

Enhancing and Reconstructing Digitized Handwriting.

by

David James Swain

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

©David James Swain and VPI & SU 1997

APPROVED:

Advisor: Dr. Roger W. Ehrich

Dr. John W. Roach

Dr. Donald C. Allison

August, 1997
Blacksburg, Virginia

Enhancing and Reconstructing Digitized Handwriting.

by

David James Swain

Project Advisor: Dr. Roger W. Ehrich

Computer Science

(ABSTRACT)

This thesis involves the restoration, reconstruction, and enhancement of a digitized library of hand-written documents. Imaging systems that perform this digitization often degrade the quality of the original documents. Many techniques exist for reconstructing, restoring, and enhancing digital images; however, many require *a priori* knowledge of the imaging system. In this study, only partial *a priori* knowledge is available, and therefore unknown parameters must be estimated before restoration, reconstruction, or enhancement is possible.

The imaging system used to digitize the documents library has degraded the images in several ways. First, it has introduced a ringing that is apparent around each stroke. Second, the system has eliminated strokes of narrow widths. To restore these images, the imaging system is modeled by estimating the point spread function from sample impulse responses, and the image noise is estimated in an attempt to apply standard linear restoration techniques. The applicability of these techniques is investigated in the first part of this thesis. Then nonlinear filters, structural techniques, and enhancement techniques are applied to obtain substantial improvements in image quality.

ACKNOWLEDGEMENTS

I would like thank Professor Roger Ehrich, my advisor, for the countless hours of advice, proof reading, and direction that he provided throughout this research project. I would like to thank my committee members Professors John Roach and Donald Allison for their support and suggestions. I would like to thank Scott Guyer for answering all of my questions related to L^AT_EX and the ETD project. I would like to thank the computer science department at Virginia Tech and VTLS, Inc. for funding me throughout my graduate studies. And last, I would like to thank my parents for always providing moral support and encouragement.

To my parents, George and Linda Swain.

TABLE OF CONTENTS

1	Introduction	1
2	Background	3
2.1	Image Enhancement	3
2.2	Image Restoration	4
2.2.1	Overview of the Linear Model	4
2.2.2	The Linear Position Invariant Model for Continuous Functions	5
2.2.3	The Linear Position Invariant Model for Discrete Functions	7
2.3	Image Reconstruction	8
2.3.1	Image Morphology	9
2.3.2	Gap Filling	12
3	Image Degradations	13
3.1	Overview of Image Degradations	13
3.2	Evaluating the Degradations Present in the Handwriting	14
3.2.1	Observed Degradations	14
3.2.2	Strategy for Removing the Observed Degradations	18
4	Filtering Techniques	19
4.1	Inverse Filter	19
4.1.1	Implementation of Inverse Filtering	20
4.1.2	Results of Inverse Filtering	24
4.2	Nonlinear Filtering	26

CONTENTS

4.2.1	Implementation of Nonlinear Filtering	27
4.2.2	Results of Nonlinear Filtering	34
5	Enhancement and Structural Filtering	36
5.1	Contrast Stretching	36
5.2	Structural Filtering	39
5.2.1	Morphological Closing	39
5.2.2	Gap Filling	41
5.3	Histogram Modification	42
5.4	Final Experiments	43
6	Concluding Remarks	54
A	Source Code	55
A.1	Inverse Filtering Script	56
A.2	Matrix Class	61
A.3	Image Class	81
A.4	Mask Class	92
A.5	Script for Experiment 1	98
A.6	Script for Experiment 2	98
A.7	Script for Experiment 3	99
A.8	Source Code for Driver Used in Experiment 1	100
A.9	Source Code for Driver Used in Experiment 2	105
A.10	Source Code for Driver Used in Experiment 3	109
A.11	Source Code for the Nonlinear Filter Implementation	114
A.12	Driver Program for the Erosion Operation	120
A.13	Driver Program for the Dilate Operation	122
A.14	Driver Program for the Gap Filling Operation	126

LIST OF FIGURES

2.1	The general linear image system model with additive noise	4
3.1	Sample document illustrating the poor image quality in the degraded images	15
3.2	Magnified portion of a sample document showing the edge ringing and missing strokes.	16
3.3	Graph of scan lines intersecting handwriting strokes taken from Figure 3.2	17
3.4	Imaging systems used to create the digital handwriting images. Transformation $T1$ corresponds to the recording of image data onto photographic film. Transformation $T2$ corresponds to digitization of the film data by a Polaroid scanner.	17
4.1	Cross section of the average impulse response	21
4.2	Results of passing images through the estimated model. The left column contains sample images with the ringing manually removed. The middle column contains output of the estimated degradation model. The right column contains the unaltered samples.	22
4.3	Magnitude of the two-dimension Fourier transform of the estimated degradation model	24
4.4	Magnitude of the inverse filter.	25
4.5	(<i>Left</i>) Sample image from the document library. (<i>Right</i>) Result of applying the inverse filter to the sample.	25
4.6	Perceptron for two pattern classes.	32
4.7	(<i>Left</i>) Sample images with standard deviations of noise equal to 0 (top), 10 (middle), and 30 (bottom). (<i>Right</i>) Result of applying the nonlinear filter to the sample on the left.	35

LIST OF FIGURES

5.1	Sample image used to illustrate the effects of enhancement and reconstruction techniques.	37
5.2	(a) Original sample taken from the boxed region in Figure 5.1. (b) Result with $t = 20$. (c) Result with $t = 40$. (d) Result with $t = 60$. (e) Result with $t = 80$. (f) Result with $t = 100$	38
5.3	Structuring element used to perform gray level erosion.	39
5.4	(Left) Result from the contrast stretch. (Right) Result of applying a gray level erosion to the image on the left.	40
5.5	Structuring element used to perform gray level erosion.	40
5.6	(Left) Result from the erosion stage. (Right) Result of applying a gray level dilation to the image on the left.	41
5.7	Filter used to in the gap filling operation.	41
5.8	(Left) Result from the initial contrast stretch. (Right) Result of applying three iterations of the gap filling operation to the image on the left.	42
5.9	(Left) Result from the morphological closing. (Right) Result of contrast stretch and brightness adjustment applied to the figure on the left.	43
5.10	(Left) Result from the gap filling. (Right) Result of contrast stretch and brightness adjustment applied to the figure on the left.	43
5.11	Sample from the document library.	44
5.12	Result from the enhancement and reconstruction techniques.	44
5.13	(Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.	45
5.14	(Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.	46
5.15	(Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.	47
5.16	(Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.	48
5.17	(Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.	49

LIST OF FIGURES

5.18	(<i>Top</i>) Sample from the document library. (<i>Bottom</i>) Result from enhancement and reconstruction techniques.	50
5.19	(<i>Top</i>) Sample from the document library. (<i>Bottom</i>) Result from enhancement and reconstruction techniques.	51
5.20	(<i>Top</i>) Sample from the document library. (<i>Bottom</i>) Result from enhancement and reconstruction techniques.	52
5.21	(<i>Top</i>) Sample from the document library. (<i>Bottom</i>) Result from enhancement and reconstruction techniques.	53

LIST OF TABLES

4.1	Variance calculation of the random sample at each pixel position	21
4.2	Results of noise estimation experiment.	23
4.3	Results from Experiment 1 using the local mean as the background estimator. . . .	28
4.4	Results from Experiment 1 using the local median as the background estimator. . .	29
4.5	Results from Experiment 1 using the local median of local means as the background estimator.	29
4.6	Results of Experiment 2 which investigates the local median of local means as a potential background estimator.	30
4.7	Results of Experiment 3 which investigates the local median as a potential background estimator.	31

Chapter 1

Introduction

In recent years, the rapid development of technology coupled with decreasing costs has enticed many industries and organizations to archive documents digitally to facilitate access over computer networks. A typical method of implementing this strategy is to digitize the documents using a scanner. Often this digitization process degrades or corrupts the quality of the documents. Printed text documents can usually be digitized into computer text documents using optical character recognition software. This software creates editable ASCII text files from the scanned image by reading and classifying each character. However, some documents such as handwritten documents cannot be converted into ASCII text and thus must be stored as bitmaps. In such cases if degradations have been introduced, then the bitmaps must be further processed to create digital representations that are visually faithful to the original documents.

This research project focuses on correcting the degradations in an archive of documents that have been stored as bitmaps. The documents contain handwriting strokes and therefore are not amenable to optical character recognition. The original documents have been scanned from microfilm using a Polaroid scanner. The image acquisition process has introduced edge defocusing, ringing, and noise. Thin faint strokes have also been lost. These degradations reduce the quality of the images.

The fundamental objective of this thesis is to remove the degradations present in the archived documents to improve document readability. To achieve this, two techniques are investigated. The first technique examined is the linear restoration technique called inverse filtering which seeks to sharpen edges and remove the ringing using a linear degradation model. This technique requires two

CHAPTER 1. INTRODUCTION

steps. First, a model of the linear imaging system that produced the degraded images is constructed. Next, the inverse filter is constructed from this model and applied to sample data.

For numerous reasons, inverse filtering turns out not to be viable, so a second technique is attempted. This combines image preprocessing, image enhancement, and image reconstruction techniques. The image is processed using a nonlinear filter to remove the ringing around the edge of each stroke. This prepares the images so that the application of the desired enhancement techniques are more effective. The image enhancement stages apply methods to the preprocessed images which strengthen the visual effect of foreground information. The image reconstruction stage applies structural reconstruction methods to connect fragmented strokes where possible. Two particular structural operations are applied and evaluated for the purpose of reconstructing lost foreground information. These include an image morphological closing operation and a gap filling operation. Results are presented for both techniques.

Both of the techniques mentioned in the above paragraph are applied to sample images to evaluate the improvements achieved. The techniques are evaluated with respect to their overall intended purpose which is to improve document readability. Discussion regarding the advantages, disadvantages, limitations, and future considerations is presented.

Chapter 2

Background

When undertaking an image processing project, it is necessary to clarify terminology. In this project, distinctions are recognized between image enhancement, image restoration, and image reconstruction. These terms are often confused and used interchangeably in verbal conversation. Indeed, these topics are related in some undeniable aspects. For example, image enhancement is generally a byproduct of image restoration or of image reconstruction. However, this is not a symmetric relationship as image enhancement does not produce restoration or reconstruction. Because this study implements techniques from each of these areas of image processing, it is necessary to provide background information for these topics. This chapter provides this background information in an effort to remove any possible ambiguities.

2.1 Image Enhancement

Image enhancement is widely considered to be any process applied to an image to improve its visual appearance. Image enhancement often includes an improvement in the perceptual aspects of an image such as image intelligibility or visual appearance [Lim, 1990]. For these reasons, enhancement techniques usually apply heuristic procedures to alter an image according to the psychophysical aspects of the human visual system [Gonzalez and Woods, 1992]. Further, enhancement techniques tend to be simple, qualitative, and ad hoc. Therefore, these techniques usually are very application-dependent and perform well only for a specific class of images. The goodness criteria for enhancement techniques are almost always subjective, as conversion from these criteria to objective measures is

too complex [Lim, 1990]. In this study, the primary enhancement method is histogram modification for improving contrast and brightness.

2.2 Image Restoration

In contrast to image enhancement, image restoration is any process to recover an image by using a model of the degradation process. Therefore, restoration procedures require accurate models of the processes that introduced degradations, and they attempt to apply the inverse of these processes to the degraded image [Gonzalez and Woods, 1992]. For this reason, image restoration algorithms generally are more mathematical and complex than image enhancement algorithms. This section presents the development of a linear imaging system. This is the most common model applied to image systems that introduce degradations.

2.2.1 Overview of the Linear Model

The first technique evaluated in this study applies a linear restoration procedure to the degraded images in an attempt to improve image quality. A linear restoration procedure is built around the general linear imaging system model with additive noise. This is depicted in Figure 2.1.

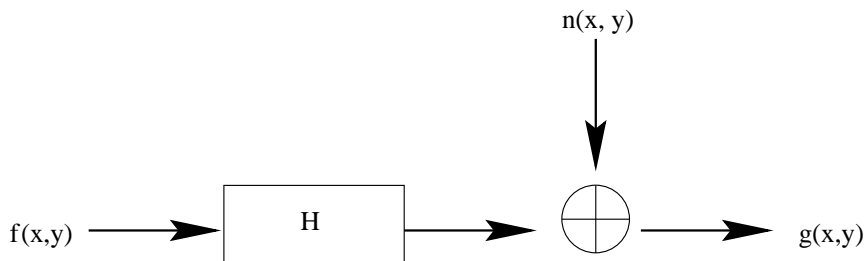


Figure 2.1: The general linear image system model with additive noise

A mathematical statement of Figure 2.1 is given by

$$g(x, y) = h(x, y) * f(x, y) + n(x, y) \quad (2.1)$$

where h denotes the image system model, f denotes the input image, n denotes the noise image, g denotes the output image, and (x, y) denotes the spatial coordinates of the images. One interpreta-

CHAPTER 2. BACKGROUND

tion of Equation 2.1 is that g is formed by adding the convolution of h with f to a noise image n . This provides the simple linear model with additive noise that will be used as the foundation of the image restoration technique that is investigated in this study. The applicability of this model will be discussed in Section 3.2.

Two properties of a linear system that will be used to simplify the model developed in this study include additivity and homogeneity. The additive property states that a response to two inputs is equal to the sum of the two responses [Gonzalez and Woods, 1992]. Mathematically, this is given by

$$H [k_1 f_1(x, y) + k_2 f_2(x, y)] = H [k_1 f_1(x, y)] + H [k_2 f_2(x, y)]. \quad (2.2)$$

The homogeneity property states that the response of a constant multiple of any input is equal to the response multiplied by the constant [Gonzalez and Woods, 1992]. Mathematically, this is given by

$$H [k_1 f_1(x, y)] = k_1 H [f_1(x, y)]. \quad (2.3)$$

A third property that will be used in the model developed in this study is position invariance. This property states that the response of a system at any point in an image depends only on the input values in a local neighborhood about that point and not on the image position. Mathematically, H is position invariant if

$$H [f(x - \alpha, y - \beta)] = g(x - \alpha, y - \beta) \quad (2.4)$$

for any $f(x, y)$ and any α and β . Applied to image restoration, position invariance means that the system model responds uniformly across the entire image.

The main advantage of the linear imaging system model is that it allows a wealth of knowledge from the field of linear systems theory to be applied to image restoration problems. Sometimes non-linear or space variant models are more accurate models. However, they often introduce mathematical difficulties that have no solutions or are mathematically intractable [Gonzalez and Woods, 1992]. For these reasons, image restoration techniques approximate imaging systems using linear position invariant systems throughout the historical literature.

2.2.2 The Linear Position Invariant Model for Continuous Functions

The linear position invariant imaging system model can be derived within the domain of continuous functions. The following derivation is given in [Gonzalez and Woods, 1992]. Let $f(x, y)$ be an ideal

CHAPTER 2. BACKGROUND

image that can be specified by the two-dimensional sampling process

$$f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta) \delta(x - \alpha, y - \beta) d\alpha d\beta. \quad (2.5)$$

Here, δ is the *Dirac delta function*, and f_c is the original continuous image. It has unit volume in an infinitesimal region about (x, y) and is zero elsewhere. For this reason, the Dirac delta function is often referred to as a two-dimensional impulse function [Gonzalez and Woods, 1992]. It follows from Equation 2.5 that f at any point (x, y) is the infinite sum of the responses of an image region to unit stimuli.

Now, a noiseless model of an imaging system is given by

$$g(x, y) = H[f(x, y)] = H \left[\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta) \delta(x - \alpha, y - \beta) d\alpha d\beta \right]. \quad (2.6)$$

In Equation 2.6, H is an operator that models the imaging system, and $g(x, y)$ is the output image. If H is a linear operator, then applying the property of additivity to Equation 2.6 yields

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} H[f_c(\alpha, \beta) \delta(x - \alpha, y - \beta)] d\alpha d\beta. \quad (2.7)$$

Further, since $f_c(\alpha, \beta)$ is independent of x and y , the property of homogeneity can be applied to Equation 2.7 giving

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta) H[\delta(x - \alpha, y - \beta)] d\alpha d\beta. \quad (2.8)$$

Equation 2.8 yields important results. The term

$$h(x, \alpha, y, \beta) = H[\delta(x - \alpha, y - \beta)] \quad (2.9)$$

is the impulse response of H . This states that if H is applied to an impulse response of strength 1 in the absence of noise, then the response of the system is given by h . In optics, h is referred to as the *point spread function* as it specifies how a system responds or spreads an intense point source of light [Duda and Hart, 1973]. If Equation 2.9 is substituted into Equation 2.8, the continuous model for linear imaging systems becomes

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta) h(x, \alpha, y, \beta) d\alpha d\beta. \quad (2.10)$$

Equation 2.10 is the *superposition* or *Fredholm integral of the first kind*. This provides a fundamental result in linear system theory. It states that the impulse response completely specifies the

CHAPTER 2. BACKGROUND

linear operator H . Applied to image restoration, this states that if the point spread function of a linear system is known, then the response to any input can be calculated. In other words, knowledge of the point spread function can be used to determine the output for any input point (x, y) [Rosenfeld and Kak, 1982].

If H is a position invariant operator, then Equation 2.9 can be written as

$$H[\delta(x - \alpha, y - \beta)] = h(x - \alpha, y - \beta). \quad (2.11)$$

Substituting this result into Equation 2.10 yields the convolution integral given by

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta)h(x - \alpha, y - \beta)d\alpha d\beta. \quad (2.12)$$

Equation 2.12 gives the linear position invariant imaging system model in the continuous domain in the absence of noise. In the presence of additive noise, the model becomes

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_c(\alpha, \beta)h(x - \alpha, y - \beta)d\alpha d\beta + \eta(x, y) \quad (2.13)$$

where $\eta(x, y)$ denotes the noise at coordinate (x, y) .

2.2.3 The Linear Position Invariant Model for Discrete Functions

Digital image processing deals with the processing of quantized image information known only at discrete sampling coordinates. Sampling refers to the measurement of image intensities on a regular sampling lattice, whereas quantization refers to amplitude digitization [Gonzalez and Woods, 1992]. To model a digital imaging system, the continuous linear position invariant imaging system model is developed for the discrete domain.

The discrete formulation of the linear position invariant imaging system is constructed around the convolution theorem. In the continuous domain, the convolution theorem states that

$$h(x, y) * f(x, y) \iff H(u, v)F(u, v) \quad (2.14)$$

where $h(x, y), H(u, v), f(x, y), F(u, v)$, and $h(x, y) * f(x, y), H(u, v)F(u, v)$ are Fourier transform pairs and $*$ denotes the convolution operation. In the discrete domain, the $*$ denotes a circular convolution, and therefore, to make the theorem parallel the continuous behavior, h and f must be periodic with periods large enough so that h and f do not overlap at repetitions. Let $h(x, y)$ denote the linear imaging system operator, and $f(x, y)$ denote the input image. If h is of size $A \times B$ and f

CHAPTER 2. BACKGROUND

is of size $C \times D$, then the images must be extended to at least $(A + C) - 1 \times (B + D) - 1$ to avoid period overlap. The extended functions are created by padding h and f with zeros resulting in

$$h_e(x, y) = \begin{cases} h(x, y) & (0 \leq x \leq A - 1) \text{ and } (0 \leq y \leq B - 1) \\ 0 & (A \leq x \leq A + C - 1) \text{ or } (B \leq y \leq B + D - 1) \end{cases} \quad (2.15)$$

and

$$f_e(x, y) = \begin{cases} f(x, y) & (0 \leq x \leq C - 1) \text{ and } (0 \leq y \leq D - 1) \\ 0 & (C \leq x \leq A + C - 1) \text{ or } (D \leq y \leq B + D - 1) \end{cases} \quad (2.16)$$

Since h_e and f_e are now two-dimensional periodic $M \times N$ functions where $M = A + C - 1$ and $N = B + D - 1$, the discrete formulation of Equation 2.12 can be stated as

$$g_e(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_e(m, n) h_e(x - m, y - n). \quad (2.17)$$

In the presence of noise, an $M \times N$ noise term is added to Equation 2.17 yielding a complete linear position invariant imaging system model given by

$$g_e(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_e(m, n) h_e(x - m, y - n) + \eta_e(x, y). \quad (2.18)$$

This is the model that underlies the image restoration procedure that is used in this study to attempt to restore the handwriting images to their original quality. This model makes it possible to compute solutions algebraically. Many techniques exist for making this problem tractable. A more complete discussion of some these techniques is given in [Gonzalez and Woods, 1992].

2.3 Image Reconstruction

Another process that may be applied to the handwriting images to improve readability is called image reconstruction. For the purpose of this study, image reconstruction is any process to recover image content that has been eliminated or lost. Usually, such information can be inferred from the context in a local neighborhood about an image point.

The handwriting images studied in this project have been formed in such a way that a significant portion of the foreground data has been eliminated. Therefore, image reconstruction techniques are applied to reintroduce some of these lost structures. The structural techniques that are used include a morphological closing operation and an ad hoc gap filling operation. Both of these techniques rely on contextual information in a local neighborhood of an image point. The remainder of this section develops the theoretical background for these structural operations.

2.3.1 Image Morphology

Image morphology is a digital image processing technique that is based on shape. Since handwriting basically consists of various well defined shapes, morphological techniques are expected to provide a desirable reconstruction. Image morphology offers a unified and powerful approach to solving various imaging processing problems [Gonzalez and Woods, 1992]. In this study, it is used to attempt to reconnect fragmented handwriting strokes.

Morphology has its foundation in set theory. As applied to images, sets represent shapes on binary or gray tone images [Haralick et al., 1987]. A gray-scale image is represented as a set over Euclidean 3-space. Each pixel of the image is represented as a 3-tuple where the first two components are the pixel coordinates, and the third is the pixel gray level intensity. Morphological transformations can be applied to sets of any dimension. These higher dimensions can be used to abstract additional image information such as color or time. The two fundamental morphological operations are dilation and erosion. Most other operations are defined in terms of these operations.

Dilation and Erosion

Dilation as a morphological transformation combines two sets using vector addition on the elements of the set. Minkowski proposed dilation as a set theoretic operation in 1903 when he used it to characterize integral measures of certain open sets. Dilation was first introduced as an image processing operator in the late 1950's. These early studies used dilation as a smoothing operation [Haralick et al., 1987]. Dilation can be defined as follows. If A and B are two sets contained in N -space with $a = (a_1, \dots, a_n) \in A$ and $b = (b_1, \dots, b_n) \in B$, then the dilation of A by B results in the vector sum of all possible pairs of a and b . Mathematically, the dilation of A by B where $A, B \in E^n$ is

$$A \oplus B = \{c \in E^n | c = a + b \text{ for some } a \in A \text{ and } b \in B\}. \quad (2.19)$$

In morphological terminology, A is the image undergoing transformation, and B is called the structuring element. However, since addition is commutative, dilation is also commutative, and therefore, the roles of A and B are symmetric.

Erosion as a morphological transformation combines two sets using vector subtraction on elements of the set. It is straightforward to show that erosion provides the morphological dual of dilation [Haralick et al., 1987]. Erosion can be defined as follows. If A and B are two sets contained in

CHAPTER 2. BACKGROUND

N -space with $a = (a_1, \dots, a_n) \in A$ and $b = (b_1, \dots, b_n) \in B$, then the erosion of A by B results in the set of all x for which $x + b \in A$ for all $b \in B$. Mathematically, the erosion of A by B where $A, B \in E^n$ is

$$A \ominus B = \{x \in E^n | x + b \in A \text{ for every } b \in B\}. \quad (2.20)$$

An alternative but equivalent definition of erosion that is usually used in literature is to express the erosion of A by B as a difference of elements of A and B . This is given by

$$A \ominus B = \{x \in E^n | \text{for every } b \in B \text{ there exists an } a \in A \text{ such that } x = a - b\}. \quad (2.21)$$

In this case, the roles of A and B are not symmetric because vector subtraction is not commutative.

Dilation and erosion provide building blocks for many basic morphological techniques. The dilation operation is used to construct opening, closing, region filling, and connected components computations. The erosion operation is used to construct opening, closing, boundary extraction, or skeleton computations. The pair is introduced here for the purpose of building a closing operation.

Opening and Closing

The opening and closing operations are rudimentary morphological techniques that are constructed exclusively from the dilation/erosion pair. The opening transformation tends to smooth contours, break narrow isthmuses, and eliminate thin protrusions [Gonzalez and Woods, 1992]. The opening of a set A by the structuring element B , denoted by $A \circ B$, is the erosion of A by B followed by the dilation of the result by B . This is given by

$$A \circ B = (A \ominus B) \oplus B. \quad (2.22)$$

The closing operation also smoothes image section contours; however, it tends to reconnect narrow breaks and fill gaps [Gonzalez and Woods, 1992]. For this reason, this technique will be explored as a possible means for reconstructing lost handwriting strokes. The closing of a set A by the structuring element B , denoted by $A \ominus B$, is the dilation of A by B followed by the erosion of the result by B . This is given by

$$A \bullet B = (A \oplus B) \ominus B. \quad (2.23)$$

The opening and closing operations are also morphological duals and therefore have been introduced together.

CHAPTER 2. BACKGROUND

Extensions to Gray-Scale Imaging

The preceding discussion of morphology has been built using binary images as a basis. Extensions to gray-scale images are straightforward. The presentation given is from [Gonzalez and Woods, 1992]. The discussions that follow use the digital image functions $f(x, y)$ and $b(x, y)$. The function $f(x, y)$ denotes the input image, and $b(x, y)$ denotes the structuring element. Further, the functions are discrete so that $(x, y) \in Z \times Z$ and $f(x, y), b(x, y) \in Z$.

The extension of the dilation operation to gray-scale imaging is introduced first. This operation basically sets the structuring element over a region and produces a set of values by performing an addition of the gray level in the image to the corresponding value in the structuring element. A maximum value calculation is then computed over this set producing the result. This operation is defined as

$$(f \oplus b)(s, t) = \max \{f(s - x, t - y) + b(x, y) \mid (s - x), (t - y) \in D_f; (x, y) \in D_b\}. \quad (2.24)$$

In the above definition, D_f and D_b are the respective domains of f and b . Gray-scale dilation provides two effects. First, since the operation is defined using the *max* function, a structuring element with all positive values tends to brighten an image. Second, dark image details are reduced or eliminated.

As in the binary case, the gray-scale erosion transformation provides the morphological dual to the dilation transformation. This operation places the structuring element over a region and produces a set of values by performing a subtraction of the structuring element value from the corresponding value in the input image. Then, a minimum value calculation is computed over this set. This operation is defined as

$$(f \ominus b)(s, t) = \min \{f(s + x, t + y) - b(x, y) \mid (s + x), (t + y) \in D_f; (x, y) \in D_b\} \quad (2.25)$$

As in the dilation definition, D_f and D_b are the respective domains of f and b . Gray-scale erosion also provides two effects. First, since the operation is defined using the *min* function, a structuring element with all positive values tends to reduce the brightness of an image. Second, bright regions with less area than the structuring element are reduced.

Gray-scale operations that are built on this transformation pair are specified no differently for gray-scale images as they were specified for binary images. That is, the opening and closing transformations are defined for gray-scale images exactly as they were defined for binary images. The

CHAPTER 2. BACKGROUND

definition of a gray-scale opening is given in Equation 2.22. The definition of a gray-scale closing is given in Equation 2.23. This gray-scale closing operation provides a possible stroke reconstruction technique that is examined later.

2.3.2 Gap Filling

The gap filling operation used in this study uses local neighborhoods to detect narrow breaks. Once detected, the breaks are filled by interpolating the stroke intensity values in the neighborhood. This is an ad hoc procedure that is designed explicitly for the handwriting images. For this reason, no theoretical development of the procedure is presented here. The implementation of the procedure is discussed in Section 5.2.2.

Chapter 3

Image Degradations

Since this project focuses on removing image degradations from an archive of images, it is necessary to develop the concept of an image degradation. The Second College Edition of the American Heritage Dictionary defines a degradation as a process of transition from a higher quality to a lesser quality. All image capturing processes involve various types of degradations. This chapter provides an overview of image degradations. It then focuses on image evaluation, and how this applies to this study. Image evaluation is the technique used to determine what degradations are present in the handwriting images. These degradations are presented visually and discussed.

3.1 Overview of Image Degradations

Image degradations reduce the quality of an image with respect to some purpose. They may be deterministic or nondeterministic. Deterministic degradations can be specified by specifying the characteristics of the systems that cause them. One example of a deterministic degradation is image blur. In general, image blur may result from many processes including motion or lens misfocus [Ekstrom, 1984]. The result of a nondeterministic imaging system cannot be uniquely specified by the system and its input. Nondeterministic degradations are difficult to model. One example of a nondeterministic degradation is an image degraded by atmospheric turbulence.

3.2 Evaluating the Degradations Present in the Handwriting

Image evaluation is the process of examining and judging image quality. For image processing, image evaluation involves identifying degradations and specifying their causes. Harris states that a necessary requirement for meaningful image evaluation is a specification of the purpose for which the images are recorded. With a definite statement of purpose established, image quality can be truly measured by the extent to which images serve this purpose [James L. Harris, 1966]. The handwriting images in our library have been recorded for the explicit purpose of the dissemination of rare documents. Therefore, image quality for this study is measured qualitatively with respect to this purpose.

3.2.1 Observed Degradations

Image evaluation reveals several degradations of concern in this study. These include the edge blur and ringing effect, additive noise, and lost information. The edge blur and ringing effect are likely introduced by the scanner; therefore, they are modeled with the imaging system. Additive noise is inherent in imaging systems due to various sources such as sensor noise or quantization noise. Noise is estimated in Section 4.1.1 because the image restoration technique used is sensitive to noise. Lost image features can only be recovered using local context. For example, the gaps appearing in the handwriting strokes are evidenced only by the shape of the stroke segments on either side of the gap. This requires *a priori* knowledge of the physical handwriting process.

The combined effect of these degradations is to reduce image quality. Figure 3.1 is a large portion of an image in the library presented to show the overall effects of the degradations to a reader.

CHAPTER 3. IMAGE DEGRADATIONS

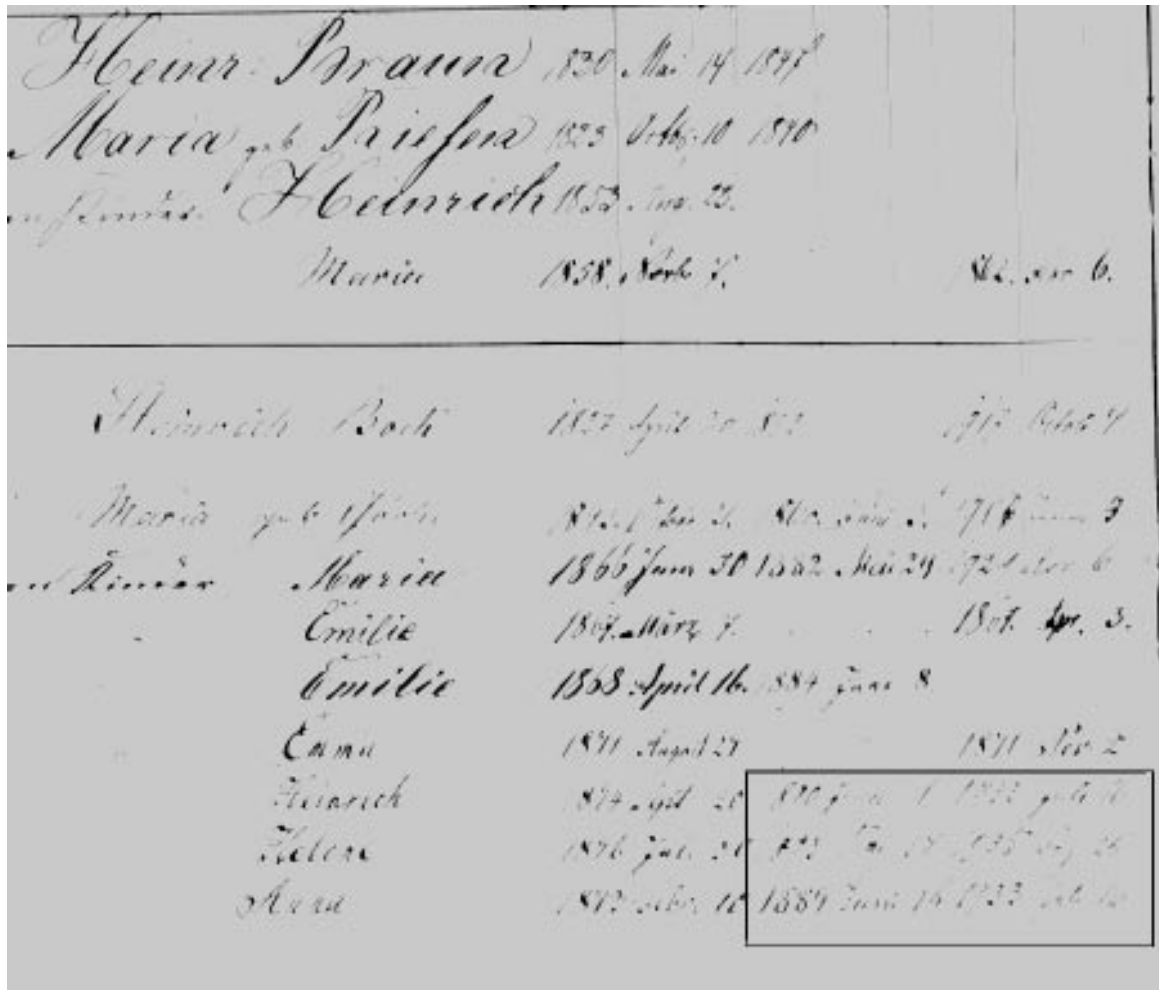


Figure 3.1: Sample document illustrating the poor image quality in the degraded images

CHAPTER 3. IMAGE DEGRADATIONS

Figure 3.2 is a magnified portion of Figure 3.1 presented to show the edge ringing and stroke fragmentation. Figure 3.3 provides a different view of the ringing effect around stroke edges. This

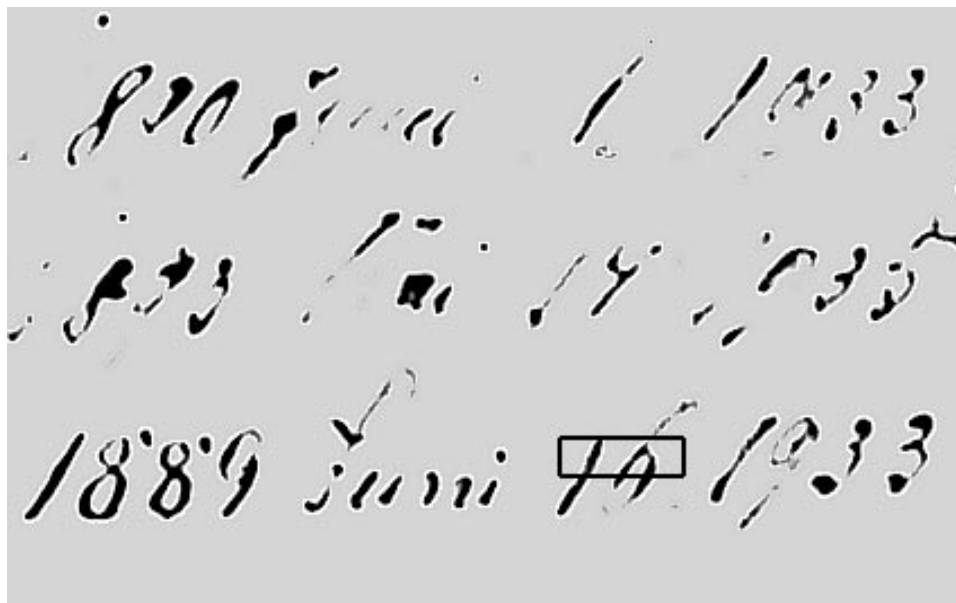


Figure 3.2: Magnified portion of a sample document showing the edge ringing and missing strokes.

presents an intensity profile of a scan line that crosses through the small boxed region in Figure 3.2. The dotted line corresponds to a scan line that does not intersect stroke information.

Before continuing, it is important to understand the process that introduced these degradations. First, the original documents have been captured or recorded on photographic film. The sensors used to record the image data restrict the recording to a limited range of energy intensities. This limited range depends upon the film exposure curve. As a result of this restriction, energy levels outside this range are lost. In this project, this recording onto film is an unknown process; since there is no way to determine the film used, light levels, or the development process, there is no way to model these degradations. Second, the digitization of the film data using a Polaroid scanner introduces noise and systematic degradation. These processes are represented in Figure 3.4.

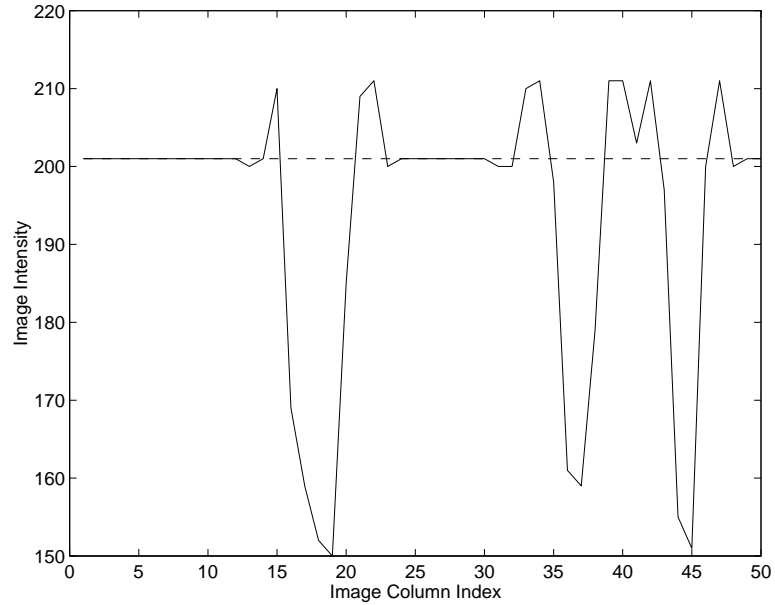


Figure 3.3: Graph of scan lines intersecting handwriting strokes taken from Figure 3.2

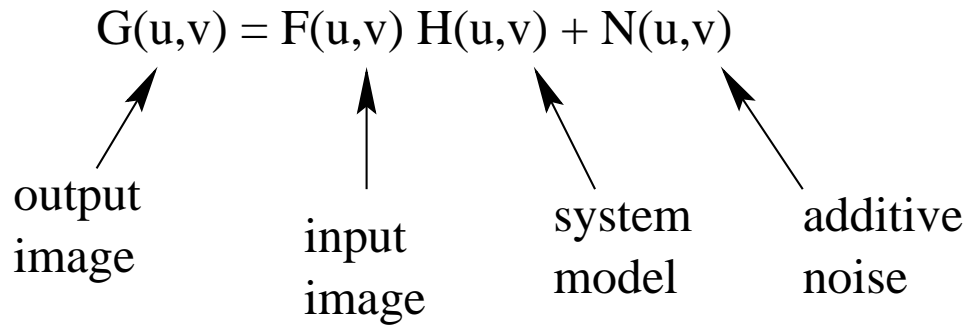


Figure 3.4: Imaging systems used to create the digital handwriting images. Transformation $T1$ corresponds to the recording of image data onto photographic film. Transformation $T2$ corresponds to digitization of the film data by a Polaroid scanner.

CHAPTER 3. IMAGE DEGRADATIONS

This process is modeled as a linear position invariant imaging system. The system can be simplified using the property of additivity by considering both transformations as one process.

If both the input and output of this process were available, the process could be modeled. No input-output pairs are available in this study, so assumptions will have to be made in order to build a model. If an accurate model can be realized, some degradations introduced in this process can be removed. Others, such as quantization noise that results from compressing a scene into a fixed intensity range, are irreversible.

3.2.2 Strategy for Removing the Observed Degradations

Many techniques are available to remove image degradations. The first technique to be investigated is a linear restoration technique called inverse filtering. This technique is chosen because in images with no noise and well defined models, it provides perfect restoration. Since the handwriting images contain low noise levels, it might be expected that this technique will perform well. The second approach is designed explicitly for the handwriting images in hope that domain knowledge can be used to produce good results.

Chapter 4

Filtering Techniques

This chapter discusses two specific filtering approaches for removing the ringing and defocusing of edges around each handwriting stroke. The candidate approaches that are investigated include inverse filtering and nonlinear filtering. The theory of each technique is discussed briefly. Then, the specific implementation is discussed, followed by the results of applying this implementation to the images being studied.

4.1 Inverse Filter

Inverse filtering is the most direct form of degradation removal. Image degradations are removed from an image by inverting the transformation that introduced them. This process relies on two assumptions. First, the transformation resulting in the degradation must be specified. Second, this transformation must be invertible. Inverse filtering can be described mathematically in the Fourier domain using the noiseless imaging system model

$$G(u, v) = H(u, v)F(u, v). \quad (4.1)$$

Here, F is the original image, H is the degradation process, G is the degraded image, and (u, v) gives the coordinates in the frequency domain. Using this model, F is obtained from G by computing

$$F(u, v) = \frac{G(u, v)}{H(u, v)}. \quad (4.2)$$

CHAPTER 4. FILTERING TECHNIQUES

The corresponding corrected image is calculated by taking the two-dimensional inverse Fourier transform of F [Gonzalez and Woods, 1992]. The inverse filter is given by

$$\frac{1}{H(u, v)} \tag{4.3}$$

where the notation denotes an element-by-element inversion of H . That is, each element of Equation 4.3 is the reciprocal of the corresponding element in H . Inverse filtering can be formulated in the presence of noise by adding a noise term to Equation 4.2. This is given by

$$F(u, v) = \frac{G(u, v)}{H(u, v)} - \frac{N(u, v)}{H(u, v)}. \tag{4.4}$$

The inverse filter gives a perfect restoration in the absence of noise and in the absence of zero crossings in H . However, most images contain both noise and zero crossings. Zero crossings can cause two effects. First, if H contains zeros at particular frequencies, then the reciprocals cannot be computed at these points. Second, in areas where H just approaches zero, inherent noise can dominate the restoration because these areas in F approach infinity [Gonzalez and Woods, 1992]. Many modifications have been suggested to overcome the inadequacies of the inverse filtering method. However, these modifications are ad hoc and implementation-dependent [Sondhi, 1972].

Another problem with the inverse filter arises in this project. The technique depends on the calculation of H and can be sensitive to inaccuracies in this calculation. Because the input-output pairs of the imaging system are unavailable, our calculation of H must rely on assumptions that may not be valid. However, without these assumptions, H cannot be computed.

4.1.1 Implementation of Inverse Filtering

The first step in implementing inverse filtering is to compute H . Linear systems theory states that a linear imaging system is completely specified by its impulse response. Therefore, to compute H accurately, it is necessary only to collect its response to a single isolated impulse. However, since the input to our system is unknown, there is no way of determining where a single isolated impulse exists. Therefore, H must be estimated.

To estimate H , small local disturbances are extracted from background regions of image samples. Each sample is assumed to be the response to an isolated impulse. The ensemble collected consists of thirty 5×5 neighborhoods. The samples are averaged to reduce the effects of outliers. This is done by simply averaging each of the 25 pixel values across the thirty samples. The average

CHAPTER 4. FILTERING TECHNIQUES

background value is then subtracted from each pixel in the average response. The final step is to shift the response so that it has a mean value of one. This gives it the property that if convolved with a constant image, the image remains unchanged. This provides an estimate of the operator h in Equation 2.13. A cross-section of the result is given in Figure 4.1. The two-dimensional Fourier transform of h yields the calculated H .

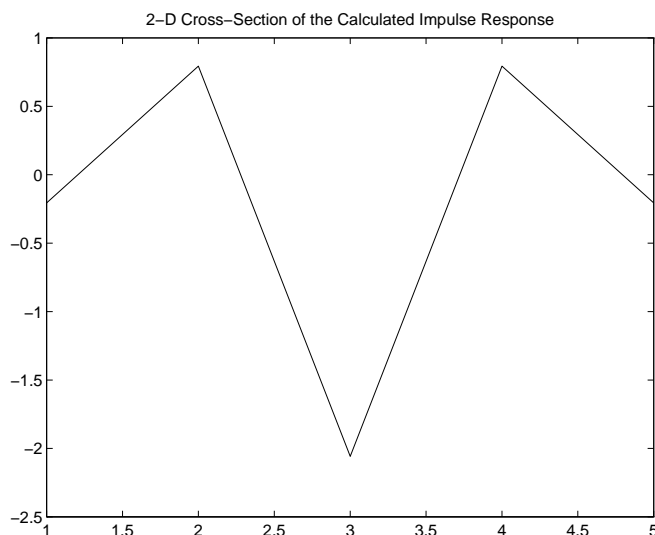


Figure 4.1: Cross section of the average impulse response

To support our assumption, the variance at each pixel position is computed after the samples are averaged. The results are given in Table 4.1. Note that variance is detected only at the center

0	0	0	0	0
0	0	0	0	0
0	0	1.2849	0	0
0	0	0	0	0
0	0	0	0	0

Table 4.1: Variance calculation of the random sample at each pixel position

pixel and is small considering the magnitude of the samples.

A final measure taken to determine if h is accurate is to apply it to sample data and compare

CHAPTER 4. FILTERING TECHNIQUES

the results to real samples. The results should contain the same degradations present in the real samples. Each of the images processed by the model is a portion of a real sample that has been manually altered to remove the edge defocus. Each altered image is convolved with h , and the result is compared to the original sample qualitatively. The results are given in Figure 4.2.

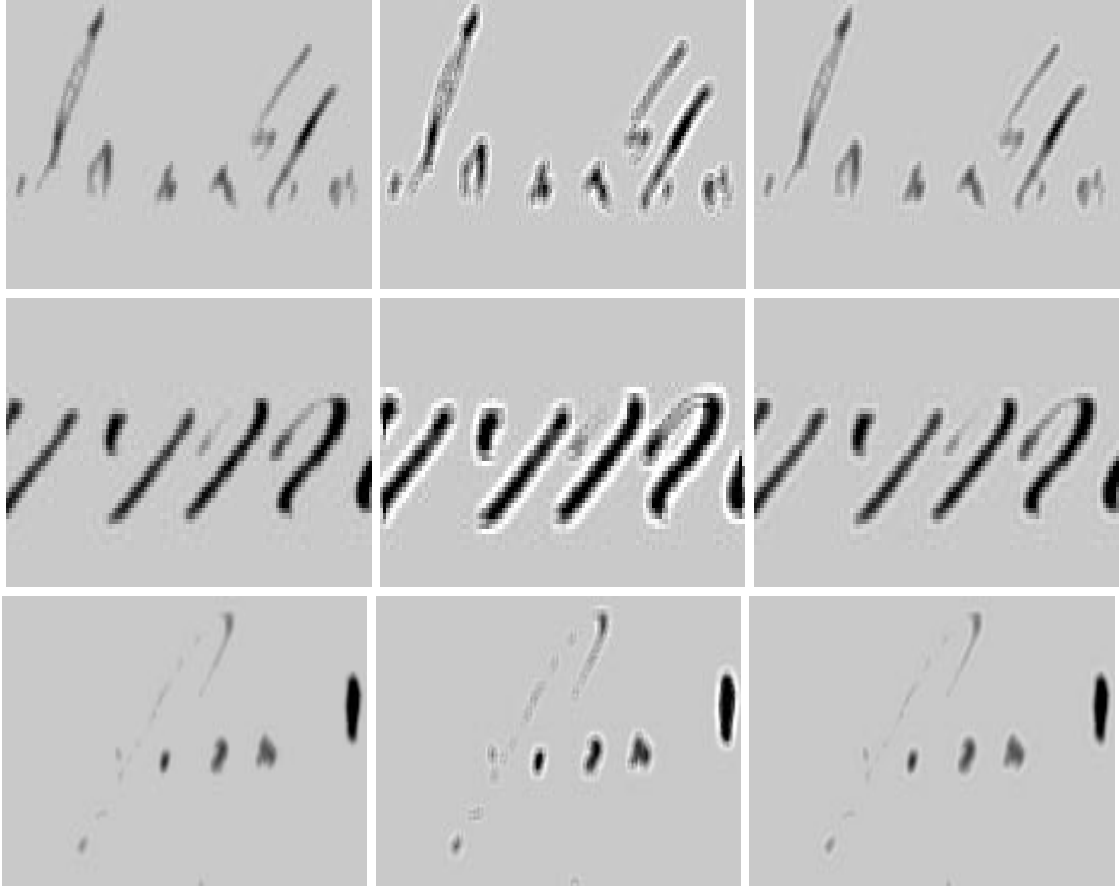


Figure 4.2: Results of passing images through the estimated model. The left column contains sample images with the ringing manually removed. The middle column contains output of the estimated degradation model. The right column contains the unaltered samples.

These results confirm that h introduces the edge defocus. However, the model does defocus the edges more strongly than the real system. This discrepancy is possibly the result of noise or inaccuracies in the scaling of h .

CHAPTER 4. FILTERING TECHNIQUES

Before constructing the inverse filter, an estimate of the additive noise is needed. To obtain this estimate, seven background sections of size 64×64 are extracted from the image library. Variance in gray levels within these sections is attributed to noise. An experiment is run to compute the variance in these samples. The results given in Table 4.1.1 suggest that the images contain a negligible amount

Sample	Image Mean
1	201.000000
2	200.993164
3	201.000000
4	200.936523
5	201.000000
6	200.993896
7	200.990967
Sample Mean	200.98779
Sample Variance	0.01709

Table 4.2: Results of noise estimation experiment.

of noise. Therefore, the additive noise term was not included in the inverse filter.

The Algorithm

Now that h has been computed, the inverse filter can be applied. The implementation used is a five step algorithm. In step one, the image data and h are retrieved and stored in matrix form. In step two, the two-dimensional Fourier transforms of the image data and h are computed giving G and H . The transform of h is padded with zeros so that G and H are the same size. The inverse filter is then calculated by performing an element-by-element inversion of H ; handling of singularities is described later. In step four, the inverse filter is applied to the transformed image G . This operation is an element-by-element multiplication resulting in the two-dimensional Fourier transform of the restored image. Thus, the fifth step is to compute the two-dimensional inverse Fourier transform of

$$\frac{G(u, v)}{H(u, v)}. \quad (4.5)$$

This implementation is written in the *Matlab* scripting language and is run using *Matlab* version 4.2c. *Matlab* is a mathematical tool that operates on matrices and is therefore ideal for image processing. Further, it offers excellent graphing capabilities which allow for visualization of image data. The script used to implement the inverse filtering algorithm is given in Appendix A.1.

4.1.2 Results of Inverse Filtering

Figures 4.3 - 4.5 show the results of attempting restoration using inverse filtering. The Fourier transform of h has many zero crossings which indicates that the restoration will not be ideal. The magnitude of this transform is shown in Figure 4.3. In areas where this transform approaches zero,

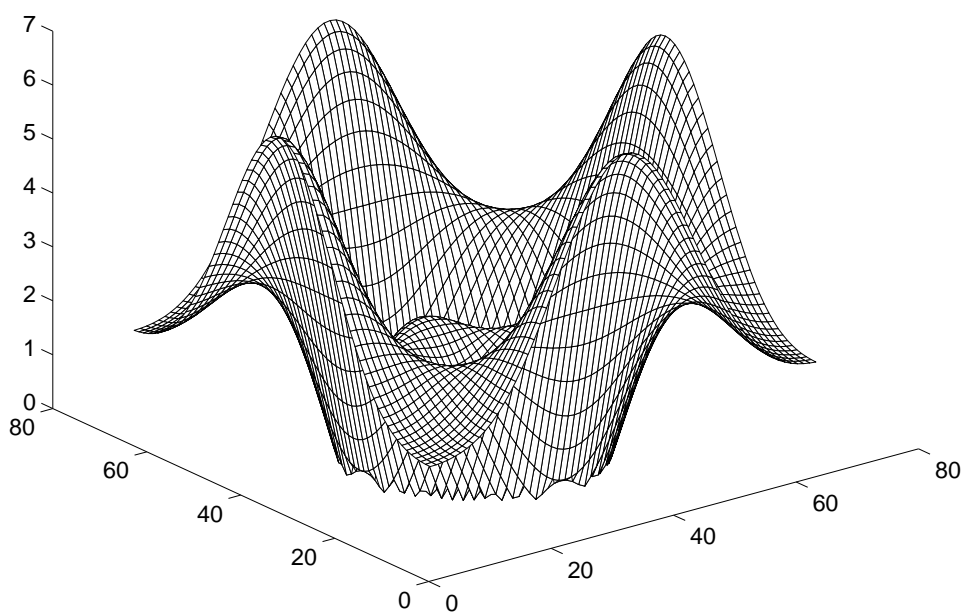


Figure 4.3: Magnitude of the two-dimension Fourier transform of the estimated degradation model

the inversion calculation will result in large peaks that will dominate the restoration. In areas where the transform is zero, the inverse filter cannot be directly applied. The magnitude of the calculated inverse filter is shown in Figure 4.4.

Figure 4.5 shows the results of applying the inverse filter to some sample images. The spikes in the inverse filter caused by zero crossings dominate the restoration filter causing severe ringing to be introduced.

CHAPTER 4. FILTERING TECHNIQUES

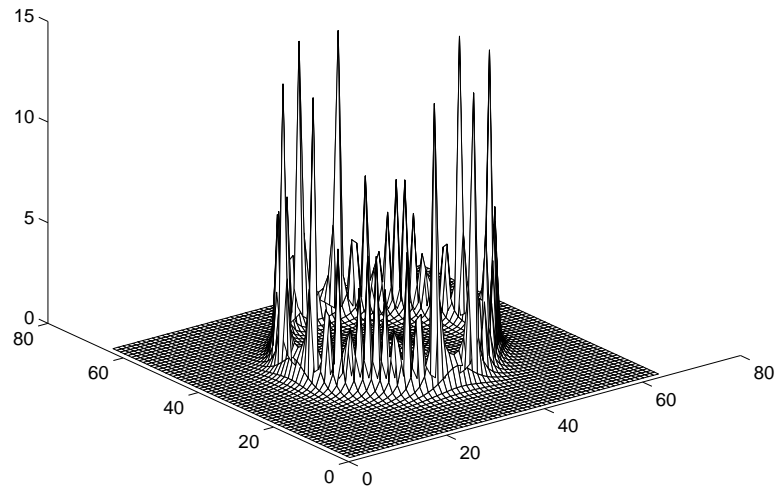


Figure 4.4: Magnitude of the inverse filter.



Figure 4.5: (Left) Sample image from the document library. (Right) Result of applying the inverse filter to the sample.

CHAPTER 4. FILTERING TECHNIQUES

To reduce this effect, two ad hoc techniques are attempted. First, the inverse filter is applied to sample image transforms at all frequencies other than the ones where spikes exist. Logically, this technique consists of examining each frequency's magnitude to determine if it is above a threshold. If it is above the threshold, then the magnitude is set to one, and the phase is left unchanged. Otherwise, the component is left unaltered. The experimental threshold value t is limited to the magnitude range $1 \leq t \leq 10$. This technique results in an inverse filter that passes certain image frequencies without performing any filtering. With this modification, the inverse filter did remove the ringing that was previously introduced, but it did not remove the edge defocus. The second technique applied consists of smoothing the inverse filter with the transform of a Gaussian function of the form

$$gauss(x, y) = e^{-\lambda^2 \pi (x^2 + y^2)}. \quad (4.6)$$

This technique removes the edge defocus without introducing ringing as λ approaches zero. However, this technique also smoothes the image so that foreground structures are reduced or completely eliminated.

Despite these efforts, the inverse filter did not improve the results substantially, and the technique was abandoned. It is very likely that the inverse filter is extremely sensitive to the estimated H . Since physical access to the Polaroid scanner was not available, it was not possible to obtain a better characterization of the scanner behavior.

4.2 Nonlinear Filtering

The second filtering technique that is investigated is nonlinear filtering. Nonlinear filtering is a spatial filtering technique that operates on neighborhoods about a pixel. However, these operations do not necessarily compute the usual weighted sum about the target pixel as do linear spatial filters. For example, a typical linear 3×3 spatial filter given by

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

computes a result about the target pixel with the formula

$$t = w_1 x_1 + w_2 x_2 + \dots + w_9 x_9 \quad (4.7)$$

CHAPTER 4. FILTERING TECHNIQUES

where the w_i are weights for each pixel in a local neighborhood, and x_i are the actual pixel values. [Gonzalez and Woods, 1992]. Typical examples include median, maximum value, or minimum value computations in a local neighborhood. Nonlinear filtering can be used to produce specific results such as noise reduction or gray level modification, but without any guidance from the extensive theory of linear systems.

4.2.1 Implementation of Nonlinear Filtering

The nonlinear filter for this project is designed to remove the edge ring apparent in the handwriting images. To accomplish this, the filter must be capable of distinguishing between foreground pixels and background pixels. Foreground pixels are defined as pixels contained within a handwriting stroke. Background pixels are then defined as pixels that are not contained in a handwriting stroke. Using this definition, all pixels comprising the edge ring should be classified as background. The second task that the filter must perform is to compute the gray level of the background in a local area of each pixel. This is used to set any pixel classified as background to the estimated background intensity. If the classification technique and the background estimator are accurate, the edge ring will be eliminated by this technique.

Developing a Background Estimator

After much experimentation, a method of estimating the background intensity at each image coordinate from a local neighborhood around that coordinate was determined. Using this procedure, a “background image” was computed against which image pixels could be compared. With images such as ours with uniform background intensity, we could measure the quality of the estimator by computing the variance of this “background image”. Three different techniques for estimating the background values were attempted, and these are described next.

1. *Local Neighborhood Mean*

A background pixel is the mean of an $n \times n$ block centered about a given pixel.

2. *Local Neighborhood Median*

A background pixel is the median of an $n \times n$ block centered about a given pixel.

3. *Local Median of Local Means*

First, each pixel is replaced by a local $n \times n$ block mean as in Method 1. Then, using this

CHAPTER 4. FILTERING TECHNIQUES

image, each pixel is replaced by a local $m \times m$ block median as in Method 2.

Experiment 1 evaluates all three potential estimators using window sizes ranging from 7 to 15, noise with a standard deviation ranging from 10 to 30, and noise distributions varying from uniform to exponential. The noise is uncorrelated additive noise manually added to sample images using *Adobe Photoshop version 3.0*. Note that in the trial computing the local median of local means, the window varied is the averaging window. The median window is held constant at 35×35 pixels. Section A.8 gives the source code for the driver program for this experiment.

Table 4.3 gives the results of the experiment trial using the mean as the estimator. These results

Win Size	Std. Dev. of Noise	Noise Dist.	Mean	Var.
7	10	Uniform	195.087673	275.968164
7	10	Gaussian	195.264572	277.863674
7	30	Uniform	195.031244	233.850649
7	30	Gaussian	195.819333	277.777687
15	10	Uniform	195.084583	122.349548
15	10	Gaussian	195.285100	123.023684
15	30	Uniform	195.064175	124.619340
15	30	Gaussian	195.755221	115.078105

Table 4.3: Results from Experiment 1 using the local mean as the background estimator.

indicate that the mean statistic is not a good estimator of the image background in local regions in the presence of noise. The variance is significant, and the mean is five to six levels removed from the actual background value.

Table 4.4 presents the results of the experiment trial using the local median as the background estimator. These results indicate that as the median window size increases, the estimator improves. We concluded that we should experiment with larger windows.

Table 4.5 presents the results of the experiment trials using the local median of local means as the background estimator. The results show that increasing the size of the averaging window reduces the quality of the estimator. It is concluded that the size of the median window should be varied while the averaging window is held small. This situation is evaluated in the next experiment.

Experiment 2 evaluates the quality of the local median of local means as a background estimator when the averaging window size ranges from 3 to 5, the median window size ranges from 25 to 50, and the standard deviation of the noise ranges from 0 to 30. Section A.9 gives the source code for

CHAPTER 4. FILTERING TECHNIQUES

Win Size	Std. Dev. of Noise	Noise Dist.	Mean	Var.
7	10	Uniform	197.115672	216.195392
7	10	Gaussian	197.224771	219.063811
7	30	Uniform	196.327521	233.850649
7	30	Gaussian	197.286251	261.044868
15	10	Uniform	199.521440	13.810453
15	10	Gaussian	199.534854	17.606693
15	30	Uniform	198.017527	35.692682
15	30	Gaussian	198.806617	47.861591

Table 4.4: Results from Experiment 1 using the local median as the background estimator.

Win Size	Std. Dev. of Noise	Noise Dist.	Mean	Var.
7	10	Uniform	200.190786	2.698594
7	10	Gaussian	200.193902	2.688266
7	30	Uniform	199.636661	3.438942
7	30	Gaussian	199.884581	3.188450
15	10	Uniform	197.850427	25.880658
15	10	Gaussian	197.946687	25.552418
15	30	Uniform	197.642099	25.076787
15	30	Gaussian	198.220148	21.488128

Table 4.5: Results from Experiment 1 using the local median of local means as the background estimator.

the driver program for this experiment. Table 4.6 gives the results of Experiment 2. This data shows that the estimator performs inconsistently. For low levels of noise, the estimator improves as the averaging window size increases. For significant levels of noise, the estimator gets worse as the averaging window size increases.

Experiment 3 evaluates the local median estimator where the median window size ranges from 25 to 75, and the standard deviation of the noise ranges from 0 to 30. Section A.10 gives the source code for the driver program for this experiment. The results are given in Table 4.7. This technique is chosen as the background estimator for the nonlinear filter for several reasons. It outperforms the other methods investigated, and it performs consistently for different noise levels. It is also conceptually simple. The background is estimated using a sufficiently large window about any pixel; we selected this technique using a 31×31 pixel window.

CHAPTER 4. FILTERING TECHNIQUES

Ave Win Size	Med Win Size	Std. Dev. of Noise	Mean	Var.
3	25	0	200.934730	0.069662
3	25	10	200.584798	1.126294
3	25	30	199.816410	4.303710
3	50	0	200.788694	1.849442
3	50	10	200.176928	5.940212
3	50	30	199.700335	7.441408
5	25	0	201.000000	0.000000
5	25	10	200.562754	0.245715
5	25	30	199.578359	0.976074
5	50	0	201.000000	0.000000
5	50	10	200.557121	0.248671
5	50	30	200.028616	0.849588

Table 4.6: Results of Experiment 2 which investigates the local median of local means as a potential background estimator.

Background/Foreground Classification

Given a background image and the original image, the next step is to classify each pixel as background or foreground, based on local context. Because of the potential size of local context windows, we sought a simple linear classification technique for determining background or foreground. Since the classifier needed to distinguish between two suspected linear separable sets, a perceptron model was implemented to produce a decision function. The decision function approach maintains simplicity in that it encapsulates all logic needed to distinguish between the two sets with a simple linear function.

A perceptron is a learning machine that has been trained to distinguish patterns. Patterns that are known members of the partitioned classes are called training patterns, and a set from each class is known as a training set. The process of using a training set to obtain a decision function is called training [Gonzalez and Woods, 1992].

Our two pattern class perceptron is a learning machine in its simplest form. The machine learns a decision function that dichotomizes two linearly separable sets [Gonzalez and Woods, 1992]. Figure 4.6 illustrates the perceptron in this simple form. The response of the perceptron is generated from

CHAPTER 4. FILTERING TECHNIQUES

Med Win Size	Std. Dev. of Noise	Mean	Var.
25	0	200.984865	0.014907
25	10	199.944780	2.106904
25	30	198.437782	14.565045
51	0	201.000000	0.000000
51	10	200.089560	0.621428
51	30	198.707713	5.371683
75	0	201.000000	0.000000
75	10	200.155466	0.308999
75	30	198.959669	2.886587

Table 4.7: Results of Experiment 3 which investigates the local median as a potential background estimator.

a weighted sum of the inputs. This calculation is called the decision function and is given by

$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1}. \quad (4.8)$$

The weighting coefficients w_i modify the inputs and send the result to a threshold element called the *activation function*. The activation function maps its input to the final output. When $d(\mathbf{x}) > 0$, this indicates that pattern \mathbf{x} belongs to class ω_1 . When $d(\mathbf{x}) < 0$, this indicates that pattern \mathbf{x} belongs to class ω_2 . When $d(\mathbf{x}) = 0$, this indicates that pattern \mathbf{x} lies on the decision surface separating the two classes causing the result to be indeterminate. The decision boundary implemented by a particular perceptron can be obtained by setting Equation 4.8 equal to zero.

$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1} = 0 \quad (4.9)$$

The output of the activation element depends on the polarity of the decision function output. Typically, the output is given by

$$O = \begin{cases} +1 & \text{if } d(\mathbf{x}) > 0 \\ -1 & \text{if } d(\mathbf{x}) < 0 \end{cases} \quad (4.10)$$

To derive training sets for each class, a one-dimensional 3-pixel pattern is built from the target pixels and its horizontal neighbors. Any pixel is classified as background if the 3-pixel pattern centered at the target pixel contains at least two members with an intensity level above or equal to

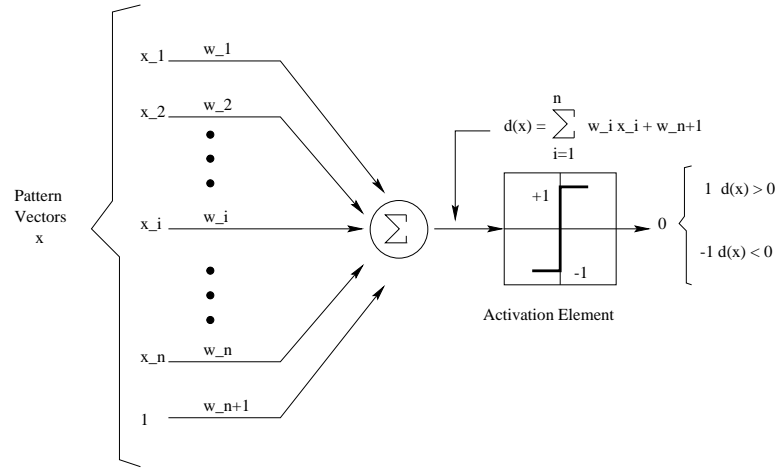


Figure 4.6: Perceptron for two pattern classes.

the estimated background at this point. The pixel is classified as foreground if the 3-pixel pattern contains at least two members with an intensity level below the estimated background at the center point. Because of the wide range of intensity values in the image, a training set described above would be enormous if all potential intensities fitting the description were included. In an effort to reduce this complexity, the patterns are constructed as binary patterns. This fits our needs logically as the actual pixel values are irrelevant to the classifier. It simply needs to know if a pixel is above the estimated background or below it regardless of how far above or below. Therefore, the patterns are constructed with a 1 to represent a pixel with intensity greater or equal to the estimated background, and with a -1 to represent a pixel with intensity less than the estimated background.

The training patterns constructed are then augmented with a $(n + 1)^{st}$ element which is always equal to one. This augmentation is simply a step in the training procedure. The training set for the background class is given by

$$\omega_1 = \{(1, 1, 1, 1)^T, (1, 1, -1, 1)^T, (-1, 1, 1, 1)^T\}. \quad (4.11)$$

The training set for the foreground class is given by

$$\omega_2 = \{(1, -1, -1, 1)^T, (-1, -1, -1, 1)^T, (-1, -1, 1, 1)^T\}. \quad (4.12)$$

The decision function is generated with the following procedure. Let $\mathbf{w}(1)$ be the initial weight vector for two training sets of augmented patterns belonging to pattern classes ω_1 and ω_2 . The

CHAPTER 4. FILTERING TECHNIQUES

vector $\mathbf{w}(1)$ may be arbitrarily chosen. At the k^{th} iteration, if $\mathbf{y}(k) \in \omega_1$ and $\mathbf{w}^T(k)\mathbf{y}(k) \leq 0$, replace $\mathbf{w}(k)$ by

$$\mathbf{w}(k+1) = \mathbf{w}(k) + c\mathbf{y}(k) \quad (4.13)$$

where c is a positive correction increment. If $\mathbf{y}(k) \in \omega_2$ and $\mathbf{w}^T(k)\mathbf{y}(k) \geq 0$, replace $\mathbf{w}(k)$ by

$$\mathbf{w}(k+1) = \mathbf{w}(k) - c\mathbf{y}(k). \quad (4.14)$$

Otherwise, leave $\mathbf{w}(k)$ unchanged; that is

$$\mathbf{w}(k+1) = \mathbf{w}(k). \quad (4.15)$$

This training algorithm only modifies \mathbf{w} if a pattern from the training set is misclassified. The algorithm converges when both training sets are cycled through the algorithm, and no changes are made.

For the training sets given in Equations 4.11 and 4.12, generating the decision function takes only twelve iterations when the correction increment is set to one, and the initial weight vector $\mathbf{w}(0)$ is chosen to be the zero vector. The training process is illustrated below.

$$\begin{array}{ll} \mathbf{w}^T(0)\mathbf{y}(0) = 0 & \mathbf{w}(1) = (1, 1, 1, 1)^T \\ \mathbf{w}^T(1)\mathbf{y}(1) = 0 & \mathbf{w}(2) = (2, 0, 0, 2)^T \\ \mathbf{w}^T(2)\mathbf{y}(2) = 0 & \mathbf{w}(3) = (1, 1, 1, 3)^T \\ \mathbf{w}^T(3)\mathbf{y}(3) = 2 & \mathbf{w}(4) = (0, 2, 2, 2)^T \\ \mathbf{w}^T(4)\mathbf{y}(4) = -2 & \mathbf{w}(5) = \mathbf{w}(4) \\ \mathbf{w}^T(5)\mathbf{y}(5) = 2 & \mathbf{w}(6) = (1, 3, 1, 1)^T \\ \mathbf{w}^T(6)\mathbf{y}(0) = 5 & \mathbf{w}(7) = \mathbf{w}(6) \\ \mathbf{w}^T(7)\mathbf{y}(1) = 4 & \mathbf{w}(8) = \mathbf{w}(7) \\ \mathbf{w}^T(8)\mathbf{y}(2) = 4 & \mathbf{w}(9) = \mathbf{w}(8) \\ \mathbf{w}^T(9)\mathbf{y}(3) = -2 & \mathbf{w}(10) = \mathbf{w}(9) \\ \mathbf{w}^T(10)\mathbf{y}(4) = -4 & \mathbf{w}(11) = \mathbf{w}(10) \\ \mathbf{w}^T(11)\mathbf{y}(5) = -2 & \mathbf{w}(12) = \mathbf{w}(11) \end{array}$$

CHAPTER 4. FILTERING TECHNIQUES

The decision function generated is given by Equation 4.16.

$$d(\mathbf{x}) = x_1 + 3 * x_2 + x_3 + 1 \quad (4.16)$$

Now that these two techniques are in place, the implementation of the nonlinear filter is straightforward. First, an image containing the calculated background value at every pixel is generated using the median technique from Experiment 3. Next, for each pixel (x, y) in the input image, a pattern is derived by comparing the pixels $(x - 1, y)$, (x, y) , and $(x + 1, y)$ to the estimated median value centered at (x, y) . For each of these three pixels, if the intensity is greater than or equal to the median, then the pattern position is marked as positive with a 1. If the pixel value is less than the median, then the pattern position is marked as negative with a -1 . The pattern is then fed into the decision function given in Equation 4.16. A positive output from the decision function means that the pixel is a member of the background class ω_1 . Therefore, the pixel is set to the estimated background value from the median calculation centered at that point. A negative output from the decision function indicates that the pixel is a member of the foreground class ω_2 . Thus, the input pixel value at (x, y) is retained in the resulting image at the corresponding position. An indeterminate output results in retaining the input pixel value also. The implementation of this procedure is written in C++ using the classes described in the Appendix in Sections A.2 - A.4. The source code is given in the Appendix in Section A.11.

4.2.2 Results of Nonlinear Filtering

To assess the performance of the nonlinear filter, we apply it to three images with additive uniform uncorrelated noise with standard deviation from 0 to 30. Performance is measured qualitatively with respect to removing the edge defocus. Figure 4.7 presents the original samples side-by-side with the filtered results. The results show that the filter removed the edge defocus even in the presence of significant levels of noise. This suggests the method can be used throughout the document library regardless of noise levels. The technique must now be applied to full images from the document library to validate that the technique will perform uniformly on large images with varying brightness levels.

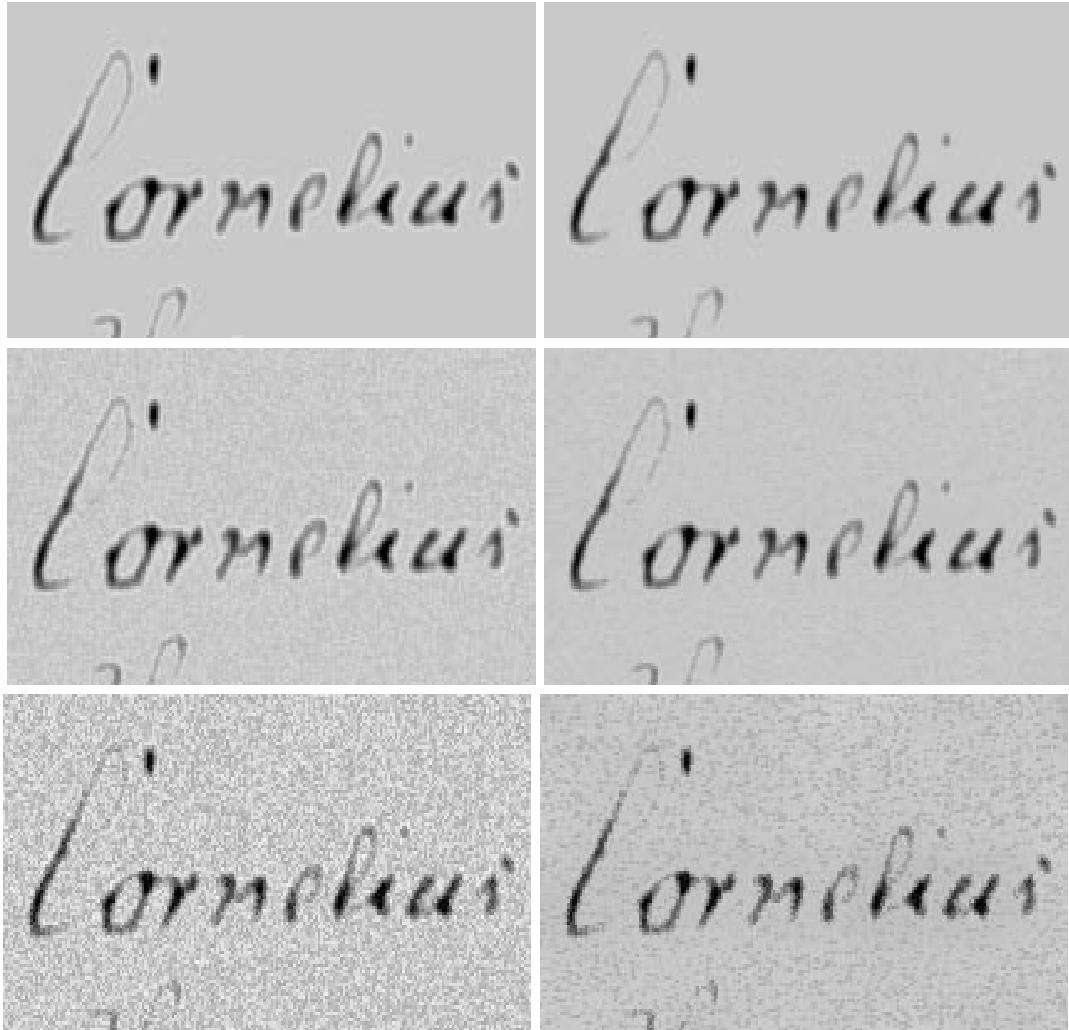


Figure 4.7: (*Left*) Sample images with standard deviations of noise equal to 0 (top), 10 (middle), and 30 (bottom). (*Right*) Result of applying the nonlinear filter to the sample on the left.

Chapter 5

Enhancement and Structural Filtering

The nonlinear filtering technique developed in Section 4.2.2 provides a preprocessing tool that prepares the images for the final enhancement and reconstruction methods used to improve the image quality. In this section, we apply contrast stretching, brightness adjustment, gray-scale image morphology, and gap filling in order to generate the final results. The merit of each step used is illustrated using the portion of Figure 5.1 that is enclosed in the boxed area. This portion is chosen because of its poor quality and lack of stroke data. These attributes make it an excellent sample to illustrate the benefits of each technique that is applied.

5.1 Contrast Stretching

The first step in producing the final results is to apply a contrast stretch to the results of the nonlinear filter. A contrast stretch produces an image of higher contrast by spreading out the intensities that are in the image. The implementation used takes a single input value t which is used to compute new intervals between intensities. If x denotes an image intensity, and m denotes the midtone of the image, then the contrast stretch calculates a new value for x , denoted by \hat{x} , with

$$\hat{x} = m + \left\lfloor \frac{(x - m)}{(100 - t) * 10^{-2}} \right\rfloor. \quad (5.1)$$

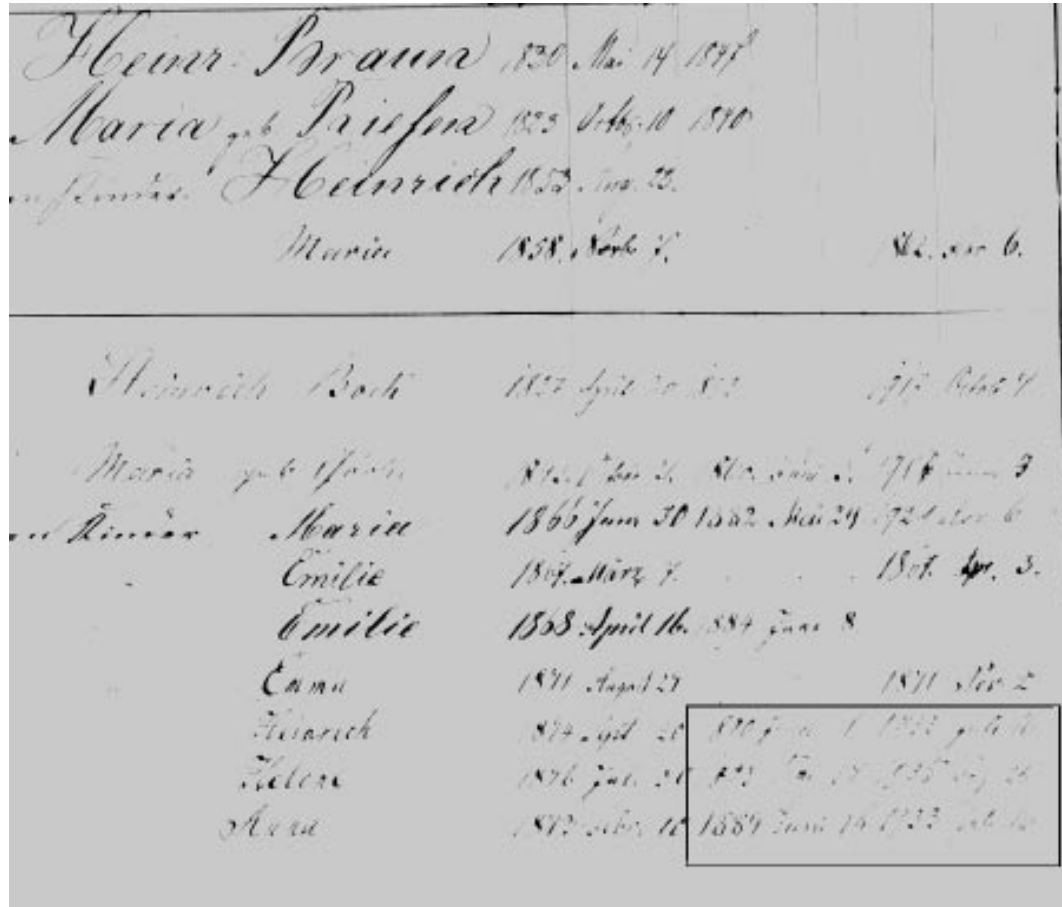


Figure 5.1: Sample image used to illustrate the effects of enhancement and reconstruction techniques.

CHAPTER 5. ENHANCEMENT AND STRUCTURAL FILTERING

The resulting values are maintained within the gray level scale of $[0, 255]$. If $t = 100$, then a threshold is computed at the midtone. The effect produced by applying this contrast stretch is to darken or strengthen stroke structures and to brighten the background. This also makes it apparent where stroke information has been lost.

Figure 5.2 demonstrates the effects of increasing t on the boxed region from Figure 5.1. The

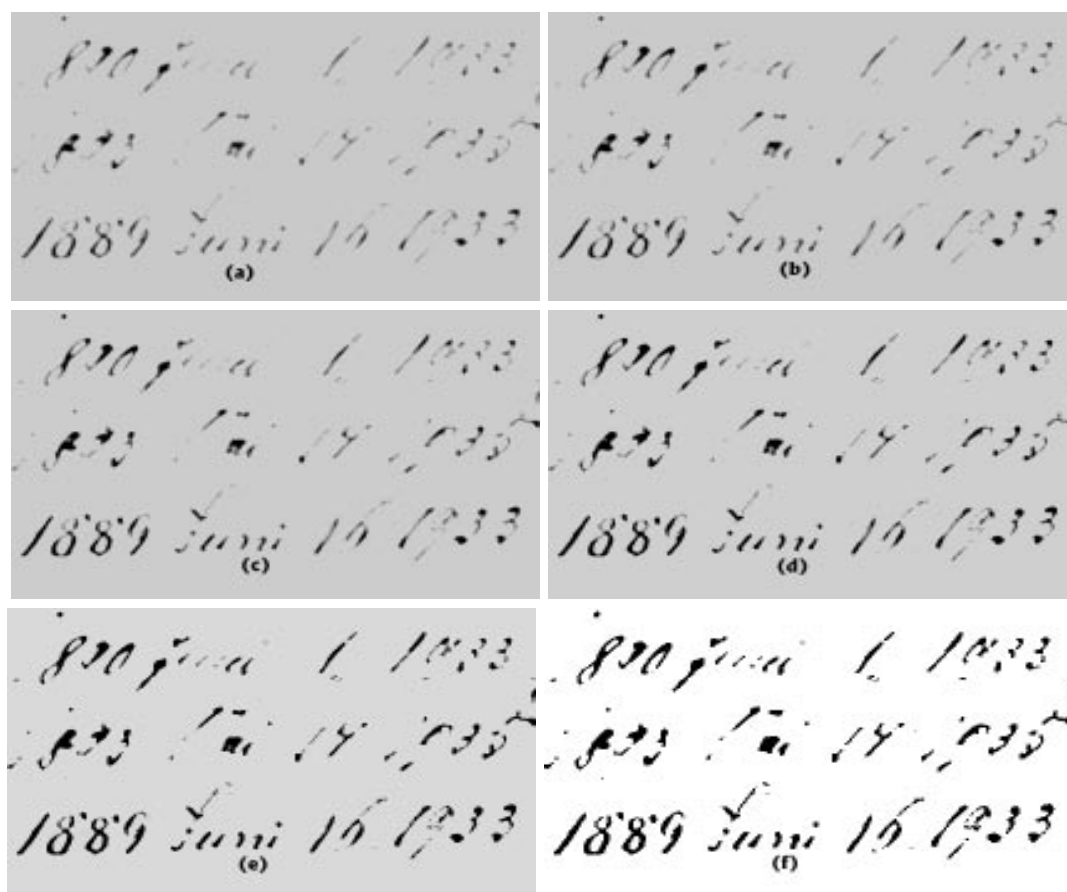


Figure 5.2: (a) Original sample taken from the boxed region in Figure 5.1. (b) Result with $t = 20$. (c) Result with $t = 40$. (d) Result with $t = 60$. (e) Result with $t = 80$. (f) Result with $t = 100$.

contrast stretch strengthens the foreground information. If the stretch is sufficiently large, then the image is reduced to a two-tone image. Since this drastic loss of gray levels is undesirable, t is chosen to prevent this from occurring. To compute the final results, $t = 80$. This offers a compromise

between strengthening the stroke information without such a drastic loss of gray levels.

5.2 Structural Filtering

The next step taken to produce the final results is to apply a structural technique to reconstruct lost foreground structures. Two techniques are considered. They include morphological closing and gap filling. Each is discussed next.

5.2.1 Morphological Closing

The morphological closing operation is applied to the output from the contrast stretch to attempt to reconnect fragmented strokes in areas with narrow breaks. The closing operation is stated mathematically in Equation 2.23. The first step in the closing operation is to compute a gray level erosion according to Equation 2.25. To prevent the technique from modifying the gray level tone of the handwriting strokes, the elements of the structuring element are set to zero. This reduces the erosion to a minimum calculation of the elements specified by the structuring element. The structuring element used is shown using a 7×7 window in Figure 5.3. This structuring element

					0	0
				0	0	0
			0	0	0	
		0	0	0		
	0	0	0			
0	0	0				
0	0					

Figure 5.3: Structuring element used to perform gray level erosion.

is chosen because the majority of the breaks span in the diagonal and horizontal directions. This element successfully reconstructs in both of these directions. The driver program for this operation is given in the Appendix in Section A.12.

The benefits of one level of erosion using this structuring element are demonstrated in Figure 5.4. Comparing the boxed regions reveals that some narrow breaks have indeed been reconnected. Multiple iterations will reconnect even more breaks; however, multiple iterations will also close hollow regions and thicken strokes. For this reason, only one iteration is used.



Figure 5.4: (Left) Result from the contrast stretch. (Right) Result of applying a gray level erosion to the image on the left.

Notice in Figure 5.4 that the erosion technique has significantly widened the handwriting strokes. To reduce this effect and complete the closing operation, dilation is applied to the results from the erosion stage. As with the erosion technique, the dilation defined in Equation 2.24 is constructed so that it does not alter the gray level tones of the handwriting strokes. This reduces the dilation operation to a maximum calculation of the image pixels corresponding to the structuring element. The structuring element used to perform the dilation technique is shown in Figure 5.5 using a 7×7 window. This structuring element is chosen so that strokes can be thinned without undoing the

			0			
		0	0	0		
			0			

Figure 5.5: Structuring element used to perform gray level erosion.

reconstruction achieved by the erosion stage. The driver program used to produce the dilation operation is given in the Appendix in Section A.13.

The results of the dilation operation are given in Figure 5.6. The boxed regions show that the strokes have been thinned without significant re-fragmentation. The resulting effect is to restore the strokes to roughly their original width.



Figure 5.6: (Left) Result from the erosion stage. (Right) Result of applying a gray level dilation to the image on the left.

5.2.2 Gap Filling

An alternative structural approach to the closing operation is to apply a gap filling filter. This filter examines local image context to determine if fragmentation exists. If so, then the gap is filled by interpolating the adjacent stroke data.

The filter is constructed using a 5×5 window. The image is processed by using the filter to examine the image data in three one-dimensional directions. The directions and their order of precedence are shown in Figure 5.7 using an a , b , and c . The pixel marked with an x is compared to

		c		a
b		x		b
a		c		

Figure 5.7: Filter used to in the gap filling operation.

the estimated background value which is obtained using the same background estimator developed for the nonlinear filter. If the value is equivalent to the estimated background, there is potential fragmentation. To determine if this is the case, the marked pixels are examined in order to determine if a stroke exists in one of these directions. A stroke exists if all a 's, b 's, or c 's are below the estimated background, and the normal to candidate direction contains all background values. Gaps are filled by interpolating the values of the a 's, b 's, or c 's corresponding to the direction where the stroke

is detected. The source code implementing this technique is presented in the Appendix in Section A.14.

The gap filling operation is applied iteratively to an image. The results of the gap filling filter after three iterations are presented in Figure 5.8. The gaps that are filled are set to maximum

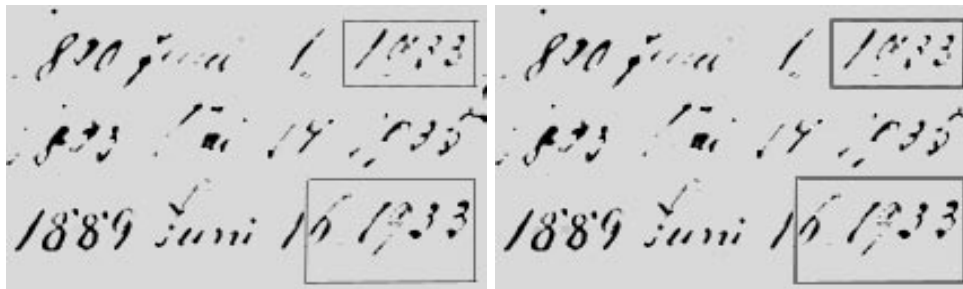


Figure 5.8: (*Left*) Result from the initial contrast stretch. (*Right*) Result of applying three iterations of the gap filling operation to the image on the left.

intensity to illustrate the benefits of this technique. If this technique were used instead of the closing operation, then these values would be interpolated using the stroke context on either side.

The closing operation and the gap filling technique provide comparable reconstructions. They do not provide complete reconstructions. Each technique must be applied to a relatively small window because larger windows connect structures incorrectly. This is a result of the physical structure of characters and cannot be avoided. For example, the “l” written in script intersects itself. If a large window is used in either technique, gaps will be filled above and below this intersection inappropriately. These situations are numerous over the character set, and they limit the reconstruction that can be performed.

5.3 Histogram Modification

The last step in producing the final results involves another contrast stretch and a brightness adjustment. This is applied to both the results of either the morphological closing operation and the gap filling operation. The contrast stretch is the same operation from Section 5.1 with $t = 80$. The brightness adjustment performs a linear shift to the image histogram. The linear shift moves the background value from the contrast stretched image back to the original background value of

the sample. For this reason, the actual shift is image dependent. Collectively, these enhancements restore the original appearance of the image despite the various processing steps applied. The results of these enhancements are given in Figures 5.9 and 5.10.



Figure 5.9: (Left) Result from the morphological closing. (Right) Result of contrast stretch and brightness adjustment applied to the figure on the left.

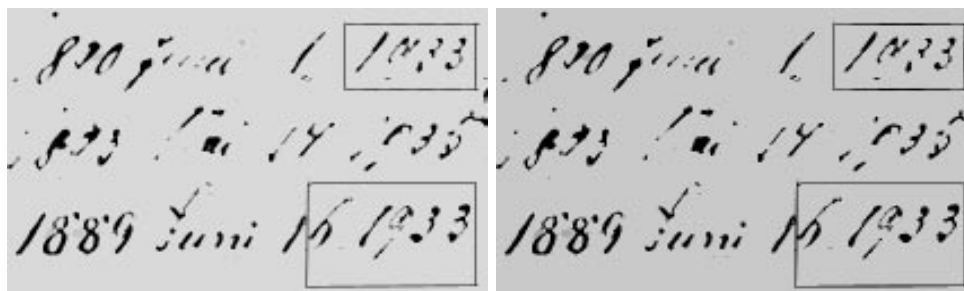


Figure 5.10: (Left) Result from the gap filling. (Right) Result of contrast stretch and brightness adjustment applied to the figure on the left.

5.4 Final Experiments

In order to demonstrate the capability of our image processing technique, we apply it to a larger data sample. We use the morphological closing operation as the structural technique because it appears to produce slightly better results. First, the nonlinear filter is applied to the samples. Next, a contrast stretch is applied with $t = 80$. The closing operation is applied to the results of the

CHAPTER 5. ENHANCEMENT AND STRUCTURAL FILTERING

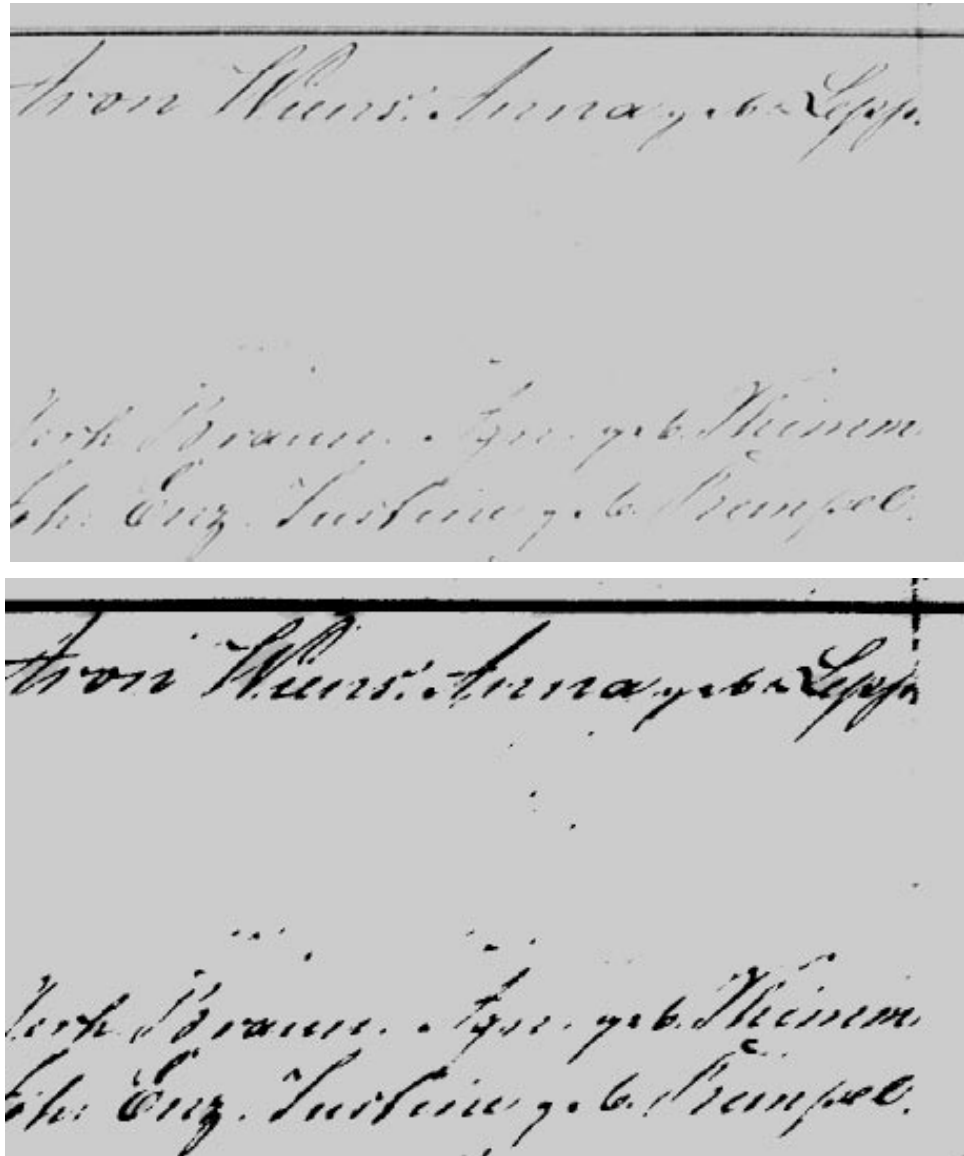


Figure 5.13: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

CHAPTER 5. ENHANCEMENT AND STRUCTURAL FILTERING

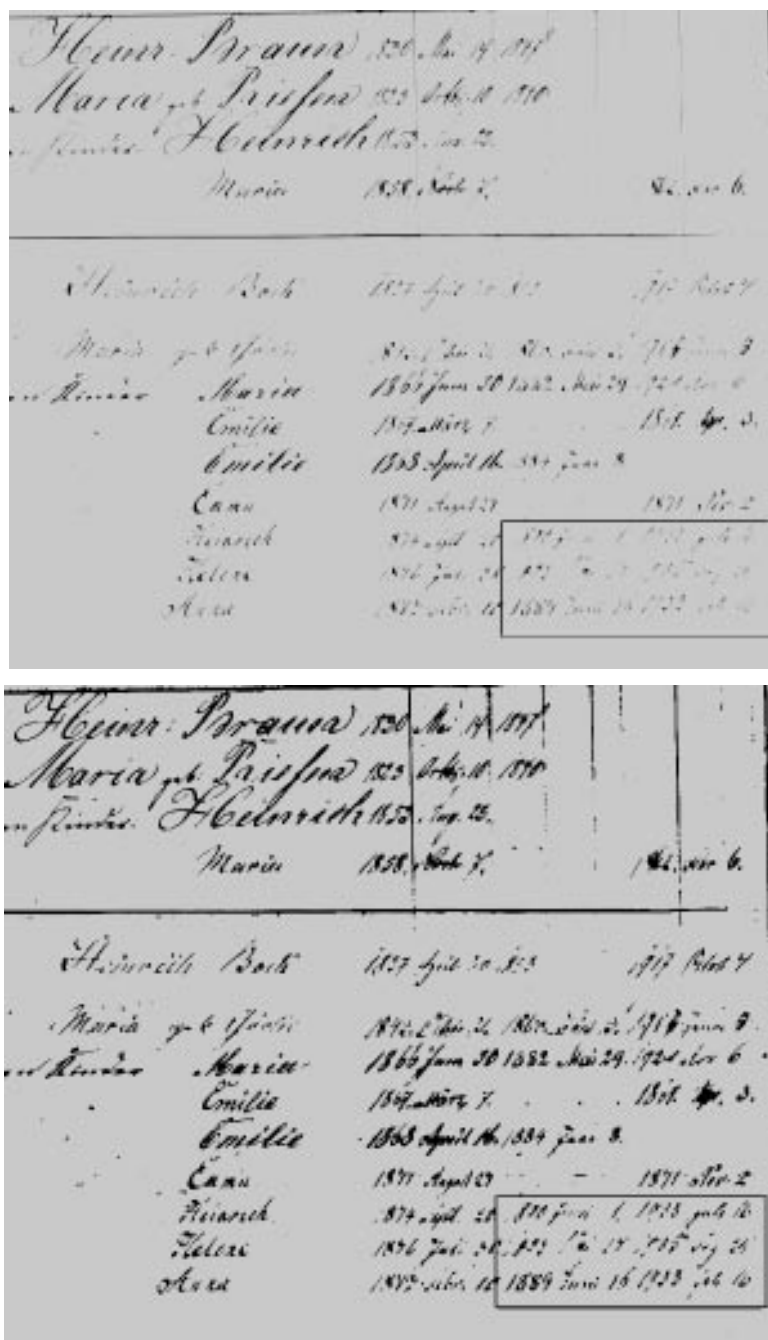


Figure 5.14: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

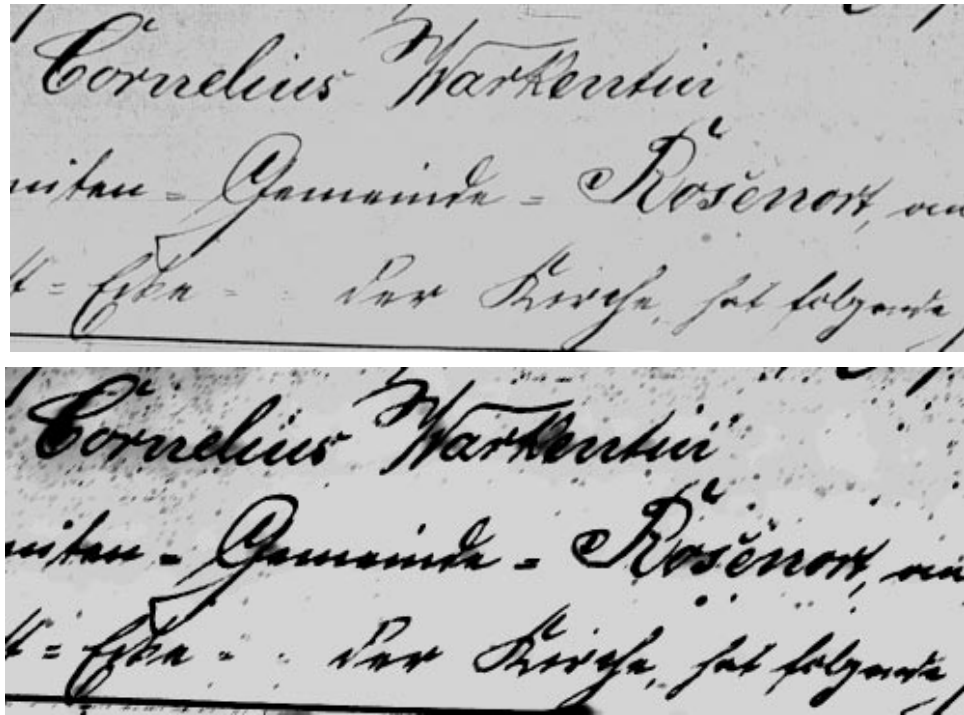


Figure 5.15: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

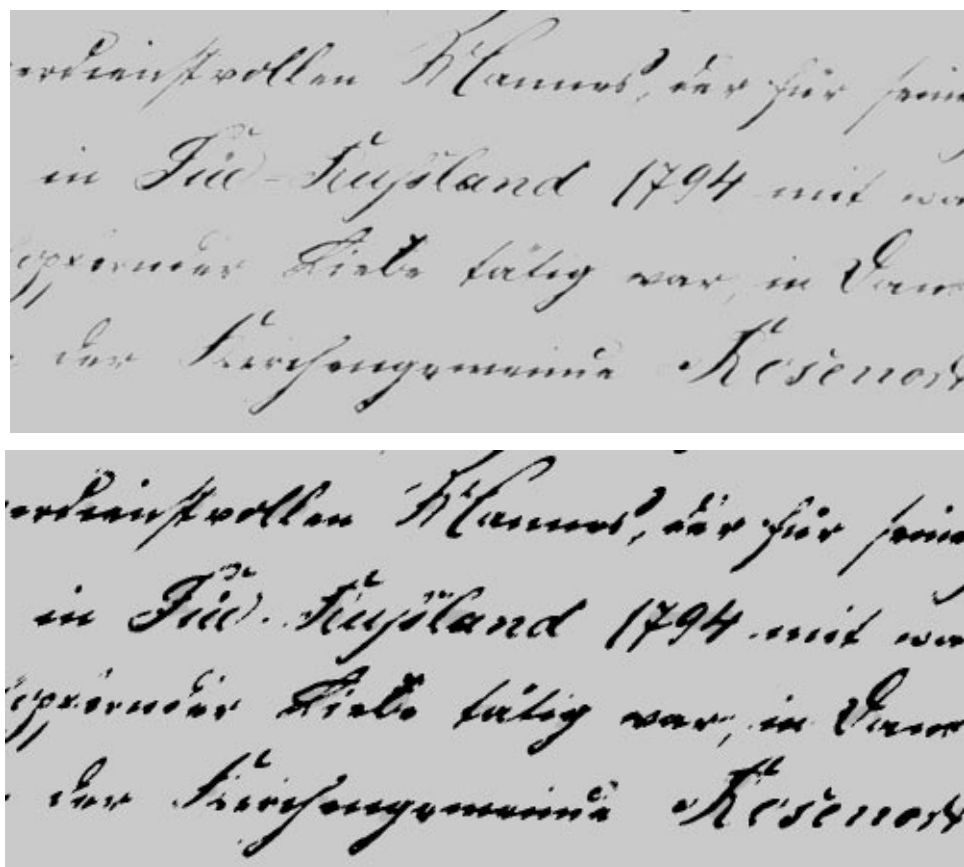


Figure 5.16: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

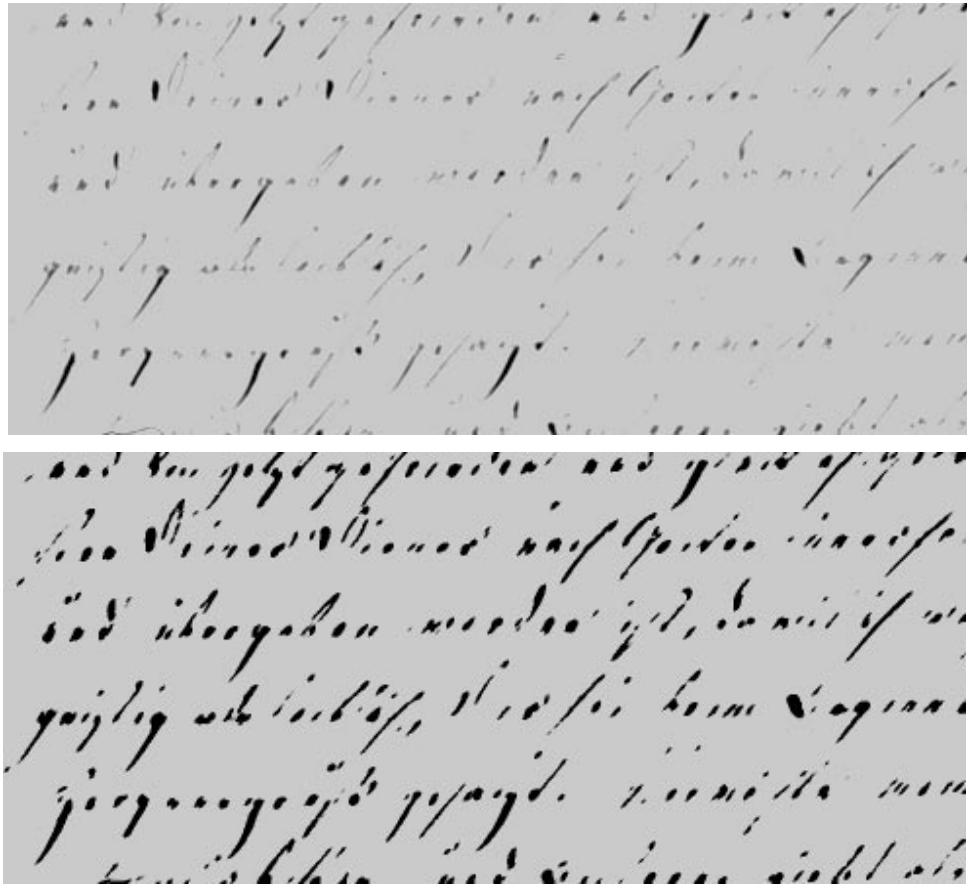


Figure 5.17: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

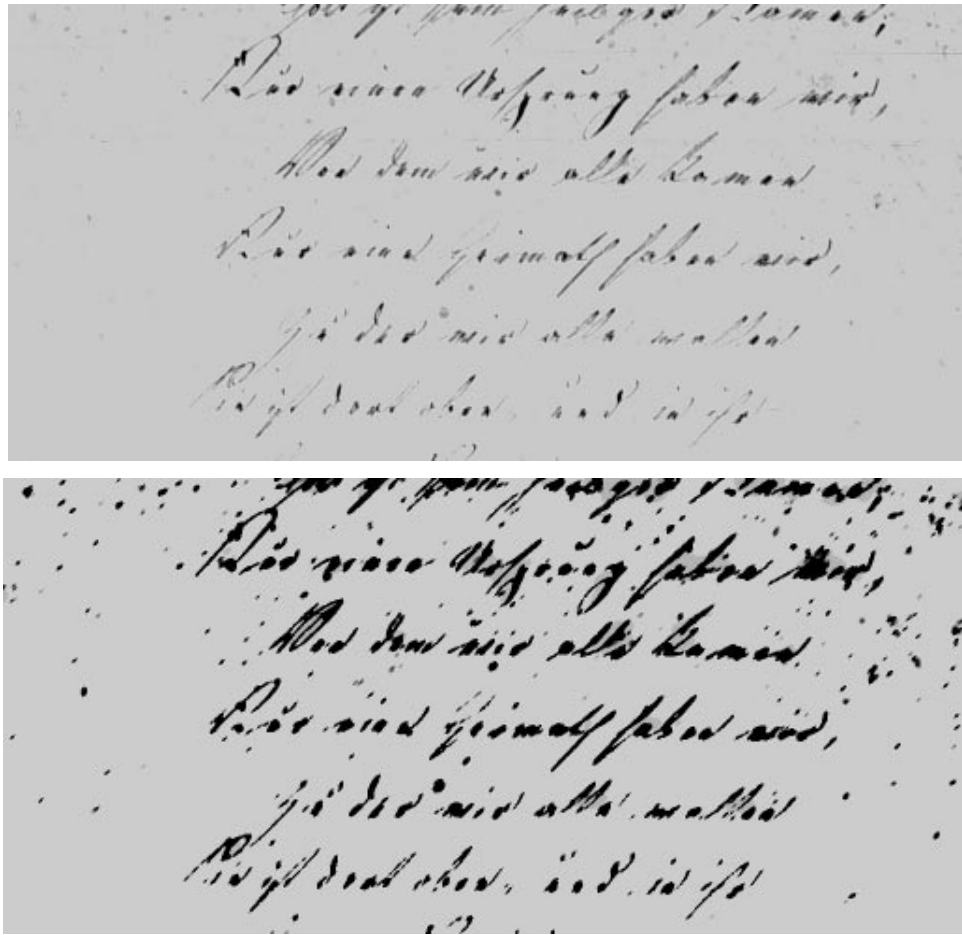


Figure 5.18: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

CHAPTER 5. ENHANCEMENT AND STRUCTURAL FILTERING

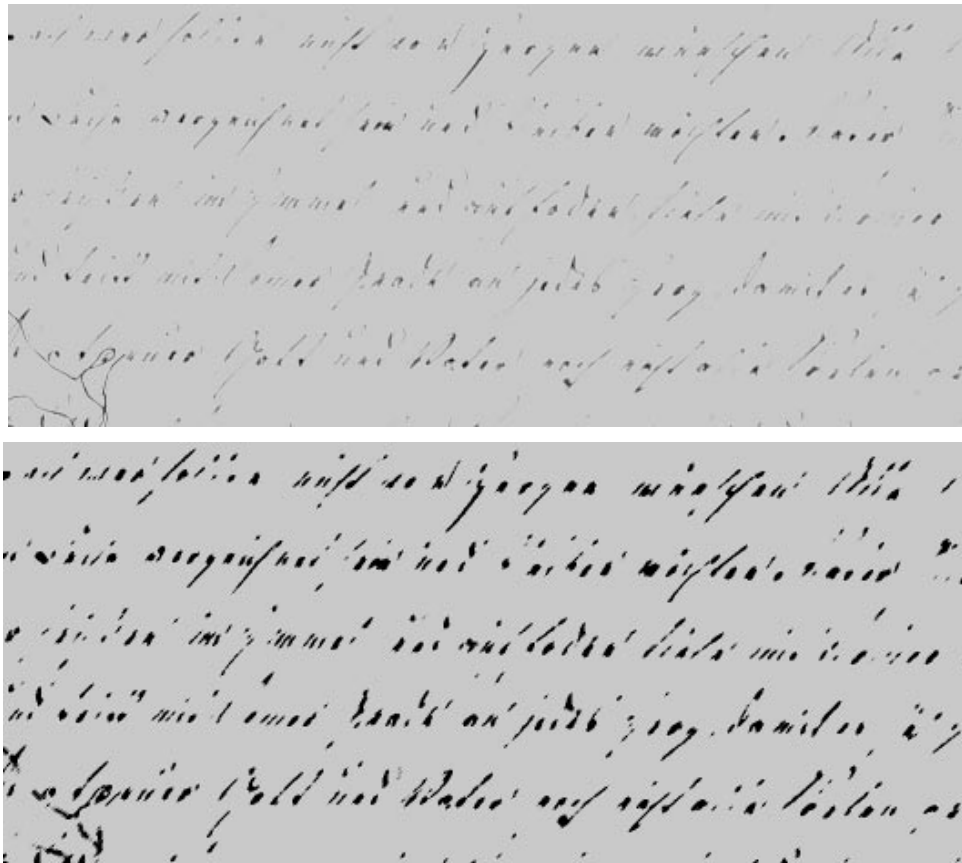


Figure 5.19: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

CHAPTER 5. ENHANCEMENT AND STRUCTURAL FILTERING

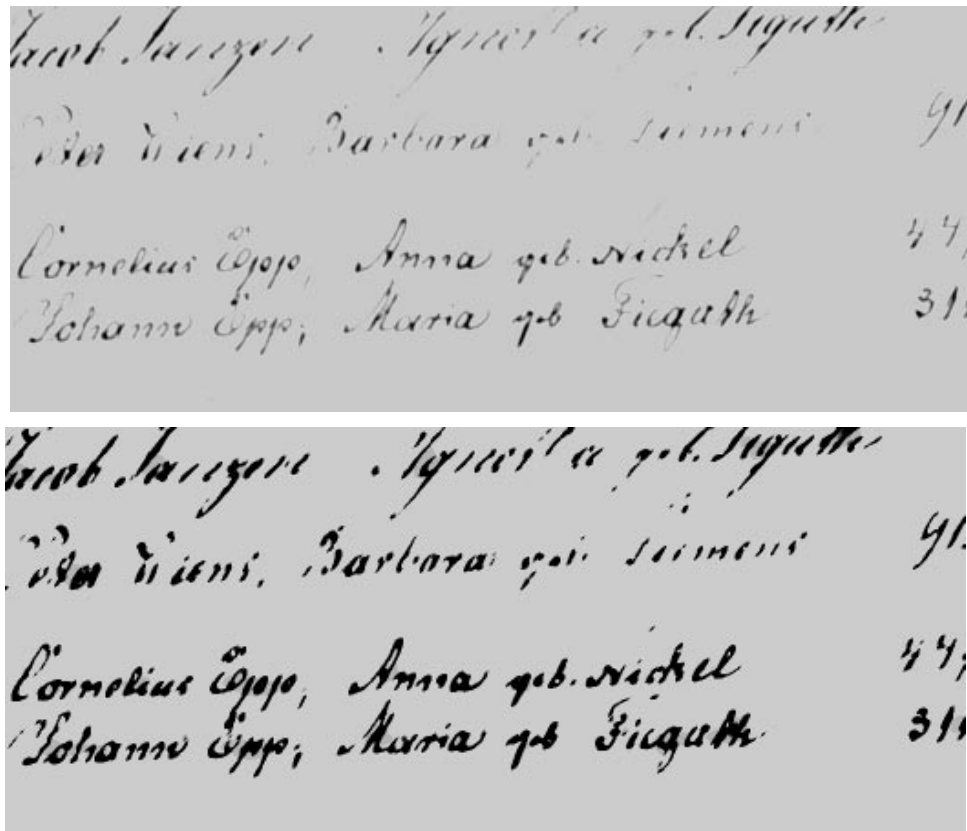


Figure 5.20: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

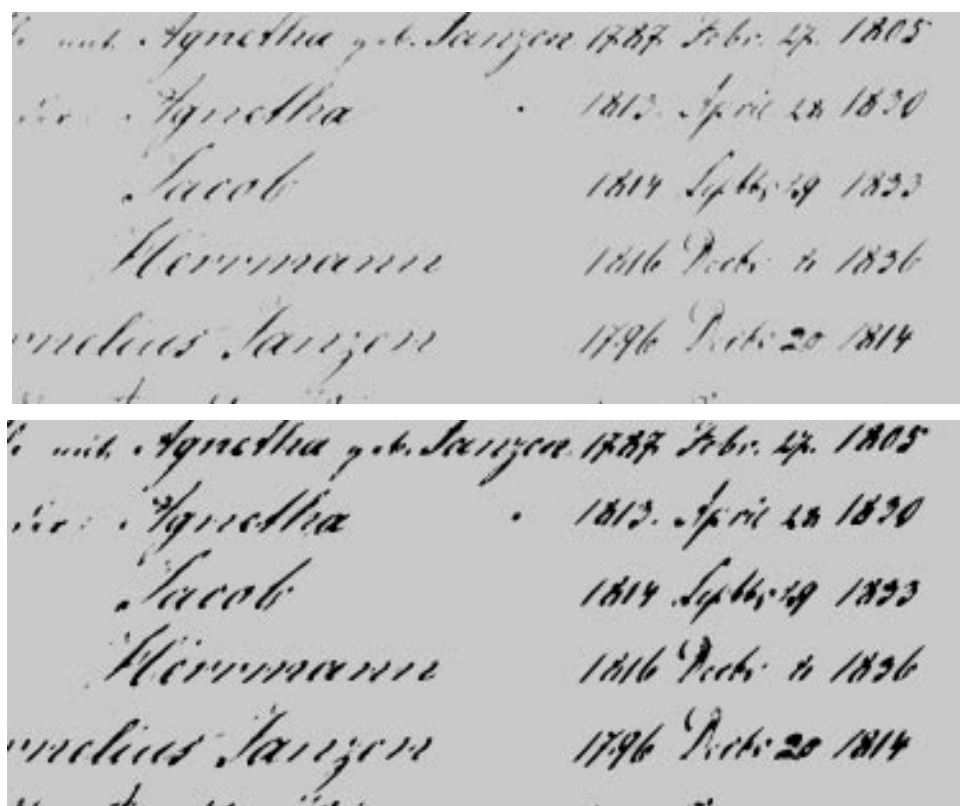


Figure 5.21: (Top) Sample from the document library. (Bottom) Result from enhancement and reconstruction techniques.

Chapter 6

Concluding Remarks

The objective of this research project is to devise an image processing technique to improve the readability and image quality of an archive of digitized handwriting images. The examples at the end of the previous chapter demonstrate that the technique developed does indeed attain this goal. The technique accomplishes this improvement using several image processing techniques.

One limitation of the technique used is that it does not fully reconstruct character strokes. One technique that may improve the reconstruction is a character recognition technique. However, because the strokes vary in sizes, shapes, widths, and tones, developing such a technique would be difficult.

Suggestions for future work include improving the image system model and improving the structural reconstruction techniques. Improving the image system model may allow for other classical restoration techniques to be applied more effectively. The morphological closing operation may be improved by experimenting with other structural elements. These would have to be designed to bridge linear gaps without significantly widening the strokes. In places, where the missing stroke is suspected to be curved, further morphology is not likely beneficial. It may also be possible to extend the logic in the gap filling filter to allow a larger window to be used. This would allow larger gaps to be filled. This would also be difficult as many problem areas exist that would be filled incorrectly by larger windows.

Appendix A

Source Code

APPENDIX A. SOURCE CODE

A.1 Inverse Filtering Script

```
%
% inverse.m
% Jimmy Swain                      Oct. 25, 1996
%
%       The is a MATLAB script to apply an inverse filter
% to an image. It is derived with the following formula.
%
% usage: inverse(img_file, [rows, cols], lambda)
%
% img_file -- image used to embed mask into in order to perform
%           transformations.
% rows -- number of rows in the image
% cols -- number of cols in the image
% lambda -- lambda value used to compute a Gaussian smoothing
% function
%

function [h, i] = inverse(img_file, dimen, lambda)

%
% Determine what parameters have been passed in.
%
if nargin<3, lambda = []; end

%
% Open the image.  If unsuccessful, ouput an error.
%
imgFile = fopen(img_file, 'r');
psfFile = fopen('psf.f', 'r');
flag = imgFile + psfFile;
if 0 > flag,
    fprintf(1, 'Error opening files.\n');
    return;
end

%
```

APPENDIX A. SOURCE CODE

```
% Read in each pixel in the image matrix.
%
sz = fscanf(psfFile, '%d', [1, 2]);
scale = fscanf(psfFile, '%lf', 1);
h = fscanf(psfFile, '%lf', [sz(1), sz(2)]);
g = fread(imgFile, dimen, 'uint8');

%
% Compute the inverse filter.
%
H = fftshift(fft2(h, dimen(1), dimen(2)));
I = 1 ./ H;

%
% Construct a Gaussian and smooth the inverse filter
%
if lambda ~= [],
    for i=1:dimen(1)
        for j=1:dimen(2)
            gauss(i,j) = ( exp(-1 * lambda^2 * pi * (i^2 + j^2)) );
        end
    end
    figure(1);
    mesh(gauss);
    Gauss = fftshift(fft2(gauss));
    normal = max(max(Gauss));
    Gauss = 1/normal .* Gauss;
    figure(2);
    mesh(abs(Gauss));
    I = Gauss .* I;
end

%
% Take the Fourier transform of the degraded image.
%
G = fftshift(fft2(g));
figure(3);
```

APPENDIX A. SOURCE CODE

```
mesh(abs(I));
print -dps invfil.ps
figure(4);
mesh(abs(H));
print -dps psffil.ps

%
% Compute the result in the Fourier domain.
%
F = I .* G;

%
% Take the inverse Fourier transform to get the estimate of
% the original image.
%
f = round ( abs ( ifft2(F) ) );

%
% Bound the result.
%
for i = 1 : dimen(1)
    for j = 1 : dimen(2)

        if f(i,j) > 255
            f(i,j) = 255;
        elseif f(i,j) < 0
            f(i,j) = 0;
        else
            f(i,j) = f(i,j);
        end

    end

end

%
% Write the result to a new file.
%
```

APPENDIX A. SOURCE CODE

```
img_file = sprintf('I%s', img_file);
outFile = fopen (img_file, 'w');
if -1 == outFile
    printf('Error opening file %s.\n', img_file);
    return;
end
fwrite(outFile, f, 'uint8');

%
% Take the inverse Fourier transform of the inverse filter.
%
i = real ( fftshift(fft2(I)) );
figure(5);
mesh(i);
iImg = round ( i );

%
% Bound the result so that we can view the image.
%
for i = 1 : dimen(1)
    for j = 1 : dimen(2)

        if iImg(i,j) > 255
            iImg(i,j) = 255;
        elseif iImg(i,j) < 0
            iImg(i,j) = 0;
        else
            iImg(i,j) = iImg(i,j);
        end

    end

end

%
% Write the result to a new file.
%
outFile = fopen ('invfil.raw', 'w');
```

APPENDIX A. SOURCE CODE

```
if -1 == outFile
    printf('Error opening file %s.\n', img_file);
    return;
end
fwrite(outFile, iImg, 'uint8');

fclose('all');
end
```

APPENDIX A. SOURCE CODE

A.2 Matrix Class

```

/*****
:   matrix.h Jimmy Swain
:
:   This file contains the Matrix class definition.
:
:   Copyright 1996. All Rights Reserved.
*****/

#ifndef MATRIX_H
#define MATRIX_H

#include <stdio.h>
#include <fstream.h>
#include "error.h"
#include "math.h"

const int MAX_INTENSITY = 256; /* defined constant for max gray value */
const int NO_VALUE = 201; /* defined constant used when operations */
/* exceed matrix boundaries as a dummy value */
class Matrix
{
public:
    Matrix(int,int,double); /* constructor for a matrix */
    ~Matrix(); /* destructor for a matrix */
    void GetDimen(int&,int&); /* return the matrix dimensions */
    double Get(int,int); /* return an entry */
    void Set(int,int,double); /* set an entry */
    void Convolve(Matrix&, Matrix&); /* convolves 2 matrices */
    void Scale(double); /* multiply matrix by a scalar */
    void Show(void); /* display the matrix to stdout */
    double Mean(void); /* calculate the mean value in matrix */
    double Variance(double); /* calculate the variance */
    double Median(void); /* calculate the median in the matrix */
    double BucketMedian(void); /* calculate the median in the matrix */
    double Median(double [], double[], int, int[]); /* calculate the median */
    void CopyData(Matrix&,int,int); /* copies a block into a matrix */

```

APPENDIX A. SOURCE CODE

```
void GetCol(int,int,int,double[]); /* copies a block into a matrix */
void GetTally(int,int,int,int[]); /* copies a block into a matrix */
void Dilate(Matrix&,int,Matrix&); /* Morphological dilate operation */
void Erode(Matrix&,int,Matrix&); /* Morphological erode operation */
void Write(ofstream&,int&,const int);/* Write a matrix to disk */

private:
    int rows; /* rows in the matrix */
    int cols; /* cols in the matrix */

    double* data; /* pointer to the matrix data */
/* represented as a stacked */
/* vector */
};

#endif

/*****
: matrix.c Jimmy Swain
:
: This file contains the member functions for the Matrix object.
:
: Copyright 1996. All rights reserved.
*****/

#include "matrix.h"

/*
: Matrix()
:
: Constructor function used to allocate and initialize storage
: for a matrix.
:
: Args: r gives the number of rows
: c gives the number of columns
: initial gives the initialization value
*/
```

APPENDIX A. SOURCE CODE

```
Matrix::Matrix(int r, int c, double initial)
{
    int i; /* loop index */

    /* Allocate the memory needed to store the image. */
    rows = r;
    cols = c;
    data = new double [rows * cols];
    if (NULL == data)
    {
        throw MEMORY_ERROR;
    }

    /* Initialize the matrix to contain zero values */
    for ( i=0; i < rows * cols; i++ )
    {
        data[i] = initial;
    }
}

/*
: ~Matrix()
:
: Destructor function: used to deallocate any dynamically created
: memory.
:
*/
Matrix::~Matrix()
{
    delete [] data;
}

/*
: GetDimen()
:
: Returns the dimensions of the matrix
:
```

APPENDIX A. SOURCE CODE

```
: Args: r returns the rows by reference
: c returns the columns by reference
*/
void Matrix::GetDimen(int &r, int &c)
{
    r = rows;
    c = cols;

    return;
}

/*
: Get()
:
: Returns the entry at matrix position (i,j) from the
: stacked vector representation.
:
: Args: i is the row entry
: j is the column entry
*/
double Matrix::Get(int i, int j)
{
    if ( 0 > i )
    {
        i = 0;
    }
    if ( 0 > j )
    {
        j = 0;
    }
    if ( rows <= i )
    {
        i = rows - 1;
    }
    if ( cols <= j )
    {
        j = cols - 1;
    }
}
```

APPENDIX A. SOURCE CODE

```
    }

    /* Return the data value */
    return data[ i * cols + j ];
}

/*
: Set()
:
: Sets the entry at matrix position (i,j) with the
: value passed in as newValue.
:
: Args: i is the row entry
: j is the col entry
: newValue is the new entry to be added to the matrix at (i,j)
*/
void Matrix::Set(int i, int j, double newValue)
{
    if( (0 > i) || (0 > j) || (rows <= i) || (cols <= j) )
    {
        return;
    }

    data[ i * cols + j ] = newValue;

    return;
}

/*
: Convolve()
:
: Convolves the matrix with another matrix that is passed in as
: input. The result is returned in the reference parameter result.
:
: Args: matrix is the matrix to convolve this with
: result is the result of the convolution
*/
```

APPENDIX A. SOURCE CODE

```
void Matrix::Convolve(Matrix &matrix, Matrix &result)
{
    int xtarget; /* the x-center pixel in mask */
    int ytarget; /* the y-center pixel in mask */
    int rows2; /* # of rows in matrix */
    int cols2; /* # of columns in matrix */
    int i,j,m,n; /* loop indices */
    double sum; /* temp storage for calculations */
    int x,y; /* image pixel coordinates */

    /* Retrieve all data that is needed to perform the convolution */
    matrix.GetDimen(rows2, cols2);
    xtarget = rows2/2;
    ytarget = cols2/2;

    /* Perform the two dimensional convolution. */
    for(i=0; i<rows; i++)
    {
        for(j=0; j<cols; j++)
        {
            sum = 0.0;
            for(m=0; m<rows2; m++)
            {
                for(n=0; n<cols2; n++)
                {
                    x = i-xtarget+m;
                    y = j-ytarget+n;
                    if ((0<=x) && (0<=y) && (rows>x) && (cols>y))
                    {
                        sum += ( this->Get(x,y) * matrix.Get(m,n) );
                    }
                }
            }
            else
            {
                sum += ( NO_VALUE * matrix.Get(m,n) );
            }
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
        /* Set the result for this entry in the output parameter */
        result.Set(i,j,sum);
    }
}

return;
}

/*
: Scale()
:
: Multiplies the matrix by a scalar.
:
: Args: value is the scale factor
*/
void Matrix::Scale(double value)
{
    int i,j; /* loop indices */

    /* Apply the scalar to each entry. */
    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols; j++)
        {
            data[ cols * i + j ] *= value;
        }
    }

    return;
}

/*
: Show()
:
: This function displays the matrix in a row x col format. Thus,
: it is only useful for small matrices.
```

APPENDIX A. SOURCE CODE

```
:
*/
void Matrix::Show()
{
    int i,j; /* loop indices */

    /* Display each entry. */
    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols; j++)
        {
            printf("%6.3f ", data[ cols * i + j ]);
        }
        printf("\n");
    }
    return;
}

/*
: Mean()
:
: Calculates the mean of the matrix
:
*/
double Matrix::Mean(void)
{
    int i,j; /* loop indices */
    double mean = 0.0; /* stores the mean */

    /* Apply the scalar to each entry. */
    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols; j++)
        {
            mean += data[ cols * i + j ];
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
    return ( (mean)/(rows*cols) );
}

/*
: Variance()
:
: Calculates the variance of the matrix
:
*/
double Matrix::Variance(double mean)
{
    int i,j; /* loop indices */
    double var = 0.0; /* stores the mean */

    /* Apply the scalar to each entry. */
    for (i=0; i<rows; i++)
    {
        for (j=0; j<cols; j++)
        {
            var += pow( (data[ cols * i + j ] - mean ), 2 );
        }
    }
    var = var / ( (rows * cols) - 1 );

    return ( var );
}

/*
: Median()
:
: Calculates the median of the matrix by sorting
: the array using the bubblesort algorithm given in
: Computer Algorithms by Sara Baase.
:
*/
double Matrix::Median(void)
```

APPENDIX A. SOURCE CODE

```
{
  int i,j; /* loop indices */
  int numPairs; /* the number of pairs to be compared */
  int didSwitch; /* true if an element is switched */
  double temp; /* temp storage */

  /* Apply the scalar to each entry. */
  j = rows * cols;
  numPairs = j;
  didSwitch = 1;
  while(didSwitch)
  {
    numPairs--;
    didSwitch = 0;
    for(i=0; i<numPairs; i++)
    {
      if( data[i] > data[i+1] )
      {
        temp = data[i];
data[i] = data[i+1];
data[i+1] = temp;
didSwitch = 1;
      }
    }
  }

  /* determine the median */
  if( 0 == j%2 )
  {
    /* Matrix is even */
    return ( (data[j/2] + data[j/2-1]) / 2 );
  }
  else
  {
    return ( data[j/2] );
  }
}
```

APPENDIX A. SOURCE CODE

```
/*
: Median()
:
: Calculates the median of the matrix using a bucket sort.
: One column is added to the buckets and one column is
: removed. The tally is kept externally.
*/
double Matrix::Median(double in[], double out[], int sz, int tally[])
{
    int i,j; /* loop indices */
    int left, right; /* temp storage used in computations */
    int index1, index2; /* temp storage used in computations */
    int sum1; /* no of pixels in sort */
    int sum2; /* no of pixels in sort */

    /* determine the elements needed to compute the median */
    j = sz * sz;
    if( 0 == j%2 )
    {
        index1 = j / 2;
        index2 = j / 2 + 1;
    }
    else
    {
        index1 = j / 2 + 1;
        index2 = j / 2 + 1;
    }

    /* Switch the new cols in and old cols out */
    for ( i=0; i<sz; i++ ) {
        tally [ (int)in[i] ] ++;
        tally [ (int)out[i] ] --;
    }

    /* Determine the left and right members used to compute the median */
    sum1 = sum2 = 0;
```

APPENDIX A. SOURCE CODE

```
left = right = -1;
for ( i=0; i<MAX_INTENSITY; i++ ) {
    sum1 += tally [ i ];
    sum2 += tally [ i ];
    if ( (sum1 >= index1) && (left == -1) ) {
        left = i;
    }

    if ( sum2 >= index2 ) {
        right = i;
        break;
    }

}

/* determine the median */
return ( (double)(left + right) / 2.0 );
}

/*
: BucketMedian()
:
: Calculates the median of the matrix using a bucket sort.
*/
double Matrix::BucketMedian(void)
{
    int i,j; /* loop indices */
    int left, right; /* temp storage used in computations */
    int index1, index2; /* temp storage used in computations */
    int sum1; /* no of pixels in sort */
    int sum2; /* no of pixels in sort */
    int buckets[MAX_INTENSITY] = {0}; /* buckets */

    /* determine the elements needed to compute the median */
    j = rows * cols;
    if( 0 == j%2 )
    {
```

APPENDIX A. SOURCE CODE

```
    index1 = (j - 1) / 2;
    index2 = j / 2;
}
else
{
    index1 = j / 2;
    index2 = j / 2;
}

/* Switch the new cols in and old cols out */
for ( i=0; i<rows; i++ ) {
    for ( j=0; j<cols; j++ ) {
        buckets [ (int) this->Get ( i, j ) ] ++;
    }
}

/* Determine the left and right members used to compute the median */
sum1 = sum2 = 0;
left = right = -1;
for ( i=0; i<MAX_INTENSITY; i++ ) {
    sum1 += buckets [ i ];
    sum2 += buckets [ i ];
    if ( (sum1 >= index1) && (left == -1) ) {
        left = i;
    }

    if ( sum2 >= index2 ) {
        right = i;
        break;
    }
}

/* determine the median */
return ( (double)(left + right) / 2.0 );
}
```

APPENDIX A. SOURCE CODE

```
/*
: CopyData()
:
: Copies a block of data into the matrix that is passed in as
: input. The result is returned in the reference parameter result.
:
: Args:  matrix is the matrix to convolve this with
:        xbegin is the row to start copying from
:        ybegin is the col to start copying from
*/
void Matrix::CopyData(Matrix &matrix, int xbegin, int ybegin)
{
    int rows; /* # of rows in matrix */
    int cols; /* # of columns in matrix */
    int i,j; /* loop indices */

    /* Retrieve all data that is needed to perform the convolution */
    matrix.GetDimen(rows, cols);

    /* Copy the data. */
    for(i=0; i<rows; i++)
    {
        for(j=0; j<cols; j++)
        {
            matrix.Set(i, j, this->Get(xbegin+i, ybegin+j));
        }
    }
    return;
}

/*
: GetCol()
:
: Returns a matrix column of a specified size about a point
:
: Args:  x is the row to start copying from

```

APPENDIX A. SOURCE CODE

```
:      y is the col to start copying from
: sz is the size to return
: col is the array to return the result in
*/
void Matrix::GetCol(int x, int y, int sz, double col[])
{
    int top; /* top position to start copying from */
    int bottom; /* bottom position to start copying from */
    int i,j; /* loop indices */

    /* Set variables */
    top = x - sz / 2;
    bottom = x + sz / 2;
    j=0;

    /* Copy the data. */
    for(i=top; i<=bottom; i++) {
        col[j++] = this.Get(i, y);
    }
    return;
}

/*
: GetTally()
:
: Return the histogram of a square region
:
: Args:  x is the row to start copying from
:        y is the col to start copying from
: sz is the size to return
: col is the array to return the result in
*/
void Matrix::GetTally(int x, int y, int sz, int col[])
{
    int offset; /* offset from center pixel */
    double value; /* pixel value */
    int index; /* pixel value cast to an int */
```

APPENDIX A. SOURCE CODE

```
int i,j; /* loop indices */

/* Set variables */
offset = sz / 2;

/* Copy the data. */
for(i=x-offset; i<=x+offset; i++) {
    for(j=y-offset; j<=y+offset; j++) {
        value = this.Get(i, j);
        value += 0.5;
        index = (int) floor ( value );
        col[ index ]++;
    }
}
return;
}

/*
: Dilate()
:
: Perform the morphological dilation operation
:
: Args:  se is the structuring element
: sz gives the size of se
: res returns the result
*/
void Matrix::Dilate( Matrix& se, int sz, Matrix& res )
{
    int i,j,k,x,y,m,n; /* loop indices */
    int r, c; /* size of r */
    double* calcs; /* stores result of each calculation */
    double max; /* maximum value in the calcs array */
    int sr, sc; /* size of struct element */
    int str, fin; /* start and finishing indices for se */

    /* Verify that r is the same size as this */
    res.GetDimen ( r, c );
```

APPENDIX A. SOURCE CODE

```
if ( ( r != this->rows) || ( c != this->cols) )
{
    return;
}

/* Allocate memory for each of the calculations */
x = 0;
se.GetDimen ( sr, sc );
str = 0 - sr/2;
fin = 0 + sr/2;
x = sr * sc;
calcs = new double [ x ];
if ( !calcs )
{
    throw MEMORY_ERROR;
}

/* Calculate the dilation operation at each point */
for ( x=0; x<rows; x++ )
{
    for ( y=0; y<cols; y++ )
    {
        /* Get each calculation for the sliding window */
        k = 0;
        for ( i=str; i<fin; i++ )
        {
            for ( j=str; j<fin; j++ )
            {
m = i + fin;
n = j + fin;
//calcs[ k ] = this->Get ( x - i, y - j ) + se.Get ( m, n );
if ( se.Get( m, n ) )
{
    calcs[ k ] = this->Get ( x - i, y - j );
    k++;
}
            }
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
    }

    /* Find the max */
    max = calcs [ 0 ];
    for ( i=1; i<k; i++ )
    {
if ( max < calcs [ i ] )
{
    max = calcs [ i ];
}
    }

    /* Set the pixel in the result image to the new value */
    if ( 0 > max )
    {
max = 0.0;
    }
    else if ( MAX_INTENSITY < max )
    {
max = MAX_INTENSITY;
    }
    res.Set ( x, y, max );
}
}

/*
: Erode()
:
: Perform the morphological erosion operation
:
: Args:  x is the x coordinates of the structuring element
:        y is the y coordinates of the structuring element
: sz gives the size of the x,y arrays
: res returns the result
*/
void Matrix::Erode( Matrix& se, int sz, Matrix& res )
```

APPENDIX A. SOURCE CODE

```
{
    int i,j,k,x,y,m,n; /* loop indices */
    int r, c; /* size of r */
    double* calcs; /* stores result of each calculation */
    double min; /* minimum value in the calcs array */
    int sr, sc; /* size of struct element */
    int str, fin; /* start and finishing indices for se */

    /* Verify that r is the same size as this */
    res.GetDimen ( r, c );
    if ( ( r != this->rows) || ( c != this->cols) )
    {
        return;
    }

    /* Allocate memory for each of the calculations */
    se.GetDimen ( sr, sc );
    str = 0 - sr/2;
    fin = 0 + sr/2;
    x = sr * sc;
    calcs = new double [ x ];
    if ( !calcs )
    {
        throw MEMORY_ERROR;
    }

    /* Calculate the dilation operation at each point */
    for ( x=0; x<rows; x++ )
    {
        for ( y=0; y<cols; y++ )
        {
            /* Get each calculation for the sliding window */
            k = 0;
            for ( i=str; i<fin; i++ )
            {
                for ( j=str; j<fin; j++ )
                {
```

APPENDIX A. SOURCE CODE

```
m = i + fin;
n = j + fin;
//calcs[ k ] = this->Get ( x + i, y + j ) - se.Get ( m, n );
if ( se.Get ( m, n ) )
{
    calcs[ k ] = this->Get ( x + i, y + j );
    k++;
}
}
}

/* Find the min */
min = calcs [ 0 ];
for ( i=1; i<k; i++ )
{
if ( min > calcs [ i ] )
{
    min = calcs [ i ];
}
}

/* Set the pixel in the result image to the new value */
if ( 0 > min )
{
min = 0.0;
}
else if ( MAX_INTENSITY < min )
{
min = MAX_INTENSITY;
}
res.Set ( x, y, min );
}
}

/*
: Write()
```

APPENDIX A. SOURCE CODE

```
:
: Writes the matrix to disk in raw format. Raw format is a
: stream of binary digits.
:
: Args: outfile is the stream to write the file to
*/
void Matrix::Write(ofstream& outfile, int& written, const int max)
{
    int i,j; /* loop indices */
    int intensity; /* pixel intensity */
    uchar byte; /* one byte of data */

    /* Write the image data. */
    for(i=0; i < this->rows; i++)
    {
        for(j=0; j < this->cols; j++)
        {
            intensity = (int) floor( ( this->Get(i,j) ) + 0.5);
            byte = (uchar) intensity;
            if ( written < max )
            {
                outfile.put( byte );
written++;
            }
        }
    }

    outfile.flush();

    return;
}
```

A.3 Image Class

```
/*****
```

APPENDIX A. SOURCE CODE

```
: image.h Jimmy Swain
:
: This file contains the Image class definition.
:
: Copyright 1996.
*****/

#ifndef IMAGE_H
#define IMAGE_H

#include <stdio.h>
#include "error.h"
#include "string.h"
#include "math.h"
#include "matrix.h"

#define MAX 255
#define MIN 0
#define BACKGROUND 201

typedef enum
{
    RAW, PPM
}
Format;

class Image
{
public:
    Image(int,int,double); /* constructor for an image */
    ~Image(); /* destructor for an image */
    void Load(Format,char*); /* loads an image from disk */
    void Write(Format,char*); /* writes image to disk */
    void CopyData(Matrix&); /* copy data from the image */
    void CopyBlock(Matrix&,int,int); /* copy data from the image */
    void SetData(Matrix&); /* copy data to the image */
    double Get(int,int); /* get a pixel value */
};
```

APPENDIX A. SOURCE CODE

```
void Set(int,int,double); /* set a pixel value */

Matrix* matrix; /* pointer to a image matrix */
private:
    int height; /* size of the matrix */
    int width; /* size of the matrix */
};

#endif

/*****
: image.c Jimmy Swain
:
: This file contains the member functions for the Image object.
:
: Copyright 1996. All rights reserved.
*****/

#include "image.h"
#include <fstream.h>

/*
: Image()
:
: Constructor function used to instantiate an image object.
:
: Args: h is the number of rows of pixels in the image
: w is the number of cols of pixels in the image
: initial is the initial value for the pixels
*/
Image::Image(int h, int w, double initial)
{
    /* Allocate the memory needed to store the image. */
    matrix = new Matrix(h, w, initial);
    if (NULL == matrix)
    {
```

APPENDIX A. SOURCE CODE

```
        throw MEMORY_ERROR;
    }

    /* Initialize other private members */
    height = h;
    width = w;
}

/*
: ~Image()
:
: Destructor function: used to deallocate any dynamically created
: memory.
*/
Image::~Image() {
    delete matrix;
}

/*
: Load()
:
: Loads an image from disk in either raw or ppm format. Raw format is a
: stream of binary digits. Ppm format is as follows:
:
: P3
: # Comment
: width height
: max intensity
: pixel value ...
:
: Note that pixel values are stored with RGB components, so each pixel
: is really written three times.
:
: Args: mode is the type of file to read in (either raw or ppm)
: filename is the file to read in
*/
void Image::Load(Format mode, char* filename)
```

APPENDIX A. SOURCE CODE

```
{
  ifstream infile; /* input file */
  int i,j; /* loop indices */
  int intensity; /* pixel intensity */
  uchar byte; /* one byte of data */
  char buffer[82]; /* storage for a line */
  int val1, val2, val3; /* generic integer storage */

  /* Determine which mode we are writing in */
  switch (mode)
  {
    case RAW:
      /* Open the input file. */
      infile.open(filename, ios::in | ios::bin, 0);
      if (0 == infile)
      {
        throw FILE_ERROR;
      }
      break;

    case PPM:
      /* Open the input file. */
      infile.open(filename);
      if (0 == infile)
      {
        throw FILE_ERROR;
      }

      /* Read the ppm header. */
      infile >> buffer;
      infile.read(buffer, 24);
      infile >> val1 >> val2;
      infile >> val3;

      /* Check to see if enough memory was allocated */
      if ( (val2 != height) || (val1 != width) )
      {
```

APPENDIX A. SOURCE CODE

```
printf("%d x %d \n", val2, val1);
printf("%d x %d \n", height, width);
throw MEMORY_ERROR;
    }

    break;
}

/* Load the image data. */
for(i=0; i < this->height; i++)
{
    for(j=0; j < this->width; j++)
    {
infile.get(byte);
intensity = (int) byte;
if ( PPM == mode )
{
    infile.get(byte);
    intensity = (int) byte;
    infile.get(byte);
    intensity = (int) byte;
}
this->matrix->Set(i,j, (double) intensity);
    }
}

infile.close();

return;
}

/*
: Write()
:
: Writes the image to disk in either raw or ppm format. Raw format is a
: stream of binary digits. Ppm format is as follows:
:
```

APPENDIX A. SOURCE CODE

```
: P3
: # Comment
: rows cols
: max intensity
: pixel value ...
:
: Note that pixel values are written with RGB components, so each pixel
: is really written three times.
:
: Args: mode is the format that we want to write in
: filename is the file to write to
*/
void Image::Write(Format mode, char* filename)
{
    ofstream outfile; /* output file */
    char tempFile[81]; /* temp storage for filename */
    char path[81]; /* complete filename */
    int i,j; /* loop indices */
    int intensity; /* pixel intensity */
    uchar byte; /* one byte of data */

    /* Retrieve the filename without an extension */
    sscanf(filename,"%[^.]",tempFile);

    /* Determine which mode we are writing in */
    switch (mode)
    {
        case RAW:
            /* Open the output file. */
            sprintf(path, "%s.raw", tempFile);
            outfile.open(path, ios::out | ios::bin, 0);
            if (0 == outfile)
            {
                throw FILE_ERROR;
            }

            break;
```

APPENDIX A. SOURCE CODE

```
case PPM:

    /* Open the output file. */
    sprintf(path, "%s.ppm", tempFile);
    outfile.open(path);
    if (0 == outfile)
    {
        throw FILE_ERROR;
    }

    /* Write the ppm header. */
    outfile << "P6" << endl;
    outfile << "# CREATOR: recover v1.0" << endl;
    outfile << width << " " << height << endl;
    outfile << MAX << endl;

    break;
}

/* Write the image data. */
for(i=0; i < this->height; i++)
{
    for(j=0; j < this->width; j++)
    {
        intensity = (int) floor( ( this->matrix->Get(i,j) ) + 0.5);
        byte = (uchar) intensity;
        outfile.put( byte );
        if(PPM == mode)
        {
            outfile.put( byte );
            outfile.put( byte );
        }
    }
}

outfile.flush();
```

APPENDIX A. SOURCE CODE

```
    outfile.close();

    return;
}

/*
: CopyData
:
: This function copies the image data to a Matrix object and
: returns the result by reference.
:
: Args: result returns the image data
*/
void Image::CopyData(Matrix &result)
{
    int i, j; /* loop indices */

    /* Copy the data pixel by pixel to the new matrix */
    for ( i=0; i < this->height; i++)
    {
        for ( j=0; j < this->width; j++)
        {
            result.Set(i, j, this->matrix->Get(i,j) );
        }
    }

    return;
}

/*
: CopyBlock
:
: This function copies a block of image data to a Matrix object and
: returns the result by reference.
:
: Args: result returns the image data
*/
```

APPENDIX A. SOURCE CODE

```
void Image::CopyBlock(Matrix &result, int xstart, int ystart)
{
    /* Copy the data pixel by pixel to the new matrix */
    this->matrix->CopyData(result, xstart, ystart);

    return;
}
/*
: SetData
:
: This function copies the matrix data to an Image object.
: Data integrity is maintained by forcing values to fall in
: an acceptable range defined by [MIN, MAX].
:
: Args: result contains the new image data
*/
void Image::SetData(Matrix &result)
{
    int i, j; /* loop indices */
    double value; /* storage for matrix entry */

    /* Copy the data pixel by pixel to the new matrix */
    for ( i=0; i < this->height; i++)
    {
        for ( j=0; j < this->width; j++)
        {
            value = result.Get(i, j);
            if (MAX < value)
            {
                this->matrix->Set(i, j, MAX);
            }
            else if (MIN > value)
            {
                this->matrix->Set(i, j, MIN);
            }
            else
            {

```

APPENDIX A. SOURCE CODE

```
        this->matrix->Set(i, j, value);
    }
}

return;
}

/*
: Get()
:
: Gets a pixel value from the image object.
:
: Args: i is the row value in the image
: j is the col value in the image
*/
double Image::Get(int i, int j)
{
    return ( this->matrix->Get(i,j) );
}

/*
: Set()
:
: Sets a pixel value in the image object.
:
: Args: i is the row value in the image
: j is the col value in the image
: value is the intensity to set
*/
void Image::Set(int i, int j, double value)
{
    this->matrix->Set(i,j,value);
    return;
}
```

APPENDIX A. SOURCE CODE

A.4 Mask Class

```

/*****
:   mask.h Jimmy Swain
:
:   This file contains the Image class definition.
:
:   Copyright 1996.
*****/

#ifndef MASK_H
#define MASK_H

#include <stdio.h>
#include <fstream.h>
#include "error.h"
#include "string.h"
#include "math.h"
#include "image.h"
#include "matrix.h"

#define MAX_INTENSITY 255
#define BACKGROUND 201
#define MASK_PATH      "/u2/jswain/jswain/masks/"

class Mask
{
public:
    Mask(const char*); /* constructor for an image      */
    ~Mask(); /* destructor for an image                */
    double Apply(Matrix&,int,int); /* apply a mask to an image pixel */
    double GetScale(); /* return the scale factor      */
    void SetScale(double); /* set the scale factor      */
    int GetRows(); /* return the number of rows */
    int GetCols(); /* return the number of cols */
    void CopyData(Matrix&); /* copies mask into matrix */

```

APPENDIX A. SOURCE CODE

```
private:
    int rows; /* rows in the mask */
    int cols; /* cols in the mask */
    double scaleFactor; /* scaleFactor          */
    Matrix* matrix; /* pointer to a image matrix */
};

#endif

/*****
: mask.c Jimmy Swain
:
: This file contains the member functions for the Mask object.
:
: Copyright 1996. All rights reserved.
*****/

#include "mask.h"

/*
: Mask()
:
: Constructor function used to read a mask from an ascii file.
: The file is assumed to have the dimensions on the first line,
: the scale factor on the second line, and the mask data following.
*/
Mask::Mask(const char* file)
{
    ifstream infile; /* the infile */
    char path[81]; /* contains the complete filename */
    int i, j; /* loop indices */
    double value; /* temp storage for a mask value */

    /* Open the image file. */
    sprintf(path, "%s%s", MASK_PATH, file);
    infile.open(path, ios::in | ios::bin, 0);
    if (0 == infile)
```

APPENDIX A. SOURCE CODE

```
{
    throw FILE_ERROR;
}

/* Read in the dimensions and the scale factor */
infile >> this->rows >> this->cols;
infile >> this->scaleFactor;

/* Allocate the memory needed to store the image. */
matrix = new Matrix(rows, cols, 0.0);
if (NULL == matrix)
{
    throw MEMORY_ERROR;
}

/* Read the image data from infile. */
for (i = 0; i < this->rows; i++)
{
    for (j=0; j < this->cols; j++)
    {
        infile >> value;
        this->matrix->Set(i, j, value);
    }
}

infile.close();

return;
}

/*
: ~Mask()
:
: Destructor function: used to deallocate any dynamically created
: memory.
*/
Mask::~Mask()
```

APPENDIX A. SOURCE CODE

```
{
    delete matrix;
}

/*
: Apply()
:
: Apply the mask to a pixel in the input image and
: return the result.
:
*/
double Mask::Apply(Matrix &mat, int x, int y)
{
    int xstart; /* the x-starting pixel in Image */
    int ystart; /* the y-starting pixel in Image */
    int m,n; /* loop indices */
    double sum; /* temp storage for calculations */
    double intensity; /* temp storage for intensity */

    /*
       Compute the start and finish positions in the image. Also
       retrieve the image dimensions.
    */
    xstart = x - rows/2;
    ystart = y - cols/2;
    sum = 0.0;

    /* Perform the two dimensional convolution. */
    for(m=0; m<rows; m++)
    {
        for(n=0; n<cols; n++)
        {
            /* Sum up the intensities with the mask weights */
            intensity = mat.Get(xstart + m, ystart + n);
            sum += ( intensity * this->matrix->Get(m,n) );
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
    /* Return the result. */
    return (scaleFactor * sum);
}

/*
: GetScale()
:
: Returns the scale factor.
*/
double Mask::GetScale()
{
    return (scaleFactor);
}

/*
: SetScale()
:
: Set the scale factor.
*/
void Mask::SetScale(double newValue)
{
    scaleFactor = newValue;
}

/*
: GetRows()
:
: Return the rows in the mask
*/
int Mask::GetRows()
{
    return (this->rows);
}

/*
```

APPENDIX A. SOURCE CODE

```
: GetCols()
:
: Return the cols in the mask
*/
int Mask::GetCols()
{
    return (this->cols);
}

/*
: CopyData
:
: Copies mask data to a matrix and returns by reference
:
: Args: result returns the mask data
*/
void Mask::CopyData(Matrix &result)
{
    int i, j; /* loop indices */

    /* Copy the data from matrix to matrix */
    for ( i=0; i < this->rows; i++)
    {
        for ( j=0; j < this->cols; j++)
        {
            result.Set(i, j, this->matrix->Get(i, j) );
        }
    }
}
```

APPENDIX A. SOURCE CODE

A.5 Script for Experiment 1

```
time stats.exe s001.raw 213 105 7 > exp1.dat
time stats.exe s003.raw 213 105 7 >> exp1.dat
time stats.exe s004.raw 213 105 7 >> exp1.dat
time stats.exe s006.raw 213 105 7 >> exp1.dat
mv me*.raw 1.0/7.0
time stats.exe s001.raw 213 105 15 >> exp1.dat
time stats.exe s003.raw 213 105 15 >> exp1.dat
time stats.exe s004.raw 213 105 15 >> exp1.dat
time stats.exe s006.raw 213 105 15 >> exp1.dat
mv me*.raw 1.0/15.0
mv exp1.dat 1.0
```

A.6 Script for Experiment 2

```
nice -15 time median2.exe s001.raw 213 105 3 25 > medave0602.dat
nice -15 time median2.exe s003.raw 213 105 3 25 >> medave0602.dat
nice -15 time median2.exe s150x148.raw 150 148 3 25 >> medave0602.dat
nice -15 time median2.exe s181x71.raw 181 71 3 25 >> medave0602.dat
nice -15 time median2.exe s213x105.raw 213 105 3 25 >> medave0602.dat
mv m*.raw 1.2/25.3
nice -15 time median2.exe s001.raw 213 105 5 25 >> medave0602.dat
nice -15 time median2.exe s003.raw 213 105 5 25 >> medave0602.dat
nice -15 time median2.exe s150x148.raw 150 148 5 25 >> medave0602.dat
nice -15 time median2.exe s181x71.raw 181 71 5 25 >> medave0602.dat
nice -15 time median2.exe s213x105.raw 213 105 5 25 >> medave0602.dat
mv m*.raw 1.2/25.5
nice -15 time median2.exe s001.raw 213 105 3 50 >> medave0602.dat
nice -15 time median2.exe s003.raw 213 105 3 50 >> medave0602.dat
nice -15 time median2.exe s150x148.raw 150 148 3 50 >> medave0602.dat
nice -15 time median2.exe s181x71.raw 181 71 3 50 >> medave0602.dat
nice -15 time median2.exe s213x105.raw 213 105 3 50 >> medave0602.dat
mv m*.raw 1.2/50.3
nice -15 time median2.exe s001.raw 213 105 5 50 >> medave0602.dat
nice -15 time median2.exe s003.raw 213 105 5 50 >> medave0602.dat
nice -15 time median2.exe s150x148.raw 150 148 5 50 >> medave0602.dat
```

APPENDIX A. SOURCE CODE

```
nice -15 time median2.exe s181x71.raw 181 71 5 50 >> medave0602.dat
nice -15 time median2.exe s213x105.raw 213 105 5 50 >> medave0602.dat
mv m*.raw 1.2/50.5
mv *.dat 1.2
```

A.7 Script for Experiment 3

```
nice -15 time median4.exe s001.raw 213 105 25 > med0602.dat
nice -15 time median4.exe s003.raw 213 105 25 >> med0602.dat
nice -15 time median4.exe s150x148.raw 150 148 25 >> med0602.dat
nice -15 time median4.exe s181x71.raw 181 71 25 >> med0602.dat
nice -15 time median4.exe s213x105.raw 213 105 25 >> med0602.dat
mv m*.raw 1.4/25
nice -15 time median4.exe s001.raw 213 105 51 >> med0602.dat
nice -15 time median4.exe s003.raw 213 105 51 >> med0602.dat
nice -15 time median4.exe s150x148.raw 150 148 51 >> med0602.dat
nice -15 time median4.exe s181x71.raw 181 71 51 >> med0602.dat
nice -15 time median4.exe s213x105.raw 213 105 51 >> med0602.dat
mv m*.raw 1.4/50
nice -15 time median4.exe s001.raw 213 105 63 >> med0602.dat
nice -15 time median4.exe s003.raw 213 105 63 >> med0602.dat
nice -15 time median4.exe s150x148.raw 150 148 63 >> med0602.dat
nice -15 time median4.exe s181x71.raw 181 71 63 >> med0602.dat
nice -15 time median4.exe s213x105.raw 213 105 63 >> med0602.dat
mv m*.raw 1.4/63
nice -15 time median4.exe s001.raw 213 105 75 >> med0602.dat
nice -15 time median4.exe s003.raw 213 105 75 >> med0602.dat
nice -15 time median4.exe s150x148.raw 150 148 75 >> med0602.dat
nice -15 time median4.exe s181x71.raw 181 71 75 >> med0602.dat
nice -15 time median4.exe s213x105.raw 213 105 75 >> med0602.dat
mv m*.raw 1.4/75
mv *.dat 1.4
```

APPENDIX A. SOURCE CODE

A.8 Source Code for Driver Used in Experiment 1

```
/*
  filter.h Jimmy Swain

  This file contains all symbolic constants, enumerations, and
  include files needed for the non-linear filtering system.

  Copyright 1997.
*/

#ifndef STATS_H
#define STATS_H

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "error.h"
#include "image.h"
#include "mask.h"

#define MAX_DIMEN 512
#define BIG_BKG_SZ 35
enum { HORIZONTAL, VERTICAL, DIAG_45, DIAG_N45 };

bool isNormalBackground ( int, int, double, Image&, int, int );

#endif

/*****
: stats.c          Jimmy Swain
:
: This file is designed to compute local area stats across
: an image. . It takes four command line parameters. The first
: is the image file in raw format. The second and third give
: the size of the image. The fourth gives the window size
*****/
```

APPENDIX A. SOURCE CODE

```
: used to compute the stats. The stats that are considered
: include the mean, the median, and the median of the mean.
:
: Copyright 1997. All rights reserved.
*****/

#include "stats.h"

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* mean; /* mean image */
    Image* median; /* median image */
    Image* median2; /* median of average image */
    Image* aveImg; /* image of average block values */
    Matrix* buffer; /* temp storage for a data block */
    Matrix* buffer2; /* temp storage for a data block */
    Matrix* aveMask; /* averaging mask for a 7x7 block */
    Matrix* ave; /* the result of the filter operation */
    Matrix* tempAve; /* temp storage for an image block */
    char fileName[31]; /* argument containing filename */
    char fileNo[4]; /* unique id number for the file */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    int window; /* columns in the image */
    double initValue; /* initial value for double variables */
    int x,y; /* loop indices */

    /* Check the command line arguments for correctness */
    if (5 > argc)
    {
        printf("Usage: stats.exe <image_file> <rows> <cols> <window size>\n");
        printf("%d \n", argc);
        exit(0);
    }

    /* Get the command line parameters */
```

APPENDIX A. SOURCE CODE

```
cols = atoi(argv[2]);
rows = atoi(argv[3]);
window = atoi(argv[4]);
strncpy(fileNo, &(argv[1][1]), (size_t)(3 * sizeof(char)));
fileNo[3] = '\0';
initValue = (double)(window * window);
initValue = (double)(1 / initValue);

/*
   Check the validity of the command line arguments. The rows and
   cols must contain positive values and further are limited in size
   to values less than MAX_DIMEN.
*/
if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
mean = new Image(rows, cols, 0.0);
median = new Image(rows, cols, 0.0);
median2 = new Image(rows, cols, 0.0);
aveImg = new Image(rows, cols, 0.0);
if( (!img) || (!mean) || (!median) || (!median2) || (!aveImg) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Create a image of 7x7 averages */
```

APPENDIX A. SOURCE CODE

```
ave = new Matrix(rows, cols, 0.0);
tempAve = new Matrix(rows, cols, 0.0);
aveMask = new Matrix(window, window, initValue);
if( (NULL == aveMask) || (NULL == ave) || (NULL == tempAve) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}
img->CopyData(*ave);
ave->Convolve(*aveMask, *tempAve);
aveImg->SetData(*tempAve);
delete ave;
delete tempAve;
delete aveMask;

buffer = new Matrix(window, window, 0.0);
buffer2 = new Matrix(BIG_BKG_SZ, BIG_BKG_SZ, 0.0);
if( (!buffer) || (!buffer2) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Calculate stats around each image pixel */
for(x=0; x<rows; x++)
{
    for(y=0; y<cols; y++)
    {
img->CopyBlock(*buffer, x-window/2, y-window/2);
aveImg->CopyBlock(*buffer2, x-BIG_BKG_SZ/2, y-BIG_BKG_SZ/2);
mean->Set(x, y, buffer->Mean());
median->Set(x, y, buffer->Median());
median2->Set(x, y, buffer2->Median());

        } /* End y for loop */
    } /* End x for loop */
```

APPENDIX A. SOURCE CODE

```
/* Copy the matrix from result to img and re-process if needed */
printf("Computing stats . . . \n");
initValue = mean->matrix->Mean();
printf("Mean: %f %f\n", initValue, mean->matrix->Variance(initValue) );
initValue = median->matrix->Mean();
printf("Median: %f %f\n", initValue, median->matrix->Variance(initValue));
initValue = median2->matrix->Mean();
printf("MedAve: %f %f\n", initValue, median2->matrix->Variance(initValue));

/* Write the image in PPM format */
sprintf(fileName, "mean%s.ppm", fileNo);
mean->Write(RAW, fileName);
sprintf(fileName, "median%s.ppm", fileNo);
median->Write(RAW, fileName);
sprintf(fileName, "median2%s.ppm", fileNo);
median2->Write(RAW, fileName);

}
catch (int error)
{
    _exit_(error);
    exit(0);
}
catch (...)
{
    _exit_(UNEXPLAINED);
    exit(0);
}

/* Clean up memory by eallocating the images */
delete img;
delete mean;
delete median;
delete median2;
delete aveImg;
delete buffer;
delete buffer2;
```

APPENDIX A. SOURCE CODE

}

A.9 Source Code for Driver Used in Experiment 2

```

/*****
:   median2.c           Jimmy Swain
:
:   This file is designed to compute an average image from
:   an input image, and then to compute a median from this
:   average image. It takes five command line parameters. The
:   first is the image file in raw format. The second and third
:   give the size of the image. The fourth gives the window size
:   used to compute the averages. The fifth gives the window size
:   of the median window.
:
:   Copyright 1997. All rights reserved.
*****/

#include "stats.h"

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* median; /* median image */
    Image* aveImg; /* image of average block values */
    Matrix* aveMask; /* averaging mask */
    Matrix* tempAve; /* temp storage for an image block */
    char fileName[31]; /* argument containing filename */
    char fileNo[4]; /* unique id number for the file */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    int aveWin; /* averaging window size */
    int medWin; /* median window size */
    double initValue; /* initial value for double variables */
    double value; /* value for double variables */
    int x,y; /* loop indices */

```

APPENDIX A. SOURCE CODE

```
int histogram[MAX_INTENSITY] = {0}; /* region histogram */
double* in; /* pointer to a column */
double* out; /* pointer to a column */
int leftmost; /* offset to col to remove */
int rightmost; /* offset to col to add */

/* Check the command line arguments for correctness */
if (5 > argc)
{
    printf("Usage: median2.exe <image_file> <rows> <cols> ");
    printf("<mean win> <median win>\n");
    printf("%d \n", argc);
    exit(0);
}

/* Get the command line parameters */
cols = atoi(argv[2]);
rows = atoi(argv[3]);
aveWin = atoi(argv[4]);
medWin = atoi(argv[5]);
strncpy(fileNo, &(argv[1][1]), (size_t)(3 * sizeof(char)));
fileNo[3] = '\0';
initValue = (double)(aveWin * aveWin);
initValue = (double)(1 / initValue);

/*
    Check the validity of the command line arguments. The rows and
    cols must contain positive values and further are limited in size
    to values less than MAX_DIMEN.
*/
if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
```

APPENDIX A. SOURCE CODE

```
median = new Image(rows, cols, 0.0);
aveImg = new Image(rows, cols, 0.0);
if( (!img) || (!median) || (!aveImg) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Create a image of 7x7 averages */
    tempAve = new Matrix(rows, cols, 0.0);
    aveMask = new Matrix(aveWin, aveWin, initValue);
    in = new double [ medWin ];
    out = new double [ medWin ];
    if( (!aveMask) || (!tempAve) || (!in) || (!out) )
    {
        printf("Error allocating memory for images\n");
        exit(0);
    }
    img->matrix->Convolve(*aveMask, *tempAve);
    aveImg->SetData(*tempAve);
    delete tempAve;
    delete aveMask;

    /* Calculate median of averages around each image pixel */
    leftmost = medWin / 2;
    rightmost = medWin / 2 + 1;
    for(x=0; x<rows; x++)
    {
        y=0;
        memset( (char*) histogram, 0, (sizeof(int) * MAX_INTENSITY) );
        memset( (char*) in, 0, (sizeof(double) * medWin) );
    }
}
```

APPENDIX A. SOURCE CODE

```
    memset( (char*) out, 0, (sizeof(double) * medWin) );
    aveImg->matrix->GetTally(x, y, medWin, histogram);

    for(y=0; y<cols; y++)
    {
value = aveImg->matrix->Median(in, out, medWin, histogram);
median->Set(x, y, value);
aveImg->matrix->GetCol(x, y-leftmost, medWin, out);
aveImg->matrix->GetCol(x, y+rightmost, medWin, in);

        } /* End y for loop */

    } /* End x for loop */

    /* Compute the results */
    printf("Computing stats . . . \n");

    /* Write the image to disk */
    sprintf(fileName, "m%s.raw", fileNo);
    median->Write(RAW, fileName);

    /* Output image stats */
    value = median->matrix->Mean();
    printf(".....\n" );
    printf("F I L E :  %s\n", fileName );
    printf("S T A T S :  %f", value );
    printf("\t%f\n", median->matrix->Variance(value) );
    printf(".....\n" );
}
catch (int error)
{
    _exit_(error);
    exit(0);
}
catch (...)
{
    _exit_(UNEXPLAINED);
}
```

APPENDIX A. SOURCE CODE

```
    exit(0);
}

/* Clean up memory by eallocating the images */
delete img;
delete median;
delete aveImg;
}
```

A.10 Source Code for Driver Used in Experiment 3

```
/******
: median4.c          Jimmy Swain
:
: This file is designed to compute local area stats across
: an image. It takes four command line parameters. The first
: is the image file in raw format. The second and third give
: the size of the image. The fourth gives the window size
: used to compute the stats. The stats that are considered
: include the mean, the median, and the median of the mean.
:
: Copyright 1997. All rights reserved.
:*****/

#include "stats.h"

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* median; /* median image */
    Matrix* block; /* temp storage for an image block */
    char fileName[31]; /* argument containing filename */
    char fileNo[4]; /* unique id number for the file */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    int window; /* columns in the image */
}
```

APPENDIX A. SOURCE CODE

```
double value; /* value for double variables      */
int x,y; /* loop indices                        */
int histogram[MAX_INTENSITY] = {0}; /* region histogram */
double* in; /* pointer to a column             */
double* out; /* pointer to a column            */
int leftmost; /* offset to col to remove       */
int rightmost; /* offset to col to add         */

/* Check the command line arguments for correctness */
if (5 > argc)
{
    printf("Usage: median4.exe <image_file> <rows> <cols> <window size>\n");
    printf("%d \n", argc);
    exit(0);
}

/* Get the command line parameters */
cols = atoi(argv[2]);
rows = atoi(argv[3]);
window = atoi(argv[4]);
strncpy(fileNo, &(argv[1][1]), (size_t)(3 * sizeof(char)));
fileNo[3] = '\0';

/*
    Check the validity of the command line arguments. The rows and
    cols must contain positive values and further are limited in size
    to values less than MAX_DIMEN.
*/
if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
median = new Image(rows, cols, 0.0);
if( (!img) || (!median) )
```

APPENDIX A. SOURCE CODE

```
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Create a matrix of size window x window */
    block = new Matrix(window, window, 0.0);
    in = new double [window];
    out = new double [window];
    if( (!block) || (!in) || (!out) )
    {
        printf("Error allocating memory for images\n");
        exit(0);
    }

    /* Calculate stats around each image pixel */
    leftmost = window / 2;
    rightmost = window / 2 + 1;
    for(x=0; x<rows; x++)
    {
        y=0;
        memset( (char*) histogram, 0, (sizeof(int) * MAX_INTENSITY) );
        memset( (char*) in, 0, (sizeof(double) * window) );
        memset( (char*) out, 0, (sizeof(double) * window) );
        img->matrix->GetTally(x, y, window, histogram);

        for(y=0; y<cols; y++)
        {
            value = img->matrix->Median(in, out, window, histogram);
            median->Set(x, y, value);
            img->matrix->GetCol(x, y-leftmost, window, out);
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
img->matrix->GetCol(x, y+rightmost, window, in);

    } /* End y for loop */

} /* End x for loop */

/* Copy the matrix from result to img and re-process if needed */
printf("Computing stats . . . \n");

/* Write the image in PPM format */
sprintf(fileName, "m%s.raw", fileNo);
median->Write(RAW, fileName);

/* Output image stats */
value = median->matrix->Mean();
printf(".....\n" );
printf("F I L E :  %s\n", fileName );
printf("S T A T S :  %f", value );
printf("\t%f\n", median->matrix->Variance(value) );
printf(".....\n" );
}
catch (int error)
{
    _exit_(error);
    exit(0);
}
catch (...)
{
    _exit_(UNEXPLAINED);
    exit(0);
}

/* Clean up memory by eallocating the images */
delete img;
delete median;
delete block;
}
```

APPENDIX A. SOURCE CODE

A.11 Source Code for the Nonlinear Filter Implementation

```

/*****
: nl.c          Jimmy Swain
:
: This file is designed to filter an image using a simple
: two class neural network. The classes are derived
: by thresholding one intensity level below the median
: of the image. Class one corresponds to the background,
: and class two corresponds to stroke pixels. All pixels
: recognized as class one pixels get set to the median of
: a 30 by 30 window centered at the pixel.
:
: Copyright 1997. All rights reserved.
*****/

#include "stats.h"

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* median; /* median image */
    Matrix* result; /* resulting image after filtering */
    char fileNo[4]; /* unique id number for the file */
    char fileName[25]; /* file name */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    int window; /* columns in the image */
    double value; /* value for double variables */
    int x,y,i; /* loop indices */
    int histogram[MAX_INTENSITY] = {0}; /* region histogram */
    double* in; /* pointer to a column */
    double* out; /* pointer to a column */
    int leftmost; /* offset to col to remove */
    int rightmost; /* offset to col to add */
    double decision; /* decision value of nn */
    double pattern[3]; /* pixel pattern fed to the nn */
    ofstream outfile; /* outfile for resulting image */

```

APPENDIX A. SOURCE CODE

```
int bytes; /* bytes written to the output stream */

/* Check the command line arguments for correctness */
if (5 > argc)
{
    printf("Usage: nl.exe <image_file> <rows> <cols> <window size>\n");
    printf("%d \n", argc);
    exit(0);
}

/* Get the command line parameters */
cols = atoi(argv[2]);
rows = atoi(argv[3]);
window = atoi(argv[4]);
strncpy(fileNo, &(argv[1][1]), (size_t)(3 * sizeof(char)));
fileNo[3] = '\0';

/*
    Check the validity of the command line arguments. The rows and
    cols must contain positive values and further are limited in size
    to values less than MAX_DIMEN.
*/
//if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
if ((rows <= 0) || (cols <= 0))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
median = new Image(rows, cols, 0.0);
if( (!img) || (!median) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}
```

APPENDIX A. SOURCE CODE

```
/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Create a matrix of size window x window */
    in = new double [window];
    out = new double [window];
    if( (!in) || (!out) )
    {
        printf("Error allocating memory for images\n");
        exit(0);
    }

    /* Calculate stats around each image pixel */
    leftmost = window / 2;
    rightmost = window / 2 + 1;
    for(x=0; x<rows; x++)
    {
        y=0;
        memset( (char*) histogram, 0, (sizeof(int) * MAX_INTENSITY) );
        memset( (char*) in, 0, (sizeof(double) * window) );
        memset( (char*) out, 0, (sizeof(double) * window) );
        img->matrix->GetTally(x, y, window, histogram);

        for(y=0; y<cols; y++)
        {
            value = img->matrix->Median(in, out, window, histogram);
            median->Set(x, y, value);
            img->matrix->GetCol(x, y-leftmost, window, out);
            img->matrix->GetCol(x, y+rightmost, window, in);

            } /* End y for loop */

        } /* End x for loop */
}
```

APPENDIX A. SOURCE CODE

```
/* Clean up memory */
delete in;
delete out;

/* Allocate storage needed for the next step */
result = new Matrix(10, cols, 0.0);
if( !result )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Open the output file */
bytes = 0;
sprintf ( fileName, "r%s.raw", fileNo );
outfile.open( fileName, ios::out | ios::bin, 0 );
if ( 0 == outfile )
{
    throw FILE_ERROR;
}

/* Apply the neural network */
for(x=0; x<rows; x++)
{
    for(y=0; y<cols; y++)
    {
value = median->Get(x,y);
value--;
for(i=-1; i<2; i++)
{
    pattern[i+1] = img->Get(x, y+i);
    if ( pattern[i+1] >= value )
    {
        pattern[i+1] = 1;
    }
else
{
```

APPENDIX A. SOURCE CODE

```
    pattern[i+1] = -1;
  }
}

/* Apply the decision function */
value++;
decision = pattern[0] + 3 * pattern[1] + pattern[2] + 1;
    if ( 0 <= decision )
{
    result->Set(x%10,y,value);
}
else /* ( 0 > decision ) */
{
    result->Set(x%10,y,img->Get(x,y));
}

    } /* End y for loop */

    /* If the buffer is full, write it to disk */
    if ( (9 == x%10) || !((x+1) < rows) )
    {
        result->Write( outfile, bytes, rows*cols );
    }

} /* End x for loop */

/* Close the outfile */
outfile.flush();
outfile.close();
}
catch (int error)
{
    _exit_(error);
    exit(0);
}
catch (...)
```

APPENDIX A. SOURCE CODE

```
{
    _exit_(UNEXPLAINED);
    exit(0);
}

/* Clean up memory by eallocating the images */
delete img;
delete median;
delete result;
}
```

A.12 Driver Program for the Erosion Operation

```

/*****
:   erode.c           Jimmy Swain
:
:   This file is designed to compute the morphological erosion
:   operation on an image.  It takes three command line
:   parameters.  The first is the image file in raw format.
:   The second and third give the size of the image.
:
:   Copyright 1997.  All rights reserved.
*****/

#include "morph.h"
#define SZ 7

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* res; /* the resulting image */
    Matrix* struc; /* structuring element */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    double value; /* input value for structuring element */
    int i,j; /* loop indices */
    fstream infile; /* structuring element file */

    /* Check the command line arguments for correctness */
    if (4 != argc)
    {
        printf("Usage: erode.exe <image_file> <rows> <cols>\n");
        printf("%d \n", argc);
        exit(0);
    }

    /* Get the command line parameters */
    cols = atoi(argv[2]);
    rows = atoi(argv[3]);

```

APPENDIX A. SOURCE CODE

```
/*
    Check the validity of the command line arguments. The rows and
    cols must contain positive values and further are limited in size
    to values less than MAX_DIMEN.
if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}
*/

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
res = new Image(rows, cols, 0.0);
struc = new Matrix(SZ, SZ, 0.0);
if( (!img) || (!res) || (!struc) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Set the structuring Element */
infile.open ( "erode.se", ios::in, 0 );
for ( i=0; i<SZ; i++ )
{
    for ( j=0; j<SZ; j++ )
    {
        infile >> value;
        struc->Set ( i, j, value );
    }
}
infile.close();

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
```

APPENDIX A. SOURCE CODE

```
img->Load(RAW, argv[1]);

/* Perform the erosion */
img->matrix->Erode ( *struc, SZ, *(res->matrix) );
res->Write ( RAW, "erode.ppm" );

}
catch (int error)
{
    _exit_(error);
    exit(0);
}
catch (...)
{
    _exit_(UNEXPLAINED);
    exit(0);
}

/* Clean up memory by eallocating the images */
delete img;
delete res;
delete struc;
}
```

A.13 Driver Program for the Dilate Operation

```
/******
: dilate.c          Jimmy Swain
:
: This file is designed to compute the morphological dilation
: operation on an image. It takes three command line
: parameters. The first is the image file in raw format.
: The second and third give the size of the image.
:
: Copyright 1997. All rights reserved.
*****/
```

APPENDIX A. SOURCE CODE

```
#include "morph.h"
#define SZ 7

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* res; /* the resulting image */
    Matrix* struc; /* structuring element */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    double value; /* input value for structuring element */
    int i,j; /* loop indices */
    ifstream infile; /* infile to read struc element from */

    /* Check the command line arguments for correctness */
    if (4 != argc)
    {
        printf("Usage: erode.exe <image_file> <rows> <cols>\n");
        printf("%d \n", argc);
        exit(0);
    }

    /* Get the command line parameters */
    cols = atoi(argv[2]);
    rows = atoi(argv[3]);

    /*
     Check the validity of the command line arguments. The rows and
     cols must contain positive values and further are limited in size
     to values less than MAX_DIMEN.
    if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
    {
        printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
    }
    */

    /* Allocate the storage needed for the two images */
```

APPENDIX A. SOURCE CODE

```
img = new Image(rows, cols, 0.0);
res = new Image(rows, cols, 0.0);
struc = new Matrix(SZ, SZ, 0.0);
if( (!img) || (!res) || (!struc) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Set the structuring Element */
infile.open ( "dilate.se", ios::in, 0 );
for ( i=0; i<SZ; i++ )
{
    for ( j=0; j<SZ; j++ )
    {
        infile >> value;
        struc->Set ( i, j, value );
    }
}
infile.close();

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Perform the erosion */
    img->matrix->Dilate ( *struc, SZ, *(res->matrix) );
    res->Write ( RAW, "dilate.raw" );

}
catch (int error)
{
    _exit_(error);
    exit(0);
}
```

APPENDIX A. SOURCE CODE

```
catch (...)  
{  
    _exit_(UNEXPLAINED);  
    exit(0);  
}  
  
/* Clean up memory by eallocating the images */  
delete img;  
delete res;  
delete struc;  
}
```

A.14 Driver Program for the Gap Filling Operation

```

/*****
: fill.c          Jimmy Swain
:
: This file is designed to filter an image using a simple
: gap filler. The logic examines scan lines about a pixel
: of a specified size to see if the center pixel is sand-
: wached between stroke data. If this is the case, and
: the scan line normal to the current direction is all
: background values, then the pixel is interpolated to a
: stroke value. If one of these conditions fail, then the
: value is passed unaltered. The directions that are
: examined include diagonal (/), horizontal (-), and
: vertical (|).
:
: Copyright 1997. All rights reserved.
*****/

#include "stats.h"

int main(int argc, char* argv[])
{
    Image* img; /* the original image */
    Image* median; /* median image */
    Matrix* result; /* resulting image after filtering */
    char fileNo[4]; /* unique id number for the file */
    char fileName[25]; /* file name */
    int rows; /* rows in the image */
    int cols; /* columns in the image */
    int window; /* columns in the image */
    double value; /* value for double variables */
    int x,y,i; /* loop indices */
    int histogram[MAX_INTENSITY] = {0}; /* region histogram */
    double* in; /* pointer to a column */
    double* out; /* pointer to a column */
    int leftmost; /* offset to col to remove */
    int rightmost; /* offset to col to add */

```

APPENDIX A. SOURCE CODE

```
double medValue; /* median of input image      */
double target; /* target value slightly below median */
ofstream outfile; /* outfile for resulting image */
int bytes; /* bytes written to the output stream */
double top; /* top summation */
double bottom; /* bottom summation */
double left; /* left summation */
double right; /* right summation */
double bottom_left; /* bottom left summation */
double bottom_right; /* bottom right summation */
double top_left; /* top left summation */
double top_right; /* top right summation */
double newValue; /* new value for result */
int offset; /* index offset */
int count; /* pixels used in summations */
int changes; /* changes made in the gap filling */

/* Check the command line arguments for correctness */
if (6 > argc)
{
    printf("Usage: nl.exe <image_file> <rows> <cols> <med win> <discon win>\n");
    printf("%d \n", argc);
    exit(0);
}

/* Get the command line parameters */
cols = atoi(argv[2]);
rows = atoi(argv[3]);
window = atoi(argv[4]);
strncpy(fileNo, &(argv[1][1]), (size_t)(3 * sizeof(char)));
fileNo[3] = '\0';

/*
    Check the validity of the command line arguments. The rows and
    cols must contain positive values and further are limited in size
    to values less than MAX_DIMEN.
*/
```

APPENDIX A. SOURCE CODE

```
//if ((rows <= 0) || (cols <= 0) || (rows > MAX_DIMEN) || (cols > MAX_DIMEN))
if ((rows <= 0) || (cols <= 0))
{
    printf("Dimensions violate range constraints: [%d,%d]\n", 0, MAX_DIMEN);
}

/* Allocate the storage needed for the two images */
img = new Image(rows, cols, 0.0);
median = new Image(rows, cols, 0.0);
if( (!img) || (!median) )
{
    printf("Error allocating memory for images\n");
    exit(0);
}

/* Perform the statistics calculations. */
try
{
    /* Load image data for each file */
    img->Load(RAW, argv[1]);

    /* Create a matrix of size window x window */
    in = new double [window];
    out = new double [window];
    if( (!in) || (!out) )
    {
        printf("Error allocating memory for images\n");
        exit(0);
    }

    /* Create a median image */
    leftmost = window / 2;
    rightmost = window / 2 + 1;
    for(x=0; x<rows; x++)
    {
        y=0;
        memset( (char*) histogram, 0, (sizeof(int) * MAX_INTENSITY) );
    }
}
```

APPENDIX A. SOURCE CODE

```
    memset( (char*) in, 0, (sizeof(double) * window) );
    memset( (char*) out, 0, (sizeof(double) * window) );
    img->matrix->GetTally(x, y, window, histogram);

    for(y=0; y<cols; y++)
    {
value = img->matrix->Median(in, out, window, histogram);
median->Set(x, y, value);
img->matrix->GetCol(x, y-leftmost, window, out);
img->matrix->GetCol(x, y+rightmost, window, in);

        } /* End y for loop */
    } /* End x for loop */

    /* Clean up memory */
    delete in;
    delete out;

    /* Allocate storage needed for the next step */
    result = new Matrix(10, cols, 0.0);
    if( !result )
    {
        printf("Error allocating memory for images\n");
        exit(0);
    }

    /* Open the output file */
    bytes = 0;
    sprintf ( fileName, "rc%s.raw", fileNo );
    outfile.open( fileName, ios::out | ios::bin, 0 );
    if ( 0 == outfile )
    {
        throw FILE_ERROR;
    }

    /* Apply the neural network */
    window = atoi(argv[5]);
```

APPENDIX A. SOURCE CODE

```
    offset = window/2;
    changes = 0;
    for(x=0; x<rows; x++)
    {
        for(y=0; y<cols; y++)
        {
/* Initialize our summation values */
top = bottom = left = right = 0.0;
top_right = top_left = bottom_right = bottom_left = 0.0;
count = 0;

/* Get the pixel value that we are examining and the local median */
value = img->Get(x, y);
medValue = median->Get(x, y);
target = medValue - 2;

/*
    Sum up the regions in the horizontal and vertical
    directions. We do not include the middle three as
    we wish to reconnect somewhat larger breaks.
        */
for(i=0; i<=offset; i++ )
{
    // if ( (-1 != i) && (0 != i) && (1 != i) )
    if ( 0 != i )
    {
        top += img->Get(x-i, y);
        bottom += img->Get(x+i, y);
        left += img->Get(x, y-i);
        right += img->Get(x, y+i);
        top_right += img->Get(x-i, y+i);
        top_left += img->Get(x-i, y-i);
        bottom_left += img->Get(x+i, y-i);
        bottom_right += img->Get(x+i, y+i);
        count++;
    }
}
}
```

APPENDIX A. SOURCE CODE

```
if (!count)
{
    count = 1;
}
top /= count;
bottom /= count;
left /= count;
right /= count;
bottom_left /= count;
bottom_right /= count;
top_left /= count;
top_right /= count;

/* Determine if this is a narrow break */
if( medValue == value )
{
    if( (target > left) && (target > right) )
    {
        if ( isNormalBackground(HORIZONTAL, offset, medValue, *img, x, y) )
        {
            //newValue = ( left + right ) / 2;
            newValue = 0;
            changes++;
        }
        else
        {
            newValue = value;
        }
    }
    else if( (target > bottom_left) && (target > top_right) )
    {
        if ( isNormalBackground(DIAG_45, offset, medValue, *img, x, y) )
        {
            //newValue = ( bottom_left + top_right ) / 2;
            newValue = 0;
            changes++;
        }
    }
}
```

APPENDIX A. SOURCE CODE

```
    else
    {
        newValue = value;
    }
}
else if( (target > top) && (target > bottom) )
{
    if ( isNormalBackground(VERTICAL, offset, medValue, *img, x, y) )
    {
        //newValue = ( bottom + top ) / 2;
        newValue = 0;
        changes++;
    }
    else
    {
        newValue = value;
    }
}
/*
else if( (target > top_left) && (target > bottom_right) )
{
    if ( isNormalBackground(DIAG_N45, 3*offset, medValue, *img, x, y) )
    {
        //newValue = ( top_left + bottom_right ) / 2;
        newValue = 0;
        changes++;
    }
    else
    {
        newValue = value;
    }
}
*/
else
{
    newValue = value;
}
}
```

APPENDIX A. SOURCE CODE

```
}
else if ( (medValue-1) > value )
{
    if( 5 <= value )
    {
        newValue = value - 5;
    }
    else
    {
        newValue = value;
    }
}
else
{
    newValue = value;
}

/* Store the new computation */
result->Set(x%10, y, newValue);

    } /* End y for loop */

    /* If the buffer is full, write it to disk */
    if ( (9 == x%10) || !((x+1) < rows) )
    {
        result->Write( outfile, bytes, rows*cols );
    }

} /* End x for loop */

/* Output the # of changes made */
printf("Number of changes: %d.\n", changes);

/* Close the outfile */
outfile.flush();
outfile.close();
```

APPENDIX A. SOURCE CODE

```
    }
    catch (int error)
    {
        _exit_(error);
        exit(0);
    }
    catch (...)
    {
        _exit_(UNEXPLAINED);
        exit(0);
    }

    /* Clean up memory by deallocating the images */
    delete img;
    delete median;
    delete result;
}

/*
: isNormalBackground
:
: this function determines if the scan line in the normal
: direction to the specified scan line is set to all
: background values.
:
: returns TRUE if normal is all background
: returns FALSE otherwise
*/
bool isNormalBackground (
int direction, /* direction we are looking at */
int offset, /* 1/2 of the decision window size */
double bg, /* background value */
Image& i, /* image we are scanning */
int x, /* x coordinate of center value */
int y /* y coordinate of center value */
)
{
```

APPENDIX A. SOURCE CODE

```
int j; /* loop index */
double pixel; /* stores one pixel from scan line */

/* Examine each value on the perpendicular scan line */
for ( j=-offset; j<=offset; j++ )
{
    switch ( direction )
    {
        case HORIZONTAL:
            pixel = i.Get ( x + j, y );
            break;
        case VERTICAL:
            pixel = i.Get ( x, y + j );
            break;
        case DIAG_45:
            pixel = i.Get ( x + j, y + j );
            break;
        case DIAG_N45:
            pixel = i.Get ( x - j, y + j );
            break;
    }

    if ( bg != pixel )
    {
        return FALSE;
    }
}

/* If we reach this point, then the scan line is entirely background */
return TRUE;
}
```

REFERENCES

- [Andrews and Hunt, 1977] Andrews, H. C. and Hunt, B. R. (1977). *Digital Image Restoration*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
- [Bates and McDonnell, 1986] Bates, R. H. T. and McDonnell, M. J. (1986). *Image Restoration and Reconstruction*. Oxford University Press, New York.
- [Berriel et al., 1983] Berriel, L. R., Bescos, J., and Santisteban, A. (1983). Image restoration for a defocused optical system. *Applied Optics*, 22(18):2772–2780.
- [Bracewell, 1978] Bracewell, R. N. (1978). *The Fourier Transform and Its Applications*. McGraw-Hill Book Company, New York.
- [Duda and Hart, 1973] Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley and Sons, Inc., New York.
- [Ekstrom, 1984] Ekstrom, M. P., editor (1984). *Digital Image Processing Techniques*. Academic Press, Inc., Orlando, Florida.
- [Giannakis, 1989] Giannakis, G. B. (1989). Signal reconstruction from multiple correlations: Frequency and time domain approaches. *Journal of the Optical Society of America*, 6(5):682–696.
- [Gonzalez and Woods, 1992] Gonzalez, R. C. and Woods, R. E. (1992). *Digital Image Processing*. Addison-Wesley Press, Reading, Massachusetts.
- [Haralick et al., 1987] Haralick, R. M., Sternberg, S. R., and Zhuang, X. (1987). Image analysis using mathematical morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(4):532–550.
- [Horner, 1969] Horner, J. L. (1969). Optical spatial filtering with the least mean-square-error filter. *Journal of the Optical Society of America*, 59:553–558.
- [Inbar and Marom, 1993] Inbar, H. and Marom, E. (1993). Matched, phase-only, or inverse filtering with joint-transform correlators. *Optics Letters*, 18(19):1657–1659.
- [Jain, 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., New York.
- [James L. Harris, 1966] James L. Harris, S. (1966). Image evaluation and restoration. *Journal of the Optical Society of America*, 56(5):569–574.

REFERENCES

- [Lim, 1990] Lim, J. S. (1990). *Two-Dimensional Signal and Image Processing*. Prentice Hall, Englewood Cliffs, New Jersey.
- [Meinel, 1986] Meinel, E. S. (1986). Origins of linear and nonlinear recursive restoration algorithms. *Journal of the Optical Society of America*, 3(6):787-799.
- [Rino, 1969] Rino, C. L. (1969). Bandlimited image restoration by linear mean-square estimation. *Journal of the Optical Society of America*, 59:547-553.
- [Robbins and Huang, 1972] Robbins, G. M. and Huang, T. S. (1972). Inverse filtering for linear shift-variant imaging systems. *Proceedings of the IEEE*, 60(7):862-872.
- [Rosenfeld and Kak, 1982] Rosenfeld, A. and Kak, A. C. (1982). *Digital Picture Processing*. Academic Press, Inc., New York, New York.
- [Slepian, 1967] Slepian, D. (1967). Linear least-squares filtering of distorted images. *Journal of the Optical Society of America*, 57(7-12):918-922.
- [Som, 1971] Som, S. C. (1971). Analysis of the effect of linear smear on photographic images. *Journal of the Optical Society of America*, 61(7):859-864.
- [Sondhi, 1972] Sondhi, M. M. (1972). Image restoration: The removal of spatially invariant degradations. *Proceedings of the IEEE*, 60(7):842-853.
- [Stark, 1987] Stark, H., editor (1987). *Image Recovery: Theory and Application*. Academic Press, Inc., Orlando, Florida.
- [Tou and Gonzalez, 1974] Tou, J. T. and Gonzalez, R. C. (1974). *Pattern Recognition Principles*. Addison-Wesley Publishing Company, Reading, Massachusetts.

VITA

David James Swain, the son of George C. and Linda W. Swain, was born January 5, 1973. He was born and raised in Eden, North Carolina. He graduated from John Motley Morehead High School in 1991. He received the Junius C. and Eliza Brown academic scholarship to attend Wake Forest University. He received a B.S. in Computer Science and Mathematics from this institution in May, 1995. This thesis completes a M. S. in Computer Science from Virginia Polytechnic Institute and State University.