. Introduction

People have been interested in the individuality of fingerprints for years. Artifacts show that ancient people were likely aware of the individuality of fingerprints [1]. In 1864 a paper was published by an English plant morphologist describing his study of ridge and pre-structures [1]. Though, it was not until 1888 that Sir Francis Galton introduced the use of minutiae features for fingerprint matching [2]. By the early twentieth century, fingerprint recognition by minutiae was formally accepted [1]. In the 1960s, the FBI began development of an automated system to compare fingerprints [1]. There has been difference of opinion on the number of minutiae to establish a positive identification [3]. The number of minutiae used varies by country and is established by observation, not scientific study [3].

The individuality of fingerprints has not been formally proven; it is an observation, not a proven scientific fact. There is growing public and legal concern about the uniqueness of fingerprints [3]. The real question at hand is whether or not each person has a unique fingerprint and also whether the method of matching location of minutiae is detailed enough to produce accurate identifications. Some papers have speculated that the fingerprints of identical twins are 95% similar [3]. It has also been speculated that comparison of a partial latent fingerprint may not be able to show the difference in very similar prints [3].

The method described in this thesis, addresses some of the problems and concerns in the current processes. The method that we have developed is global in nature since the data sampled to construct the fractal is taken from the entire image. This is distinct from the traditional methods that are based on local features such as minutiae. Using this unique approach and the

inherent speed of our method current databases could be compared to one another to offer some proof that at least those prints are unique.

A. Background (literature review)

Comparison of fingerprints can be accomplished either manually or automatically. Human fingerprint examiners use minutiae in the fingerprint to compare two prints and decide if they came from the same finger. The accepted standard in the US is twelve minutiae for a fingerprint to be considered a match [3].

The matching of fingerprints is made difficult because the same finger can produce different fingerprint images due to the following factors [3].

- Displacement: The fingerprint will not always be in the same position in the image due to differences in finger position during capture.
- Rotation: The fingerprint will not always have the same orientation in the image due to the finger being twisted from normal during capture.
- Partial overlap: A portion of the fingerprint may be missing because the finger was placed over the edge of the sensor.
- Non-linear distortion: These differences in the image are caused by trying to obtain an image on a two-dimensional surface from a three-dimensional finger. The skin elasticity of the print will cause differences in the image produced from separate acquisitions. These differences can also be produced by the user applying torque to the finger during acquisition causing ridge distortion.

- Pressure and skin condition: Pressure, skin condition, sweat, hydration and skin disease can cause the fingerprint image to be different between acquisitions of the same print.

- Noise: Problems created when the image is obtained can cause disturbances or noise in the image. Examples would be residues left on the glass of a sensor or other disruptions in the surface on which the finger was placed. These problems would cause a distorted or smudged area of the fingerprint.

- Feature extraction error: Algorithms to extract minutiae and ridge details can cause measurement errors. Algorithms to enhance and extract features are aggressive and can sometimes add false or distorted details. This would only be an issue with automated systems.

Matching by a human examiner is time consuming and has questionable reliability. Even though the examiners are trained, they will each have their own standards that make their examination of fingerprints different. The manual matching of fingerprints is a subjective process and producing a quantitative measure of a match is difficult if not impossible.

Due to the difficulties and limitations of manual matching, many automated methods have been developed for the analysis of fingerprints. Automated methods for matching can be divided into three classes; correlation-based, minutiae-based and ridge feature-based [3]. All of these methods are based on the location of features in the fingerprint so they are dependent on the displacement and the orientation of the fingerprint.

To allow the automated matching of fingerprints, they must be aligned to account for displacement and differences in orientation. To align a fingerprint, the center of the print or core must be found. The core of the fingerprint is defined as "the north most point of the innermost ridge line [4]". This point is often just the middle of the overall structure of the print. Some classes of fingerprint do not have a well defined center, such as the arch type, and are difficult to align (see Figure I-1). Figure I-2 shows the core of two different classes of fingerprints.

Figure I-1. The image above shows fingerprints from the five classes. (Image from Figure 3.3 on p84 of Handbook [3]. With permission.)

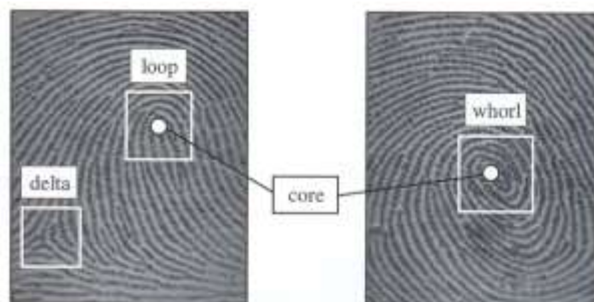


Figure I-2. Image showing the cores of two different classes of print (Image from Figure 3.2 on p84 of Handbook [3])

Correlation-based Techniques

Correlation methods involve the superposition of one print over another and the comparison of their intensities directly. The simplest measure of the differences in the intensities is the sum of the squares of the differences in intensities. The sum of the squares is given by Equation I-1 for images represented by matrices T and I [3].

Equation I-1. Equation for correlation comparison.

$$\|T - I\|^2 = \|T\|^2 + \|I\|^2 - 2T^T I$$

In this equation the second term in the final expression is just -2 times the cross correlation of the images. From this observation it is seen that maximizing the correlation minimizes the distance. High correlation implies that the images are likely from the same finger. The displacement and rotation of the image will affect the correlation so the correlation must be maximized as a function of core position and rotation.

Differences in pressure and skin elasticity along with different collection environments and methods affect the image and cause difficulty with the described correlation comparison. The use of better correlation methods such as normalized cross-correlation can help to negate the effects from these factors [5].

Calculation of the maximum correlation of a relatively small image over the displacement and rotation values is extremely time consuming operation. Because of this, the correlations are often calculated on a local and not a global area. Each of the local regions in the template image are extracted and correlated with the whole input image [6].

Minutiae-based Methods

To understand minutiae methods one first has to examine how minutiae are defined and extracted from a fingerprint image. By definition the word minutia means a small detail. In fingerprints these 'small details' are the discontinuities in the ridges of the fingerprint. Minutiae come in many forms [2] and some typical types are shown in Figure I-3.

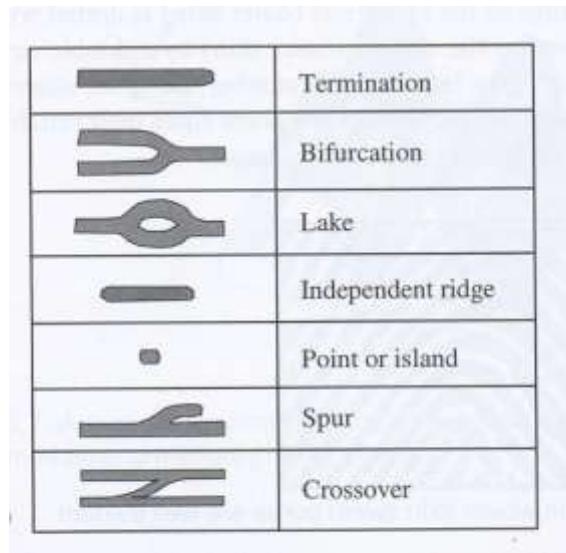


Figure I-3. Illustration of the different types of minutiae in fingerprints. (Image from Figure 3.4 on p85 of Handbook [3])

There is some difference of opinion about which of these minutiae should be used to identify a print. American National Standards Institute (ANSI) suggests there should be four classes of minutiae [7]: *terminations, bifurcations, trifurcations (crossovers), and other*. The FBI's model considers only terminations and bifurcations [8]. If one examines the negative of a fingerprint image, terminations become bifurcations (see Figure I-4). Where a ridge terminates, the corresponding valley bifurcates creating a feature called *termination/bifurcation duality* [3].

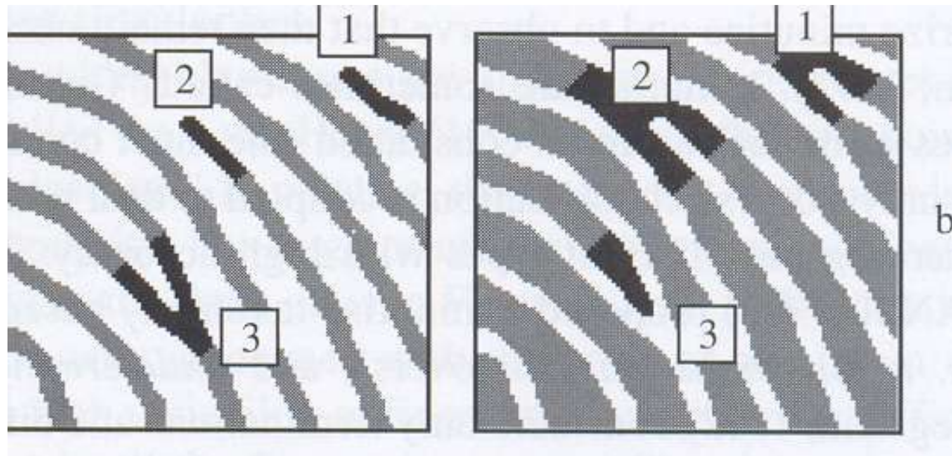


Figure I-4. Images that illustrate the termination/bifurcation duality principle. (Image from Figure 3.5 on p86 of Handbook [3])

The information about a minutia used for comparison includes the class, position (or coordinates), and its angle or orientation to the horizon. Extracting the minutiae from the fingerprint involves determining the orientation. The most common method for determining the orientation of minutiae in the print is to use an orientation image [9]. Each element in the orientation image's array corresponds to the average ridge orientation in that element's neighborhood. When creating the orientation matrix, a reliability matrix is also created showing the reliability of the orientation estimate. The reliability value can be used to determine high and low quality (or noisy) portions of the image. Figure I-5 shows the orientation image for part of a print. An illustration of the orientation and reliability is also shown.

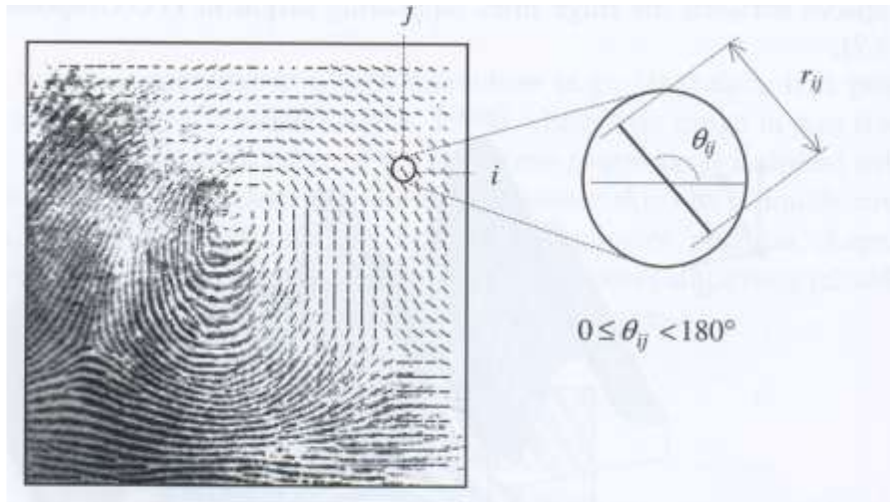


Figure I-5. Partial image of the average orientation map for a print. The illustration to the right shows what the orientation and reliability matrices describe. (Image from Figure 3.8 on p88 of Handbook [3])

Most methods for computing the orientation of the ridges involves taking the gradient of a pixel map. The phase angle of the gradient is the direction of max intensity change of the pixels. The gradient estimate of one pixel is on too small of a scale to be used. To solve this problem, a method was proposed to determine the average phase angle over an *n by n* region [10]. Equation I-2 shows how the estimate **d** is computed. Where $r_{i,j}$ is the distance from the calculated point and $\theta_{i,j}$ is the angle.

Equation I-2. Equation to determine the orientation image components.

$$\boldsymbol{d} = \left[\frac{1}{n^2}\sum_{i,j} r_{i,j}\cos 2\theta_{i,j}\,,\frac{1}{n^2}\sum_{i,j} r_{i,j}\sin 2\theta_{i,j}\right]$$

Once an orientation image for a print is obtained, the next step in the minutiae matching process is singularity detection. Singularities are usually detected using the Poincaré method

[11]. The Poincaré index is computed for each element in the orientation image by algebraically summing orientation differences for its adjacent elements. Equation I-3 shows how the index is calculated and a list of the singularities certain indices detect [12]. Figure I-6 shows an illustration of the Poincaré index for some singularities. The detection of singularities is used to determine the core of the fingerprint image. The core is used to align the fingerprint image for comparison.

Equation I-3. Equation for the Poincaré index.

$$P_{G,C}(i,j) = \sum_{k=0}^{7} angle\left(d_k, d_{(k+1) mod\, 8}\right)$$

$$P_{G,C}(i,j) = \begin{cases} 0° & if\ [i,j] does\ not\ belong\ to\ any\ singular\ region \\ 360° & if\ [i,j] belongs\ to\ a\ whrol\ type\ singular\ region \\ 180° & if\ [i,j] belongs\ to\ a\ loop\ type\ singular\ region \\ -180° & if\ [i,j] belongs\ to\ a\ delta\ type\ singular\ region \end{cases}$$

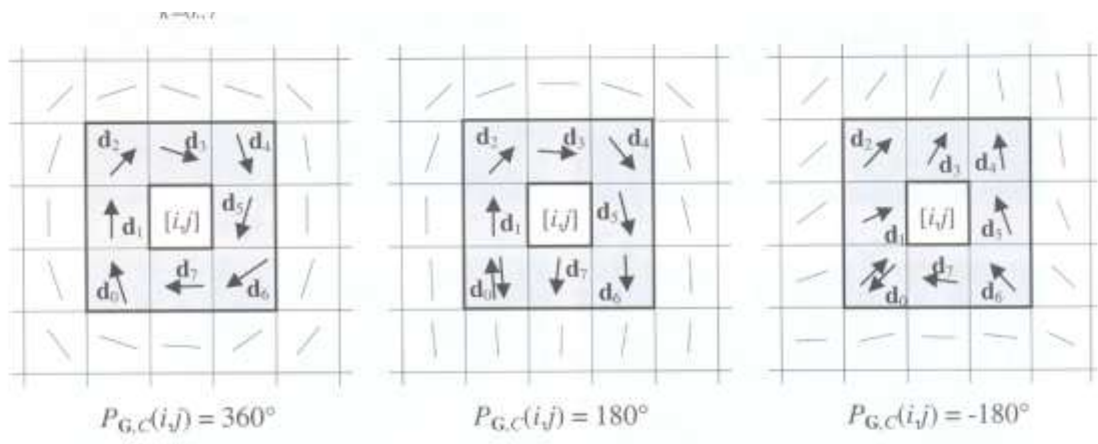


Figure I-6. Images showing the Poincaré index for some singularities in the orientation image. (Image from Figure 3.15 on p98 of Handbook [3])

The next step in the minutiae based method is the detection of the minutiae in the print image. Since most images of fingerprints are in a grayscale form, minutiae detection is simplified by conversion to a binary image. Each of the ridges in the binary image is then reduced to one pixel to further simplify minutiae detection. Figure I-7 shows a fingerprint image, its binary version and the reduced image [13].

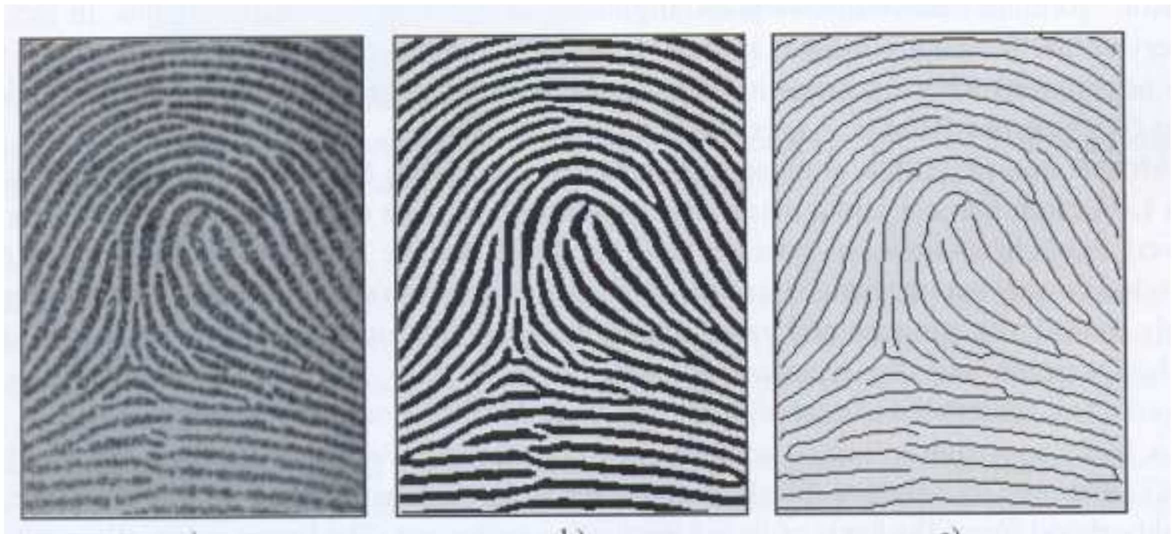


Figure I-7. Images showing the conversion of a grayscale image to its reduced binary form. (Image from Figure 3.31 on p113 of Handbook [3])

Using the thinned binary image, minutiae are detected by calculating the *crossing number* for each element of the thinned image. The crossing number is half the sum of the differences between pairs of adjacent pixels in the eight pixels neighboring the pixel [14]. Equation I-4 shows how the crossing number is obtained. A crossing number other than 2 indicates a minutia at that pixel. Figure I-8 illustrates some minutiae and their corresponding crossing numbers.

Equation I-4. Equation for the crossing number.

$$cn(p) = \frac{1}{2}\sum_{i=1}^{8} |(p_{i \bmod 8}) - (p_{i-1})|$$

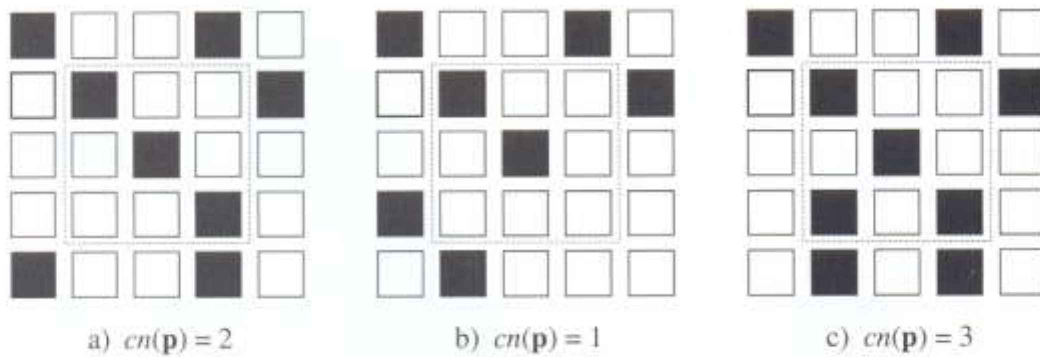


Figure I-8. Images illustrating the crossing number for various minutiae. (Image from Figure 3.36 on p119 of Handbook [3])

The thinning process can introduce some false minutiae in corrupted areas of the fingerprint. For example, arrow sections of a continuous ridge may disappear during thinning and create two terminations. Algorithms have been created to detect and remove false minutiae from the thinned image [15]. Figure I-9 shows common false minutiae and their corrected forms after application of the algorithm.

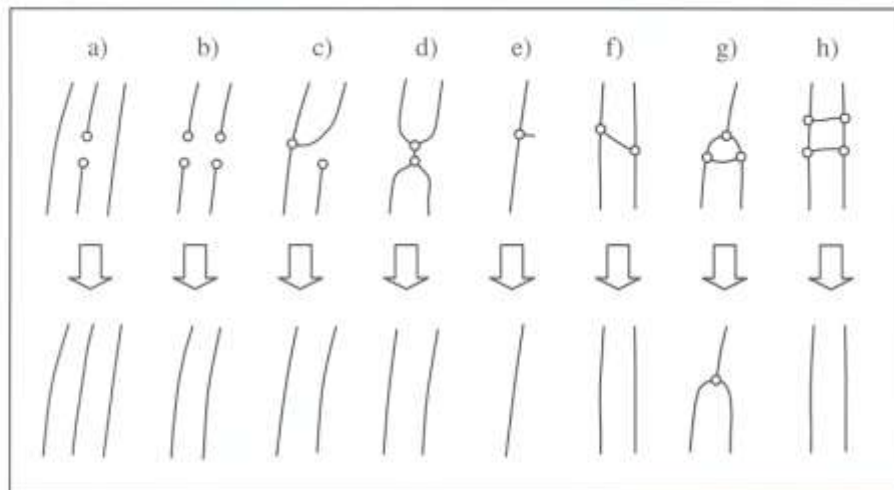


Figure I-9. Image that shows some typical false minutiae and their repaired forms. (Image from Figure 3.41 on p125 of Handbook [3])

The list of minutiae obtained along with their positions in the image and their orientation can be used to make comparisons to other prints. Distortion tolerant transforms are applied to maximize the number of matched minutiae and negate to effects of distortion. Figure I-10 shows some minutiae from two prints with the first print's minutiae as circles and the second as x. Gray circles indicate a match because the minutiae from the two prints are within tolerance levels.

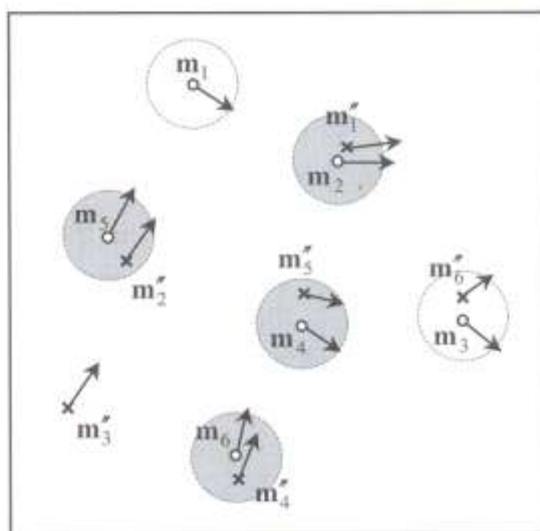


Figure I-10. Image of matching minutiae from two images. (Image from Figure 4.4 on p144 of Handbook [3])

The solution is trivial when the two prints are correctly aligned. This happens when both cores have the same displacement and the rotations are also equal. For prints that are not correctly aligned, the maximization can be solved for the (Δx, Δy, θ) variables using a least squares approach [16]. Solving this maximization using the brute force approach may be the most obvious, but is computationally prohibitive.

Most different methods of fingerprint matching concentrate on handling the distortion and rotation problems with minutiae matching. These methods take some different approaches to maximizing the number of matching minutia by obtaining the correct alignment. Some methods for determining matches include relaxation, algebraic and operational research solutions, tree pruning, energy minimization, and Hough transforms.

In relaxation, confidence intervals for pairs of points are adjusted until certain criteria are satisfied [17]. The algebraic approach proposes that exact alignment can be accomplished using affine transformation [18]. For tree pruning, correspondence between points is found by searching over a tree of possible matches while applying pruning methods to reduce the search space [19]. Energy approaches associate energy with each solution to the matching problem, and then minimizes the energy using a stochastic algorithm [20]. The Hough approach converts the matching problem into a peak detection problem in Hough space [21].

Pre-alignment of the fingerprint images offers a great improvement in the speed of the matching process. Even though methods with built-in alignment offer more robust algorithms for noisy images, they do not offer great enough throughput for some applications such as AFIS. There are two types of pre-alignment, absolute and relative.

In absolute pre-alignment, templates in the database are pre-aligned and stored. The input image is then aligned just once before being compared to all images in the database. The main difficulty is in the registration of the input image, as a mistake here will result in a matching error. The registration process depends on core detection, which can be difficult.

In relative pre-alignment, the input image is aligned with each template in the database before matching. Relative pre-alignment offers a speed increase over methods without pre-

alignment but cannot compete with absolute pre-alignment. Relative pre-alignment is more effective than absolute because features of the template can be used in the registration process.

Ridge Feature Based Techniques

Ridge feature based methods can be used in conjunction with or instead of minutiae based methods. When used in conjunction with minutiae methods, they can increase accuracy and robustness. Sometimes ridge feature based methods are used instead of minutiae methods because of difficulties in extracting minutiae from poor quality images. Below are some of the other features of a print that can be used for matching [3]:

1. Size and silhouette shape.
2. Number, type and position of singularities.
3. Spatial relationship and geometrical attributes of the ridge lines [22].
4. Shape features [23].
5. Global and local texture information.
6. Sweat pores [24].
7. Fractal features [25].

Approaches numbered 1 and 2 are unstable and highly dependent on the image collection method. Out of the others, global and local texture information and fractal features merit further consideration in this thesis because they have the closest relationship to the method proposed.

Textures are characterized by properties of an image such scale, orientation, frequency, symmetry, and isotropy. One method proposed looks at the global textures using the Fourier domain. In this method, a "wedge-ring detector" is used to produce a feature vector that is

independent of translation, rotation and scale [26]. This is difficult because specific ridge features and orientations show up as small changes in the frequency domain, while features such as ridge frequency dominate. In global analysis, most spatial information is lost.

Most local texture analysis is performed on the orientation and frequency images described earlier. One method proposed that an area of interest in the print be tessellated with respect to the core [27]. A feature vector is then obtained from ordered enumeration of the features found in each sector of the tessellation. This feature vector contains both global information and local details from the sectors. A Gabor filterbank is used to decompose the information in the sectors. Figure I-11 illustrates the process described.

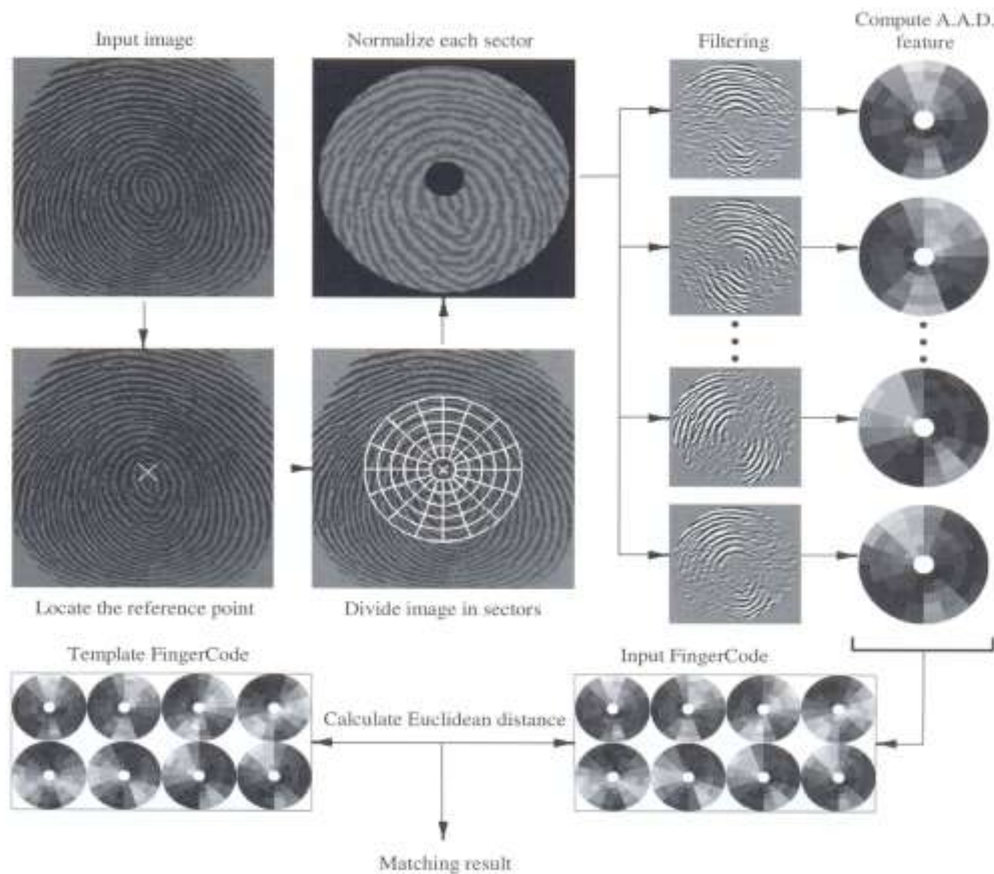


Figure I-11. Illustration of an local texture matching method. (Image from Figure 4.19 on p167 of Handbook [3])

The matching method based on fractal features uses existing algorithms to calculate Estimates of Fractal Dimension (EFD) [25]. The box counting algorithm to obtain these EFDs divides the image into boxes of length l containing at least one sample of light intensity. Plotting the log of this intensity versus the log of l gives a slope that estimates the fractal dimensions.

The EFD can be calculated for each pixel along with the whole image. The EFDs for each pixel comprise a matrix that is referred to as the *fractal dimension map* of the image. These maps can be used to compare images of low quality. For comparison, a grid EFD is used. The grid EFD bins the number of boxes N of a given size that have the same intensity. Plotting the log of N versus the log of the intensity gives a slope as a measure of the fractal dimension. Table I-1 shows a couple of points on two different person's fingers. Notice how the fractal dimension for the different points on the same finger is close while the fractal dimension between persons is noticeably different.

Table I-1. Table showing the estimated fractal dimensions for some prints. (Data from Table 1 in Polikarpova's paper [25]).

	Person 1	Person 2
Point 1	2.45	2.51
Point 2	2.46	2.51

Even though there are differences in the EFD for the two different prints, the measures are reasonably close. With a difference this small it is likely, that given a large set of prints, two different prints would produce the same fractal dimension measure.

B. Observations

By surveying the current methods for analyzing fingerprints, it is clear that a more reliable method needs to be developed. Many current methods only use small portions of the print to identify them. A global approach that uses the whole print and is not affected by many of the typical problems was developed and is described in this thesis.

Using an iterated function sequence to create a fractal from the image of a fingerprint helps with many of the difficulties in fingerprint analysis. The fractal has mathematical properties that allow it to be analyzed much faster and with less difficulty than the image itself. Information about the error in the random walk and the properties of the fractal affects the reliability of the match.

C. Statement of the Problem

A fast algorithm that provides a fractal that is representative of the fingerprint that produced it is needed. Further, this algorithm needs to easily account for differences in images from the same print, including independence of displacement and rotation. This fractal should then be able to be matched to another print using the same process.

D. Goal of Research

The goal of this research is to provide a fast algorithm to produce a fractal representative of the fingerprint from which it came. The algorithm also should provide solutions to many of the current problems with fingerprint matching using a fractal method. The goal is to produce an algorithm that is unaffected by the typical variable that affect multiple acquisitions of the same print. This method addresses the following acquisition variables from the list presented in the literature review:

- Displacement
- Rotation
- Partial Overlap
- Non-linear distortion.
- Pressure and skin condition
- Noise
- Feature extraction errors.

E. Products of Research

The products of this research include the algorithm with the properties described above. The research also produced a program that implements the algorithm in C. The research also provided many fractals and data to be analyzed for verification of the method.

II. Experimentation

A. Approach & Design of Methodology

The goal of the research described in chapter one is to address some of the problems with current methods for fingerprint analysis, by developing a new method. This method should also provide some assurance to the reliability of fingerprints as a source of identification. Before the process was designed, many of the weaknesses of other minutiae based methods were examined. These weaknesses are detailed in the Literature review earlier in the document.

The design of this process allows it to address many of the concerns with fingerprint identification. Some examples of the items addressed in the program are independence of orientation, use of the entire image not just select points, and the ability to provide a quantitative measure of a match. Many computerized matching methods have difficulty if a fingerprint is rotated more than 15° [3]. An algorithm that was independent of orientation of the fingerprint would be far more useful.

Most current automated and manual methods depend on the comparison of minutiae in fingerprints for their matching. The difficulty with this is the question of how many minutiae to use for an accurate match and also if that number is available in a partial print [3]. A method that used data from the entire print instead of just a few key minutiae would answer both of the questions raised.

Offering a quantitative measure for the comparison of fingerprints would offer reassurance on their use for identification. A process based entirely on mathematical operations that have a defined error would allow one to use this error to produce a probability of a match.

B. Fractal Analysis

A fractal is a figure that can easily be broken into smaller parts. Each of these parts will have a self-similar pattern. Each part is a scaled copy of the whole [29].

Chaos Game

The iterated function sequence (IFS) that was used to create the fractal is the well known chaos game [29]. The chaos game starts by choosing an initial point within a square (or other geometrical shape). The game progresses by a series of moves. Each move adds a new point to the fractal. The first move places a point at a distance halfway between the initial point and one of the corners of the square. The second move places another point at a distance halfway between the second point and one of the corners. In principle this process is repeated an infinite number of time, while in practice it is repeated until fractal is developed to the desired level of detail. Figure II-1 shows a few examples of the starting moves in the chaos game for squares moving halfway to the corner.

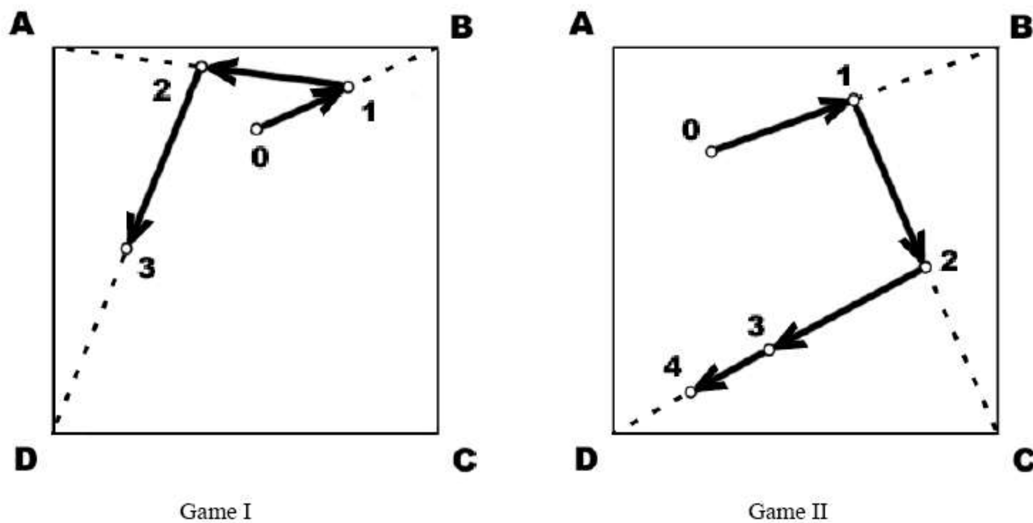


Figure II-1 Two examples of the chaos game. In the first game the moves taken were to corners BAD. For the second game the moves taken were BCDD.

The selection of the corners in each of the moves is the cruces of the procedure. In the traditional chaos game, the corners are chosen randomly. If the chaos game is played on a triangular board and the corners are chosen randomly, the fractal produced is the well known Sierpinski triangle [30]. On the other hand, if the chaos game is played on a square board, then the pattern produces is a uniform gray (see Figure II-2). While both of these patterns meet the definitions of a fractal, the properties of a fractal are more readily seen in the Sierpinski triangle.

The Sierpinski triangle displays many properties fractals posses including *self-similarity*. A self-similar fractal contains the same information as the whole in a smaller portion of the fractal pattern. The top third of the triangle repeats the same motif, and it is a scaled copy of the entire triangle. The same holds for the top third of that portion. If the number of moves used to produce the fractal was infinite then the resolution of the fractal will also be infinite. The triangular pattern exhibited in figure II-2 would be reproduced on all scales. However, since in practice, one must limit the number of moves made in the development of the fractal, the resolution of the fractal will also be limited. In practice one chooses the number of moves used to develop the fractal sufficiently large so that one obtains the desired level of resolution (see Figure II-2).

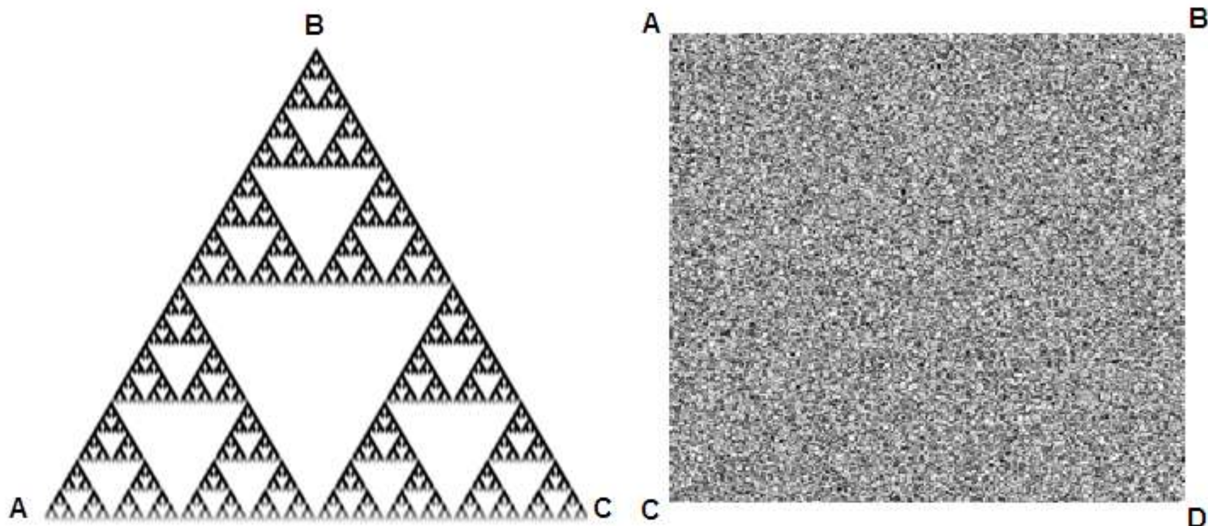


Figure II-2. Two results of the chaos game. In the first one the game was played over a triangle to produce the Sierpinski triangle. Notice the self-similarity in the triangle fractal. In the second the game was played over a square to produce a gray fractal.

Random Walk

The difference between the traditional chaos game and the method of analysis developed here lies in the choice of the corners. Instead of picking the corners randomly our method picks the corners based on information from a fingerprint image. The information from the fingerprint image is obtained by comparing two pixels in the image and using the result to select the next corner, by the rules of the Chaos Game, to approach (see Table II -1). The fingerprint image used to obtain the information is a binary image so there are only two possible values for each point in the image.

Value of points in pairing		Corner to approach
Point 1	Point 2	
Black	Black	A
Black	White	B
White	Black	C
White	White	D

Table II-II-1. Table showing the steps to take in the chaos game based on comparing two pixels in the fingerprint image

The pixels to be compared in the fingerprint are determined by a random walk through the fingerprint. The random walk through the image ensures the fractal is produced from the image as a whole and not just from a few features.

To specify two pixels to compare four values are needed. If all four values are picked randomly the resulting fractal only offers information concerning the proportion of black and white pixels in the image. In our algorithm we chose pairs of pixels that are separated by a fixed distance. As consequence only three of the four values are picked randomly, the last being the distance between the two pixels being fixed. Moreover, by varying the distance between the pixels we can construct an entire family of fractals from a single fingerprint.

At each step in the iterated function sequence used to create the fractal, two random numbers are chosen to determine the x and y coordinates of the point midway between the two pixels. This accounts for two of the three random values needed to select two pixels. A third random number is selected to specify the angle that the line connecting the two pixels makes with the horizontal axis. Using these three random values and a predetermined distance the two pixels to be compared are readily determined (see Figure II-3). This process is repeated for each step of

the iterated function sequence and constitutes a random walk through the fingerprint. After many thousands of steps in the iterated function sequence a fractal similar to the one shown in Figure II-4 is produced.

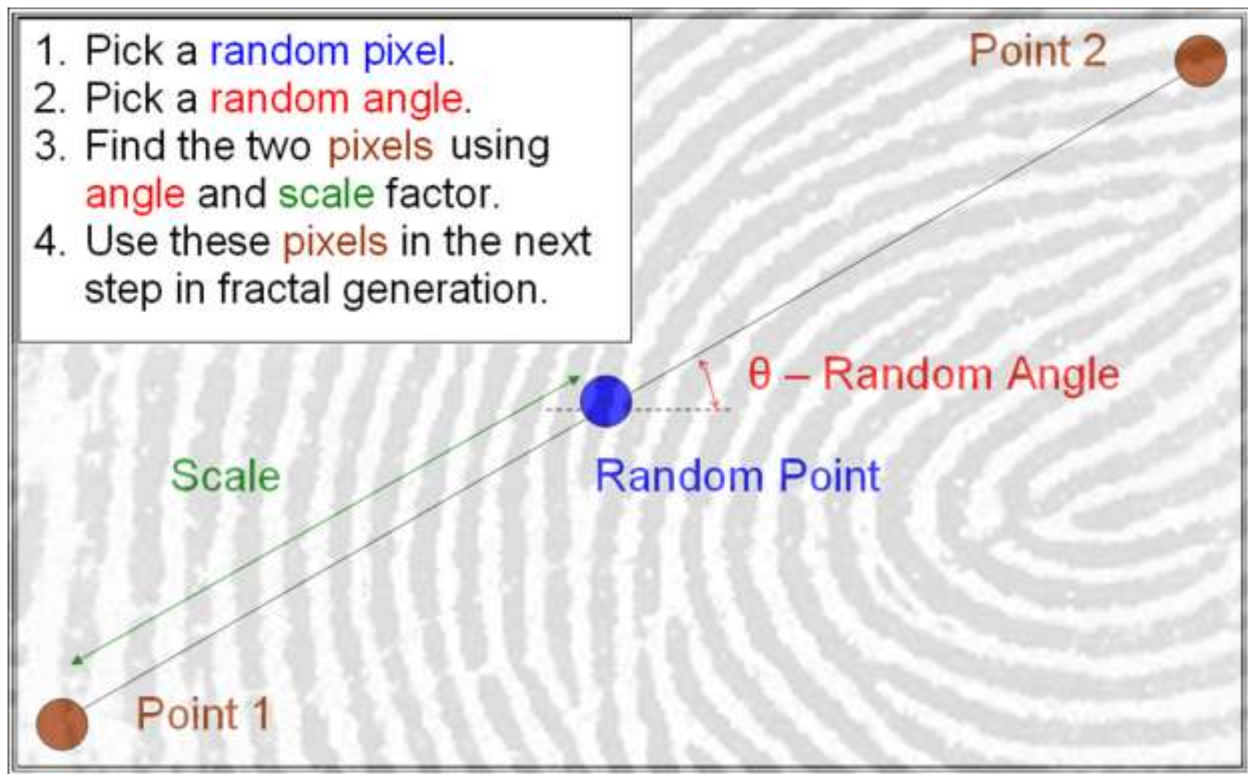


Figure II-3. This figure illustrated the process developed to pick the two points for comparison. The process is repeated for each step in the random walk. The points obtained in each step define to next corner in the chaos game according to Table II-1.

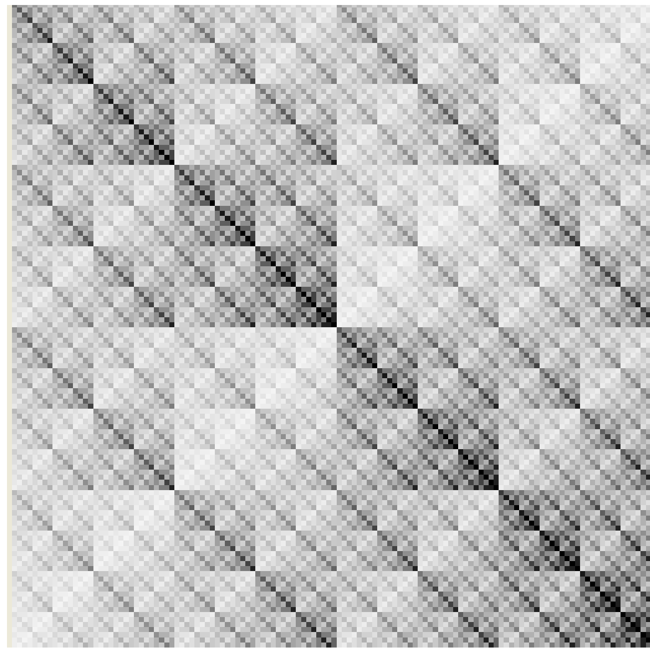


Figure II-4. The fractal produced from the described process (500,000 itterations).

Using the random walk through the fingerprint in order to determine the sequence of corners to use in the iterated function sequence defined by the chaos game, results in a self-similar fractal. It is important to observe that while the walk through the fingerprint is random, the information extracted from the fingerprint in this manner is not random. If the information extracted in this manner was random, then the resulting fractal would have been uniformly gray, however, it is clear from the fractals shown in the various figures in this section that they are not uniformly gray. The self-similarity of the fractals is very similar to that seen in the Sierpinski triangle, the difference being that instead of the upper triangle being repeat again and again, in this case it repeated motif is found in the upper quadrangle (see Figure II-5).

Scale & Scale Spectrum

As discussed in the previous section, the algorithm used for determining the sequence of corners chosen in construction the fractal using the chaos game is based upon a comparison of two pixels separated by a given distance. If this distance is varied, a different fractal is constructed. This is illustrated in Figure II-5 where two fractals are shown. The difference between these two fractals is the distance between the pixels which are used to determine the sequence of corners to use in the chaos game. In this research the distance separating the pair of pixels is referred to as the scale. This scale is different from the scale used in the fractal dimension method described in chapter one. The other methods scale is a scale of the fractal while this methods scale is over the fingerprint.

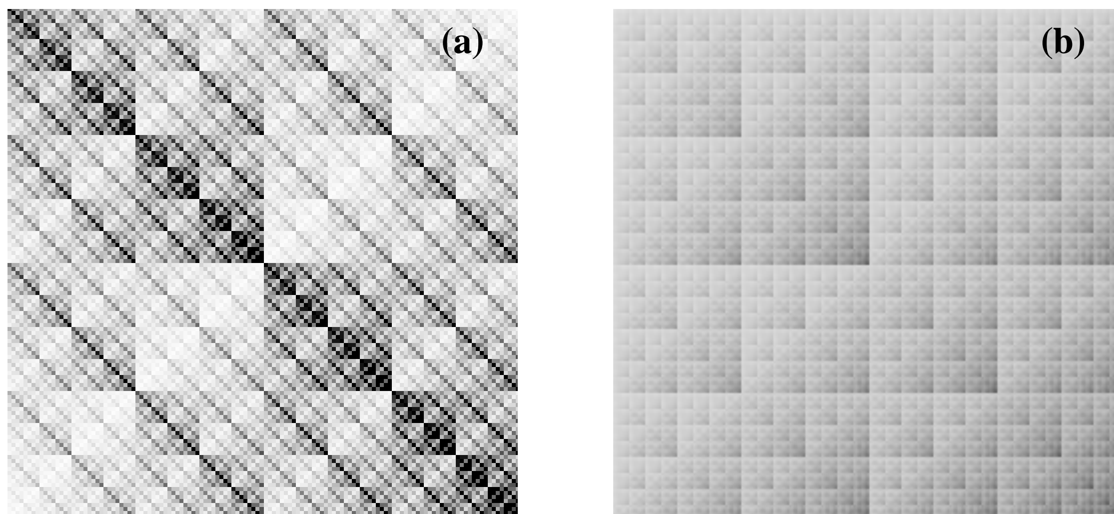


Figure II-5. The two fractals above are from the same fingerprint but on different scales. The scale is defined as half the distance between the two pixels being compared. The scale of (a) is $\lambda = 2.8$ and the scale of (b) is $\lambda = 18.2$.

The difference in the fractals is no surprise as fingerprints have features at different scales. On the large scale, fingerprints have shapes and general ridge flow that classify them into a certain category. On the medium scale, fingerprints have the individual ridges and their splits.

Finally, on the small scale, fingerprints have sweat pores that appear if the image has a high enough resolution.

The fractals that are constructed using the chaos game/ iterated function sequence can be characterized by a set of scaling parameters. When observing the self-similar behavior of these fractals, it is clear that when reducing the full fractal to fit into one of the four quadrants that the intensity (darkness) must either be increased or decreased. The factor by which the intensity is increased or decreased defines the scaling parameters just mentioned above. As there are four quadrants there will be four scaling parameters.

The scaling parameters are related to the "darkness" of each of the four quadrants. In turn, the darkness of each of the four quadrants is determined by the number of points that lie in each of these quadrants. The iterated function sequence defining the chaos game has been constructed in such a manner that if two pixels are white-white, then the associated point lies in the top left quadrant, if the two pixels are black-black then the associated point lies in the lower right quadrant. Similarly, if the two points are either white-black or black-white then the associated point lies in either the upper right or lower left quadrant. As the "darkness" of each of the quadrants is directly proportional to the number of points that lie within it, the scaling factors are seen to be equal to the probabilities that a given point is in a particular quadrant.

We label these scaling parameters α_{00}, for the white- white quadrant, α_{11} for the black black quadrant, β_{01} for the white-black quadrant and finally, β_{10} for the black-white quadrant. Next we note that the ordering of the pixels in the pairs is arbitrary, consequently, it is expected that the two scaling parameters β_{01} and β_{10} should be equal. (we have verified this numerically and these two scaling parameter have been observed to be equal to within the expected level of numerical

error). Moreover, as the scaling parameters are probabilities, their sum should be equal to unity, see Equation II-1. Thus we see that there are two relationships between these four scaling parameters and so only two will be independent.

Equation II-1. Equation relating the scaling parameters.

$$1 = \alpha_{00} + \alpha_{11} + 2\beta$$

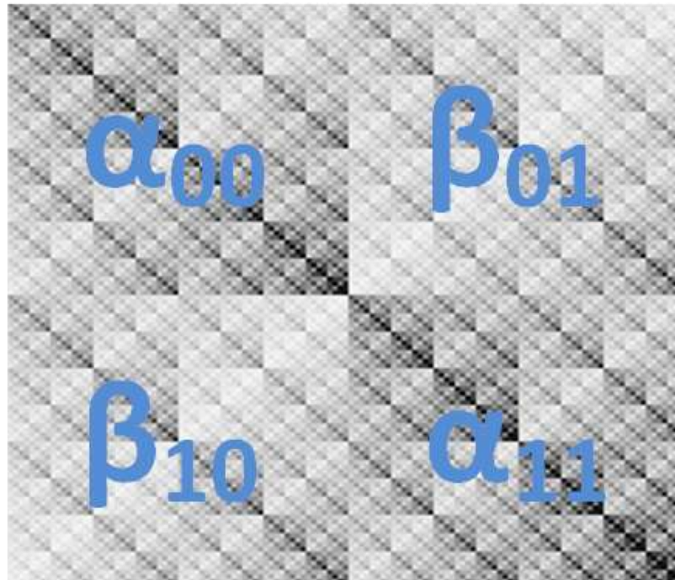


Figure II-6. Fractal with the corner probabilities labeling their respective regions.

As we have seen in the figure II-5, different choices of the scale (or distance between the pixels to be compared) result in different fractals and consequently different scaling parameters. In order to characterize a fingerprint we calculate the scaling parameters or probabilities for a wide range of scales for a given fingerprint. These probabilities are then plotted as a function of the scale. The resulting plot is called the scale spectra of the fingerprint.

Figure II-7 shows the scale spectra of a finger print, in Figure II-8 two fingerprints are shown. Their respective scale spectra are shown in Figure II-9. These scale spectra are clearly different and can be used to distinguish between the different fingerprints.

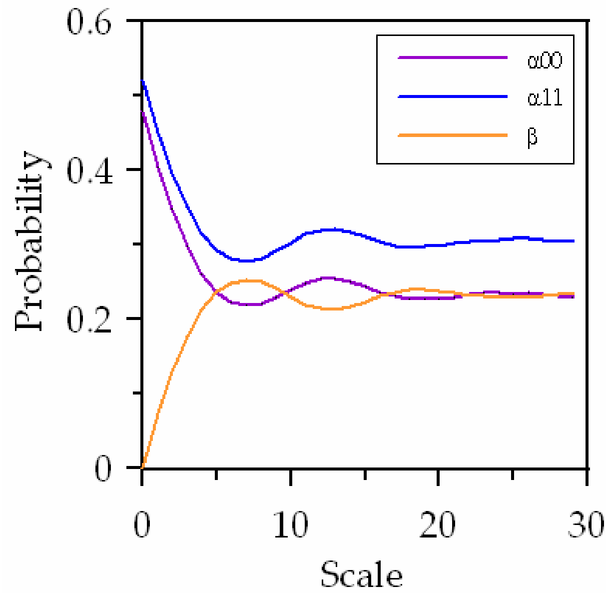


Figure II-7. Scale spectrum for a fingerprint. The spectrum is a plot of α00 is the probability of white-white α11 is the probability of black-black and β represents the equal probability white-black and black-white probabilities.

Figure II-8. Two different fingerprints for the same class of print.

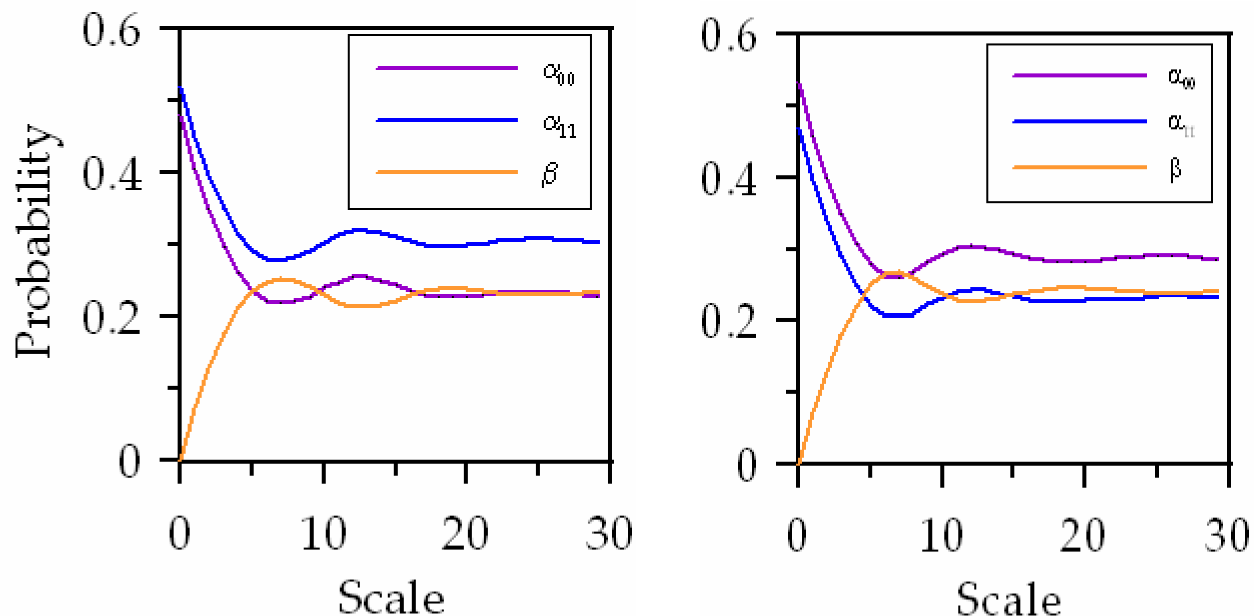


Figure II-9. Scale spectra for the two prints in Figure II-8. Notice that even though the prints are similar in type size and proportion of black to white pixels their scale spectra are noticeably different.

Normalization

Variables in the collection of fingerprints affect the image used by the process. Pressure and the resolution of the scan can affect the ridge width in the image and thus the proportion of black to white pixels in the image. To ensure these factors don't lead to false matches or rejections, the scale spectrum needs to be normalized. Normalization involving all probabilities would account for differences in the proportion of black and white pixels.

Many methods for normalizing the spectra were investigated. The goal of the normalization was to produce one spectrum per print that had an initial value of one and a final value of zero. Some of the methods to normalize the spectra required the use of the probability of a black and a white pixel being selected for a given scale. Equations II-2 through II-5 show how the probabilities are found and how the scale spectra are normalized for each scale σ.

Equation II-2. Equation for the probability of a black pixel at a given scale.

$$P_b(\sigma) = 2\alpha_{11}(\sigma) + \beta_{01}(\sigma) + \beta_{10}(\sigma)$$

Equation II-3. Equation for the probability of a white pixel at a given scale.

$$P_w(\sigma) = 2\alpha_{00}(\sigma) + \beta_{01}(\sigma) + \beta_{10}(\sigma)$$

Equation II-4. Equation to verify probabilities.

$$\frac{\alpha_{00}(\sigma) - \alpha_{11}(\sigma)}{P_w(\sigma) - P_b(\sigma)} = 1$$

Equation II-5. The first normalization equation.

$$\frac{P_b(\sigma)\alpha_{00}(\sigma) - P_w(\sigma)\alpha_{11}(\sigma)}{P_w(\sigma)P_b(\sigma)(P_w(\sigma) - P_b(\sigma))} = \frac{\beta(\sigma)}{P_w(\sigma)P_b(\sigma)}$$

This normalization worked well in theory but failed to produce the desired spectrum in practice. The spectrum starts at one but approaches a value slightly off zero (see Figure II-10). The most likely explanation is the error is magnified by the normalization and the spectrum will not go to exactly one.

When trying to show the theoretical origin of these equations, it was discovered that similar results could be obtained by using matrix properties. The trace and determinant of the 2x2 matrix shown below can be used as a normalization (see Figure II-11).

Equation II-6. Scaling parameters matrix and the equation for the determinant.

$$M = \begin{matrix} \alpha_{00} & \beta_{01} \\ \beta_{10} & \alpha_{11} \end{matrix}$$

$$\det(M) = \propto_{00} \propto_{11} - \beta_{01}\beta_{10}$$

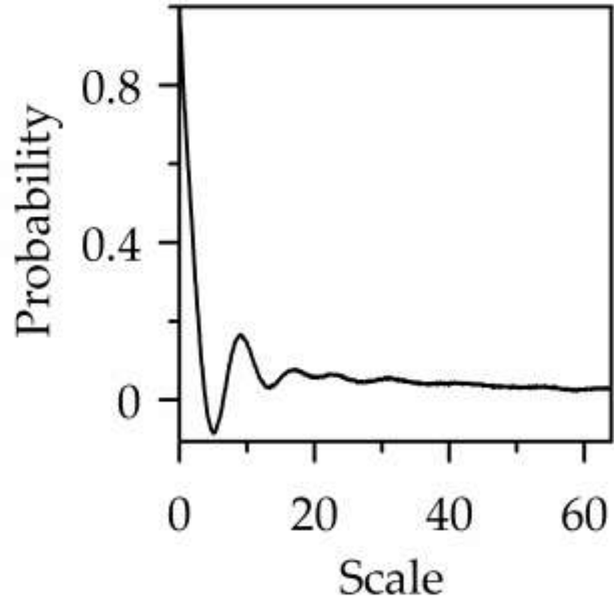


Figure II-10. Graph showing the normalized spectrum for a fingerprint (plot of equation II-5). The y-axis is the normalized probability which is the difference in two probabilities, this is why it goes negative.

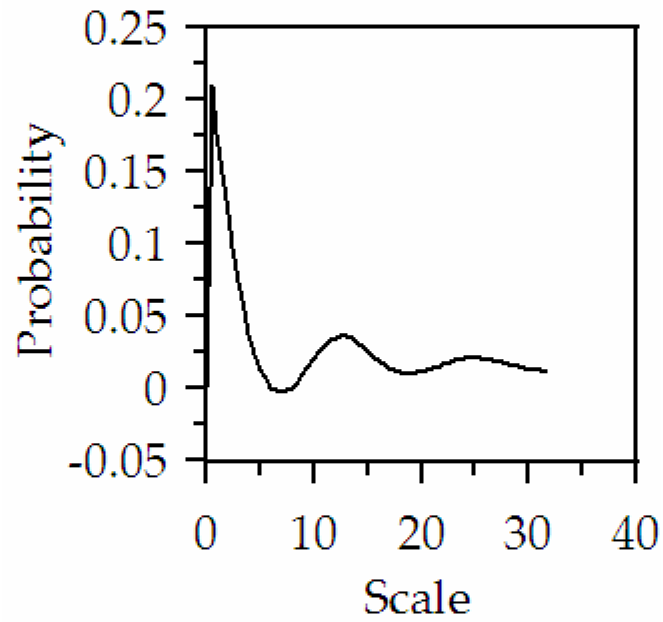


Figure II-11. Graph showing the determinant of the probability matrix M.

C. Image Preprocessing

The fingerprints that are readily available need to be preprocessed prior to using them in our analysis. Two principle factors must be taken into account. First, our algorithm requires that the images of the fingerprints be binary, that is, black and white. Often images of fingerprints are in a gray scale or in a color scale. If this is the case then they must be converted to a binary image. The second issue is that fingerprints invariably include background. This background must be removed. This is accomplished by *outlining* the fingerprint.

A related issue is the question of smudges and other "damages" in the image of the fingerprint. These regions also need to be removed. With regard to these regions, we have designed our outlining algorithm in such a manner that it not only outlines the fingerprint but also outlines the smudges and other “damage”. Figure II-13 shows a fingerprint with the pre-processing performed.

Figure II-12. Image showing a fingerprint that has been outlined with the masked pixels shown in green. The image has also been converted to binary.

Outlining

The outlining process involves creating a mask of the fingerprint images that includes the domains of uniformity to exclude from the random walk. This is accomplished by observing each pixel's surrounding pixels and masking those whose regions are near uniform (almost all black or all white). Many factors affect whether or not a pixel is masked. The number of surrounding pixels is an important factor in the decision to mask a pixel. The uniformity of the neighborhood as a boundary for masking is also a factor.

The number of surrounding pixels or the neighborhood size must be a value that allows the fingerprint to be kept and the background to be excluded. By looking at many fingerprints it has been observed that many of them have ridges and valleys 4-5 pixels wide, so a neighborhood of pixels 9x9 would ensure a non-uniform neighborhood. A boundary of within 5% of uniform was also determined by observation. Both of these values are easily adjustable for different collection methods or other factors that may affect the images used.

Determining whether a pixel should be masked is accomplished by checking the uniformity of the pixels neighborhood. This is accomplished by calculating the sum of the intensities of the NxN neighborhood of pixels. If this summation is within a certain percentage of all white or all black the pixel will be masked (see Equation II-7). In this equation $P_{i,j}$ is one of the pixels in the neighborhood. For this process to work the intensities used are either 1 if the grayscale value is greater than 128 or 0 if less than 128.

Equation II-7. Outlining threshold equation.

$$0.05(N^2) < \sum_{i=1}^{N}\sum_{j=1}^{N} P_{i,j} < N^2 - 0.05(N^2)$$

Performing the outlining calculation for the pixels near the edge presents some difficulty. The pixels on some sides of the boarder pixels are outside of the image and are not usable in the summation calculation. These pixels are often in the background of the image so the outlining calculation must be performed for them. To address this problem, the image is padded with pixels to allow the calculation of the sum. The padding pixels are just N/2 copies of the edge rows and columns (see Figure II-14).

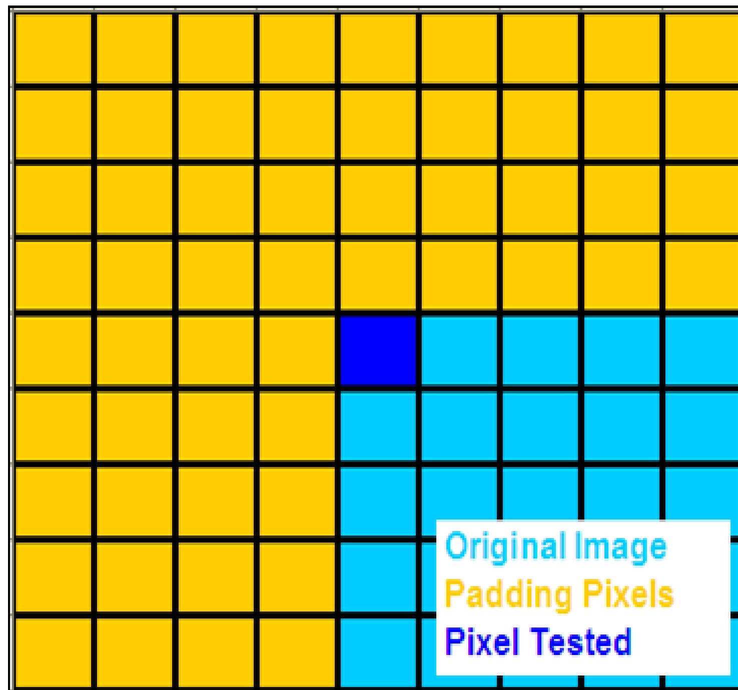


Figure II-13. Image showing the 9x9 neighborhood of pixels used to decide if the top left pixel of an image should be masked.

Old Method for Conversion to Binary

The current method to convert the grayscale image uses the average intensity to determine the binary value of a pixel. The first step is to sum the intensities of all of the pixels in the image and divide by the number of pixels in the image. Finally, each pixel in the image is compared to

the average intensity and if it is less than the average it is set to black, otherwise it is set to white (see Equation II-8).

Equation II-8. Old binary conversion decision equations. I_{cp} is the pixel in question and $I_{i,j}$ represents a pixel in the neighborhood.

$$If\ I_{cp} < \frac{\sum_{i=1}^{M}\sum_{j=1}^{N} I_{i,j}}{MN} \qquad Then\ \ Pixel = 0$$

$$If\ I_{cp} > \frac{\sum_{i=1}^{M}\sum_{j=1}^{N} I_{i,j}}{MN} \qquad Then\ \ Pixel = 1$$

New Method for Conversion to Binary

The new method for the conversion of the grayscale image is an adaptation of the NIST standard for fingerprint processing [31]. In this method the pixel is compared to an average of its neighboring pixels as opposed to the average of the entire image. This method allows proper binarization of images with light and dark regions. If the overall average is used dark regions would be set to almost all black and light regions would be set to all white regardless of the ridge features in these light and dark regions.

This method is also easily implemented during the outlining process. When the summation of the neighborhood is calculated in the outlining process it is divided by the number of pixel in the neighborhood to calculate the neighborhood average. If the central pixel is less than the average it is set to black, otherwise it is set to white (see Equation II-9).

Equation II-9. New binary conversion decision equations. I_{cp} is the pixel in question and $I_{i,j}$ represents a pixel in the neighborhood.

$$If\ I_{cp} < \frac{\sum_{i=1}^{N}\sum_{j=1}^{N} I_{i,j}}{NN} \qquad Then\ \ Pixel = 0$$

$$If\ I_{cp} > \frac{\sum_{i=1}^{N}\sum_{j=1}^{N} I_{i,j}}{NN} \qquad Then\ \ Pixel = 1$$

III. Program

Step 1 Preprocessing

A small C++ program was written in order to implement the fractal analysis of fingerprints. The implementation involves two principle tasks. These are, first, the prepossessing of the images, and second, the conversion of the digital image of the fingerprint into a scale spectra representative of the image using the iterated function sequence.

For the preprocessing step, the image needs to be converted into a form that can be used in the random walk step. The random walk requires binary values to give the desired four possible outcomes. All of the images currently in our database are grayscale. These grayscale images are converted to binary using the process described in the end of the previous chapter.

The background and smudged areas of the image also need to be addressed before the image is analyzed. This is accomplished through an outlining process that is also described in the end of the previous chapter. The masked binary image is then used as input to the fractal creation step. The implementation details of both preprocessing steps are covered by the source code for the outlining class in Appendix B.

Step 2 Fractal Creation

The next step in the process is to use the preprocessed image as input to the random walk to produce a scale spectrum. For each scale value specified in the options, the program performs the random walk incrementing the appropriate scale parameter counter at each step. This is just

an implementation of the random walk process described in the previous chapter without the fractal creation. The fractal is not created during that random walk because it is easier and faster to calculate the scale parameters directly. If the graphical mode of the program is run, the fractal is then calculated from these stored scale parameters.

. The program was designed to store the scale spectrum in a table in a text document for further analysis. Appendix A contains a user's manual for the program. Appendix B contains the source code for the key files in the program. The files that are not included deal with typical code for a Windows API and were adapted from code available online [28].   Appendix C contains descriptions of the variables and methods by class. The figures below contain some basic diagrams and flowcharts describing the program.

Diagrams

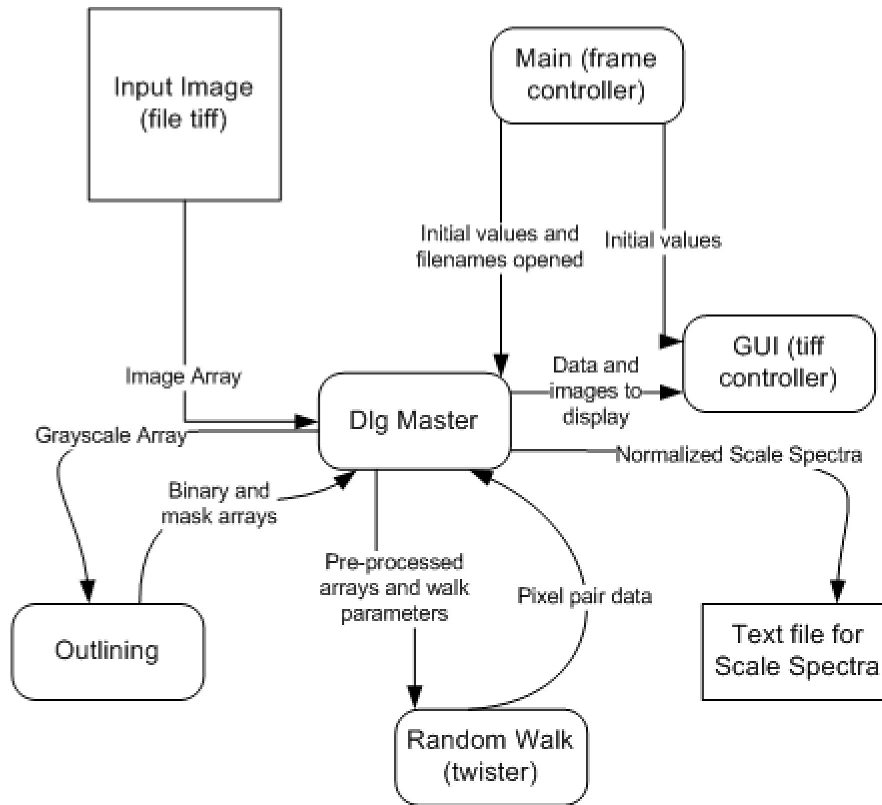


Figure III-1. Data flow diagram for program. The diagram shows how and where the data described in the algorithm chapter is exchanged by the implemented classes.

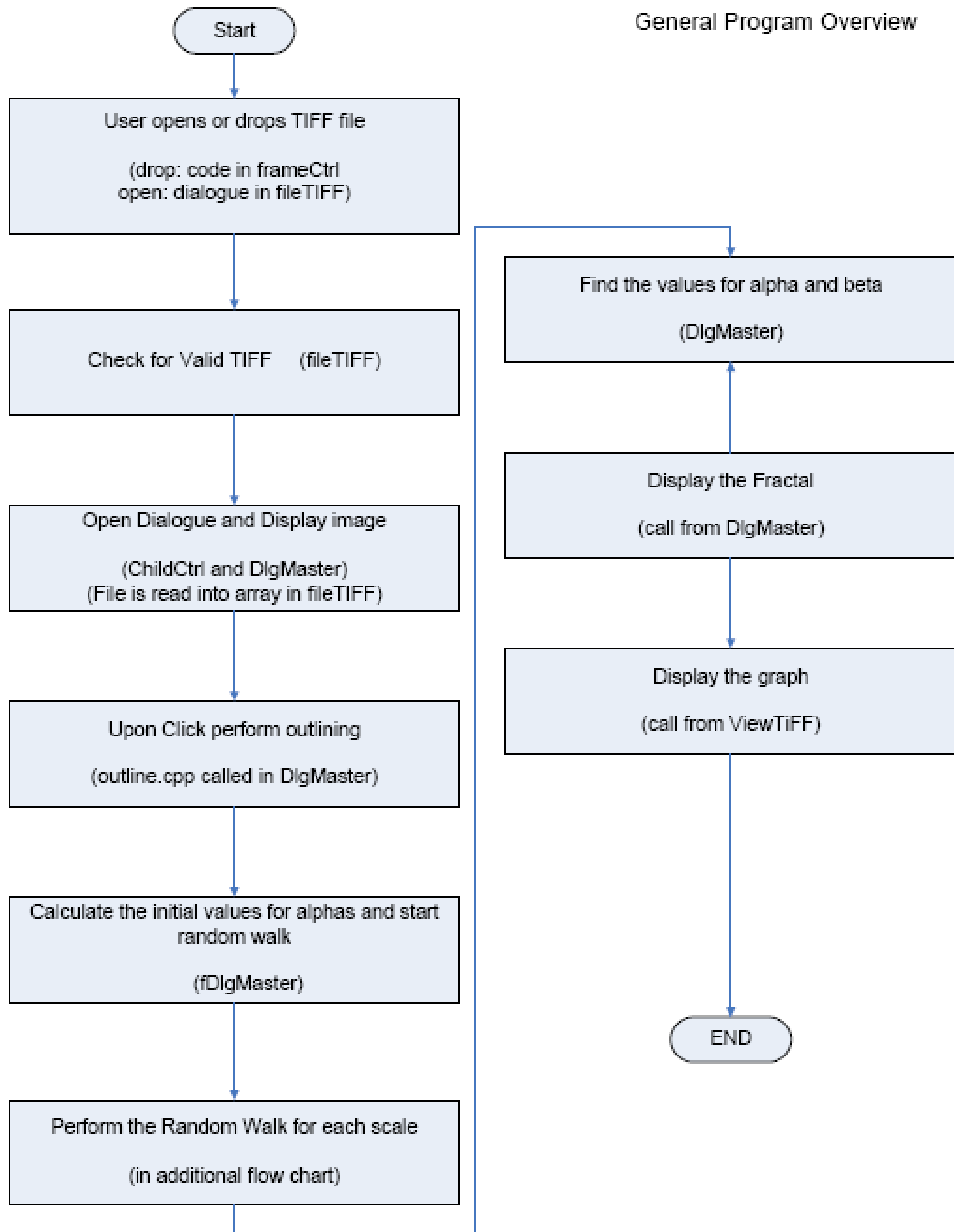


Figure III-2. General flowchart for the whole program. The flowchart shows the steps taken and classes used to analyze a typical file from opening to close.

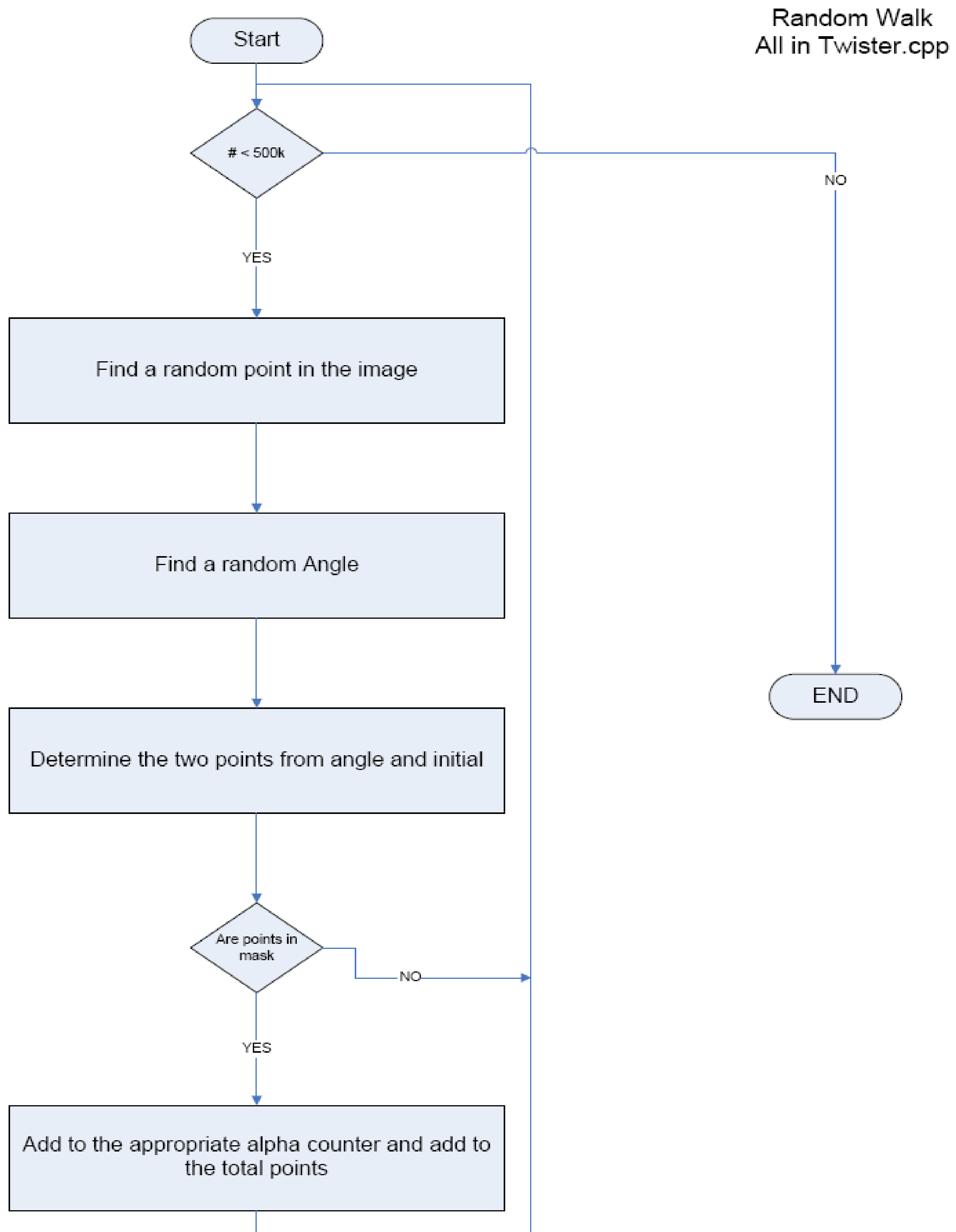


Figure III-3. Detailed flowchart for the random walk. This chart details the random walk step in the general flowchart Figure III-2

IV. Analysis

After attaining the scale spectra from the fingerprints, a method to compare the spectra and, in turn, the fingerprints they were created from was needed. There are many ways to compare two graphs. Just a few procedures are presented in this chapter as ways to compare the graphs. The first procedure developed was the direct comparison, where the difference in the two scale spectra is evaluated. The second procedure treated the spectra as an N dimensional vector and evaluated the distance for comparison. The third procedure uses the Fourier Transform of the spectra for comparison. The last procedure presented here fitted polynomials to the scale spectra for comparison of their coefficients.

The database used for the analysis consists of uncompressed grayscale TIFF images 640x640 pixels or less [3]. This database includes 7,040 fingerprints and 8 different captures of each print. This database is actually comprised of two databases from the Fingerprint Verification Competition 2000 and 2002. Details concerning these databases are described by the sponsors below.

> "Four different databases (hereinafter DB1, DB2, DB3 and DB4) were collected by using the following sensors/technologies:
>
> DB1: low-cost optical sensor "Secure Desktop Scanner" by KeyTronic*
>
> DB2: low-cost optical capacitive sensor "TouchChip" by ST Microelectronics*
>
> DB3: optical sensor "DF-90" by Identicator Technology*
>
> DB4: synthetic generation based on an evolution of the method proposed in [10].
>
> Each database is 110 fingers wide (**w**) and 8 impressions per finger deep (**d**) (880 fingerprints in all); fingers from 101 to 110 (set B) have been made available to the participants to allow parameter tuning before the submission of the algorithms; the benchmark is then constituted by fingers numbered from 1 to 100 (set A) [3]."

Table IV-1 summarizes the common features and Figure IV-1 shows a sample print.

Table IV-1. The four FVC 2000 databases taken from Handbook [3]

	Sensor Type	Image Size	Set A (w×d)	Set B (w×d)	Resolution
DB1	Low-cost Optical	300×300	100×8	10×8	500 dpi
DB2	Low-cost Capacitive	256×364	100×8	10×8	500 dpi
DB3	Optical	448×478	100×8	10×8	500 dpi
DB4	Synthetic Generator	240×320	100×8	10×8	500 dpi

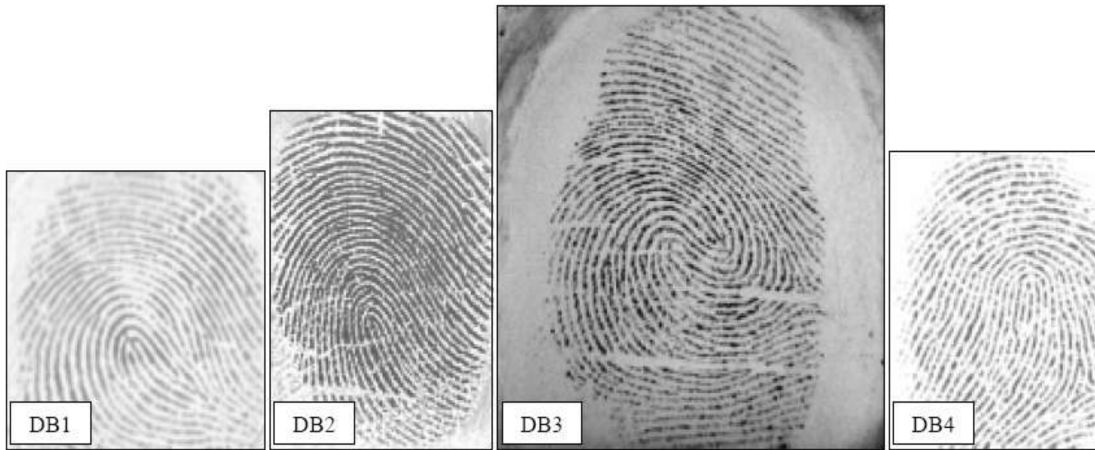


Figure IV-1.Samples of the four databases.[3]

A. **Direct Comparison**

To make decisions about fingerprint matches, the scale spectra must be compared. If the scale spectra from two fingerprint images are nearly the same, the two images came from the same fingerprint. Plotting the normalized scale spectra from two fingerprints will visually show the differences in the spectra. Figure IV-2 shows a plot of the normalized scale spectra from two fingerprints. These spectra are from the fingerprints shown in Figures II-8 and are clearly different.

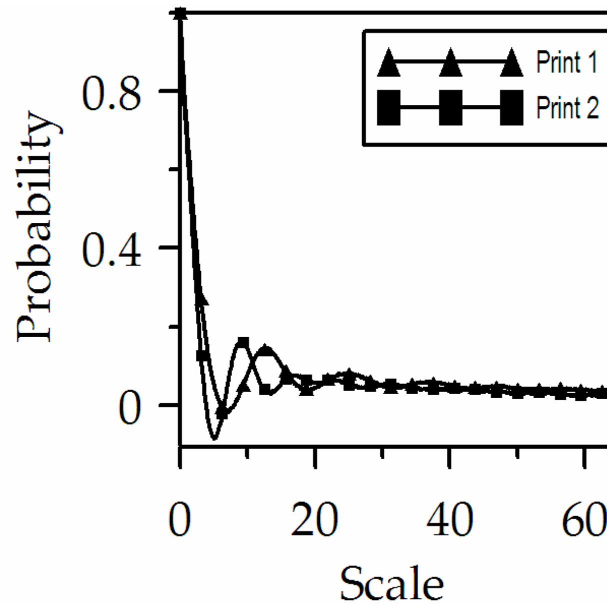


Figure IV-2. Plot the scale spectra from two prints.

One of the simplest ways to compare the scale spectra is to find the difference between the two spectra to be compared. If the difference between the two spectra is statistically small enough, the two spectra are considered to be the same.

Error Bars

To allow for the differences in random walks over the same print the error in the process has to be considered when making comparisons of scale spectra. The error in the scale spectra is

taken to be $N^{1/2}$ where N is the number of times a probability bin has been visited [32]. Using this method, the length of the random walk will control the uncertainty of a fingerprints scale spectra (See Figure IV-3).

The amount of time required to perform a random walk over the fingerprint is dependent on the length of the random walk performed. Since the time required to make a decision when comparing fingerprints is of concern, the length of the random walk is very important (See Figure IV-4). Due to the relationship between time and walk length and the relationship between walk length and uncertainty, the time to make the random walk is related to the desired uncertainty (See Figure IV-5).

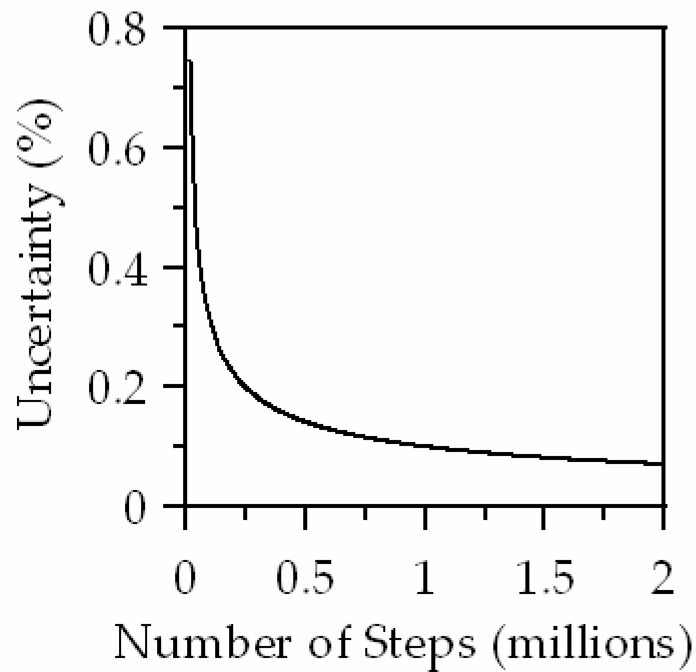


Figure IV-3. Graph showing uncertainty as a function of random walk length.

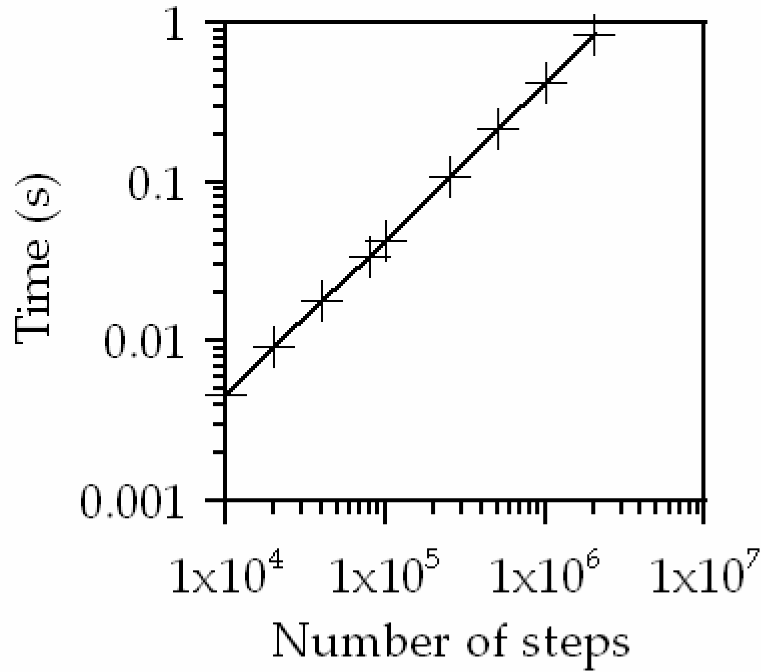


Figure IV-4. Graph showing the time necessary to perform a random walk based on the length of the random walk. The data for this graph was obtained using the described program on a 1.7GHz IBM Laptop.

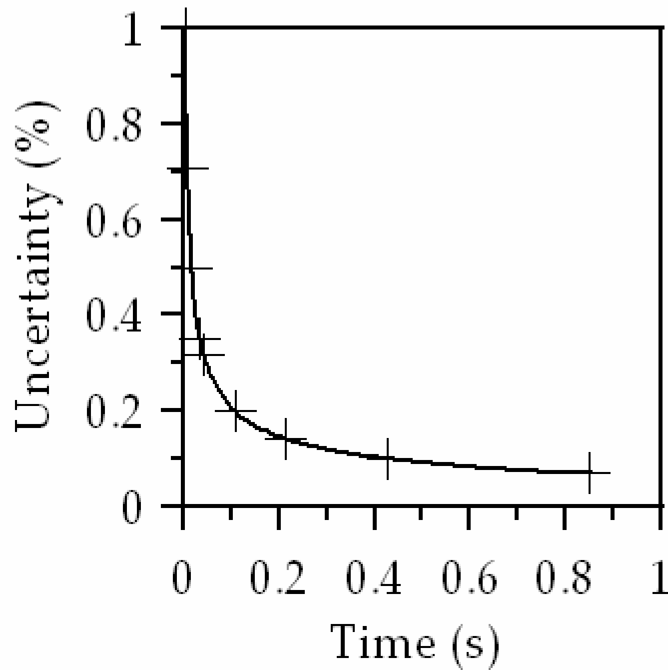


Figure IV-5. Graph showing uncertainty as a function of time to execute. Notice how that past about 0.2 seconds the gains in uncertainty are minimal due to the law of diminishing returns. So the optimum value of uncertainty is about 0.15% taking 0.2 seconds to obtain.

The above graphs illustrate that a relatively low uncertainty of about 0.15% can be obtained using a random walk that only takes 0.2 seconds to perform on a modest machine. The time cost to get a significantly smaller uncertainty is very large and likely unacceptable. This value for uncertainty corresponds to a random walk length of 500k steps, which is our default value in the program.

Using this error estimation, error bars can be added to the scale spectra graphs. If two scale spectra are within the error region for each other, they are most likely from the same fingerprint. Plotting the difference between the two spectra with an error line clearly shows whether the difference for all scales is within the error for the random walk. Figure IV-6 shows the described difference plot for two different random walks over the same image.

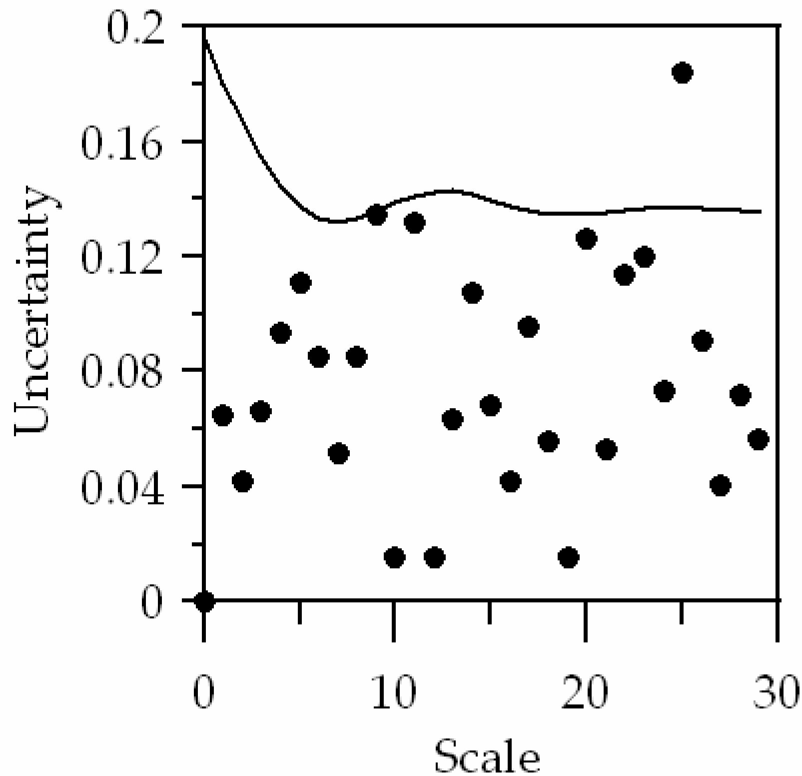


Figure IV-6. The graph shows the difference in two runs over the same image. Notice that, for all but one scale the difference is below the expected percentage uncertainty.

Orientation Independence

One of the key benefits of the developed process is its independence on the orientation of the fingerprint in the image. Many current automated fingerprint comparison algorithms depend on the fingerprint being no more than 15° from normal. This is because their minutiae match optimization is only done over a limited angle typically 15˚ [3]. This either requires the algorithm to have a complex algorithm to rotate the captured print to normal or for the device to request another acquisition of the fingerprint.

The use of a random angle to find the two points for comparison in the fingerprint allows the developed process to be independent of orientation. Figure IV-7 illustrates this independence by showing the difference in two runs of a print in which on run was based on a 90° rotation of the image.

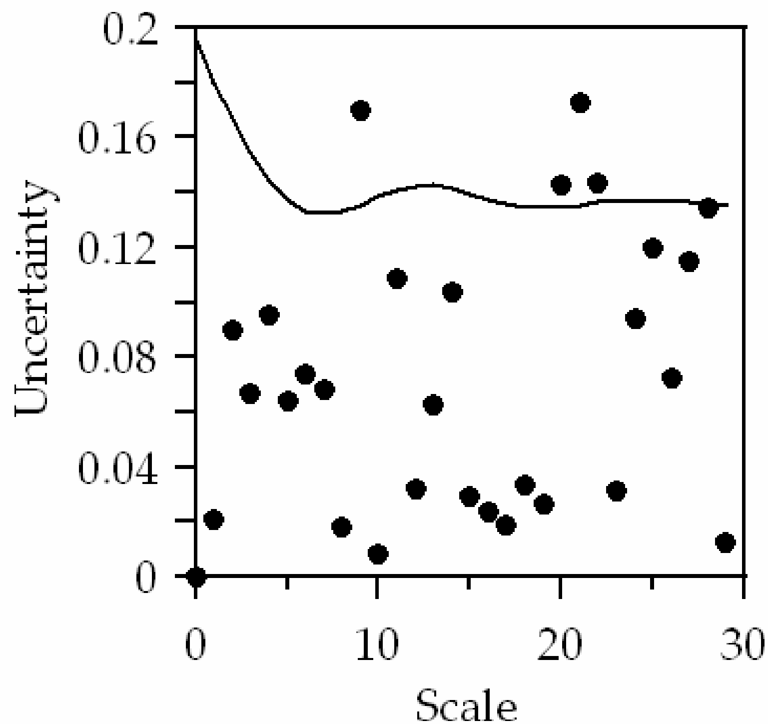


Figure IV-7. The above plot shows the difference in scale spectra from a print and its rotated version. Notice how most of the differences are below the expected percentage uncertainty.

This orientation independence is due to the random angle selection. For an example take a print rotated by φ^o from norm. When the random angle θ is selected in the random walk, there is really no difference in the randomness of the angle if it is instead $\theta+\varphi$. If a constant value is added to all of the random angles in the random walk process, they are all still random and the scale spectrum is unaffected.

Smudge Removal

Fingerprint images often have areas in the print that are smudged. These are areas in which the ridge details are not clear either because of collection or damage to the image in other ways. The developed algorithm uses the entire print to produce the scale spectrum. If the smudges of the print are included, it will cause the scale spectrum to be based in part on the smudge instead of just the actual ridges in the fingerprint. Many of the smudged prints in our database have small smudges less than 5% of the print in size.

To remove the smudged area from the fingerprint, an outlining process is used that has been described earlier in this document. The outlining process masks the areas of the fingerprint that are smudged or part of the background behind the fingerprint. If any of the points to be compared are in this masked area when the random walk is performed, they are thrown out and another random set are picked.

To illustrate the removal of a damaged area of the fingerprint, a small portion of a good fingerprint is erased in a photo editing program. Figure IV-8 shows a fingerprint image with the background outlined in green. Figure IV-9 shows the same print with a portion blacked out and it's outlined version. Figure IV-10 shows a plot of the difference in the scale spectra for the

fingerprint and the fingerprint with a 2% portion removed as shown in Figure IV-9. The blacked out area was placed in the left central portion of the fingerprint because by observation this area seems to be the most likely to be smudged in our database. Figures IV-11 shows the same print with a 5% portion removed with the smudge and background in green. Figure IV-12 shows a plot of the difference in scale spectra between the Figure IV-11 print and the original. Figure IV-13 shows the same print with a 10% portion removed with the smudge and background in green. Figure IV-14 shows the plot of the difference in scale spectra for the Figure IV-13 print and the original.

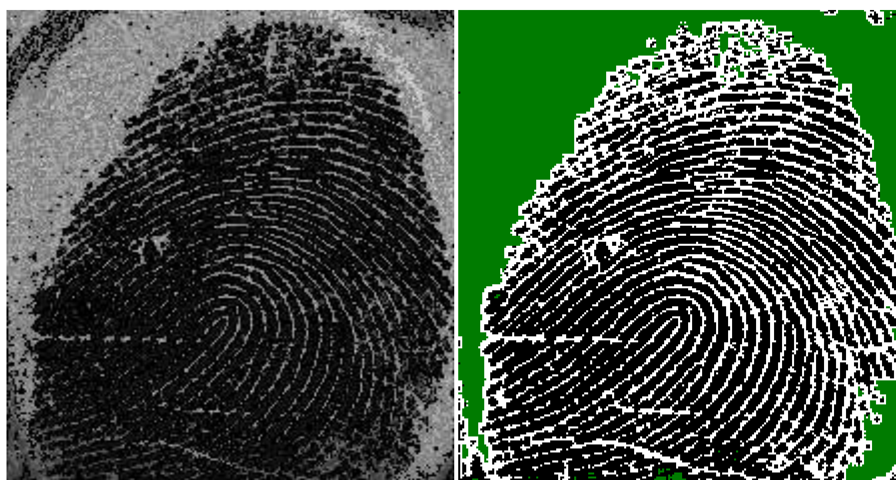


Figure IV-8. Fingerprint image and its outlined version with masked pixels shown in green.

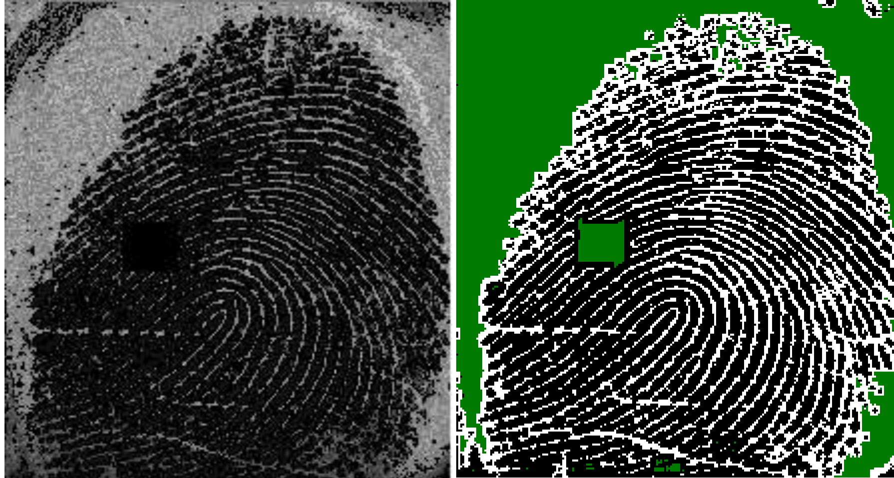


Figure IV-9. Image of a fingerprint with a small portion blacked out and its outlined version with masked pixels displayed in green.

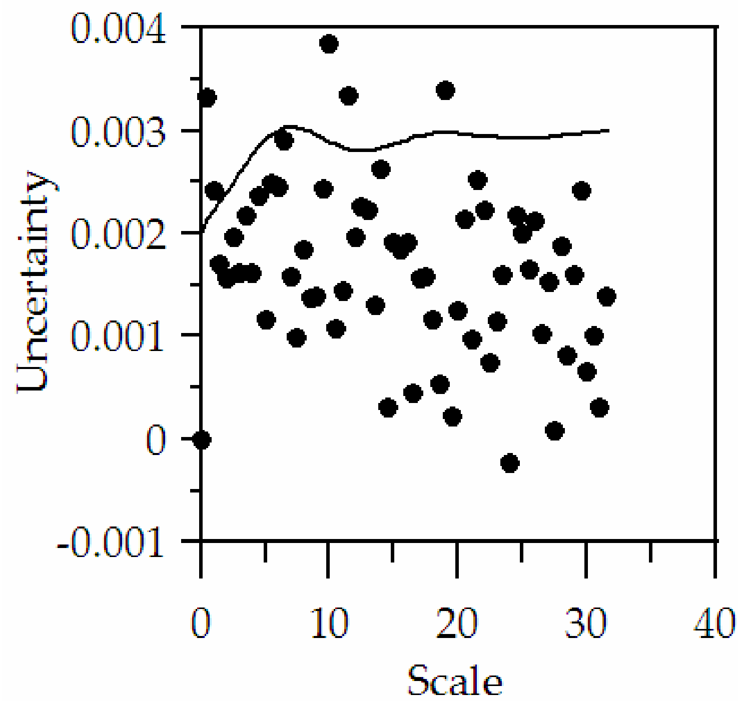


Figure IV-10. This is a plot showing the difference in the scale spectra for the fingerprint and its smudged version. Notice that most of the differences are still below the expected uncertainty.

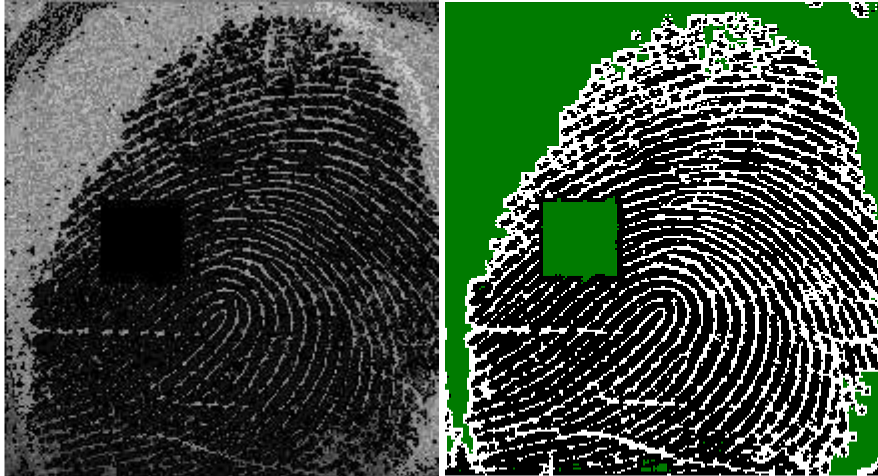


Figure IV-11. The images above show the original print with 5% smudged and the outlined version of this print. In the outlined version everything in green is masked from use in the random walk.

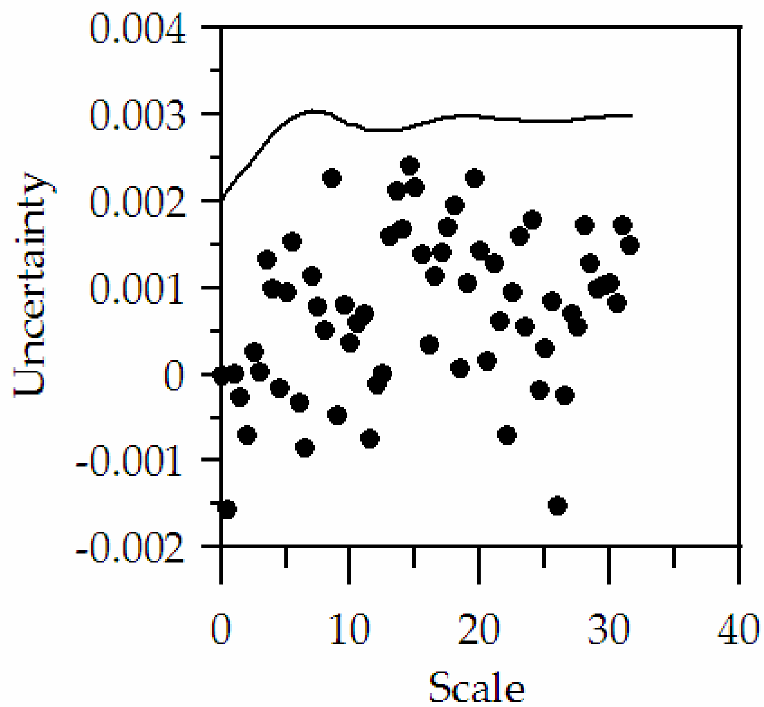


Figure IV-12. This is a plot showing the difference in the scale spectra for a fingerprint and its version with 5% smudged. Notice that in this case all of the differences are below the expected value.

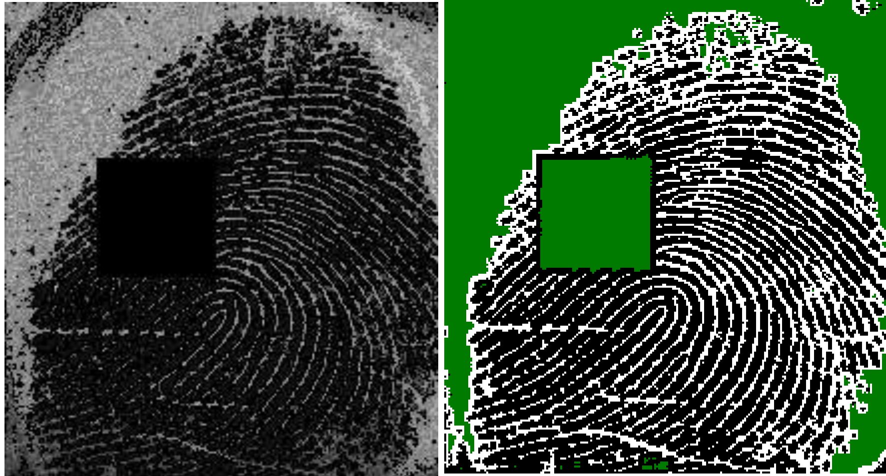


Figure IV-13.The above images show the original print with 10% smudged and it's outlined version.

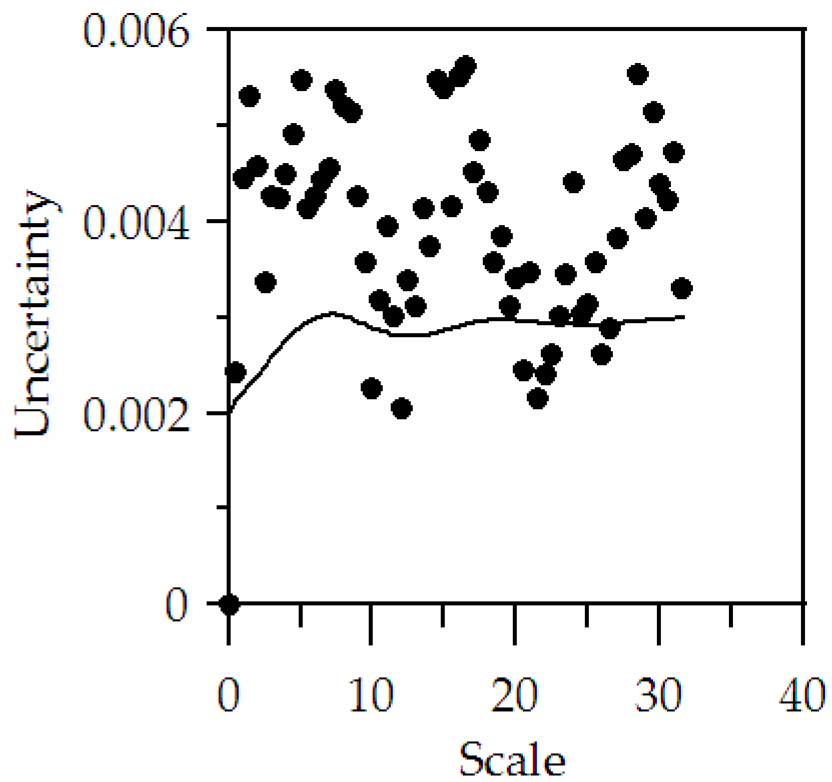


Figure IV-14. This plot shows the differences in scale spectra for a print and its smudged version with 10% removed. Notice that with this much removed a large portion of the differences are above the expected uncertainty.

From these tests, it appears that 10% is the approximate maximum for the amount of the print that can be missing without affecting the scale spectra noticeably. Even though this is the area observed to be the most likely to be smudged, some other tests of moving the smudge around the image were designed. Figure IV-15 shows the original print with 5% smudged in a different area than before. Notice that in the plot of the difference in Figure IV-16, the differences are much higher than in the first example of 5% removed Figure IV-12. Figure IV-17 shows the original print with 5 areas of 2% each removed for a total of 10% removed. Notice that in the plot of the difference Figure IV-18, the differences are much lower than those from the first example of 10% being smudged Figure IV-14.

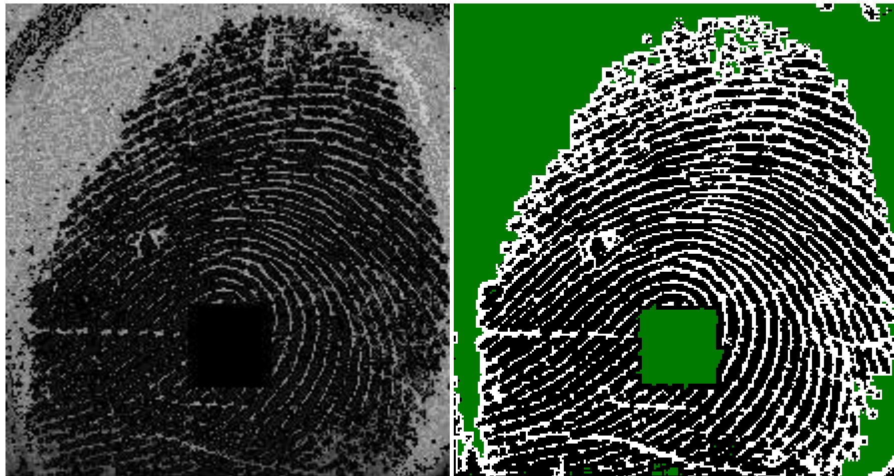


Figure IV-15. The above images show the original print with 5% smudged in a new region and it's outlined version.

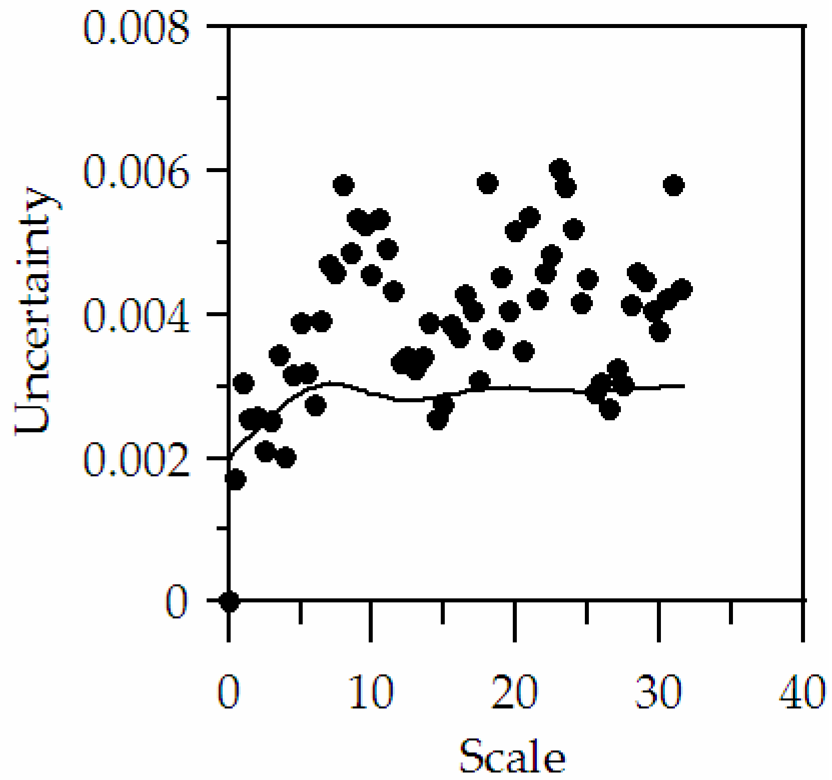


Figure IV-16. This plot shows the differences in scale spectra for a print and its smudged version with 5% removed from a new region. Notice that most of the differences are above the expected uncertainty unlike in the first print with 5% smudged.

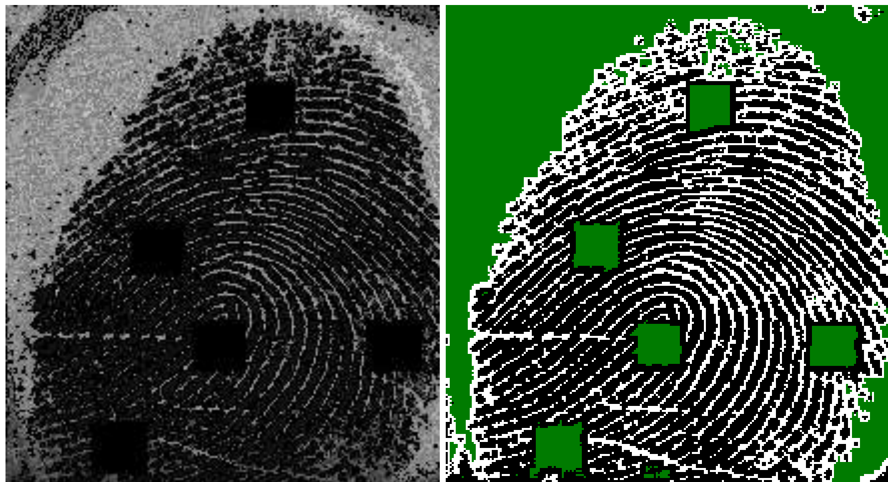


Figure IV-17. The above images show the original print with 2% smudged in a new 5 different regions for a total of 10%.

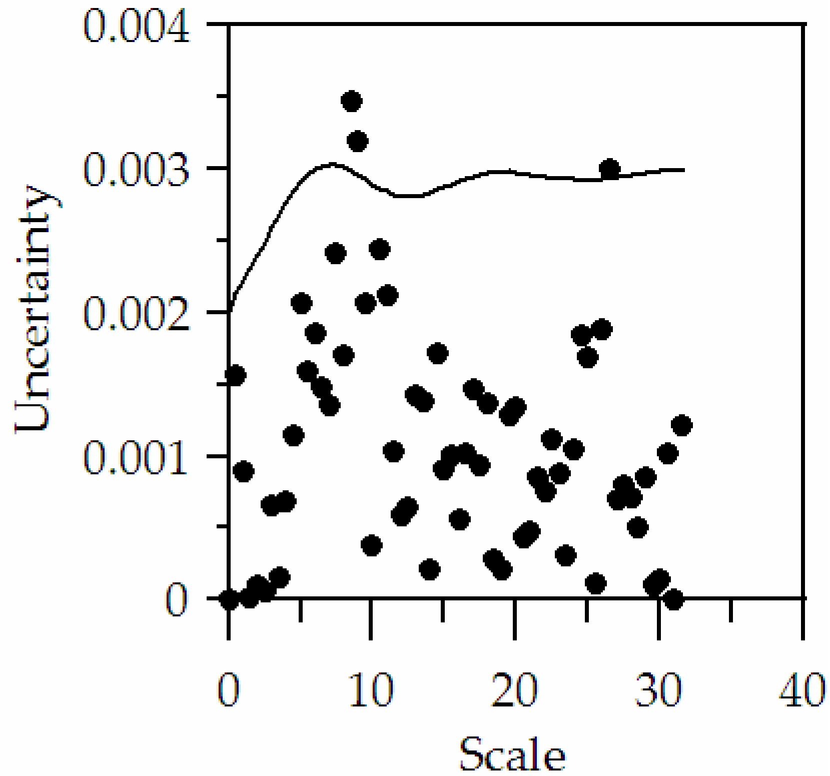


Figure IV-18. This plot shows the differences in scale spectra for a print and its smudged version with 2% removed from 5 different regions. Notice that most of the differences are below the expected uncertainty unlike in the first print with 10% smudged.

From the comparisons of different smudged prints it is clear that smudging effect on the scale spectrum is dependent on both the size of the smudge and the position on the fingerprint image. In general a single smudged portion 10% or larger appears to have a noticeable effect on the scale spectrum. Also it appears that smudges in the center of the fingerprint have a greater effect than ones on the outside of the image. Finally it appears that if there are many small smudges they have less effect than one large one equal to their total area.

Now that it has been shown that direct comparison can identify the same print even with some abnormalities we should at least show how the difference and uncertainty plot looks for two different prints. This is shown in Figure IV-19 for the two prints shown in Figure II-8.

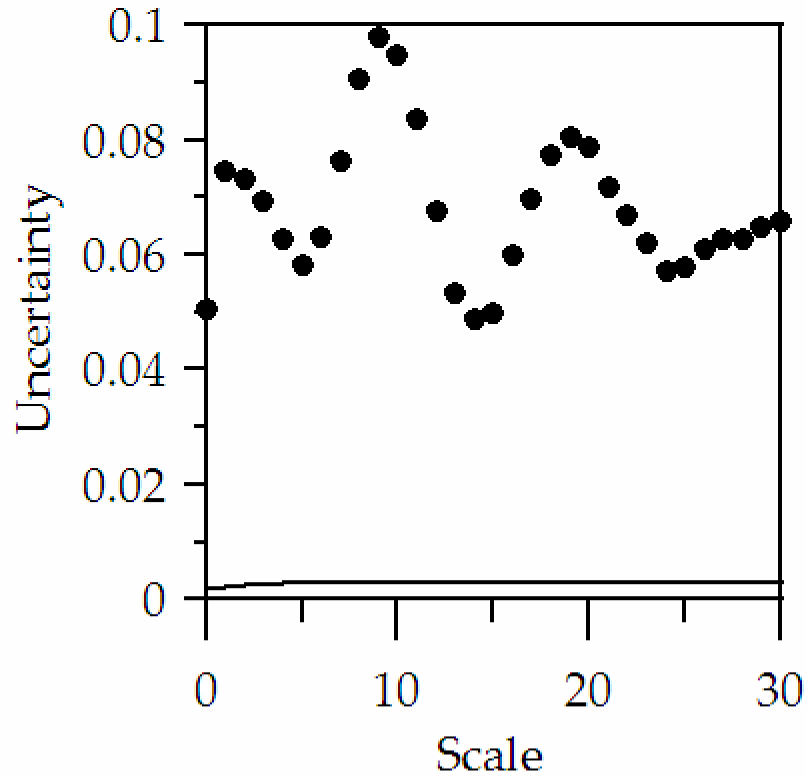


Figure IV-19. Plot showing the difference in two different prints. Notice the line clear at the bottom is the uncertainty line that almost all of the points were below for the same print.

B. Fourier Transform

When looking at the scale spectrum for a fingerprint, it is clearly a periodic damped function. Upon initial inspection, it appears to have the shape of an exponentially decaying sinusoid. To help extract the periodic information of the scale spectrum, a Discrete Fourier Transform (DFT) was used. The DFT should give us the frequencies of the periodic portions of the scale spectrum. In our case, the frequency is really number of pixels because the independent variable is scale instead of time. To obtain the Fourier Transform Coefficients from the software program the Intel Signal Processing package was used.

Initially a Discrete Cosine Transform (DCT) was used because it has no complex coefficients. The all-real coefficients were thought to be easier to compare and understand. The DCT failed to work properly due to the sharp decay from the origin. The DFT was needed due to this feature of the scale spectrum.

From examination of the scale spectra for most of the fingerprints in our database, we found the scale spectrum stays constant after a scale of approximately 30 for the prints in our database. Using this fact, a maximum scale of 32 works well and would allow the use of the Fast Fourier Transform (FFT). The coefficients of the FFT are the same as those from the DFT for series with 2^n points in them. For experimentation, a max scale of 64 was used to be sure important data was not excluded, but in practice a max scale of 32 could be used and the FFT efficiency can be exploited.

The FFT of the scale spectra can be compared in the same fashion as the spectra themselves because they are still just graphs. The advantage to taking the FFT is the information on pixel features such as the ridge spacing from the fingerprint. The peak in the FFT plot corresponds to the ridge spacing in the fingerprint. Figure IV-20 shows the FFT from the normalized scale spectra for the two prints shown in Figure II-9.

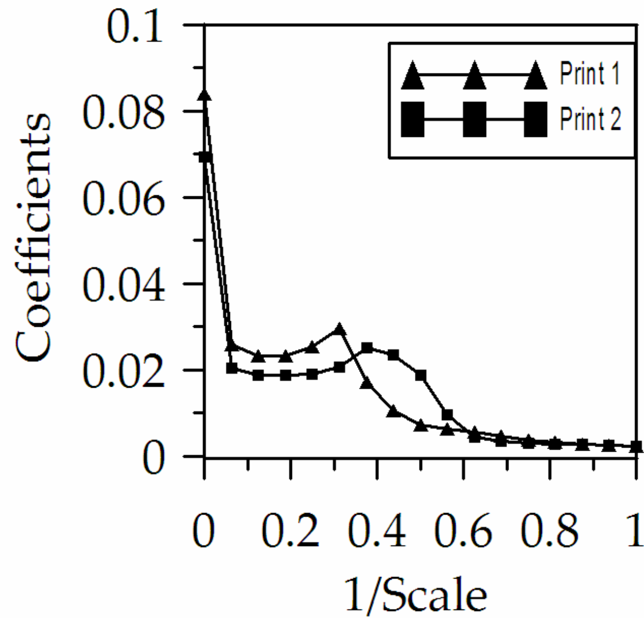


Figure IV-20. Plot showing the FFT for two different fingerprints.

C. Other Comparison Methods

Since the scale spectra to be compared are just graphs representing the fingerprint there are many more ways to compare them. Any reasonable method of comparing graphs of functions can be used to compare the scale spectra and in turn the fingerprint images that produced them.

One other method that was used was to treat the probabilities for each scale as the magnitude for a vector in R dimensional space. Where the R is the number of scales the random walk was performed over. Using this idea, the distance between the endpoints of the vectors can be found using the distance formula shown below. If the distance is small enough the fingerprints are likely the same and if they are further apart they are likely from different fingers.

Equation IV-1. Multidimensional vector distance formula. $P_{i,j}$ is the probabilities from two prints, where I is the print number and j is the scale with a max scale of R.

$$d = \sqrt{\left(P_{1,1} - P_{2,1}\right)^2 + \left(P_{1,2} - P_{2,2}\right)^2 + \cdots \left(P_{1,R} - P_{2,R}\right)^2}$$

Another method that was investigated was fitting a 5th order polynomial to the scale spectrum and using the coefficients for comparison. Figure IV-21 shows the scale spectrum for the first fingerprint shown in Figure II-8 and the 5th order polynomials that were fitted to the plots. Table IV-2 shows the polynomial coefficients from the two fingerprints in Figure II-8 along with the difference in each of the coefficients for each of the probabilities in the scale spectrum.

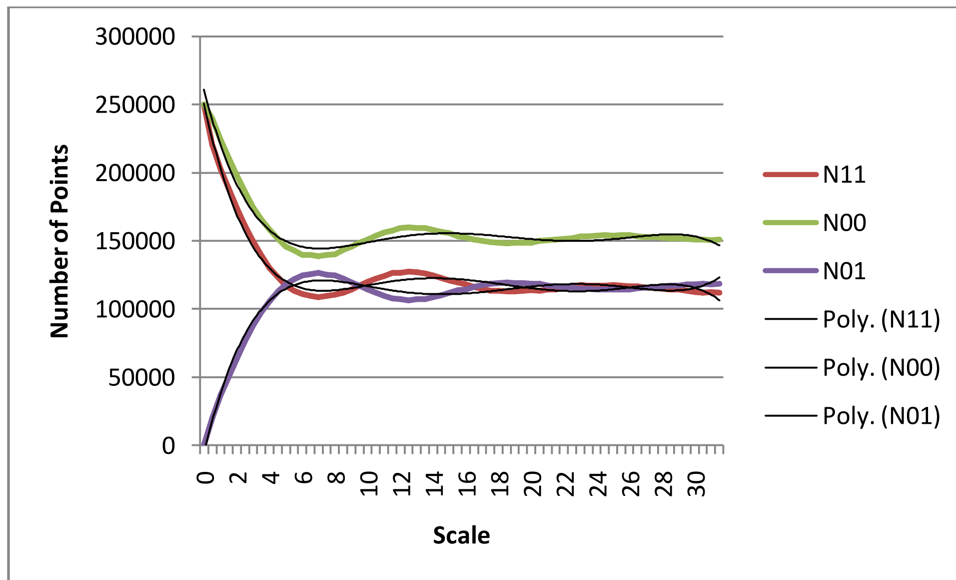


Figure IV-21. Plot showing the scale spectrum and trend lines. The trend lines are 5th order polynomials.

Table IV-2. Table showing the coefficients of the fitted polynomials. The difference in the coefficients of the polynomials for the two prints is shown in the difference row. The distance column shows the distance using the distance formula and the differences from the difference row.

N11							
Print	x^5	x^4	x^3	x^2	X	C	Distance
1	-0.004	0.797	-56.26	1830	-26721	28596	
2	-0.005	0.981	-67.51	2119	-29669	28773	
Difference	0.001	-0.184	11.25	-289	2948	-177	2967.437
N00							
Print	x^5	x^4	x^3	x^2	X	C	Distance
1	-0.005	0.95	-66.44	2140	-31203	27968	
2	-0.006	1.158	-78.9	2439	-32998	27087	
Difference	0.001	-0.208	12.46	-299	1795	881	2021.817
N01							
Print	x^5	x^4	x^3	x^2	X	C	Distance
1	0.004	-0.866	60.94	-1976	28883	-32676	
2	0.005	-1.069	73.13	-2277	31323	-29336	
Difference	-0.001	0.203	-12.19	301	-2440	-3340	4147.282

After presenting all of the above methods for the comparison of the scale spectra It can be concluded that the direct comparison or difference method offers the best solution. The distance method offers only one final value for comparison. The FFT and polynomial fit method require an additional complex calculation before comparison. The difference method is the computationally simplest method that offers multiple points of comparison between two prints.

V. Conclusions

The examples presented in this thesis show the method produces the desired results. The method uses a random walk over the whole fingerprint to produce a fractal image that is representative of the fingerprint it was created from. Further the fractal can be represented by the probabilities of each pixel pair combination. The method captures information from the print over many scales by setting the distance between the points to be compared. The probabilities can be plotted as a function of scale to produce a graph that represents the fingerprint from which it was created.

This process addresses many of the difficulties with current methods. It offers a creative solution to many of the difficulties in fingerprint analysis. Since the method is performed over the whole print, it uses all of the information available over all scales of the print to produces the scale spectra. Unlike many other methods, it can account for small scale (sweat pores) to large scale (ridge classification) features of the print.

As presented in the problem statement, this method is also fast. The method can produce the scale spectra for an entire print in less than 15 seconds. This is impressive considering it was performed on an ultra-portable laptop using an un-optimized program. Though the creation of the scale spectra is quick, the largest speed gains from this method will come in the comparison to a database. Most current methods require the alignment and comparison of the actual print images between the sample and every print in the database. The scale spectra for all of the prints in a large database could be obtained in idle times before a comparison is requested. Using the values for the spectra, a sorted list of values can be created. Then when a comparison is requested, the scale spectra is calculated and then the closest value can be found in the database through an efficient search of a few values. Searching a database for a certain pairing of about a

dozen values would be far quicker than aligning and comparing two images for each print in the database.

The quantitative measure that was proposed in the introduction has not been obtained yet. The method still has the potential to produce a quantities measure of a match, though the research just has yet not been developed that far in the comparison process. Once a comparison method is settled on mathematical analysis of the error in the random walk based on the resolution of the print could be used as the basis for a measure.

Goals of Research

The goals of the research were to produce a method that was not affected by many of the typical difficulties for automated matching. Referring to the list of difficulties presented in the introduction each of the problems are addressed by portions of the algorithm designed.

Displacement, rotation, and partial overlap, are addressed by the fractal being created from a random walk. Since the pixels are selected randomly selected the fractal is unaffected by prints being in different positions in the image. The random angle allows for any rotation of the print without an effect on the fractal (See the Analysis section).

Pressure, skin condition, and non-linear distortion are handled by the normalization of the scale spectra as described in the Experimentation section. These factors will effect the ridge width and spacing. The normalization process makes the scale spectra independent of the portion of black and white pixels (ridge width).

The noise problems are handled by the outlining algorithm which removes smudged or damaged areas of the prints along with the background. The method is not prone to feature extraction errors because features of the print are not used and therefore not extracted.

Further Research and Future Applications

This method is very open to expansion and improvement. Further work on the comparison method and the quantitative measure is needed. Once that comparison and measure are settled, the method should have far more extensive testing on large databases to determine its performance and limitations.

More work is also currently being performed on understanding how different print features affect the scale spectra. Test images have been ran through the program to determine how factors such as ridge spacing show up in the scale spectra. This work will allow the method to give information such as the ridge width and spacing using the scale spectra obtained.

One of the most interesting areas for this method is the application in new areas. With some adaptations to the random walk and scale spectra creation process, it can be applied to many other images. It can be used for other biometric features, including palm prints, faces, iris, speech and handwriting. It could also have other forensic applications such as ballistics and fracture comparison.

One other area for this method is detecting hidden features such as camouflage. Hidden features in an image or signal are usually on a different scale from their surroundings. Producing the scale spectra for portions of an image independently will allow the observations of these

differences in scale. The scale spectra of the background portions will be similar while the spectra for the abnormality will be different.

Bibliography

1. Lee H.C. and Gaensslen R.E., *Advances in Fingerprint Technology*, 2nd edition, Elsevier, New York, 2001.
2. Galton F., *Finger Prints*, McMillian, London, 1892.
3. Maltoni D., Maio D., Jain A., and Prabhakar P., *Handbook of Fingerprint Recognition*, Springer, New York, 2003.
4. Henry E., *Classification and Uses of Finger Prints*, Routledge, London, 1900.
5. Crouzil A., Massip-Pailhes L., and Castan S., “A New Correlation Criterion Based on Gradient Fields Similarity,” in *Proc. Int. Conf on Pattern Recognition* (13th), pp. 632-636, 1996.
6. Bazen A.M., Verwaaijen G.T.B., Gerez S.H., Veelenturf I.P.J., and van der Zwang B.J., “A Correlation-Based Fingerprint Verification System,” in *Proc. Workshop on Circuits Systems and Signal Processing* (ProRISC 2000), pp. 205-213, 2000.
7. ANSI, “Fingerprint Identification – Data Format for Information Interchange,” American International Standards Institute, New York, 1986.
8. Wegstein J.H., “An Automated Fingerprint Identification System,” U.S. Government Publication, Washington, DC: U.S. Dept. of Commerce, National Bureau of Standards, 1982.
9. Grasselli A., “On the Automatic Classification of Fingerprints,” in *Methodologies of Pattern Recognition*, S. Watanabe (Ed.), Academic, New York, 1969.
10. Kass M. and Witkin A., “Analyzing Oriented Patterns,” *Computer Vision, Graphics, and Image Processing*, vol. 37, no. 3, pp. 362-385, 1987.
11. Weisstein E.W., “Poincaré-Hopf Index Theorem”, http://mathworld.wolfram.com/Poincare-HopfIndexTheorem.html
12. Kawagoe M. and Tojo A., “Fingerprint Pattern Classification,” *Pattern Recognition*, vol. 17, pp. 295-303, 1984.
13. Maio D. and Maltoni D., ‘Direct Gray-Scale Minutiae Detection in Fingerprints,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 1, 1997.
14. Arcelli C. and Baja G.S.D., “ A Width Independent Fast Thinning Algorithm,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 4, no. 7, pp. 463-474, 1984.
15. Xiao Q. and Raafat H., “Fingerprint Image Post-Processing: A Combined Statistical and Structural Approach,” *Pattern Recognition*, vol. 24, no. 10, pp. 985-992, 1991.
16. Umeyama S., “Least-Square Estimation of Transformation Parameters Between Two Point Patterns,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 4, pp. 376-380, 1991.
17. Rosenfeld A. and Kak A., *Digital Picture Processing*, Academic, New York, 1976.
18. Sprinzak J. and Werman M., “Affine Point Matching,” *Pattern Recognition Letters*, vol. 15, pp. 337-339, 1994.
19. Baird H., *Model Based Image Matching Using Location*, MIT Press, Cambridge, MA, 1984.

20. Starnik J.P.P. and Backer E., "Finding Point Correspondence Using Simulated Annealing," *Pattern Recognition*, vol. 28, no. 2, pp. 231-240, 1995.
21. Stockman G., Kopstein S., and Benett S., "Matching Images to Models for Registration and Object Detection via Clustering," *IEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 4, no. 3, pp. 229-241, 1982.
22. Kaymaz E. and Mitra S., "Analysis and Matching of Degraded and Noisy Fingerprints," *Proc. Of SPIE (Applications of Digital Image Processing XV)*, vol. 1771, pp. 498-508, 1992.
23. Ceguerra A. and Koprinska I., "Integrating Local and Global Features in Automatic Fingerprint Verification," *in Proc. Int. Synp. On Fingerprint Detection and Identification*, J. Almong and E. Springer (Eds.), Israel National Police, Jerusalem, pp. 305, 1996.
24. Stosz J.D. and Alyea L.A., "Automated System for Fingerprint Authentication Using Pores and Ridge Structure," *Proc. Of SPIE (Automatic Systems for the Identification and Inspection of Humans)*, vol. 2277, pp. 210-223, 1994.
25. Polikarpova N., "On the Fractal Features in Fingerprint Analysis," in *Proc. Int. Conf. on Pattern Recognition* (13th), vol. 3, pp. 591-595, 1996.
26. Coetzee L. and Botha E.C., "Fingerprint Recognition in Low Quality Images," *Pattern Recognition*, vol. 26, no. 10, pp. 1441-1460, 1993.
27. Jain A.K., Prabhakar S., Hong L., and Pankanti S., "Filterbank-Based Fingerprint Matching," *IEEE Transactions on Image Processing*, vol. 9, pp. 846-859, 2000.
28. Milewski B. *Windows API Tutorial*, http://www.relisoft.com/win32/index.htm , 2006.
29. Barnsley M., *Fractals Everywhere*, Morgan Kaufmann, San Fransisco, 2000.
30. Sierpinski W., *Sur une courbe dont tout point est un point de ramification*, C. R. Acad. Sci., Paris, 1915.
31. NIST Image Group, http://fingerprint.nist.gov/ , 2004.
32. Weisstein. E., "Standard Error.", http://mathworld.wolfram.com/StandardError.html

Appendix A Software Manual

General description

This program converts fingerprints into fractal images. The fingerprints are assumed to be in standard grayscale TIFF format. The algorithm constructs a random walk through the fingerprint and uses this sequence of data as input to the Chaos Game that produces the fractal. The details of this (both technical and theoretical) procedure are discussed at length in the Annual Report dated September 30, 2005 NIJ(2003-RC-CX-K001).

Using the algorithm the scale spectrum defined in the Annual Report is constructed. The scale spectrum is characteristic of the fingerprint and is used to make direct comparisons between fingerprints. A variety of tools are included in this package. The first outlines the useable areas of the image of the fingerprint. Also included is a tool that generates the fractal for arbitrary scaling parameters. The batch processing mode includes a tool to produce Discrete Cosine Transforms of the scale spectrum. The program also includes a separate capability to send the results to a data file. A variety of parameters governing the algorithm must be input. These are discussed in the remainder of this guide.

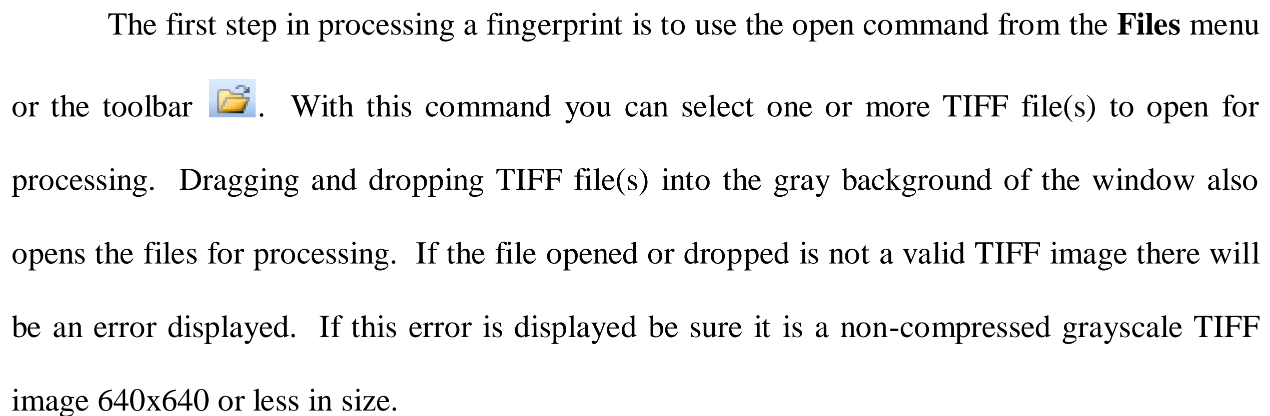
The software has been structured to be a long-term exploratory tool as well as an algorithm depository. This strategy is apparent in the look and feel of the program's implemented capabilities to date. Graphical display of calculated data, the ability to select input parameters, and real-time graphical updates to user requests via sliders and other controls are important features for us— as the developers of the underlying computational algorithms— as well as for users whose intent is for demonstration and discussion of PCG research and technologies.

Installation

Installation of the program only involves copying the executable to the folder from which you want to run the program. You must also have the "duke.dll" file in the same folder as the executable. Failure to have the "duke.dll" file will cause an error when attempting to run the program.

Getting Started

Double clicking the executable file starts the fingerprint program. Upon opening the program one sees a blank window with the typical menu and toolbars at the top (see Figure A-1).



The first step in processing a fingerprint is to use the open command from the **Files** menu or the toolbar . With this command you can select one or more TIFF file(s) to open for processing. Dragging and dropping TIFF file(s) into the gray background of the window also opens the files for processing. If the file opened or dropped is not a valid TIFF image there will be an error displayed. If this error is displayed be sure it is a non-compressed grayscale TIFF image 640x640 or less in size.

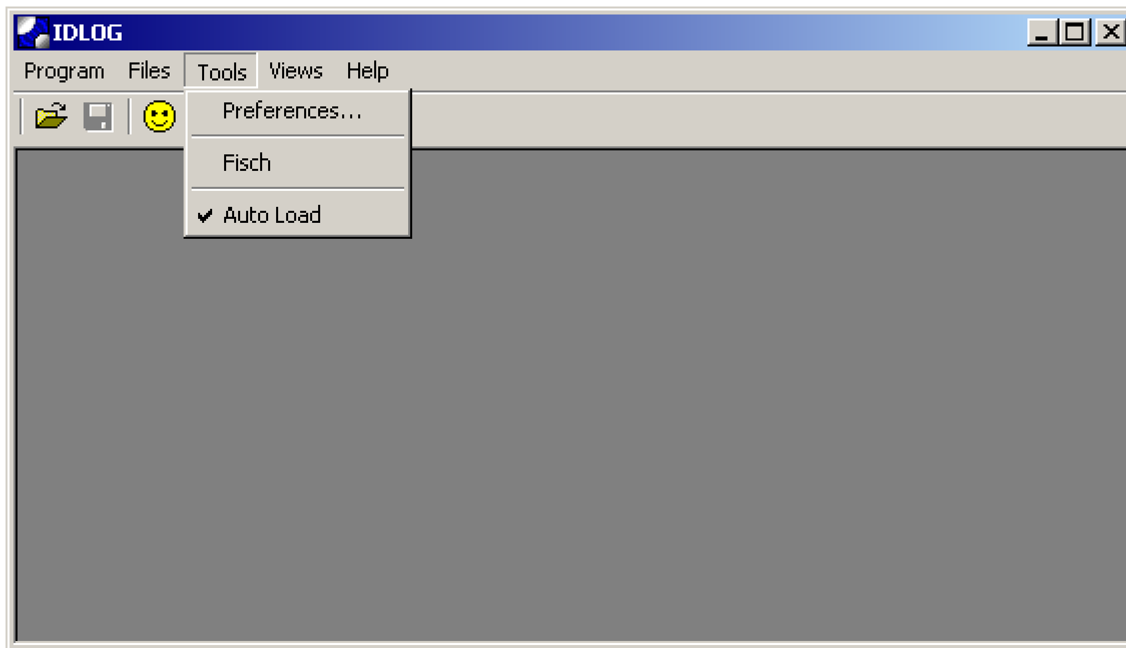


Figure A-1. The Initial program window. Notice the menu and toolbar at the top of the window. Clicking on these produces menus that are used to set options, control inputs and outputs, or process

Working with the fingerprint

When the TIFF file(s) are successfully loaded you will see a dialog with three panes, four buttons below them (see Figure A-2). Once the TIFF file(s) are opened you will see the fingerprint shown in the first pane. Clicking the **Outline** button will outline and convert the fingerprint to a binary image. The result will be displayed in the second of the three panes in the dialog. The areas in green are the masked area of the image identified by the outlining procedure and will not be used in the fractal calculations.

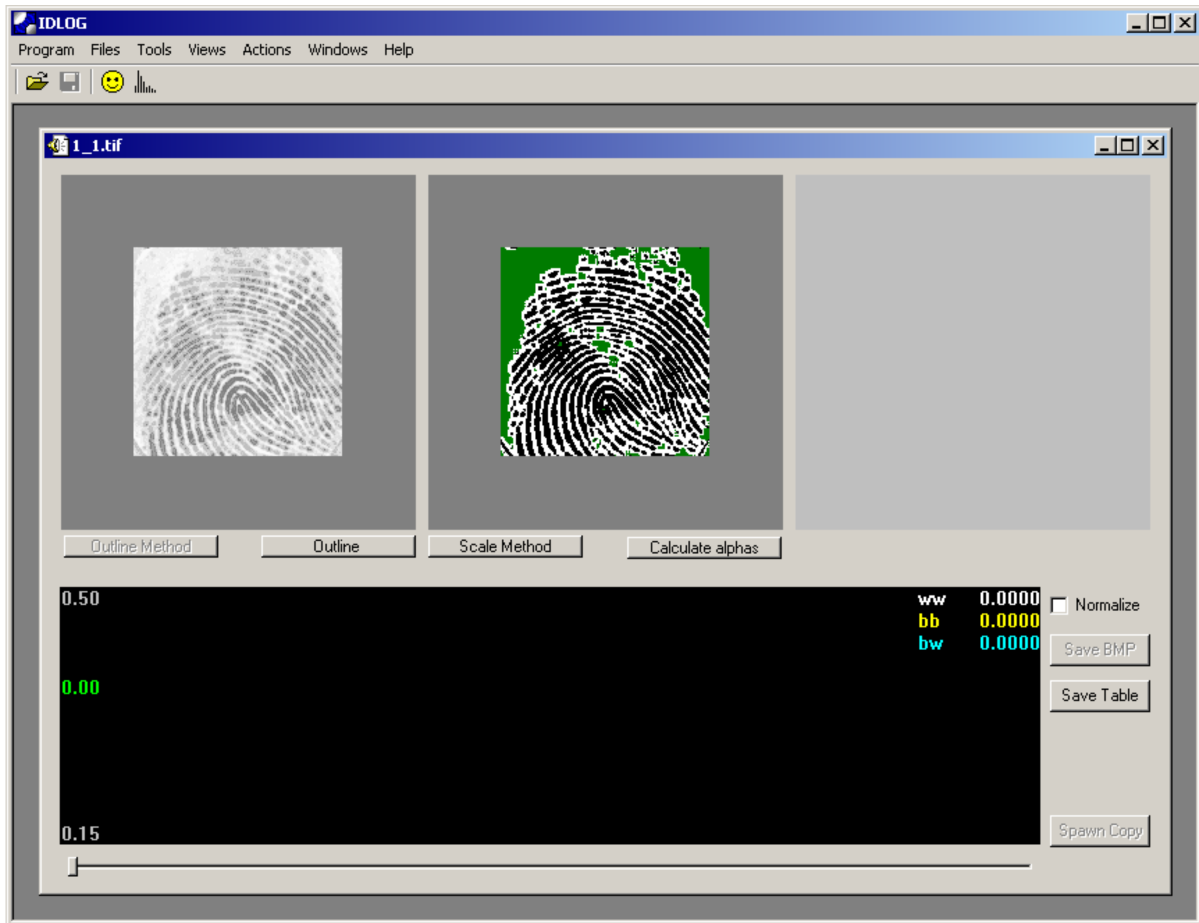


Figure A-2. TIFF image dialog. Outlining process has been done. The three panes are for displaying the fingerprint image, the outlined image, and the fractal for the scale currently selected by the slider. The buttons below the panes allow you to outline the image, change the random walk and scale boundaries (explained below), and to perform the calculations. The items that are grayed out are not fully implemented in this version of the program and are consequently disabled.

Once the outlining process has been completed you can use the **Calculate Alphas** button to do the random walks to produce the fractals for the print. If you wish to change the length of the random walk, the number of scales or the increment between scales click the **Scale Method** button to open the dialogue (see Figure A-3).

Once you have set the scale method and clicked on the **Calculate Alphas** button the program will begin the fractal generation process. As fractals are calculated for each scale they are displayed in the third box and the scale spectrum graph is updated. When the calculation is

complete for all scales (see Figure A-4) fractals for a given scale can be reviewed using the slider at the bottom of the window (see Figure A-4 and A-5).

When the calculation is complete you also have the ability to normalize the graph (subtract a value so the graphs approach zero at the end). You can also choose to save the values for the print to a text file using the **Save Table** button. This option saves the data to the selected text file in a tab separated format so it can be opened in other programs such as Excel.

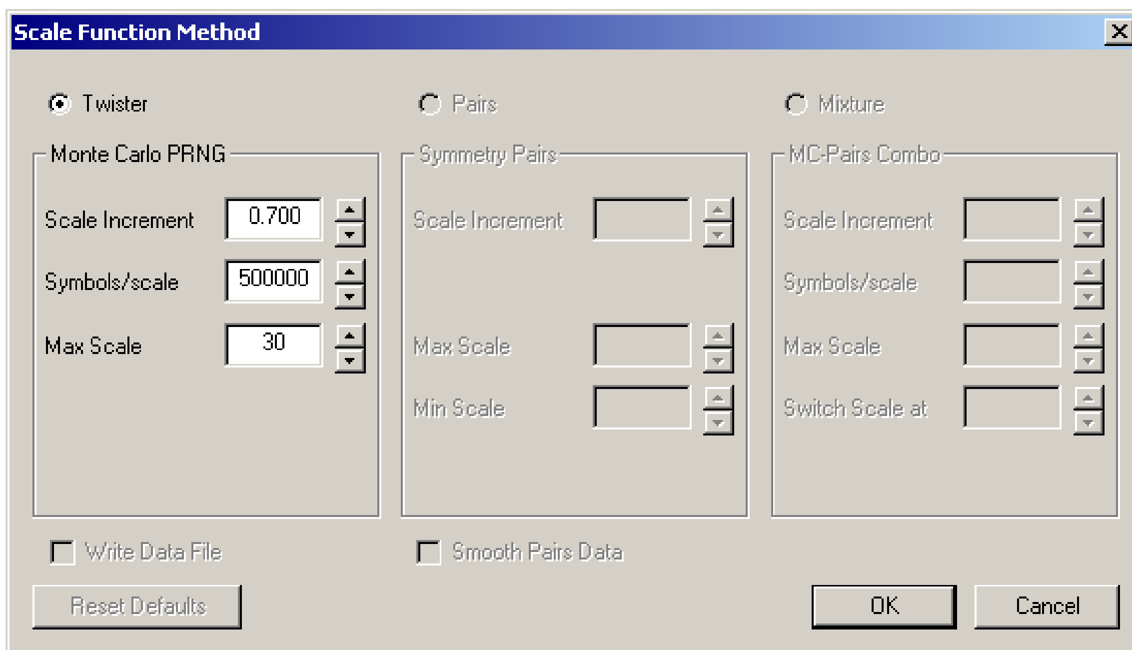


Figure A-3. Scale Method Dialog. The Max Scale box relates to the maximum distance between the pair of compared pixels. The Scale Increment relates to the step size between fractal images calculated for each scale. Finally, the Symbols/scale refers to the length of the random walk.

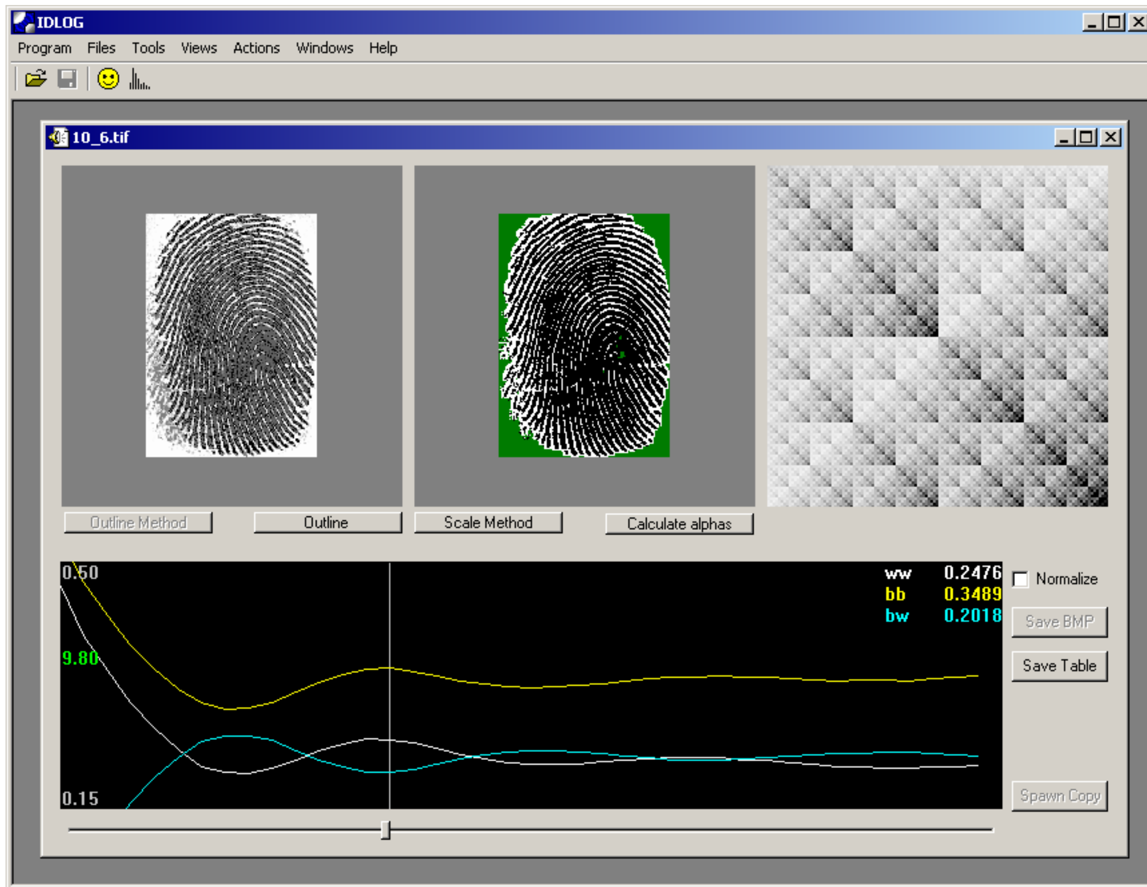


Figure A-4. Tiff dialog upon completion. The fractal for the scale indicated by the slider and the line on the scale spectrum graph is displayed in the third pane above.

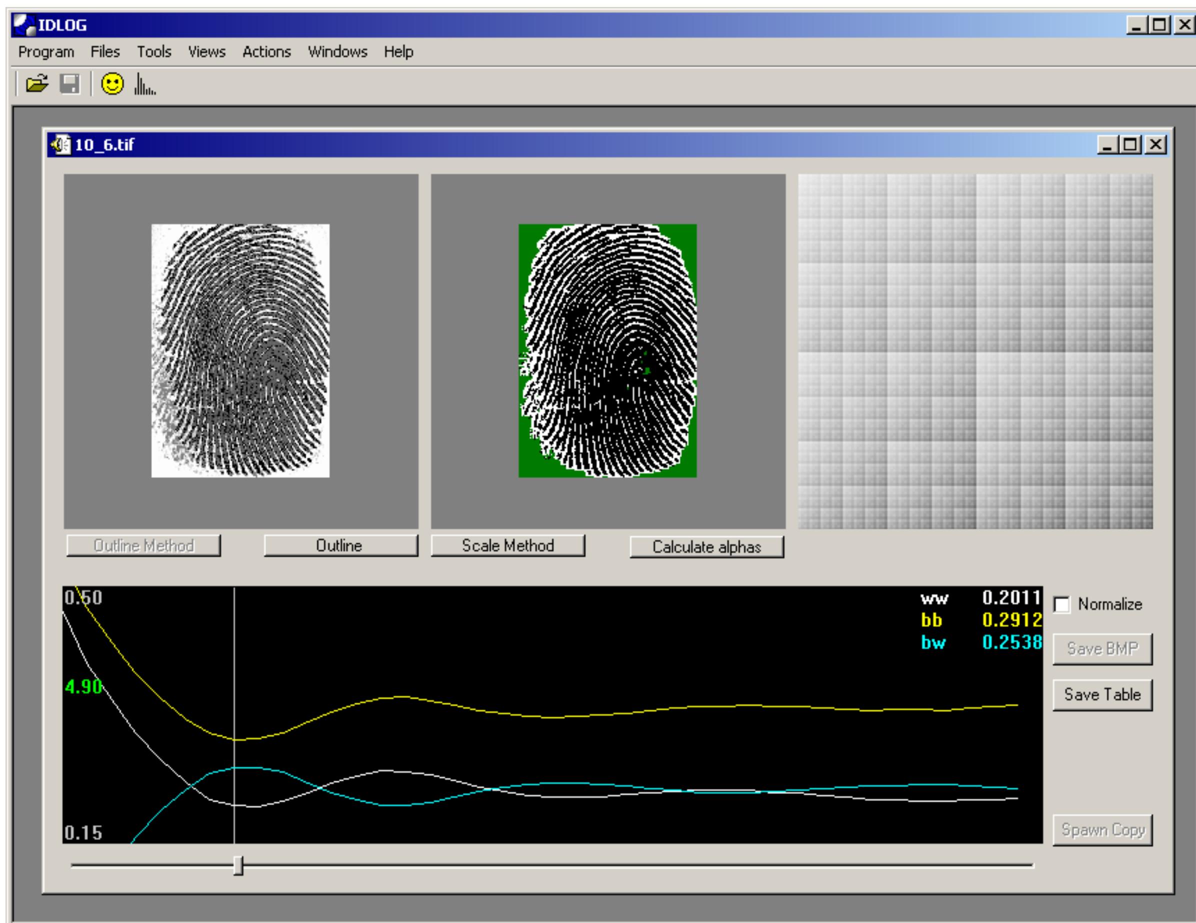
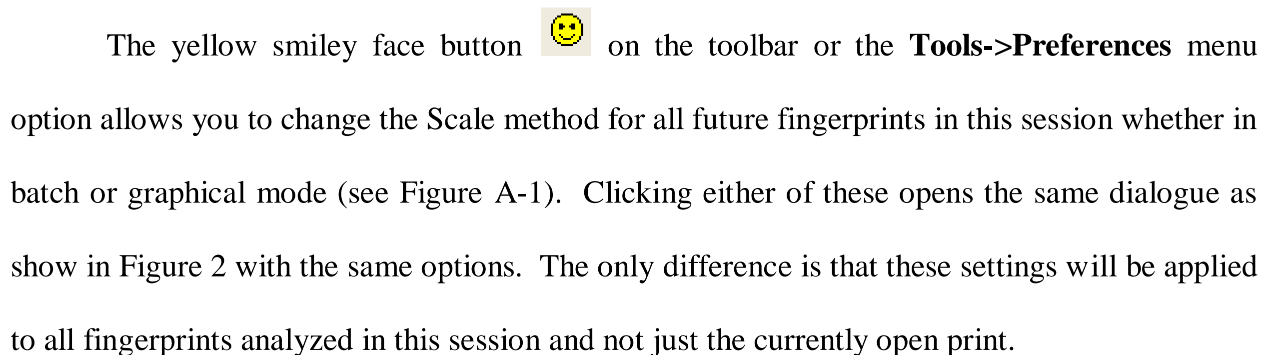


Figure A-5. Tiff dialog upon completion. This is the same fractal and run as in Figure B.4 but with a different slider position and corresponding fractal.

Global Options and Batch Processing

The yellow smiley face button  on the toolbar or the **Tools->Preferences** menu option allows you to change the Scale method for all future fingerprints in this session whether in batch or graphical mode (see Figure A-1). Clicking either of these opens the same dialogue as show in Figure 2 with the same options. The only difference is that these settings will be applied to all fingerprints analyzed in this session and not just the currently open print.

Clicking the **Files->Batch Flag** option puts the program in batch mode. In this mode when print(s) are opened or dragged in the program does not display the graphical interface, but rather begins the process for the opened files. When the process is complete for all files the program will display a completed message. You can then find the results of the batch processing in two text files for each print located in the same directory the print was loaded from. If the print's filename was *filename.tif* the two files would be *filename.txt.* The first file has the scale and scaling parameter table similar to the one saved using the Save Table button in graphical mode. The second file is the Discrete Cosine Transform of the data in the table in the first file. These files can be open in Excel or any similar program for inspection and comparison between prints.

Other Tools

Clicking the **Tools->Fisch** menu option opens a tool shown in Figure A-6 that illustrates the relationship between the scaling parameters and the fractal pattern they define. Moving the sliders around the fractal will show the various fractals produced for the scaling parameters given by the sliders.

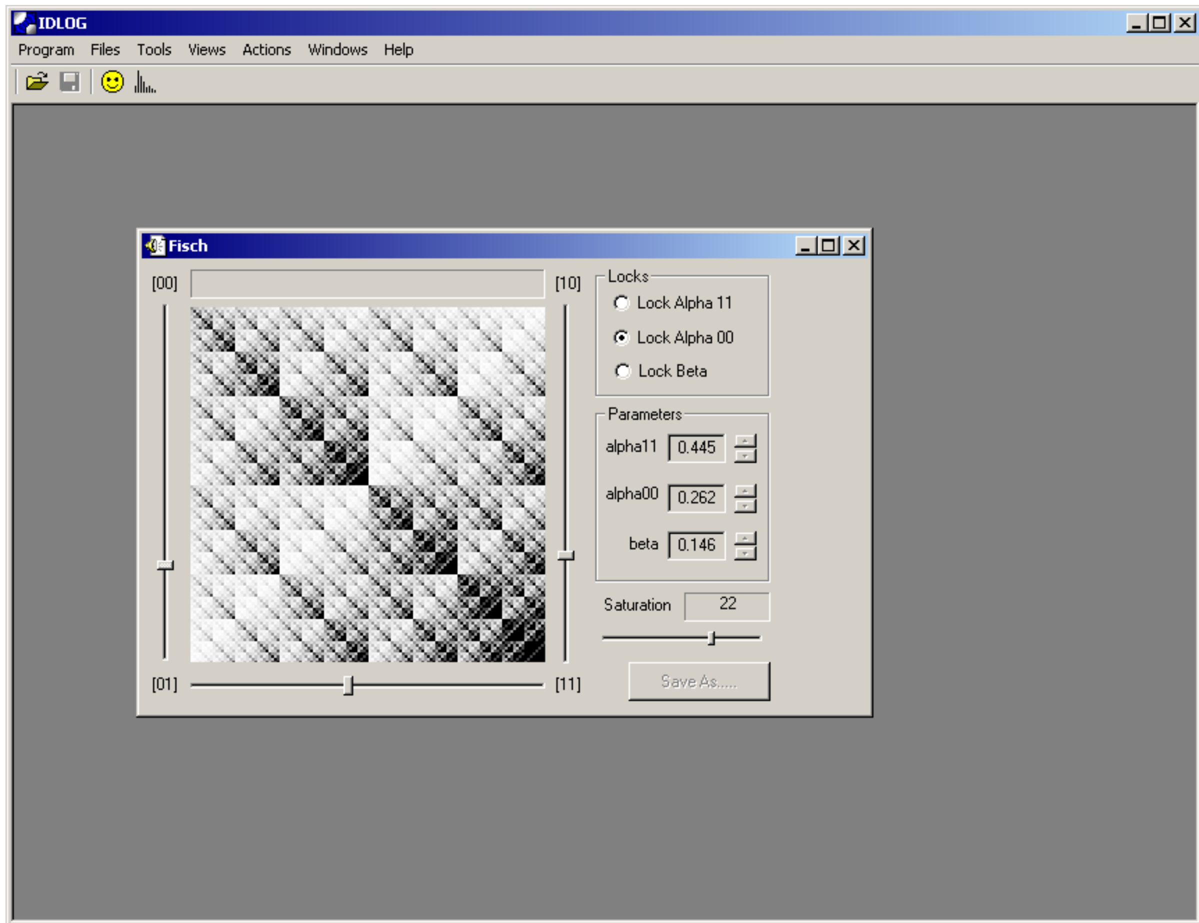


Figure A-6. Scaling parameter to fractal tool. The scaling parameter with the dot before it is set allowing only two of the sliders to be moved. Changing the dot will allow the other slider to move.

Appendix B Source Code

DlgMaster.h

```
#ifndef DLG_MASTER_H
#define DLG_MASTER_H

#include "tools.h"
#include "arrays.h"
#include "twister.h"
#include "fileTIFF.h"
#include "FrameCtl.h"
#include "ctlDMask.h"
#include "ctlDWalk.h"
#include <windows.h>


extern    TCHAR szFileDrop[MAX_PATH];   // Framectl.cpp
extern    TCHAR szTitleDrop[MAX_PATH];  // Framectl.cpp
extern    TCHAR _szFileName[MAX_PATH];  // Framectl.cpp


class DlgMaster
{
public:
     DlgMaster (DOCSTATEINFO  & pstate, IDLOGDOC *initDOC,
WalkData *walkdata, HWND hwnd);
     ~DlgMaster ();


     void OutlineMethod ();        //calls outlining
     void OnNewTiff (HWND hwnd);   //Captions window
     void SymbolGenerateMethod (); //Calls symbol gen twister
     void SymbolGenerateTwister ();//calls calc next point

     void MaskPrefsDlg (HWND hwnd);  //opens dialogue
     void    WalkPrefsDlg (HWND hwnd);  //opens dialogue

     void RunBatch();
     BOOL TxtFileWrite (TCHAR * szFileName, TCHAR * szTitleName,
bool bNormalized);
     bool CalcNextPoint(int iscale, float finc);


     //provides access to priviate variables
     float     getA11(int i) const {return
(_A11arr.GetFloat(i));}
     float getA00(int i) const {return (_A00arr.GetFloat(i));}
```

```
     float getB01(int i) const {return (_B01arr.GetFloat(i));}
     float getBn(int i) const {return (_Barr.GetFloat(i));}
     float getDIS(int i) const {return (_distance.GetFloat(i));}
     BYTE getBmpImage(int x, int y) const {return
(_bmpImage.GetByte(x,y));}
     BYTE getBmpMask(int x, int y) const {return
(_bmpMask.GetByte(x,y));}
     bool getBoolMask(int x, int y) const {return
(_boolMask.GetBool(x,y));}
     int       getImageDimX() const {return
_bmpImage.QueryDimX();}
     int       getImageDimY() const {return
_bmpImage.QueryDimY();}
     int       getMaskDimX() const {return
_bmpMask.QueryDimX();}
     int       getMaskDimY() const {return
_bmpMask.QueryDimY();}
     int       getNumScales() const {return _numScales;}

     float GetScaleIncr() const {return
_walkMethodLocal.GetScaleIncr();}
     int GetWalkMethod() const {return
_walkMethodLocal.GetMethod();}

 private:
     DlgMaster (const DlgMaster & no_copy);  // protect from
copy

     void PrepareStateForDlgCtor ();
     HWND         _hwnd;

     WalkData          &_walkMethodGlobal;   // params for
generating symbols
     WalkData          _walkMethodLocal;   // params for
generating symbols

     int               _numScales;
     int               _neighborParm;

     Byte2D_640        _bmpImage;     // for display of original
image
     Byte2D_640        _bmpMask;      // for display of masked
image

     Bool2D_640        _boolImage;    //  unmasked data for
analysis
     Bool2D_640        _boolMask;     //  mask for data used in
```

```
analysis

     Int2D_640          _scratch;      //  neighborhood
calculations

     FileTifTouch        _tifFileInfo;
     TCHAR               _szTitleName[MAX_PATH]; // Local

     //scaling parameters
     AlphaArray        _A00arr;
     AlphaArray        _A11arr;
     AlphaArray        _B01arr;
     AlphaArray        _distance;
     AlphaArray        _A00c;
     AlphaArray        _A11c;
     AlphaArray        _B01c;
     AlphaArray        _B10c;
     AlphaArray       _Tout;

     AlphaArray       _Pb;
     AlphaArray       _Pw;
     AlphaArray       _Barr;

    HWND           _hMDIChildOwner; // scroll, command
    HWND           _hFrameClient;        // scroll, command
    IDLOGCOMMON              _common;
    MenuBarCurrent      _TDSLmenu;
    DOCSTATEINFO        &_pState;             // owned by
ChildCtrl
    Twister            _MTPRNG;
    FingerTwister      _ftwister;
    MaskData           _maskMethod;


};

#endif
```

DlgMaster.cpp

```
#include "dlgmaster.h"

#include "outline.h"
#include "MiscDlgs.h"
#include "modaldlg.h"

extern "C" {
#define nsp_UsesTransform
#define nsp_UsesDct
```

```
//#include "C:\Program Files\Intel\plsuite\Include\nsp.h"
#include "nsp.h"
}


#include <windowsx.h>
#include <stdlib.h>

static const double INFLXPNT_TEMP = 0.03f;

DlgMaster::DlgMaster (   DOCSTATEINFO   & pstate, IDLOGDOC
*initDOC, WalkData *walkdata, HWND hwnd)
     :    // Stuff we get from MDI Doc level: (don't use
File*Look/Touch)
          _walkMethodGlobal        (* walkdata),
          _walkMethodLocal         (_walkMethodGlobal),
          _hwnd(hwnd),
          _hMDIChildOwner (GetParent(hwnd)),
          _hFrameClient (GetParent(_hMDIChildOwner)),
          _TDSLmenu (GetParent(_hFrameClient)),
          _tifFileInfo (_bmpImage),
          _numScales (MAX_SCALE_INDEX),           // the number
(also sizes?)
          _neighborParm (1),
          _ftwister (_MTPRNG, _boolImage, _boolMask),// size of
neighborhood; see Outline.cpp
          _maskMethod (127, out_thresh, out_thresh),
//
          _pState             (pstate)//,         // owned by
ChildCtrl
{
     PrepareStateForDlgCtor ();
     // Has PtrWavedata been redimed already?
}

DlgMaster::~DlgMaster ()
{
}


void DlgMaster::RunBatch()
{
   OnNewTiff(_hwnd);
   OutlineMethod();
   SymbolGenerateMethod();
   TxtFileWrite (_szFileName, _szTitleName, false);
```

```
}

void DlgMaster::OnNewTiff (HWND hwnd)
{
     GetFileTitle (szFileDrop, szTitleDrop, MAX_PATH+1);
     if (_tifFileInfo.DropFileOpen (hwnd, szFileDrop,
szTitleDrop))
     {
          memcpy (&_szFileName, &szFileDrop, MAX_PATH);     //
copy sz*Drop to sz*Name
          GetFileTitle (_szFileName, _szTitleName, MAX_PATH+1);
          DoCaption (GetParent(hwnd), _szTitleName) ;
//        UpdateNeedSaveStatus (false);
     }
}

void DlgMaster::SymbolGenerateTwister ()
{

  numScales  =
_walkMethodLocal.GetMaxScale()/_walkMethodLocal.GetScaleIncr();

  // Count the pixels of each type

  _ftwister.CountPixels();            // Scale = zero; alphas =
fraction of pixels

  float a00 = (float) _ftwister.GetNum00()/_ftwister.GetTotal();
  float a11 = (float) _ftwister.GetNum11()/_ftwister.GetTotal();
  float b01 =
(float)(_ftwister.GetNum01()+_ftwister.GetNum10())/(2*_ftwister.
GetTotal());

  _A00arr.SetFloat(0, a00);
  _A11arr.SetFloat(0, a11);
  _B01arr.SetFloat(0, b01);
  _distance.SetFloat(0, 0);
  _Barr.SetFloat(0,1);

  // Get parameters as a function of scale

  for (int iscale = 1; iscale < _numScales; iscale++)
    {
      CalcNextPoint(iscale, _walkMethodLocal.GetScaleIncr());
    }
```

```
}

bool DlgMaster::CalcNextPoint(int iscale, float finc)   //
returns true while points left to do.
{
  int tout;
  _numScales =
_walkMethodLocal.GetMaxScale()/_walkMethodLocal.GetScaleIncr();
// need this for printing txt file

  if( iscale >= _numScales )
    {
      return false;        // error, add a message box here for
DEBUG
    }

  if ( 0 == iscale)
    {
      _ftwister.CountPixels();// Scale = zero; alphas = fraction
of pixels
      tout = 0;
    }
  else
    {
      _ftwister.CountSymbols(iscale,
_walkMethodLocal.GetNumSymbols(), finc/2,tout); //last param is
factor to multiply scale
    }

  //set values for scaling parameters
  _A00arr.SetFloat(iscale, (float)
_ftwister.GetNum00()/_ftwister.GetTotal());
  _A11arr.SetFloat(iscale, (float)
_ftwister.GetNum11()/_ftwister.GetTotal());
  _B01arr.SetFloat(iscale,
(float)(_ftwister.GetNum01()+_ftwister.GetNum10())/(2*_ftwister.
GetTotal()));

  //set values for probability of black and white pixels
  _Pb.SetFloat(iscale,
(float)(_ftwister.GetNum11()*2+_ftwister.GetNum01()+_ftwister.Ge
tNum10())/(2*_ftwister.GetTotal()));
  _Pw.SetFloat(iscale,
(float)(_ftwister.GetNum00()*2+_ftwister.GetNum01()+_ftwister.Ge
tNum10())/(2*_ftwister.GetTotal()));

  //set normalized values
```

```
  _Barr.SetFloat(iscale,(float)1-
((pow(_B01arr.GetFloat(iscale),2)/(_A00arr.GetFloat(iscale)*_A11
arr.GetFloat(iscale)))));
  _distance.SetFloat(iscale, ((float) iscale)*finc);

  _A00c.SetFloat(iscale,_ftwister.GetNum00());
  _A11c.SetFloat(iscale,_ftwister.GetNum11());
  _B01c.SetFloat(iscale,_ftwister.GetNum10());
  _B10c.SetFloat(iscale,_ftwister.GetNum01());
  _Tout.SetFloat(iscale,tout);


  return (_numScales != (iscale + 1) );
}

void DlgMaster::OutlineMethod ()
{
  Outline outline;

  outline.ByteToBool(_bmpImage, _boolImage);   // -> binary,
based on avg unmasked image

  // create mask, allowing for noise thresholds _noisefraction

  outline.MaskSquareAggregate(_boolImage, _boolMask, _scratch,
_neighborParm, _maskMethod);

  outline.ByteToBoolMasked(_bmpImage, _boolImage, _boolMask);
// temp: no mask, just a "greyscale" of 2 color image

  outline.MaskSquareAggregate(_boolImage, _boolMask, _scratch,
_neighborParm,  _maskMethod);

  outline.ByteToBoolMasked(_bmpImage, _boolImage, _boolMask);
// temp: no mask, just a "greyscale" of 2 color image

  outline.BoolToByte(_boolImage, _bmpMask);    // temp: no mask,
just a "greyscale" of 2 color image

  //      _boolMask.FillAll(true);                     //
convention: mask is "true" if we use the pixel


}

void DlgMaster::SymbolGenerateMethod ()
{
```

```
   switch (_walkMethodLocal.GetMethod ())
   {
   case WALK_METHOD_TWISTER:
      SymbolGenerateTwister ();
      break;
   case WALK_METHOD_PAIRS:
//       SymbolGeneratePairs ();
      break;
   case WALK_METHOD_COMBO:
//       SymbolGenerateCombo ();
      break;
   }

}

void DlgMaster::MaskPrefsDlg (HWND hwnd)
{
  //  MaskData data (_maskMethod);
  MaskDataBMP data (_maskMethod, _bmpImage);
  ControllerFactory <MaskCtrl, MaskDataBMP> factory (& data);
  ModalDialog dialog (GetWindowInstance(hwnd), hwnd,
IDD_MASK_METHOD, &factory);
  if (dialog.IsOk ())
    {
      _maskMethod.CopyAll (data);
      ::InvalidateRect (hwnd, NULL, FALSE);  // Force repaint
    }
}

void DlgMaster::WalkPrefsDlg (HWND hwnd)
{
  WalkData data (_walkMethodLocal);
  ControllerFactory <WalkCtrl, WalkData> factory (& data);
  ModalDialog dialog (GetWindowInstance(hwnd), hwnd,
IDD_WALK_METHOD, &factory);
  if (dialog.IsOk ())
    {
      _walkMethodLocal.CopyAll (data);
      ::InvalidateRect (hwnd, NULL, FALSE);  // Force repaint
    }
}

BOOL DlgMaster::TxtFileWrite (TCHAR * szFileName, TCHAR *
szTitleName, bool bNormalized)
{
  FILE *            outf;
```

```
  TCHAR    *szFileNameTXT =szFileName;
  char *temp;
  temp = strchr(szFileNameTXT,int('.'));
  *temp = '\0';
  strncat(szFileNameTXT, ".txt",4);

  outf = fopen(szFileNameTXT,"w");

  int i;
  float mul_00, mul_11, mul_01;          // assymptotic values
(A(0) squared)
  float inf_00, inf_11, inf_01;          // assymptotic values
(A(0) squared)

  fprintf(outf,"\n\nFile: %s\tScale Incr:%1.2f\tMax
Scales:%d\tNum Steps%d\n", szTitleName,
       _walkMethodLocal.GetScaleIncr(),
       _walkMethodLocal.GetMaxScale(),
       _walkMethodLocal.GetNumSymbols());
  fprintf(outf,"\nScale\tN11\tN00\tN01\tN10\tNumSkip\n");
  for (i = 0; i < _numScales; i++)
    {


      fprintf(outf,"%.2f\t%.0f\t%.0f\t%.0f\t%.0f\t%.0f\n",
_distance.GetFloat(i),
           _A00c.GetFloat(i),
           _A11c.GetFloat(i),
           _B01c.GetFloat(i),
           _B10c.GetFloat(i),
           _Tout.GetFloat(i));
    }
}

fclose(outf);
return TRUE;
}
```

Twister.h

```
#ifndef TWISTER_H
#define TWISTER_H

#include "params.h"      // needed for MAX_SCALE_INDEX
#include "arrays.h"
#include "FOG/randoma.h"
```

```
#include <cassert>
#include <time.h>
#include <windows.h>               // needed for min, max, BOOL

// class to generate unifrom random numbers
class Twister
{
 public:
  Twister (int iSeed = (int) time( NULL ))
    :  _twoPI (2 * 3.1415926535),
    _min (0),
    _max (1)

    {
      TRandomInit(iSeed);
    }
  //set range
  void    SetMin (int i) {_min = i;}
  void    SetMax (int i) {_max = i;}


  float TwoPI () const {return _twoPI;}


private:

   float      _twoPI;
   int        _min;
   int        _max;

};


class FingerTwister
{
 public:
  FingerTwister (Twister & twst, Bool2D_640 & bImage, Bool2D_640
& bMask)
    :  _pTwist (twst),
    _pbImage (bImage), _pbMask (bMask),
    _iA00 (0), _iA11 (0), _iB01 (0), _iB10 (0), _iTotal(0),
    _iScale (1)
    {
    }

  BYTE GetSymbol (int i, int j) const;
```

```
  //functions to perform walk
  void CountSymbols (int iscale, int iterations, float factor,
int &tout);
  void CountPixels ();

  //functions to return results
  int GetNum00 () const {return _iA00;}
  int GetNum01 () const {return _iB01;}
  int GetNum10 () const {return _iB10;}
  int GetNum11 () const {return _iA11;}
  int GetTotal () const {return _iTotal;}


private:

   int             _iA00;     // false-false (WHITE-WHITE) (MSB-
LSB)
   int             _iA11;     // true -true  (BLACK-BLACK) (MSB-
LSB)
   int             _iB01;
   int             _iB10;
   int             _iTotal;

   int             _iScale;

   Twister        &_pTwist;

   //images to perform walk over
   Bool2D_640     &_pbImage;
   Bool2D_640     &_pbMask;
};


#endif
```

Twister.cpp

```
#include "Twister.h"
#include <math.h>



void FingerTwister::CountSymbols (int iscale, int iterations,
float factor, int &tout)
{
  _iScale = iscale;
```

```
  tout=0;
  int inum=0;
  float fScale = _iScale * factor;
  _iTotal = 0;

  _iA00 = 0;
  _iA11 = 0;
  _iB01 = 0;
  _iB10 = 0;

  int xsize = _pbImage.QueryDimX();
  int ysize = _pbImage.QueryDimY();


  //  Initialize variables associated with the output bitmap's
coordinates

  int xA, yA, xB, yB;
  float fXorg, fYorg;
  float fTheta;
  float fx, fy;
  int isymbol;

  float fXsize = (float)(xsize-1);
  float fYsize = (float)(ysize-1);

  do
    {
      fYorg = fYsize * TRandom();    //  Pick a random x and y
coordinate
      fXorg = fXsize * TRandom();

      //random angle
      fTheta = TRandom() * _pTwist.TwoPI();

      fx = fScale * cos(fTheta);
      fy = fScale * sin(fTheta);

      yA = (int)(fYorg + fy); //  Find P1, with new x and y
coordinates (closest integer)
      xA = (int)(fXorg + fx);

      yB = (int)(fYorg - fy); //  Find P2, with new x and y
coordinates (closest integer)
      xB = (int)(fXorg - fx);

      //  Check to ensure all four points are within bounds
```

```
      if(          (yA >= 0) && (yA < ysize))
     {
       if(       (xA >= 0) && (xA < xsize))
         {
           if(    (yB >= 0) && (yB < ysize))
          {
            if((xB >= 0) && (xB < xsize))
              {

                if (_pbMask.GetBool(xA, yA) &&
_pbMask.GetBool(xB, yB))  // convention: mask is "true" if we
use the pixel
               {
                 isymbol = 0;

                 if(_pbImage.GetBool(xA, yA))      // true =
black
                   {
                     isymbol += 1;
                   }

                 if(_pbImage.GetBool(xB, yB))      // most
recent point is MSB
                   {
                     isymbol += 2;
                   }
                 //increment the appropriate counter
                 switch (isymbol)
                   {
                   case 0:
                     _iA00++;
                     break;
                   case 1:
                     _iB01++;
                     break;
                   case 2:
                     _iB10++;
                     break;
                   case 3:
                     _iA11++;
                     break;
                   }

                 _iTotal++;

               }  // outline
```

```
            }     //yA
        }        //xA
       }           //yB
    }             //xB
     inum++;
    } while (_iTotal < iterations);        // loop over
iterations

  tout=inum-iterations;

}


void FingerTwister::CountPixels ()
{
  _iTotal = 0;

  _iA00 = 0;
  _iA11 = 0;
  _iB01 = 0;     // remains 0
  _iB10 = 0;     // remains 0

  int xsize = _pbImage.QueryDimX();
  int ysize = _pbImage.QueryDimY();


  //  Count the self-self symbols (black and white pixels)

  for (int j = 0; j < ysize; j++)
    {
      for (int i = 0; i < xsize; i++)
     {
       if (_pbMask.GetBool(i, j))          // don't count the
masked pixels
         {
           if(_pbImage.GetBool(i, j))      // true = black
          {
            _iA11++;
          }
           else
          {
            _iA00++;
          }
           _iTotal++;
         }
     }
```

```
    }
}
```

Outline.h
(written entirely by Dr. Lyn Ratcliff)

```
#ifndef OUTLINE_H
#define OUTLINE_H

#include "ctlDMask.h"

class Int2D_640;
class Byte2D_640;
class Bool2D_640;


class Outline
{
 public:
  Outline ()
    : _srcDimX(0), _srcDimY(0), _destDimX(0), _destDimY(0),
    _iValidPixelCount(0), _validAvg (0)
    {
    }

  void DoOutline (Byte2D_640 & source, Bool2D_640 & dest);

  void ByteToBool (Byte2D_640 & source, Bool2D_640 & dest);
  void BoolToByte (Bool2D_640 & source, Byte2D_640 & dest);

  int  GetValidPixelCount () const {return _iValidPixelCount ;}

  void ByteToBoolMasked (Byte2D_640 & source, Bool2D_640 & dest,
Bool2D_640 & bmask);

  void MaskSquareAggregate (Bool2D_640 & source, Bool2D_640 &
dest,
                   Int2D_640 & scratch, int tt, const MaskData &
maskdata);


 protected:

  int           _srcDimX, _srcDimY, _destDimX, _destDimY;
  int          _iValidPixelCount;
```

```
  float       _validAvg;

};


#endif
```

Outline.cpp
(written entirely by Dr. Lyn Ratcliff)

```
#include "outline.h"
#include "arrays.h"
//#include <windowsx.h>  // GetWindowInstance


// aggregate version of OUTLINE nask

// tt = 0, 1, 2, ...
// zz = 2*tt + 1
// 3*zz = 2*ww + 1

// neighborhood is a square of length = width = 3*zz

// eg if tt = 1, then zz = 3, ww = 4, neighborhood is  9x9
divided into 9 sub-squares of 3x3
// eg if tt = 2, then zz = 5, ww = 7, neighborhood is 15x15
divided into 9 sub-squares of 5x5

// will try tt = 1 for 640x640 images (1500x1500 images used
35x35 neighborhood)

// we will PAD the outside edges of the image by (zz + tt)
pixels, by simply extending the
// edge values into the PAD region, the same way that a surface-
mount chip leads are soldered
// onto a curcuit board; the corner values will be copied to the
four PAD corners.


// then, we aggregate (sum) data for every zz by zz sub-square,
and then do a 9 term sum
// (center and compass directions) to decide on whether to mask
the pixel.


void Outline::MaskSquareAggregate (Bool2D_640 & source,
```

```
Bool2D_640 & dest, Int2D_640 & scratch, int tt, const MaskData &
maskdata)
{
  int zz =2*tt + 1;
  int ww =3*tt + 1;
  int PAD = zz + tt;

  dest.SetDimX(source.QueryDimX());      // set dimensions of
the mask
  dest.SetDimY(source.QueryDimY());

  _srcDimX = source.QueryDimX();         // some redundancy in
these
  _srcDimY = source.QueryDimY();
  _destDimX = dest.QueryDimX();          // some redundancy in
these
  _destDimY = dest.QueryDimY();

  int scrDimX = _srcDimX + 2*PAD;
  int scrDimY = _srcDimY + 2*PAD;

  scratch.FillAll(0);                    // zero out scratch
array
  scratch.SetDimX(scrDimX);              // set dimensions of
the scratch array
  scratch.SetDimY(scrDimY);

  int tmp = _srcDimY - 1;
  int i, j, ii, jj;


  // Fill center of scratch array, indexed with offset PAD

  for (j = 0; j < _srcDimY; j++)
    {
      for (i = 0; i < _srcDimX; i++)
     {
       if (source.GetBool(i,j))
         {
           scratch.SetInt(i+PAD, j+PAD, 1);      // work with
this array in subsequent steps (?)
         }
     }
    }

  // pad top and bottom (but not corners)
```

```
  for (i = 0; i < _srcDimX; i++)
    {
      if (source.GetBool(i,0))               // pad top
     {
       for (jj = 0; jj < PAD; jj++)
         {
           scratch.SetInt(i, jj, 1);
         }
     }

      if (source.GetBool(i,tmp))             // pad bottom
     {
       for (jj = 0; jj < PAD; jj++)
         {
           scratch.SetInt(i, _srcDimY + jj, 1);
         }
     }
    }

  tmp = _srcDimX - 1;

  // pad left and right (including corners) using values from
scratch array

  for (j = 0; j < scrDimY; j++)
    {
      for (ii = 0; ii < PAD; ii++)
     {
       scratch.SetInt(ii, j, scratch.GetInt(PAD, j));
       scratch.SetInt(_srcDimX+ii, j, scratch.GetInt(tmp, j));
     }
    }

  // if we could settle on a value of tt, this routine could be
made much more efficient
  // by unrolling many of these "for" loops

  // aggregate the data in two steps (zz terms added along x
coordinate, then along y coordinate)

  int isum;
  int numExtra = zz - 1;

  for (j = 0; j < scrDimY; j++)
    {
      for (i = 0; i < (scrDimX - numExtra); i++)
     {
```

```
        isum = 0;
        for (ii = 0; ii < zz; ii++)
          {
            isum += scratch.GetInt(i+ii, j);
          }
        scratch.SetInt(i, j, isum);
      }
    }

  for (j = 0; j < (scrDimY- numExtra); j++)
    {
      for (i = 0; i < (scrDimX - numExtra); i++)
      {
        isum = 0;
        for (jj = 0; jj < zz; jj++)
          {
            isum += scratch.GetInt(i, j+jj);
          }
        scratch.SetInt(i, j, isum);
      }
    }

  // reject pixel if the neighborhood avg is outside range [(1 -
thresh)*maxagg, thresh*maxagg]

  int twoPAD = PAD+PAD;

  int maxagg = (3*zz)*(3*zz)*(1);        // *(1) because source
is bool
  int lowerTol = maxagg * maskdata.GetLoThresh();
  int upperTol = maxagg * (1.0f - maskdata.GetHiThresh());

  dest.FillAll(true);                 //

  for (j = 0; j < _srcDimY; j++)
    {
      for (i = 0; i < _srcDimX; i++)
      {
        isum = 0;

        isum += scratch.GetInt(i,       j);
        isum += scratch.GetInt(i+PAD,    j);
        isum += scratch.GetInt(i+twoPAD, j);
        isum += scratch.GetInt(i,        j+PAD);
        isum += scratch.GetInt(i+PAD,    j+PAD);
        isum += scratch.GetInt(i+twoPAD, j+PAD);
        isum += scratch.GetInt(i,        j+twoPAD);
```

```
        isum += scratch.GetInt(i+PAD,    j+twoPAD);
        isum += scratch.GetInt(i+twoPAD, j+twoPAD);

        if( (isum < lowerTol) || (isum > upperTol) )
          {
            dest.SetBool(i, j, false);
          }
      }
     }

}




void Outline::ByteToBool (Byte2D_640 & source, Bool2D_640 &
dest)
{
  dest.SetDimX(source.QueryDimX());      // set dimensions of
the mask
  dest.SetDimY(source.QueryDimY());

  _srcDimX = source.QueryDimX();      // some redundancy in
these
  _srcDimY = source.QueryDimY();
  _destDimX = dest.QueryDimX();      // some redundancy in these
  _destDimY = dest.QueryDimY();


  // An extremely crude fake outlining procedure
  bool bvalue;
  long sum = 0;
  float avg;
  int i, j;

  //calculate the average byte value
  for (j = 0; j < _srcDimY; j++)
    {
      for (i = 0; i < _srcDimX; i++)
     {
        sum += source.GetByte(i,j);
     }
    }

  avg = sum / (_srcDimY*_srcDimX);

  for (j = 0; j < _srcDimY; j++)
```

```
    {
      for (i = 0; i < _srcDimX; i++)
     {
       if (source.GetByte(i, j) < avg)
         {
           bvalue = true;
         }
       else
         {
           bvalue = false;
         }
       dest.SetBool (i, j, bvalue);
     }
    }

}

void Outline::ByteToBoolMasked (Byte2D_640 & source, Bool2D_640
& dest, Bool2D_640 & bmask)
{
  dest.SetDimX(source.QueryDimX());      // set dimensions of
the mask
  dest.SetDimY(source.QueryDimY());

  _srcDimX = source.QueryDimX();      // some redundancy in
these
  _srcDimY = source.QueryDimY();
  _destDimX = dest.QueryDimX();      // some redundancy in these
  _destDimY = dest.QueryDimY();


  // An extremely crude fake outlining procedure
  int i, j;
  bool bvalue;
  long sum = 0;
  _iValidPixelCount = 0;

  for (j = 0; j < _srcDimY; j++)     //calculate the average
byte value
    {
      for (i = 0; i < _srcDimX; i++)
     {
       if (bmask.GetBool(i,j))          // convention: mask is
"true" if we use the pixel
         {
           sum += source.GetByte(i,j);
           _iValidPixelCount++;
```

```
          }
      }
     }

  _validAvg = sum / _iValidPixelCount;   // get average over
unmasked

  for (j = 0; j < _srcDimY; j++)
    {
      for (i = 0; i < _srcDimX; i++)
     {
       if (source.GetByte(i, j) < _validAvg)
         {
           bvalue = true;
         }
       else
         {
           bvalue = false;
         }
       dest.SetBool (i, j, bvalue);
     }
    }

}

void Outline::BoolToByte (Bool2D_640 & source, Byte2D_640 &
dest)
{
  dest.SetDimX(source.QueryDimX());      // set dimensions of
the mask
  dest.SetDimY(source.QueryDimY());

  _srcDimX = source.QueryDimX();      // some redundancy in
these
  _srcDimY = source.QueryDimY();
  _destDimX = dest.QueryDimX();      // some redundancy in these
  _destDimY = dest.QueryDimY();


  // An extremely crude fake outlining procedure
  BYTE white = 255;
  BYTE black = 0;
  BYTE byvalue;

  for (int j = 0; j < _srcDimY; j++)
    {
      for (int i = 0; i < _srcDimX; i++)
```

```
      {
        if (source.GetBool(i, j))
          {
            byvalue = black;
          }
        else
          {
            byvalue = white;
          }
        dest.SetByte (i, j, byvalue);
      }
     }

}
```

Appendix C

DlgMaster **(methods)**	
`void OutlineMethod()`	**Inputs:** grayscale array (_bmpImage) **Outputs:** binary array (_boolImage) mask array (_bmpMask) **Purpose:** The method calls the outlining functions to convert the image to binary and outline it. It uses the image arrays from the private section.
`void OnNewTiff(HWND hwnd)`	**Inputs:** hwnd **Outputs:** none **Purpose:** The method gets the filename that has been opened and uses it for a caption on the window
`void SymbolGenerateMethod ()`	**Inputs:** Walk Method **Outputs:** none **Purpose:** The method calls the appropriate method to perform the walk based on the method selected (SymbolGenerateTwister is the only implemented method)
`void SymbolGenerateTwister ()`	**Inputs:** Walk parameters (_walkMethodLocal) **Outputs:** scaling parameters for scale zero (a00,a11,b01 arrays) **Purpose:** The method determines the length and step size for the walk from default or input information. The method then calls the CountPixels function for the for each of the scales until max scale.
`void MaskPrefsDlg (HWND hwnd)`	**Inputs:** hwnd **Outputs:** Mask data (_maskMethod) **Purpose:** The method opens the mask preference dialogue to gets mask data from the user.

`void WalkPrefsDlg (HWND hwnd)`	**Inputs:** hwnd **Outputs:** Walk values (_walkMethodLocal) **Purpose:** The method opens the walk preferences dialogue to get walk information from the user.
`void RunBatch ()`	**Inputs:** none **Outputs:** none) **Purpose:** The method calls the other methods necessary to perform the random walk and write the results to the file. The method takes the place of the GUI 's buttons when the batch flag is selected.
`bool TxtFileWrite (TCHAR *szFileName, TCHAR *szTitleName, bool bNormalized)`	**Inputs:** Filename, titlename, normalize flag, scaling parameters (a00,a11,b01) **Outputs:** Table of information to txt file **Purpose:** The method formats and write the scale spectra information to a text file with the same name as the print.
`bool CalcNextPoint (int iscale, float finc)`	**Inputs:** binary array (_bmpImage) mask array (_bmpMask) current scale (iscale) increment (finc) **Outputs:** scaling parameters (a00,a11,b01) **Purpose:** The method calls twister's CountPixels method to perform the random walk for the current scale. It also fills the scaling parameter arrays upon completion.
`float get*****` `BYTE get*****`	Methods to provide data from private variable to functions outside of the class.

DlgMaster **(variables)**	
`HWND _hwnd`	Pointer to the window for print information display
`WalkData &_walkMethodGlobal`	Reference to walk information for all prints.
`WalkData _walkMethodLocal`	Data for the length and scale of the random

	walk either a copy of the global or the user input data
`int _numScales`	Number of scales to perform the walk over.
`int _neighborParm`	Number of neighboring pixels to consider for outlining
`Byte2D_640 _bmpImage`	Array containing the input image.
`Byte2D_640 _bmpMask`	Array containing the mask calculated in the outlining methods for display
`Bool2D_640 _boolImage`	Array containing the binary print image.
`Bool2D_640 _boolMask`	Array containing the mask information used for analysis
`Int2D_640 _scratch`	Array used for temp in outlining process.
`FileTifTouch _tifFileInfo`	Tiff file information
`TCHAR _szTitleName[MAX_PATH]`	Title of file
`AlphaArray _***`	Arrays to store scaling parameter, current scale, number of pixels skipped and normalized scale values
`FingerTwister _ftwister`	Instance of the twister class used to perform the random walk.
`MaskData _maskMethod`	Data to use when calculating the outline.
`Other private variables`	Used for tracking parent windows and other GUI information.