ABSTRACT

BRENZOVICH, JOSEPH A. Fabric Defect Detection using a GA Tuned Wavelet Filter (under the direction of Warren J. Jasper and Jeffrey A. Joines)

The purpose of this research project is to show that a computerized system based on image processing software is capable of identifying defects in woven fabrics. Current defect detection is carried out through use of visual inspection of fabric rolls after the rolls have been doffed from the production machinery, which adds a substantial lag between defect creation and detection. Existing methods for automatic defect detection rely on methods that suffer from substantial analysis time or a low percentage of detection. The method described in this thesis represents a quick and accurate approach to automatic defect detection and is capable of identifying defects such as lines, tears, and spots. Utilizing a Genetic Algorithm (GA) as the primary means of solving the wavelet filter equations with respect to a fabric image proved adequate in the construction of a wavelet filter that was capable of removing large amounts of the fabric texture from the image, thus allowing defect segmentation algorithms to run more effectively. Although a real-time system is not developed, suggestions for constructing such a system are presented. This work provides a foundation for the development of a real-time automated defect detector based on the algorithms and methodologies employed in this work.

Fabric Defect Detection

using a GA Tuned Wavelet Filter

by

Joseph A. Brenzovich

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

TEXTILE ENGINEERING

Raleigh

2003

APPROVED BY:

_____
Dr. Kailash Misra

_____
Dr. Warren Jasper
Co-Chair of Advisory Committee

_____
Dr. Jeffrey Joines
Co-Chair of Advisory Committee

DEDICATION

To my late uncle, Michael Stephen Murdock, for his ever present support and encouragement

BIOGRAPHY

Joseph A. Brenzovich, the first child of William Sr. and Susan Brenzovich, was born on February 25, 1979 in Mechanicsville, Virginia. He received a Bachelor of Science in Textile Engineering from North Carolina State University in 2001. He is currently pursuing a Master of Science in Textile Engineering at North Carolina State University. His future position will be with Milliken and Company at Magnolia Finishing Plant, where he will be employed as a Product and Process Improvement Engineer.

ACKNOWLEDGEMENTS

I wish to express my appreciation and thanks to Dr. Warren Jasper and Dr. Jeff Joines for their guidance, support, encouragement, and patience throughout the duration of this research. I would also like to thank Dr. Kailash Misra for his help. A special thanks is extended to Dr. Jon Rust for his continual advice and the use of his image gathering equipment. Finally, my deepest thanks go to my best friend Kristin, for her support, love, patient, and unwavering encouragement.

TABLE OF CONTENTS

List of Tables

List of Figures

1.0 Introduction

The production of first quality fabric is the foremost goal of the modern textile industry. First quality fabric is totally free of major defects and virtually free of minor structural or surface defects. Second quality fabric, which is fabric that may contain a few major defects and/or several minor structural or surface defects, represents a loss in revenue for a manufacturer since the product will now sell for only 45%-65% the price of first quality fabric [Chan and Pang 2000], while using the same amount of production resources. Because production speeds are faster than ever, manufacturers must be able to identify defects, locate their source, and make the necessary corrections in less time so as to reduce the amount of second quality fabric. This in turn places a greater strain on the inspection departments of the manufacturers.

Typical textile mills employ humans to inspect and grade the fabric in the production facility. The job is monotonous, as it requires the employee to sit at an inspection frame and watch as fabric that is 5-9ft wide passes over the board at speeds ranging from 8-20 yards a minute, all the while visually scanning that wide area of fabric for possible defects. The average human inspection department is only able to find 60%-75% of existing defects [Chan and Pang 2000], which translates into a substantial amount of second quality shipped and or returned. This alone leads to a considerable reduction in production efficiency, as most customers will only accept a certain percentage of second quality fabric in their order, meaning that the production facility must spend time producing re-work to be able to meet the customer demands. In an effort to improve the efficiency of inspection departments and subsequently reduce the costs of production, textile manufacturers have begun turning to automated inspection systems.

An automated inspection system usually consists of a computer based vision system. Because they are computer based, these systems do not suffer the drawbacks of human visual inspection, such as fatigue, boredom, or inattentiveness. Automated systems are able to inspect fabric in a continuous manner without pause. Most of these automated systems are off-line or off-loom systems; they inspect rolls of fabric that have already been removed from the production machinery. Should any defects be found that are mechanical in nature (i.e., missing ends or oil spots), the lag time that exists between actual production and inspection translates into more fabric that is produced on the machine that is causing these defects. Therefore, to be more efficient, inspection systems must be implemented on-loom.

The advantages of an on-loom inspection system are numerous. Perhaps the most valuable asset to an on-loom system is that if it identifies a defect that is mechanical in nature, it would be possible to shut down the loom and correct the problem before any more off-quality fabric is produced, which translates into substantial savings for the manufacturer. Should the defect be non-mechanical in nature (yarn defects), the system can still flag the defect on a report so that the manufacturer and subsequently the customer is aware of its presence.

However, there are several disadvantages to such a system. First, instead of utilizing three or four inspection frames, an on-loom inspection system would be needed for each loom in the facility. Secondly, cost is an issue because there needs to be one inspection system per loom rather than three or four systems per facility. Thirdly, the environment in a weaving mill contains large amounts of dust, lint, and other particulate matter which can interfere with the image gathering equipment. Finally, the looms themselves create lots of

excessive vibration which could decrease image quality by adding varying levels of blurriness to the captured images.

Considerable research has been conducted to find feasible algorithms on which to base an automated inspection system. The algorithms that have been studied include Fourier analysis, Gabor filters, and wavelet filters. While the first two have been shown to be effective methods with which to detect the presence of defects, they are computationally expensive algorithms, which could necessitate a slower production speed to ensure a feasible system. A system based on wavelet filters, which are compact in nature and computationally efficient, would allow defect detection to occur at current industry production speeds of approximately 1000 picks per minute. The only drawback to such a system stems from the complexity of the filter construction, which in turn is dependent on the complexity of the fabric texture under inspection. The system must first be trained to recognize the fabric texture, as the quality of the training phase determines the quality of defect detection.

The hypothesis for this work is that given an efficient and effective method to derive the optimal wavelet filter for a given texture, a significant level of defect detection can be achieved. The subject of this study is “fabric defect detection using a GA tuned wavelet filter”.

2.0 Literature Review

2.1 Defect Detection

2.1.1 Introduction to Defect Detection

In today's textile industry, production speeds are faster than ever before. Manufacturers of woven and knitted products are able to produce more product in less time owing to substantial improvements in the technology driving the processes. Because of this increased throughput, manufacturers are spending more time on quality assurance of the outgoing product, which in turn places a greater strain on the human element. Therefore, it is highly important that a manufacturer be able to accurately gage quality during production in order to lower costs associated with quality inspection and to increase profits by producing less off quality [Sari-Sarraf and Goddard 1999].

Most textile production facilities employ human inspectors to serve as quality assurance associates. These inspectors sit in front of an inspection frame and observe the fabric as it passes over the board at speeds ranging from 8-20 meters per minute [Baykut et. al. 2000; Kumar and Pang 2002; Özdemir et. al. 1997]. Due to the monotonous nature of this position, it is not uncommon for the associate to miss defects that appear on or in the fabric structure; furthermore, a human inspector is not capable of viewing the inspection frame in its entirety at any given time due to the size of the fabric itself (most looms produce fabric between 60 and 108 inches wide) [Baykut et. al. 2000; Kumar and Pang 2002; Özdemir et. al. 1997]. Studies suggest that on average, human visual inspection is capable of catching approximately 60%-75% of all significant defects present in the fabric [Chan and Pang

2000]. Because the price of second quality fabric is a meager 45%-65% of that the first quality price, it is crucial for the manufacturer to find ways to improve the detection rate of their inspection departments [Chan and Pang 2000].

A defect with respect to the textile industry is defined as a flaw in or on the structure of the fabric [Kumar and Pang 2002; Chan and Pang 2000; Kumar and Pang 2000; Sari-Sarraf and Goddard 1999]. There are currently over 50 categories of defects known to the weaving process alone. Most of these defects appear only in the direction of motion on the loom (the warp direction) or across the width of the fabric (the pick direction) [Sari-Sarraf and Goddard 1999]. Most defects are yarn related, such as mispicks, end outs, or broken yarns. Other defects are caused by slubs, or waste, becoming trapped in the fabric structure as it is created. Additional defects are mostly machine related, and manifest themselves in the forms of structural failures (tears or holes) or machine residue (oil spots or dirt) [Sari-Sarraf and Goddard 1999]. Coupled with the size and speed of the fabric as it passes over the inspection frame, the wide range of defects serve to add complexity to visual inspection and increase the probability of missed defects [Baykut et. al. 2000; Chan and Pang 2000; Sari-Sarraf and Goddard 1999; Yang et. al. 2002].

Because of the high percentage of defects that human visual inspection misses, a more accurate and efficient method for defect detection is needed. Therefore, the textile industry has been moving towards automated inspection [Baykut et. al. 2000; Kumar and Pang 2002; Chan and Pang 2000; Kumar and Pang 2000; Sari-Sarraf and Goddard 1999; Yang et. al. 2002]. Automated fabric inspection can provide the industry with higher standards of accuracy than human inspectors, thus saving the manufacturer money in lost goods and production time. However, most of these automated systems are only available in

the form of off-line or off-loom processes that inspect fabric rolls that have been removed from the production line. The most precise and accurate, and therefore efficient, form of inspection is an on-loom system that can monitor the fabric as it is being produced [Sari-Sarraf and Goddard 1999].

Most of the on-loom systems are based on texture recognition. This allows the system to identify defects by looking for deviations from the fabric's normal texture [Baykut et. al. 2000; Kumar and Pang 2002; Chan and Pang 2000; Sari-Sarraf and Goddard 1999]. These approaches use a variety of methods to capture the texture information of the fabric and subsequently identify any defects. These methods include Fourier analysis, Gabor filters, Markov Random fields, and wavelet transforms.

2.1.2 Fourier Analysis

One proposed method to automate defect detection involves using Fourier analysis to compare the power spectrum plot of an image containing a defect with that of a defect free image. This comparison focuses on shifts in the normalized intensity between one plot and the other, which could signify the presence of a defect.

Chan and Pang [2000] studied warp or fill direction defects by comparing the spectrum plots from images with no defect and those containing defects. Their work showed that a missing or broken yarn in either direction could be detected by an upward shift in the normalized magnitude of the associated spectrum plot. This upward shift occurs because more light is being transmitted through the fabric owing to the open space left by the missing yarn.

Furthermore, the presence of a double pick or warp yarn could be detected by a downward shift in the spectrum from the original image to the defect image. Consequently,

tears or holes in the fabric, and any other two dimensional defects could be detected in either the warp or pick direction. However, detection of any one-dimensional defects is solely dependent on the orientation of the analysis; i.e. the warp or pick direction.

Another study on Fourier analysis for defect detection was done by Tsai and Huang [2003], who focused on detecting surface defects of various textured materials. Their work showed that by taking a small circular sample from the Fourier spectrum image and setting the frequency components at the center and outside the circle to zero, the repetitive global texture would be eliminated which would serve to emphasize any defects that were present. From these reconstructed images, a feature extractor (i.e., a thresholding method) could then be employed to highlight any defects that may be present.

Fourier analysis is suitable for defect detection when the defect distorts the material on a global scale, and therefore no spatial information is required. However, this analysis suffers from its lack of any spatial information from the original image, since it captures only frequency information. To be able to identify the placement of localized defects with respect to the original image, additional algorithms would be needed, thereby negating the need for Fourier analysis [Kumar and Pang 2000]. Furthermore, Fourier analysis is a computationally expensive method, with a computational time proportional to $2N^2 \log_2 N$ for two-dimensional transforms [Chan and Pang 2000], such as those previously discussed.

2.1.3 Gabor Filters

Another method used as the basis for automated defect detection systems is the Gabor Filter. Gabor filters have recently received a great deal of attention in the field of computer vision and defect detection. This has been motivated by a theory proposed by Campbell and Robson [1968] which states that the human visual cortex decomposes images captured by the

retina into several filtered images, each containing varying intensities over a narrow band of frequency and orientation. The neurons in the brain are individually tuned to a particular combination of frequency and orientation, which denotes a channel. These channels, therefore, closely resemble Gabor functions. Because of this, researchers have suggested that computer vision systems utilize these functions to more closely mimic the texture recognition abilities of mammalian brains [Kumar and Pang 2000].

Kumar and Pang [2000] have studied the usage of real Gabor functions in defect detection. They did not study imaginary Gabor functions, since these functions serve to enhance edge detection, which, in a complex textural environment, serve only to emphasize the edges to the point of washing out the background texture. The real Gabor functions serve as blob detectors, which can be employed to enhance changes in a repeating texture.

Kumar and Pang utilized a bank consisting of 16 different Gabor filters, each of a different size and orientation, as their construct for defect detection. By supplying the filter algorithm with details on the defects contained in the fabric (size, orientation, etc.), the filter construct could be considered a supervised defect segmentation algorithm. They showed that this system was effective for identifying defects of varying sizes and orientations when the filter bank was customized to the structure type of the given defect [Kumar and Pang 2000].

Another study by Kumar and Pang [2002] built upon their previous work to construct an unsupervised defect segmentation system. In this new system, a filter bank consisting of 18 Gabor filters at 3 scales and 6 orientations was used. Each of these filters was tuned to a narrow frequency and orientation range. Fabric images were passed through the filter bank, which served as a spatial mask, and the resulting magnitude information was collected.

The system would then apply a non-linear function to the magnitude information to rectify the multi-channel filter response [Kumar and Pang 2002]. The non-linear function used served to convert negative amplitudes to positive amplitudes, both of which are outputs of a Gabor filter. Once the image data has been rectified in this manner, it is compared to previously analyzed data whose source was a defect free fabric image. This allows for the filtering of background texture and the subsequent revealing of defects.

Kumar and Pang [2002] then applied a data fusion scheme to the filtered data to combine all image data into one composite image. Once this was complete, a thresholding algorithm was applied to generate a binary image for defect segmentation. They found that this system was effective at detecting different classes of defects when the filter bank was oriented in a manner similar to that of the defect in question. Since different defects appear in different sizes and orientations, the filter bank must have a component or components of similar sizes and orientations to effectively segment those defects.

Gabor filters serve as proven blob detectors. However, the filter itself must be in a similar orientation as that of the defect in question for effective defect segmentation. The methods discussed previously deal with multi-filter systems composed of numerous Gabor filters, each of a different size and orientation. Because of the complexity added to the system by these additional filters, the system as a whole suffers a severe computational penalty in return for its ability to accurately and effectively identify defects in a fabric image. Therefore, systems that do not rely on multiple filter constructs would serve as more efficient methods for defect detection.

2.1.4 Markov Random Fields

The theory behind Markov Random Fields (MRF) as it relates to image analysis states that the brightness level of a pixel is dependent on the brightness levels of neighboring pixels, providing that the image is not random noise [Baykut et. al. 2000]. Therefore, the MRF model attempts to derive the relationship between the brightness levels of pixels and their neighbors as a means of capturing the texture information contained in that image.

Baykut et. al. [2000] applied the MRF model to defect segmentation of fabric images. Their procedure consisted of first obtaining fabric images that were defect free. Next, the MRF model was applied to that image such that the image could be quantified in terms of the model; in essence, statistics that describe that particular texture in terms of the MRF model were gathered. The training phase of the algorithm was then complete.

Because their model had quantified the texture of a fabric, Baykut et. al. [2000] could then apply this filter to images of the same fabric and search for defects. The new image would be subdivided into several square windows and the filter construct would be applied to each window. From this, a probability statistic for each window could be calculated and compared with the statistic from the original training texture. Should these probabilities disagree at some confidence level, it could be concluded that the window in question contained a defect [Baykut et. al. 2000].

The MRF model has been shown to be an excellent tool for identifying the approximate location of a defect by comparing the texture in a small window of the unknown image to the texture of the training image. Though this method has been shown to be effective in detecting defects, it gives little spatial information concerning location and defect type. To be able to perform defect segmentation on the filtered image, a feature extractor

would be required. Methods that combine defect detection and segmentation into one filter would be more efficient than the MRF model.

2.1.5 Wavelet Transforms

A wavelet function is a compact, finite duration orthonormal signal that forms a basis for the signal subspace [Jasper et. al. 1996]. Wavelet based multi-resolution analysis (MRA) decomposes a signal into low and high frequency information and can be used in texture analysis since it decomposes the texture across several different scales, which serves to greatly improve analysis. The texture can now be inspected at various scales such that a feature vector comprised of significant features at each scale can be created and used as a base for defect classification [Jasper et. al. 1996].

Yang et. al. [2002] used adaptive wavelets combined with a discriminative feature extractor to perform defect segmentation. The design of the wavelet was incorporated with the design of the detector parameters to result in a high detection rate. Because they were using undecimated wavelet transforms to perform the analysis, the images were subjected to a single pass of the filter construct. The proposed detection method involves several passes through the detection algorithm that consists of the feature extractor (based on the adaptive wavelet filter) followed by the defect detector (a Euclidean distance based detector). These passes were necessary to obtain the optimal settings on the wavelet filter, and served as a training phase for the system.

The training method that Yang et. al. [2002] utilized to minimize the detection error of the feature extractor and detector combination was couched as an unconstrained optimization problem over the set of coefficients that define the wavelet filter and the Euclidean distance detector. As described in Yang et. al. [2002], a loss function (i.e. the

empirical cost for the total set of training samples) based on the measure of incorrect detection was formulated to serve as the basis for the optimization. By minimizing the empirical cost, the optimal set of coefficients for both the feature extractor and the defect detector were obtained. The results gathered by Yang et. al. suggest that the adaptive wavelet construct serves as a better feature extractor than other wavelet types (Haar, Daubechies, and other standard forms). Their results further suggest that an adaptive wavelet with 3-scale wavelet features achieved the best results (maximum detection error rate of 6.5%). Though these results were comparable to those gathered using the standard wavelet forms, the adaptive wavelet achieves the same level of performance with fewer scales of wavelet features, yielding computational savings.

Sari-Sarraf and Goddard [1999] performed a similar study to that of Yang et. al. [2002] by studying the effectiveness of a wavelet based defect detection system. In their study, Sari-Sarraf and Goddard based the wavelet portion of the detection system on Daubechies' D2 filter [Press et. al. 1992] because it closely matched the plain weave patterns of the fabrics under study. Their work proposed the creation of four two-dimensional kernels comprised of a high pass (HP) and a low pass (LP) filter: $LP(LP)^t$, $HP(LP)^t$, $LP(HP)^t$, and $HP(HP)^t$ (as seen in Figure 2.1). These filter combinations, when populated with the appropriate lattice coefficients, would result in a near zero response to a defect free fabric image and a non-zero response to an image containing a defect. This attenuation of the background texture would result in the accentuation of any defects since a defect does not represent part of the regularly repeating fabric pattern. By adding a defect segmentation algorithm, the accentuated defect would become computationally obvious, resulting in a

desirable detection rate of 89% on 26 different defects tested [Sari-Sarraf and Goddard 1999].

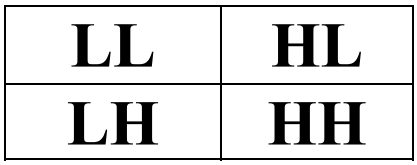


Figure 2.1: Filter Quadrant Design

In another similar study, Jasper et. al. [1996] again utilized discrete wavelet functions as the basis for the filter construct. However, they proposed the derivation of the optimal set of wavelet coefficients directly from the fabric image rather than using a pre-defined filter setup (i.e. Daubechies coefficients). Given a set of flawless fabric images, Jasper et. al. argues that because of the regular repeating nature of textile textures, designing an optimal wavelet filter for each image can be done by minimizing a quadratic cost function that is subject to orthogonality constraints. It is necessary to determine the optimal coefficients for each new texture because wavelet filters will respond to different textures in different manners, showing that there is not a general filter setup that will prove effective for all textures.

Because the wavelet filter has been optimized to the particular texture it will work with, the filter itself now represents the texture information of that fabric and can subsequently be used to characterize defects on that fabric [Jasper et. al. 1996]. Jasper et. al. go further to show that when the number of coefficients that make up the filter are equal to or close to the pixel count of the fabric repeat unit, there is significant reduction in false detection.

Wavelet filters show great promise for use in defect detection because wavelets represent frequency and spatial information together. While frequency information is adequate to identify the presence of a defect in an image, the spatial information is absolutely necessary to go further and identify the placement of the defect.

2.1.6 Conclusions

To satisfy the needs required by a robust defect detection system, the detection algorithm must be able to supply both frequency and spatial information. While frequency information is sufficient to identify the presence of a defect, spatial information is needed to effectively identify the location of that defect. It has been shown that Fourier analysis suffers from its lack of spatial information due to the nature of the Fourier transform [Jasper et. al. 1996; Kumar and Pang 2000; Sari-Sarraf and Goddard 1999]. On the other hand, Gabor filters, while containing both frequency and spatial information, suffer from a severe computational penalty due to the need for multiple filter elements in the overall construct to be able to account for various defect sizes and orientations. The MRF model suffers a similar computational penalty in that it requires a feature extraction algorithm to actually segment the defect from the background texture. However, the complexity of the equations that characterize a wavelet filter is such that an efficient search algorithm is needed to drive the search for the optimal wavelet coefficients.

2.2 Genetic Algorithms

Many problems that arise in industrial applications lack a reasonably fast solution algorithm. Typically, these problems require an optimization problem to be solved which represents different degrees of complexity and computational difficulty. While it is possible

to determine and construct efficient algorithms with which to solve these optimization problems, these solution algorithms do not guarantee the global optimal value as their result [Michalewicz 1996]. Because these problems occur in an environment that demands the best possible solution, a solution algorithm that is reasonably fast, efficient, and can guarantee the optimal solution value is required.

In recent years, much attention has been given to evolutionary computing, a field that utilizes principles of Darwinian evolution and heredity to solve difficult problems. The reason for the attention stems from the fact that these principles of evolution create a class of problem solving algorithms that can be applied to a wide range of problems. One such class of problem solvers is the Genetic Algorithm (GA). Michalewicz [1996] defines GA's as "…stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival".

Simply put, a GA is a model of nature; the algorithm behaves as nature would. A GA manages a population of individuals (possible solutions); it begins with an initial set of individuals, breeding and killing them. In natural terms, consider a set of wild animals. More often than not, the smartest and fastest of those animals, therefore the best of that population, are the ones to survive and procreate, thereby creating smarter and faster animals. Periodically, however, one of the dumb and slow animals will survive to procreate with the smart and fast animals. Additionally, nature sometimes adds a random mutation to the population, causing that individual to be smarter or dumber, faster or slower. Similarly, a GA performs these same functions on its virtual population, encouraging the production of a diverse population to find the best solution possible [Michalewicz 1996].

2.2.1 Genetic Algorithm Methodology

As discussed previously, GAs model the natural phenomena of evolution and apply those methodologies towards solving problems. Evolution, as it relates to problem solving, is the progression of the solution set towards convergence to the "optimal" solution. "Optimal" solutions in terms of GAs are the best solution found by the GA. As the solution set "evolves", areas of infeasible solutions are identified so that the algorithm does not waste time searching for a solution in the infeasible domain. By the same token, areas of promise are identified so that they can be exploited; meaning a more extensive search for the "optimal" solution takes place. This raises two key attributes possessed by GA's: global exploration and local exploitation.

There are two trains of thought when dealing with a searching algorithm for complex problems, those being global exploration and local exploitation. Global exploration is the tendency of the algorithm to examine a very diverse range of points in the solution space, never dwelling very long in one particular area. Local exploitation refers to the tendency of the algorithm to search a small area of the solution space, focusing more on that particular area than on the rest of the space [Michalewicz 1996]. Therefore, an algorithm that can balance global exploration with local exploitation will serve as a very efficient search algorithm since it will be able to search through a large area of the solution space, pausing to search further only into those areas that show promise.

Genetic algorithms represent a balance of the two above algorithms. Furthermore, a GA can be tuned to perform more exploration or more exploitation. For example, altering the genetic operators, which are functions that alter the composition of the child solution points, will alter how the GA performs [Michalewicz 1996, Houck et. al. 1996]. The

operators employed by GA's are crossover and mutation, as well as selection. These operators represent two of the fundamental issues in applying a GA (operators and selection), of which the others are representation, termination, initialization, and evaluation.

2.2.2 GA Representation

The representation of a GA refers to how the solution vectors are represented inside the GA. For the classical GA model, the solutions are represented in binary. For example, in a binary GA, the solutions would be represented by [0010110100, 1101001010, …], where the first binary string is variable x_1, the second is x_2, and so on and so forth. This form of the GA can handle floating point numbers, but must convert them into binary to operate on them. Because of this transformation, some floating point problems cannot be effectively solved using a classical GA representation. To remedy this, another GA representation can be used, such as the float GA. In this representation, the solution vectors are represented with floating point numbers, thus eliminating the need for any number system transformations and thereby increasing the effectiveness of the GA.

2.2.3 Genetic Operators

As stated previously, there are two operators employed by a GA are crossover and mutation. Crossover is a genetic operator that combines two parent vectors to produce child vectors. In the classical GA representation, simple crossover is the only form that is employed. Simple crossover crosses the parent vectors after the k^{th} position, where k is a random number in the domain $\{1..n\}$, where n is the size of the vectors in question. The GA applies this operator based on a probability of crossover, which is set by the user. This

crossover operator has been shown to improve the stability of the search and is capable of reducing the standard deviation of the best solutions found [Michalewicz 1996].

A floating point GA representation employs three different crossover algorithms: simple, arithmetical, and heuristic. Simple crossover is applied in the same manner as for the classical representation. The only difference between the two applications is that one operates on a binary bit string while the other operates on floating point variables. Arithmetical crossover is a linear combination of two parent vectors, and results in the creation of two child vectors. Given parent vectors x_1 and x_2, the resulting offspring are $x'_1 = a \cdot x_1 + (1 - a) \cdot x_2$ and $x'_2 = a \cdot x_2 + (1 - a) \cdot x_1$, where $a \in [0..1]$. Arithmetical crossover, like simple crossover, has been shown to improve search stability while reducing the standard deviation of the best solutions found [Michalewicz 1996]. The last crossover type is heuristic crossover, which is a unique crossover method because it is the only operator that uses the values of the objective function to direct the search. It produces a single offspring, and it may not produce an offspring at all. This method, given parent vectors x_1 and x_2, produces a child vector x_3 such that $x_3 = r \cdot (x_2 - x_1) + x_2$ where r is a random number between 0 and 1,and parent vector x_2 is better than or equal to parent x_1 according to the objective function. Because of the affine combination, it is possible that this method can produce an infeasible point outside the bounds on the variables, in which case a new r is generated along with a new child. This continues for a limit of w, after which no offspring will be produced by this method. Heuristic crossover aids the precision of the search by directing the search in the most promising direction and applying fine local tuning to areas of promise [Michalewicz 1996].

In general, the mutation operator requires a single parent vector and produces a single child vector. The classical GA representation applies only binary mutation. This mutation algorithm moves through the bit strings that represent the solution variables and flips bits according to a probability of mutation, which is defined by the user. Typically, this probability ranges between 0.001 and 0.005.

The four types of mutation operators for a floating point GA representation are uniform, boundary, non-uniform, and multi-non-uniform. Uniform mutation produces a child vector by making a change to a single component (variable) of the parent vector. This method replaces the i^{th} component of the parent with a random number between the range of the upper and lower bounds for that variable (i.e., $x_i = \text{unif}(a_i, b_i)$, where a_i and b_i are the lower and upper bounds, respectively). This mutation method allows the solution vectors to move freely inside of the search space, which is important in the early stages of solution evolution [Michalewicz 1996, Houck et. al. 1996].

Boundary mutation sets the i^{th} component of the parent vector equal to the value of either the upper or lower bound for that variable with equal probability. This mutation operator is designed more for problems in which the optimal solution lies along the boundaries of the solution space; however, this operator performs extremely well in the presence of constraints [Michalewicz 1996, Houck et. al. 1996].

Non-uniform mutation replaces the i^{th} component of the parent vector with a non-uniform random number. This number is generated by Equation 2.1 and is based upon the current generation number and the maximum number of generations for the run, as well as a shape parameter that is set by the user. This mutation operator aids the search by adding a fine tuning capability to the system [Michalewicz 1996]. Multi-non-uniform mutation is the

same as non-uniform, but instead of changing a single component of the parent vector, this operator changes all components [Houck et. al. 1996].

$$x_i' = \begin{cases} x_i + (b_i - x_i) f(G) & r_1 < 0.5 \\ x_i - (x_i + a_i) f(G) & r_1 \geq 0.5 \\ x_i & otherwise \end{cases} \qquad (2.1)$$

$$f(G) = \left(r_2 \left(1 - \frac{G}{G_{\max}} \right) \right)^b$$

r_1, r_2 = a uniform random number between (0,1)

G = the current generation

G_{max} = the maximum number of generations

b = a shape parameter

2.2.4 GA Selection Routine

The selection routine determines how the GA selects which solution vectors to include in the intermediate population where the crossover and mutation operators are applied. There are many different types of selection methods and ranking methods [Houck et. al. 1996]. Classical GAs use roulette wheel selection scheme which is based on the probability of selecting a solution directly on the fitness of that solution. Therefore, if the best solution in a set is four times better than the next best solution, the GA will have more selective pressure placed on it than if the best solution was only twice as good as the next solution in the ranking. This selection method limits the GA to maximization since the evaluation function used must map the solutions to an ordered set on the positive real numbers.

A tournament selection scheme randomly chooses *k* solutions from the population and evaluates them based on their fitness. The best individual is selected from this tournament which is repeated *n* times, where *n* is the population size. The GA will experience more selective pressure the higher the *k* because the probability of entering the best solution from the set into the tournament increases as *n* increases.

Other selection routines are based on ranking methods. These methods rank the individuals in the population from 1 to *n* (where *n* is the population size) according to their fitness which is defined by the evaluation function. One commonly used ranking method is the normalized geometric. As with all other ranking methods, the individuals are ranked as described previously. Then, the probability that a solution vector will be selected for the intermediate population is based upon a normalized geometric function, which has a discrete point for each solution vector present in the population (Equation 2.2).

$$P[\text{Selecting the } i^{th} \text{ individual}] = q'(1-q)^{r-1} \tag{2.2}$$

q = the probability of selecting the best individual
r = the rank of the individual, where 1 is the best
P = the population size
$q' = q/(1-(1-q)^{P})$

The amount of selective pressure placed on the GA to select the best solutions can be adjusted by altering the probability of selecting the best individual. A higher setting results in more selective pressure. In this way, it can be seen that if the best solution is two times better than the next best solution, the probability of selecting the best solution will be

identical to the case where the best solution is only one and a half times better than the next solution in the ranking.

2.2.5 Termination

The fourth issue associated with applying a GA is the termination criteria. First, the maximum number of generations can be set by the user, which acts as a cutoff point for the GA. If the maximum is set too low, the GA could be forced to return a sub-optimal solution because it was not allowed to run long enough to converge to a better solution. Conversely, if the maximum is set too high, the GA may spend several generations attempting to find a better solution when the current best solution satisfies the user's requirements. Secondly, the tolerance setting affects when the GA will stop. This setting can stop the GA before reaching the maximum number of generations if the best solution found is within the user-defined tolerance, meaning that the "optimal" solution has been found.

2.2.6 Initialization

The fifth issue is the initialization scheme that is applied. One option is to seed the initial population of the GA with previously found solutions in an attempt to search for better solutions. The remaining population members are then generated randomly. Another option, which is more commonly used, is to initialize all individuals in the population randomly.

2.2.7 Evaluation Function

Finally, the last issue associated with applying a GA is the evaluation function. This function is utilized by the GA to determine the fitness of the solutions generated as they apply to the problem at hand. While the GA will generate solutions that fall between the bounds as defined by the user, the evaluation function may further constrain the problem. In

this way, it is possible for the GA to generate solutions that do fall within the user defined bounds but are infeasible due to the constraints added by the evaluation function.

To counteract this occurrence, several things can be done. First, these infeasible solutions could be simply ignored in hopes that they will be forced out of the population in subsequent generations. Second, infeasible solutions could be immediately removed from the population. Both of these methods do not prevent or discourage the GA from generating infeasible solutions in future generations, and therefore represent an ineffective way of combating infeasible solutions. A penalty method, on the other hand, is designed to penalize infeasible solutions in an effort to force the GA to view these solutions as unfit, and therefore remove them from the population. This would also serve to discourage the GA from searching infeasible regions of the solution space.

The purpose of penalty methods is to alter the fitness value of a particular solution depending on its feasibility. Infeasible solutions are penalized each generation according to the user defined penalty settings. Doing this allows the GA to force out these infeasible solutions much faster than under normal conditions. These penalty methods can also be dynamic, meaning that they are able to adjust the value of the penalty term during a GA run. Bean's penalty method, for example, tracks the number of consecutive generations that have included infeasible solutions. Should that count exceed some user-defined threshold, the penalty term is increased by a user-defined amount. In the same manner, if the solutions become feasible, the penalty is reduced by a user-defined amount. This ability to dynamically alter the penalty term based on the feasibility of the solutions forces the GA to spend more time searching the feasible areas of the solution space [Houck et. al. 1996].

2.2.8 Hybridized Genetic Algorithms

There are some problems that exist for which the basic GA is not capable of finding very good solutions. These problems represent very complex solution spaces that impede the GA in its search for good solutions. Though GAs have been shown to be very good global explorers [Michalewicz 1996], they suffer from poor local exploitation. Even though the GA does utilize past information to direct its search [Jasper et. al. 2002], the solutions it generates are done so randomly, which limits its local exploitation abilities.

Local Improvement Procedures (LIPs) are searching techniques that can quickly converge to the local optimum in a small region of the search space [Jasper et. al. 2002]. Therefore, these techniques are very strong local exploitation searches when the starting point supplied to the algorithm is in the same region of the search space as a local optimum. However, these searches are poor global searchers since they are designed to quickly converge to a local optimum rather than explore the search space as a whole.

In terms of GA's, hybridizing involves redefining another search method into GA terms and combining it with the GA search algorithm. Though the GA represents a very powerful global search routine, it is not always capable of truly converging to the precise local optimum or basin of attraction even though it may extensively search the surrounding area. Simply put, the GA is very good at identifying the approximate location of these solutions, but somewhat lacking when it comes to converging to them. By adding a search method that is capable of converging to these solutions, the GA can now converge to the "optimal" solution. For example, a gradient descent search is a capable search algorithm for converging to a solution when it is supplied with a good starting point (the approximate location of the solution). By itself, this search method struggles when it does not start near

the best solution. Because the GA is capable of approximating the location of the best solution in a very complex solution space, adding a gradient descent search to the GA to form a hybrid results in an algorithm that can find the approximate location of the optimal solution and then zero in on the solution itself [Jasper et. al. 2002].

Coupling a LIP and its local exploitation abilities with a GA and its global searching abilities, the hybrid algorithm that results proves to be a very capable searcher when faced with a complex search space. Now that the GA hybrid can better exploit the areas surrounding the solutions it generates, the algorithm can effectively "learn", meaning that it is better able to judge the promise of the areas in the search space that it visits, thereby allowing it to focus more attention on areas that could hold the "optimal" solution. These hybrid algorithms have been successfully applied to complicated problems and have been shown to outperform other search methods [Houck et. al. 1996, 1997; Joines and Kay 2002; Chu and Beasley 1995; Renders and Flasse 1996].

2.2.9 Conclusions

Genetic algorithms serve as a powerful tool to solve difficult and complex optimization problems. They offer a wide range of internal settings that aid in tailoring the search to the problem at hand. By adjusting the settings for the selection routines, the selective pressure on the solution vectors can be adjusted to suit the needs to the search. Furthermore, by adjusting the termination generation counter and/or the tolerance, the speed of the search can be tailored to the requirements of the search (i.e. how close to the optimal solution is acceptable?). In addition, adjusting the settings for the mutation and crossover operators allows the user to customize the GA in terms of global exploration versus local exploitation, which is important based on the complexity of the solution space. Finally,

should the problem prove too complex for a simple GA to solve, hybridizing it with a local searcher combines the strengths of both algorithms and aids in overcoming the deficiencies in both routines [Jasper et. al. 2002]. Overall, GA's offer an effective and efficient search method for complicated problems.

3.0 Research Approach

Woven and knitted fabrics exhibit textures that are comprised of repeating patterns. This research aims to show that using a wavelet filter, the textural pattern can be filtered out, revealing any underlying defects in the structure or on the surface of the fabric. Since inspection procedures in textile production facilities generally require a trained human operator to detect fabric defects at slow production speeds, there is a significant need for an online automated process that can correctly identify fabric defects with a comparable level of competence to that of the human operator.

3.1 Objectives

The main objective of this study is to show that a computerized system based on image processing software is capable of identifying defects in woven or knitted fabrics. Specifically, the project objectives are as follows:

1. Determine wavelet filter coefficients for simple cases using a GA,
2. Determine wavelet filter coefficients for real fabric images using a GA,
3. Create a thresholding algorithm to convert filtered images to black and white, and
4. Identify different classes of defects that can be detected

3.2 Approach

There are two main parts to this study. The first part is the creation of a genetic algorithm methodology that is capable of determining the optimal set of wavelet coefficients that will satisfy the non-convex constraints for a specific texture. Simple cases for which the optimal solution was known were found as a preliminary screening method for the GA.

Subsequent GA attempts were always tested on these simple known cases before any other testing was performed. Further testing would only begin after the GA could successfully and repeatedly determine the known set of wavelet coefficients for these known cases. The second part of experimentation was to devise and code algorithms that were able to identify defects in real fabric samples. Multiple classes and types of defects were tested using these methods and comparison between the algorithms were made.

3.3 Software

All optimizations performed during this study were done using The Mathworks Matlab® version 6.1.0.450 release 12.1 and version 2.1 of the optimization toolbox for Matlab®. Version 5 of the Genetic Algorithm toolbox [Houck et al 1996] was used to create the GA.

3.4 Image Gathering

The fabric images were gathered using a custom image gathering system. This was constructed beginning with a Pulnix TM-1020 8-bit continuous capture grayscale camera. The camera is capable of capturing 15 frames per second at a resolution of 1018x1000 pixels. The camera was controlled by an EDT PDVa Camera Capture Card, which allowed for image capture under Microsoft Windows or Redhat Linux. To achieve the image quality necessary for defect detection, the images were backlit so that the yarns would show up dark, with spaces between them light. Backlighting was achieved using a CCS Green LED array light source. The green LED source was chosen because the camera was most sensitive to light in the green section of the spectrum. The light source was powered using a CCS PTU-

3012 pulsing power source. This power source is capable of strobing the light source using either an internal counter or external control.

3.5 External Light Control

The strobing aspect of the power supply was controlled externally. Because with internal strobing the light source is strobed continuously, it is difficult to sync the camera shutter with a strobe. Controlling the strobe externally allows for the resolution of timing issues, resulting in well lit images. The power supply was connected to the computer via a STCB-25-3 cable, available from CCS Inc. This cable connected to a high current digital I/O card in the computer. The high current model of the DIO card was necessary due to power requirements over the cable to the power supply. In accordance with the specifications sent by the manufacturer, one of the DIO output channels was connected to the STCB cable (see Figure 3.1).

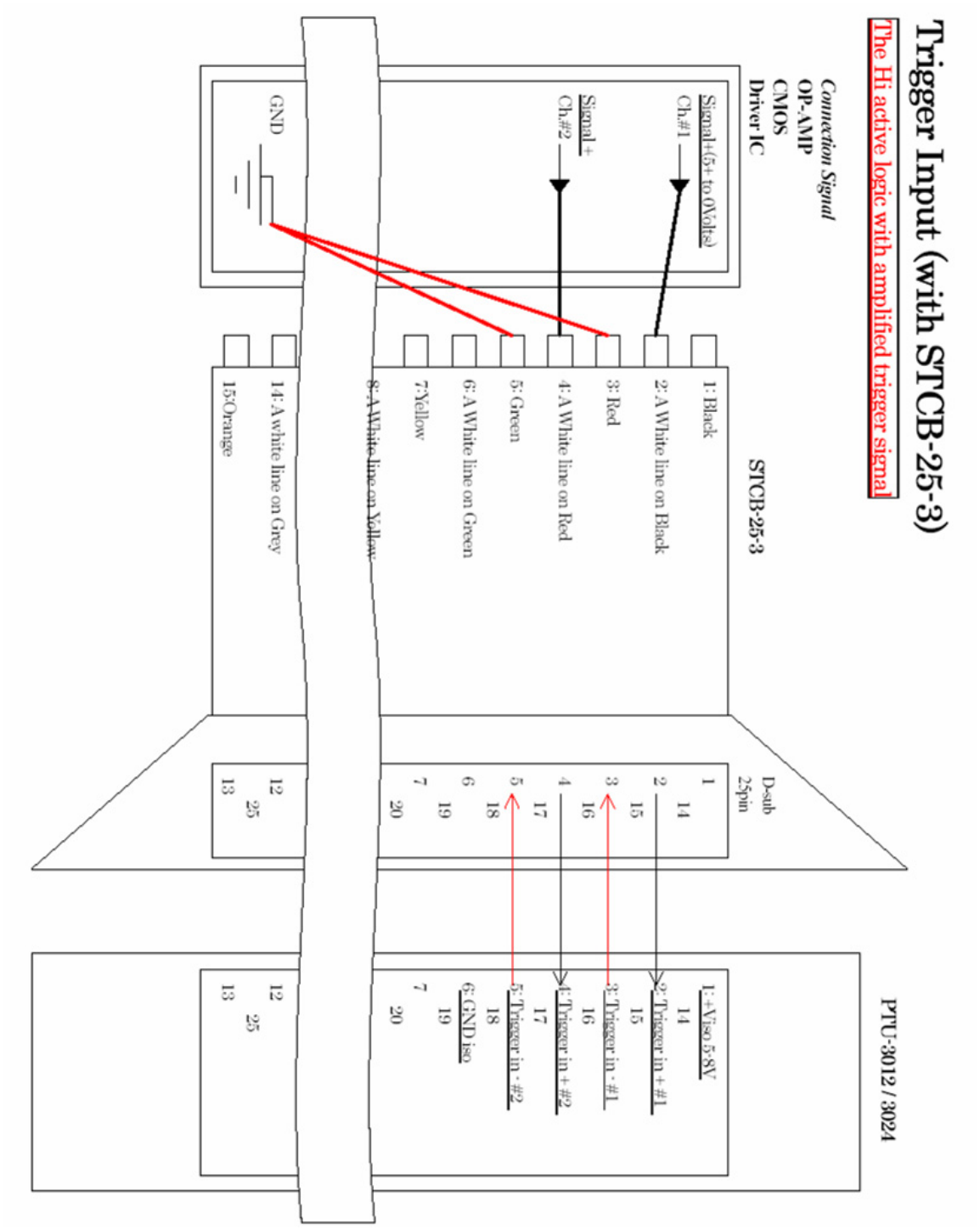


Figure 3.1: STCP-25-3 Cable Connection Diagram

4.0 Experimentation

It has been shown that wavelet filters might offer the flexibility and capability of performing fabric defect detection. As stated in Section 2.1.5, the following optimization problem (see Equation 4.1) can be solved in order to determine the best set of wavelet coefficients necessary for the wavelet filter [Jasper et. al. 1996]. However, this nonlinear, non-convex optimization problem is extremely difficult to solve.

$$\begin{aligned} \min \quad & J = c * P * P^T * c^T \\ \text{s.t.} \quad & \\ & C_j(c) = \sum_{k=0}^{n-1} c_k c_{k-2j} - \delta_{0j} = 0 \quad j = 1 \ldots n-2 \end{aligned} \tag{4.1}$$

The objective function (J) to be minimized was the 2-norm of the high frequency component of the image where n is the number of coefficients, c is a vector of wavelet coefficients contained in the solution set, and P is the array of the pixel values from the image. Genetic algorithms have been shown to perform well on a variety of difficult problems, such as the one shown in Equation 4.1.

4.1 Simple Float GA

Therefore, for the first attempt, a simple genetic algorithm (GA) using a float representation was used to determine the set of coefficients for the wavelet filter. This phase performed a directed random search on the solution space utilizing no additional algorithms (i.e., no local search) other than a very simple penalty to aid in phasing out infeasible solutions. Each solution set (i.e., set of wavelet coefficients) was subjected to two

evaluations; one determined the fitness of the solution set, while the other validated the coefficients that make up the set. The second evaluation function is needed because there exist constraints on the values of the coefficients (as seen in Equation 4.1), and the GA may produce infeasible solutions, so the number of coefficients determined the number of constraints. These constraints were necessary to satisfy orthogonality requirements, as the values of all of these constraint equations should equal zero for valid coefficients. In our experimentation, constraint values below 10^{-6} were considered zero, and therefore were considered satisfied. Since the GA cannot handle the constraints directly, the constrained optimization problem in Equation 4.1 is turned into an unconstrained problem by adding the constraints into the objective function as seen in Equation 4.2.

$$\min \quad J = c * P * P^T * c^T + \lambda^T C(c) \tag{4.2}$$

In essence, the GA minimized the sum of the squares of the product of the coefficients and the pixel values. Because of how the *P* array was set up (see Appendix 9.5), the GA minimized the *J* value for quadrant 4 in the filtered image. The λ term represented the simple penalty term which was applied to the constraint violation term, *C(c)*, which was the vector of the constraint equations, of which there were *n*/2, where *n* was the number of coefficients in the solution set. If all constraints were satisfied, then this vector would be zero.

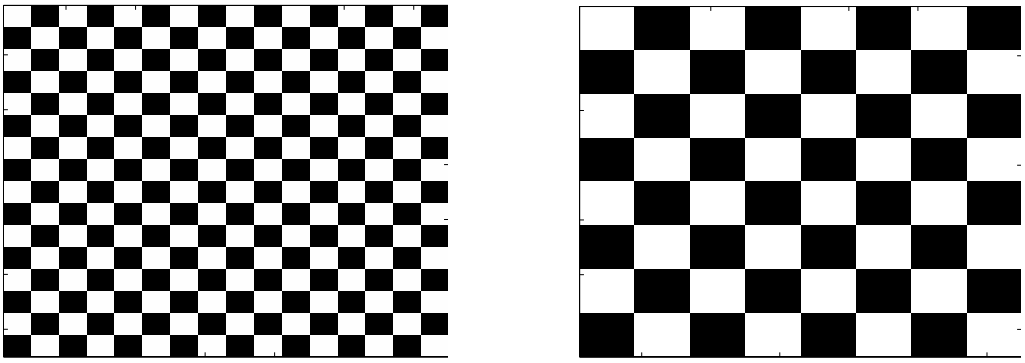
To assist in the solution search, the GA utilized both mutation and crossover operators. In the first GA phase, multi-non-uniform, non-uniform, and uniform mutation were used, as well as arithmetic, heuristic and simple crossover algorithms [Houck et al 1996]. Table 4.1 details the number of times each operator was applied, or the value used for

that setting. Note that for Multi-non-uniform (MNU) and Non-uniform (NU) mutation, there are two extra settings needed (see Section 2.2.3): the maximum number of generations and a shape parameter to determine the degree of non-uniformity to apply (3 for MNU and 6 for NU) [Houck et al 1996]. Also, heuristic crossover needs one additional setting, that being the number of retries it will undergo if it produces an infeasible child solution vector; the value used for this study was 3.

Once the GA and the necessary objective function files were developed, trial runs were begun to test the effectiveness of the simple GA. Initial experimentation began by minimizing some simple known cases (for example, a 2x2 and 4x4 black and white checkerboard image). These images were constructed by alternating 2x2 or 4x4 black and white pixel blocks as shown in Figure 4.1 (a) and (b), respectively.

Table 4.1: Genetic Algorithm Settings

Simple Crossover	2
Heuristic Crossover	2
Arithmetical Crossover	2
Boundary Mutation	4
Multi-non-uniform Mutation	6
Non-uniform Mutation	6
Uniform Mutation	4
Population Size	100
Normalized Geometric	0.08



(a) 2x2 (b) 4x4

Figure 4.1: Synthetic Images used in Experimentation

Due to the simplicity of these designs, the global optimum solution (i.e., the *J* value and coefficients {c}) was known, and therefore these images were used to validate the ability of the GA to obtain the optimal solution. The 2x2 case required a set of four wavelet coefficients and the 4x4 case required a set of eight coefficients, one coefficient per pixel in the repeat unit of the image. These images will be shown in Section 4.3 owing to the fact that every GA methodology was able to find the optimal solution. During these preliminary runs, it was noted that while this simple approach GA could find the optimal solution for the 2x2 case, it struggled with the somewhat more complex 4x4 case. Even after subjecting the GA to longer runs of 16,000 generations, it was still unable to adequately minimize the *J* value every time. Furthermore, when this simple GA was tested on a real fabric image, the GA was unable to adequately minimize the *J* value for this complex image. Even after running for a great length of time (40,000 generations), an adequate defect filter could not be obtained. This lead to the second GA iteration which incorporated a better penalty term to force out infeasible solutions from the search.

4.2 Float GA with dynamic penalty

One problem with the simple GA approach was the static penalty term λ, whose value was difficult to determine in order to perform effectively. If λ was too small, then the GA never converged to the feasible domain while too large a penalty term forced the GA to prematurely converge to a suboptimal solution. Therefore, the second attempt with the GA included a dynamic penalty term (i.e., λ would change during the course of the run) to the objective function. This was done such that the infeasible solutions could be eventually forced out while still allowing the GA to search. This new penalty term, based on the Bean's penalty method, increases in value over successive generations if infeasible solutions still exist in the set, and decreases in value if more feasible solutions are found. The logic behind this scheme was to allow the GA to skirt along the boundary of the feasible domain. The settings for this new penalty method allowed the GA to search infeasible regions for 15 generations before increasing the penalty term, which was added onto the objective function value. Since the GA was trying to minimize the objective value, a larger penalty term would force the GA to find solutions that were much more feasible. The formula for this penalty method is shown in Equation 4.3.

$$\lambda^T = \begin{cases} \beta_1 * \lambda^T & \text{if best value has been infeasible for } k \text{ generations} \\ \lambda^T / \beta_2 & \text{if best value has been feasible for } k \text{ generations} \\ \lambda^T & \text{otherwise} \end{cases} \tag{4.3}$$

As with the previous GA attempt, experimentation began with the simple cases. Once again, the optimal solution was found for the 2x2 case, but not for the 4x4 case, despite long runs (> 40,000 generations). At this time it was determined that the current GA

approach was inadequate in finding valid solutions for the problem and a newer more sophisticated method was needed. Therefore, the current GA was hybridized with a gradient search to form a better global searcher.

4.3 Float GA with Gradient Search

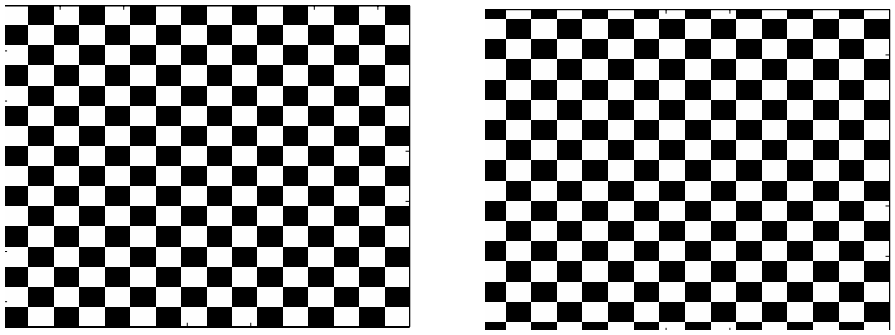
Because of the inability of the prior versions of the GA to handle the more complex images, a gradient search algorithm was added to the methodology by hybridizing the GA. Though the objective function itself would still be minimizing the sum of squares, the coefficient search would no longer be driven solely by the output of the GA. Once the GA generated a set of coefficients {c} for the image, a gradient descent search was performed to look for better solutions in the space immediately surrounding the current solution set. Consequently, this new method would gain the strengths of both the GA's global search ability and the gradient descent's local exploitation ability to quickly zero in on the local optimal solution. Also, the gradient search method (Multidimensional constrained nonlinear minimization) forced the point back to the feasible domain if it was an infeasible point.

In addition, the objective function was re-written to improve computational efficiency. This new formulation applied the original wavelet coefficient matrices to the image data directly, and then calculated the sum of the squares of the pixel values. This was done due to the difficulty associated with determining the gradients of the previous objective function as defined in Jasper et. al. [1996] and shown in Equation 4.1. The resulting formula for the new objective function is shown in Equation 4.4.

$$F = W * U * W^T$$
$$Sum = \sum_{i=1}^{n} \sum_{j=1}^{m} F(i,j)^2 \tag{4.4}$$

F represents the filtered image, *W* is the wavelet filter (see Appendix 9.6), *U* is the unfiltered image, and *n* and *m* represent the size of the image.

As with previous GA attempts, experimentation began by testing the simple synthetic cases as shown in Figures 4.2 and 4.3. In addition to the normal 2x2 synthetic image, two additional images were created to study the effect of shifting on the transformed image (i.e. – a one pixel shift down and a one pixel shift right). This was done to determine if the resulting filter would be shift invariant since for a real fabric image, capturing an image at exactly the same position would be nearly impossible.



(a) 2x2 Synthetic Image (b) 2x2 Shifted Down 1 Pixel

Figure 4.2: 2x2 Synthetic Images Used to test Hybrid GA

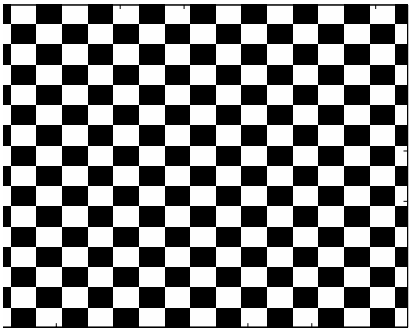


Figure 4.3: 2x2 Synthetic Shifted Right 1 Pixel

The wavelet filter produced by the coefficients {c} determined by the GA was then applied to these images. The results of this are shown below in Figure 4.4. The first image shows the result of the filtered normal 2x2 synthetic image. Because the GA was able to accurately capture the texture information in the form of the filter coefficients, the texture has been filtered out of quadrants two, three, and four, resulting in black quadrants. Any defects that had been present in those quadrants would show up as obvious flaws to the solid black background. The second image shows the filtered synthetic image that has been shifted down one pixel. Note again that quadrants two and four are black. Because the original image was shifted down one pixel, due to the properties of the filter and the simplistic repeat of the image, the compressed image shifted to quadrant 3. However, because quadrant one was still numerically equal across its space, it was concluded that any defects would still show up in the washed out quadrants (one, two, and four for this image). When the original image was shifted to the right by one pixel, a similar effect to down shifting was seen. The compressed image shifted to quadrant two, quadrant one turned white, and the other two quadrants remained black.

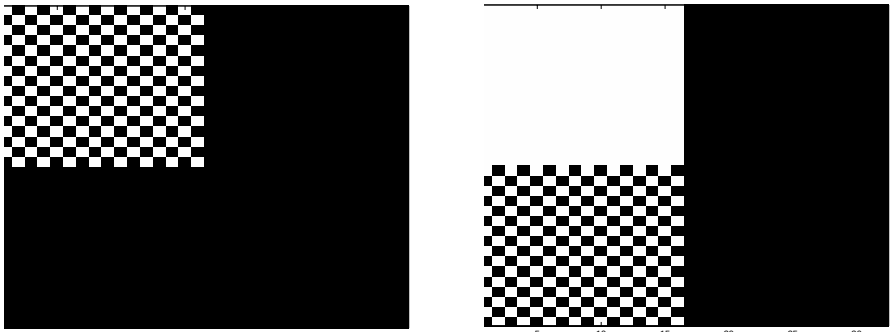


Figure 4.4: Transformed normal and down shifted synthetic images

To validate the method used here to another wavelet methodology, these same images were also subjected to a filter created using the standard Daubechies coefficients for a 4 pixel repeat unit [Press et. al. 1992]. The transformed image based on this filter is shown in Figure 4.5. Note the major difference between the gray levels of this image compared to the images filtered using the GA method. Though the Daubechies filter coefficients were considered to be a good baseline for comparison, they were not suitable for use in fabric defect detection. This was due to the fact that the desired filtered images needed to have all or nearly all of the texture washed out in quadrants two, three, and four (or whichever quadrants that do not contain the compressed original image). The Daubechies filtered image clearly shows that significant amounts of texture remained in the all four quadrants making it ineffective as a defect detector as seen in Figure 4.8.

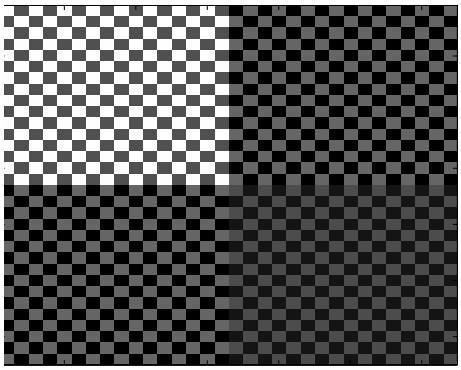
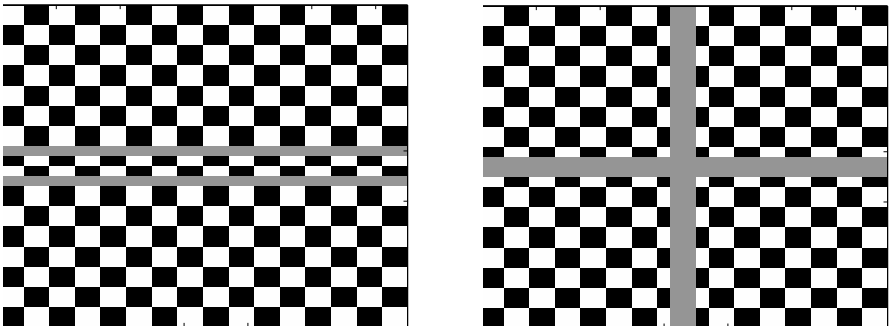


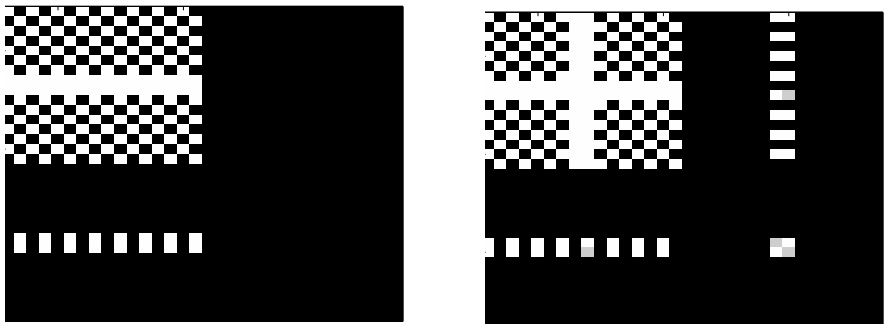
Figure 4.5: Daubechies Transform of Synthetic Image

The next step was to apply the determined filter to a synthetic image containing different defects to verify that those defects would indeed be visible. The first defect tested was a simple horizontal line defect, simulated by turning the pixels of image lines 15 and 18 to gray level 150 (the original image contained only levels 1 and 255) as seen in Figure 4.6 (a) and (b). Testing for shift invariance was also performed on this defect. The second defect tested was a cross defect, simulated by adding in a vertical and horizontal line of gray pixels (gray level 150) into the center of the image, seen in Figure 4.6. The GA-wavelet transforms of these images are shown in Figure 4.7, while the Daubechies transform of the cross defect is shown in Figure 4.8.



(a) Horizontal Line Defect (b) Cross Defect

Figure 4.6: Synthetic Images containing Defects



(a) Horizontal Line Defect (b) Cross Defect

Figure 4.7: Transforms of Defect Images

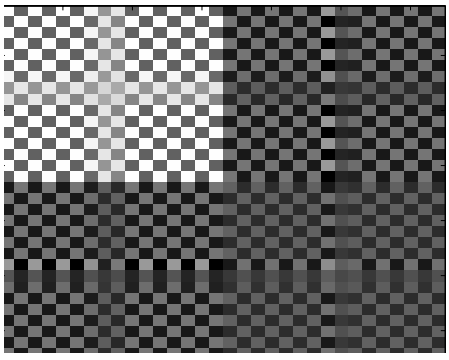


Figure 4.8: Cross Defect using Daubechies wavelet coefficients

The GA-wavelet filtered images shown in Figure 4.7 constitute a significant finding. The defect seen in the first image was of a horizontal nature, and therefore was seen in the third quadrant of the filtered image. The cross defect in the second image had both horizontal and vertical components which caused it to show up in both quadrants two (vertical) and three (horizontal). The intersection between both components was visible in quadrant four. These findings suggest that horizontal defects would appear best in quadrant three; vertical defects in quadrant 2; and two dimensional defects in quadrant four.

Next, to test the shift invariance, the horizontal line defect image was shifted down by a pixel. The transformed image for the down shifted horizontal line defect (as seen in Figure 4.9) initially appeared identical to the transform of the shifted down defect free image (as seen in Figure 4.4). When the image matrix for the defective image was viewed numerically, the defect was readily apparent as there was a significant difference between the values of neighboring pixels, signifying a non-uniform shade. By normalizing each quadrant to a 255 gray level scale, the defect became visually apparent as shown in Figure 4.9. The filtered image of the down shifted horizontal defect showed that the defect appeared in quadrant one. As discussed above for the simple 2x2 case, a one pixel shift down caused the compressed image to switch from quadrant one to quadrant three, which explains why the defect appeared in quadrant one here.

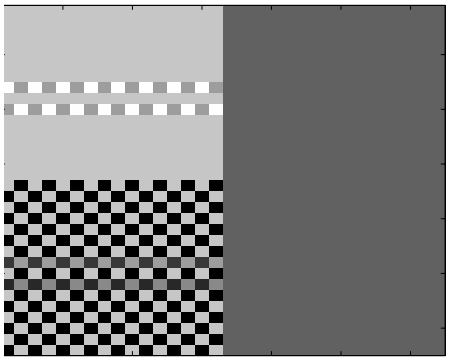


Figure 4.9: Transform Image of Horizontal Line Defect

Before experimentation continued, some time was spent analyzing the thought processes that resulted in the current GA setup. It was realized that an incorrect assumption had been made. The current setup at the time was using one set of wavelet coefficients to define the corresponding filter construct, which filtered in both the horizontal and vertical directions. However, the assumption that this filtering method would suit fabric images was incorrect; while it matched fine with simple test cases due to their symmetry, real fabric images are not truly symmetrical due to small variations in yarn structure, hairiness, and random noise.

The analysis determined that by utilizing a set of coefficients each for the horizontal and vertical directions, the GA could then optimize each coefficient set independently, which would result in a better wavelet filter construct. Therefore, the filtering process equation, shown in Equation 4.4, was modified as shown in Equation 4.5.

$$F = W_1 * U * W_2^T$$
$$Sum = \sum_{i=1}^{n} \sum_{j=1}^{m} F(i,j)^2 \tag{4.5}$$

In the new equation, W_1 is the wavelet filter constructed with the first set of wavelet coefficients, and W_2 is the filter constructed with the second set of wavelet coefficients. As shown, the sum of the squares is still applied in the same manner as before but now the GA solves for two sets of coefficients $\{c_1\}$and $\{c_2\}$.

Several trials were run utilizing the new combination of searches and the new objective function on real fabric images, and it was noted that each generation was requiring upwards of 5 minutes. Since each run of the GA required 400 or more generations, this translated into an arduous task. This long amount of time was partly due to the complexity of the filter necessary to adequately capture the texture information of the fabric, and also in part to the intricacy of the gradients which were being constructed from equations having 16 variables each (2 sets of 8 coefficients each for real fabric images). To help alleviate the load to the computer, the gradients were calculated ahead of time with respect to the coefficient variables. The gradients were then supplied directly to the gradient search in the GA for evaluation, eliminating the time spent determining them by perturbation in the gradient search algorithm.

Because of the time savings incurred from the predetermined gradients, longer runs of the GA were performed. These longer runs ran in the same amount of time as with the previous settings, and had the advantage of producing images more closely suited to achieving the goals of the project (i.e., real fabric defect detection).

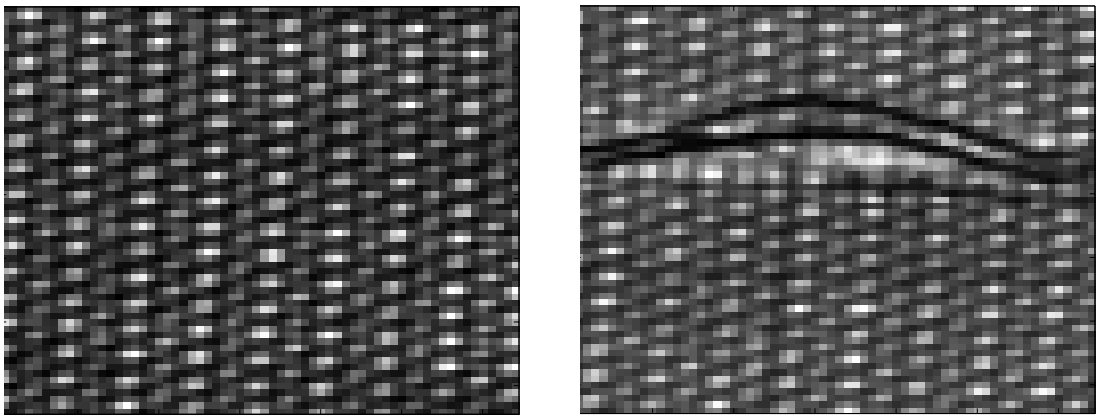


Figure 4.10: Plain weave no defect, with defect

The images shown in Figure 4.10 were taken from a sample of a plain weave that contained numerous tears throughout its length. Both images are 64x64 pixel subsets of the original capture. The difference in apparent gray level is only due to image scaling; numerically, the background textures are identical. A set of wavelet coefficients were determined using this GA methodology on the images in Figure 4.10 to create a wavelet filter. Two sets of coefficients were used as shown in Equation 4.5. Both sets of coefficients contained 8 values, one value for each pixel in the repeat unit of the fabric image (Figure 4.10). The corresponding filter was then applied to the images as seen in Figure 4.11. However, not enough of the texture was being washed out by the filter to accurately identify the presence of any defects.

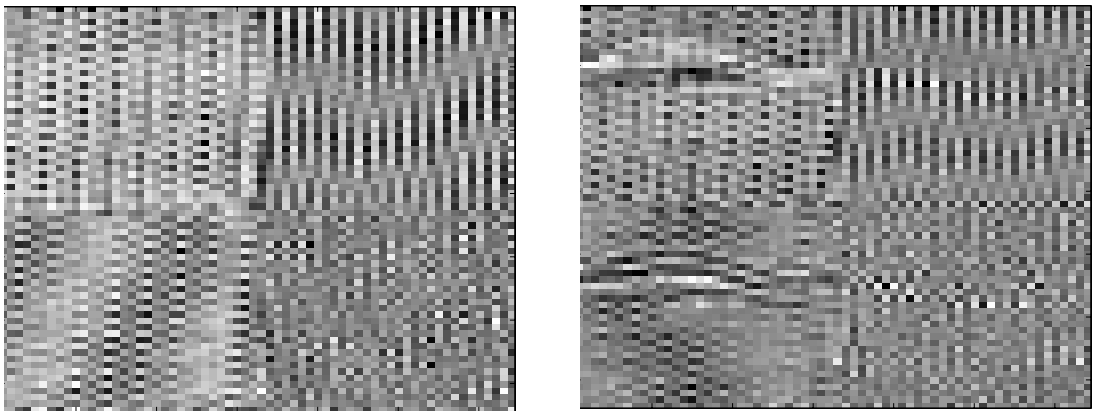


Figure 4.11: Transform of Plain weave images, without and with defect

4.4 Float GA with Entropy

While the results of the GA with gradient search were definitively better than those found previously, the filter construct was still not capable of filtering out enough of the texture to allow for reliable defect detection on real fabrics. Because of this, there was a need for a different method of image processing instead of using the sum of squares; one such objective was entropy. Entropy, in terms of image processing, is the measure of the number of bits per pixel needed to represent an image. The formula, also known as Shannon's entropy, is shown in Equation 4.6.

$$H = -\sum_{i=1}^{n} d(i) * \log_2(d(i)) \tag{4.6}$$

In this equation, n is the number of gray levels in the image; in the case of this study, n was 256 since all images used were 8-bit grayscale images. The term $d(i)$ is the normalized frequency of occurrence for each gray level, where each probability is between zero and one,

and the sum of all probabilities is one. For example, an image that contained 256 different gray levels would have an entropy value of 8 ($2^8 = 256$), whereas an image that contained only 128 different gray levels would have an entropy of 7 ($2^7 = 128$) [Leung et al 2001].

To increase the computational efficiency of the entropy formula for use in this study, the equation was re-written as shown in Equation 4.7. As in the method shown by Equation 4.6, this method uses a frequency of occurrence array; in this case, the array has not been normalized, so the counts contained in the array represent actual frequency of occurrence data rather probability data. However, after the frequency distribution has been constructed, all non-zero elements are removed since they have no bearing on the calculation, and the result is the *Bin* term shown in Equation 4.7. The reworked entropy formula proves to be more computationally efficient since it ignores the non-zero elements of the frequency distribution and does not require an extra step to convert the frequency distribution into a probability distribution.

$$Entropy = \frac{-\sum_{1}^{n} Bin * \log_2(Bin)}{Length(Bin)} + \log_2(Length(Bin)) \tag{4.7}$$

To aid in understanding entropy, the data in Table 4.2 will be used to perform a sample entropy calculation. The first step was to find the range of the data (i.e., the values range from 0 to 8 resulting in a range of 9) and then divide that range into 256 increments. Next, each pixel value is sorted into the appropriate bin, and the count of that bin is increased by one as seen in Equation 4.8, term BinWithZeros. Next, all non-zero elements of the bin

array are arranged into a row (Bin), the length is calculated, and the entropy calculation performed. The intermediate and final results are shown.

Table 4.2: Example Image Data

2	3	0	2
8	3	7	0
2	5	7	3
1	4	2	7

$$BinWithZeros = \begin{bmatrix} 2\ldots & 1\ldots & 4\ldots & 3\ldots & 1\ldots & 1\ldots & 3\ldots & 1 \end{bmatrix}^{T}$$
$$Bin = \begin{bmatrix} 2 & 1 & 4 & 3 & 1 & 1 & 3 & 1 \end{bmatrix}^{T} \qquad Length = 8$$
$$Entropy = 2.7806 \tag{4.8}$$

The fourth and final attempt during experimentation resulted in the use of entropy as the evaluation function and the problem in Equation 4.9 was minimized. Note that the same orthogonal constraints have to be satisfied. This method, when applied to an image, returned the number of bits necessary to represent the image. Unfiltered images are in 8-bit grayscale, meaning that they require 8 bits of image data per pixel to store the corresponding gray level. Since this method determined the number of different grayscale levels present in the image, it was decided that this would serve as an excellent method to evaluate images.

$$J = Entropy\left(W_1 * U * W_2{}^{T}\right) + \lambda^{T} C(c) \tag{4.9}$$

Even though it was deemed impossible to continue supplying a predetermined gradient to the GA with this new method, doing so proved unnecessary, since the

generational time was faster than before (i.e., the entropy calculation was quick). While the exact reasons for this are still unclear, speculation pins it to the simplicity of the function itself, which consists of few lines of code. Contrasted with the previous method used, which required very long equation evaluations to calculate the gradient, the entropy method is much simpler from a computational point of view, as it primarily focuses on grouping data before performing the simple entropy calculation and therefore determining the gradient numerically fast.

The filtered image of the plain weave with defect shown in Figure 4.10 using entropy can be seen in Figure 4.12. Because of the similarities between this new image and the previous filtered image shown in Figure 4.11, there was still not enough of the background texture being filtered out. The cause was related to the high zoom level used to capture the previous set of images. Therefore, a new set of images were captured for experimentation using a lower zoom level and are shown in Figure 4.13.

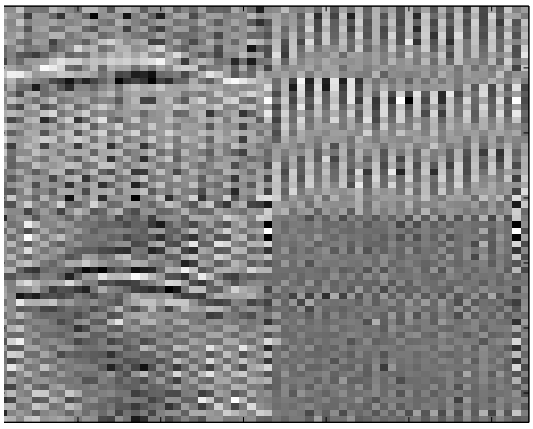
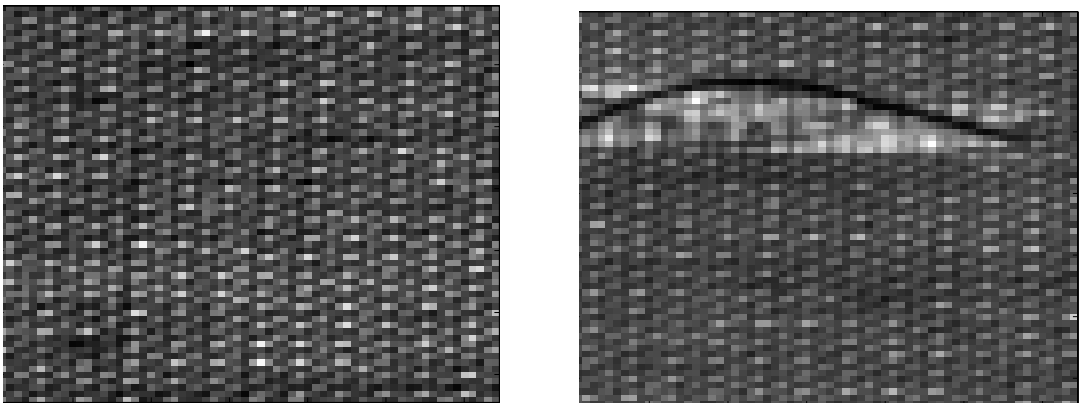


Figure 4.12: Entropy Transform of Plain weave with defect



(a) Without Defect (b) With Defect

Figure 4.13: New Plain weave images

The images in Figure 4.14 were the filtered results of the images in Figure 4.13, which represented another set of images on woven fabrics. After comparing the new images shown in Figure 4.14 to those previously collected and shown in Figure 4.11, it was readily apparent that because of the lower magnification used during image capture, the filter created by the GA was much more capable of filtering out the background texture of the images. This can be seen clearly in Figure 4.14 since the gray levels seen in the fourth quadrant are much more uniform than before. Remember, the entropy values of only the fourth quadrant are optimized. The other quadrants just use the same coefficients, and for symmetric images, that was enough.

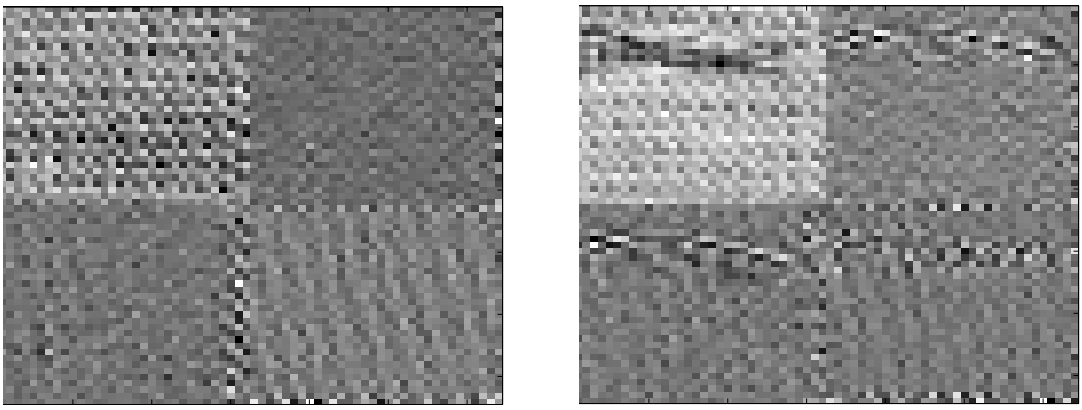


Figure 4.14: Transform of weave images, without and with defect

During this phase of experimentation, it was hypothesized that if a set of coefficients can adequately filter the fourth quadrant, additional sets of coefficients can be found to adequately filter the second and third quadrants as well. Therefore, the necessary changes were made to the objective functions (new sets were created to handle optimizations on the other quadrants) and the optimizations were run. Three separate optimization problems have to be solved (i.e., a filter for each of the three quadrants has to be found). As shown in Figure 4.14, there is a significant improvement in the uniformity of quadrants two and three under this new optimization scheme.

Now that the GA was capable of finding adequate filter coefficients for quadrants two, three, and four of all images, it was deemed time to attempt the filtering process using much larger images, ones that would encompass a much greater area of fabric and allow for better defect detection.

The image shown in Figure 4.15 is the full fabric image from which the test images shown in Figure 4.13 were taken. This full image encompassed an area of 512 by 512 pixels, which was substantially larger than anything attempted prior (largest image previously was

64 by 64). This also allowed for a greater amount of defect detection, as the image encompassed the full area of the backlighting source used. The resulting filtered image confirmed that the results given by the entropy method were of a high enough quality to attempt defect detection. The reason for this can be seen by the fairly uniform background and pronounced defect in each of the quadrants of Figure 4.16. The methods used for defect detection and their results will be discussed in a later section.

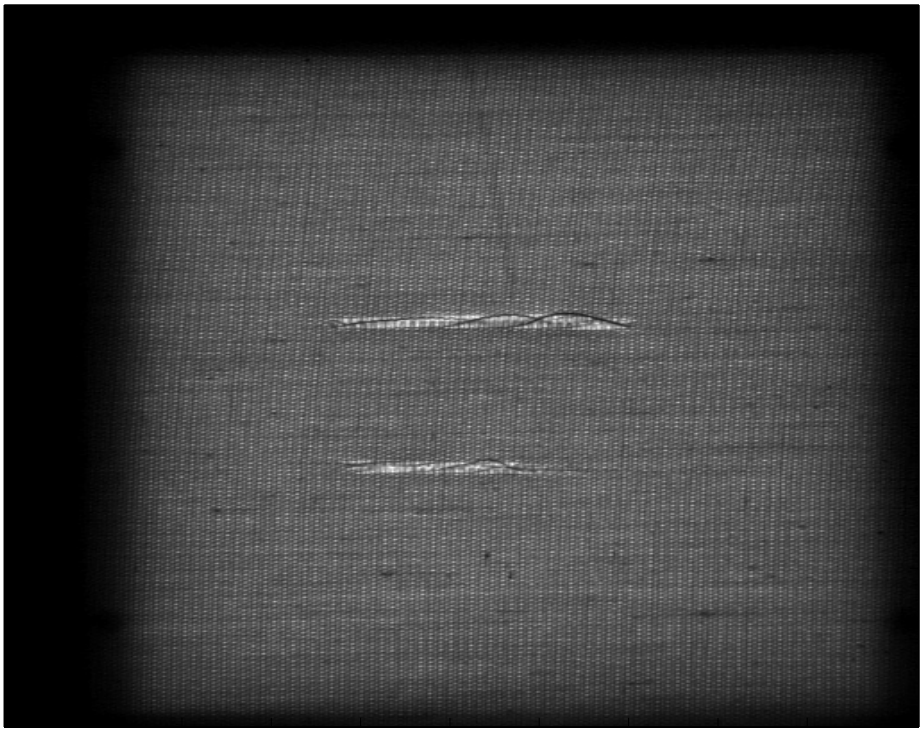


Figure 4.15: Full Plain weave image

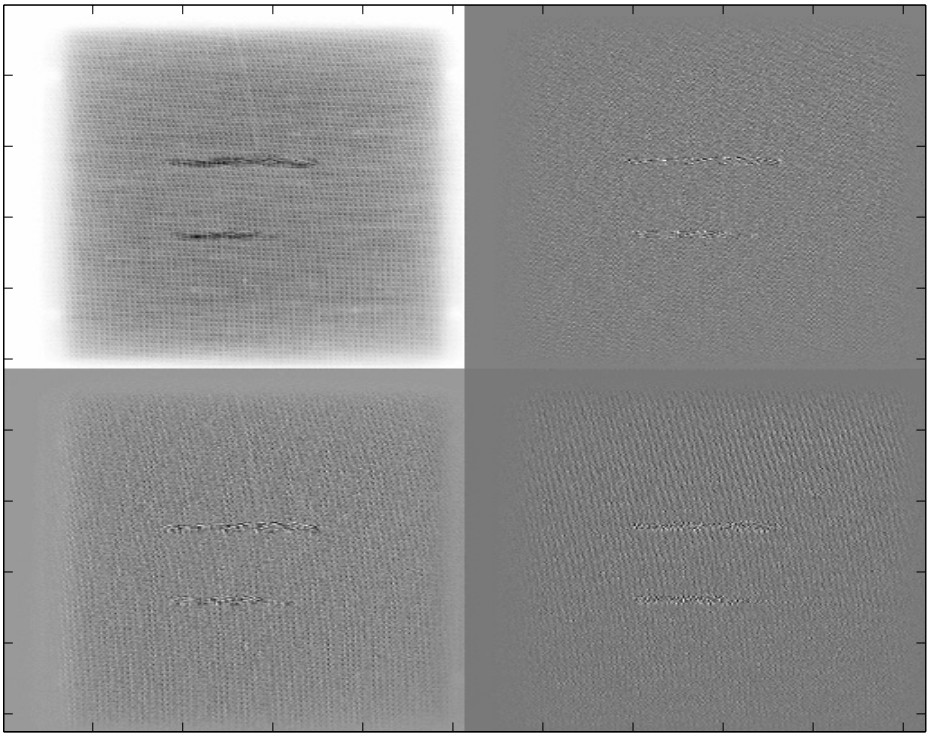


Figure 4.16: Transform of Full weave image

Experimentation with the entropy method has demonstrated promising results. Not only was the GA able to reduce the entropy of the images by at least one bit, but the resulting range of pixel values allows for the setting of a threshold value. This allows the image to be converted into a strict black and white image for defect detection, something that was not possible before with sum of squares. The reason for this new capability stemmed from the way that the data was represented with entropy. Looking at a histogram of the pixel value distributions on an unfiltered image (see Figure 4.17) it was readily apparent that no threshold value could be set due to the erratic and non-normal distribution. However, when viewing the same histogram populated with pixel values from the filtered image (see Figure 4.18) it was shown that the distribution was normalized, with the upper or lower tails of the

curve representing potential defect pixels. These tails represented potential defect pixels due to the nature of the wavelet filter. Because defects would represent a significant deviation from the normal texture of a fabric image, defects would be emphasized in the resulting filtered image. The emphasis placed on these pixels would result in large departures from the population mean, placing those values in either the upper or lower tail of the population distribution. By setting a threshold value near one of these tails (for example, a threshold value of 200 since a high percentage of the data falls below 200) and thresholding accordingly, the resulting image clearly shows where the defects lay. Further image processing algorithms could be performed on the resulting images to further enhance the defect detection as shown in later sections.

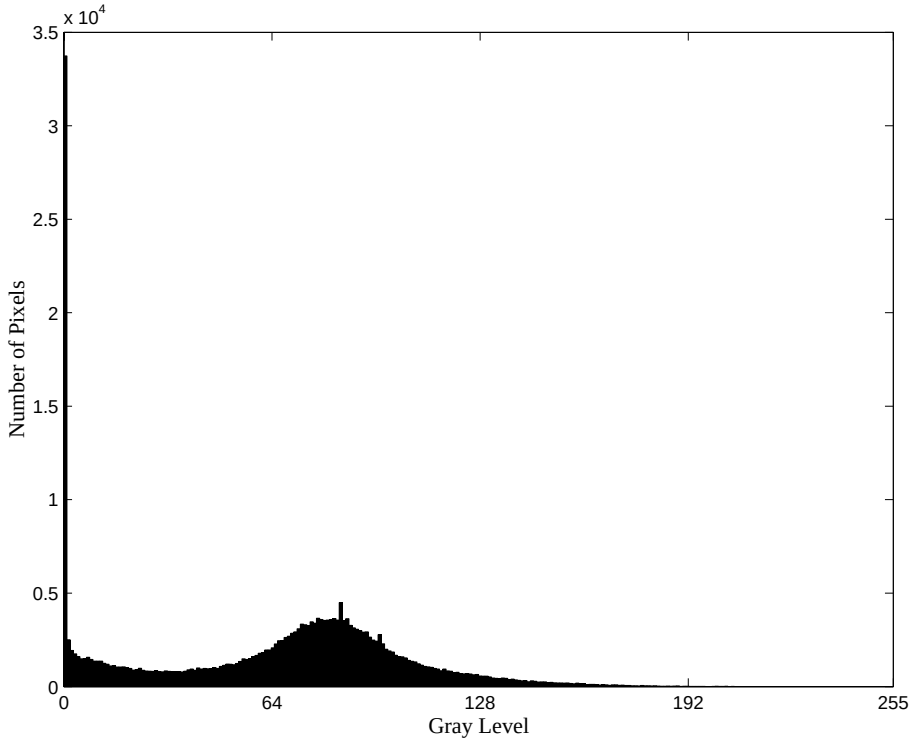


Figure 4.17: Histogram of pixel values in full weave image

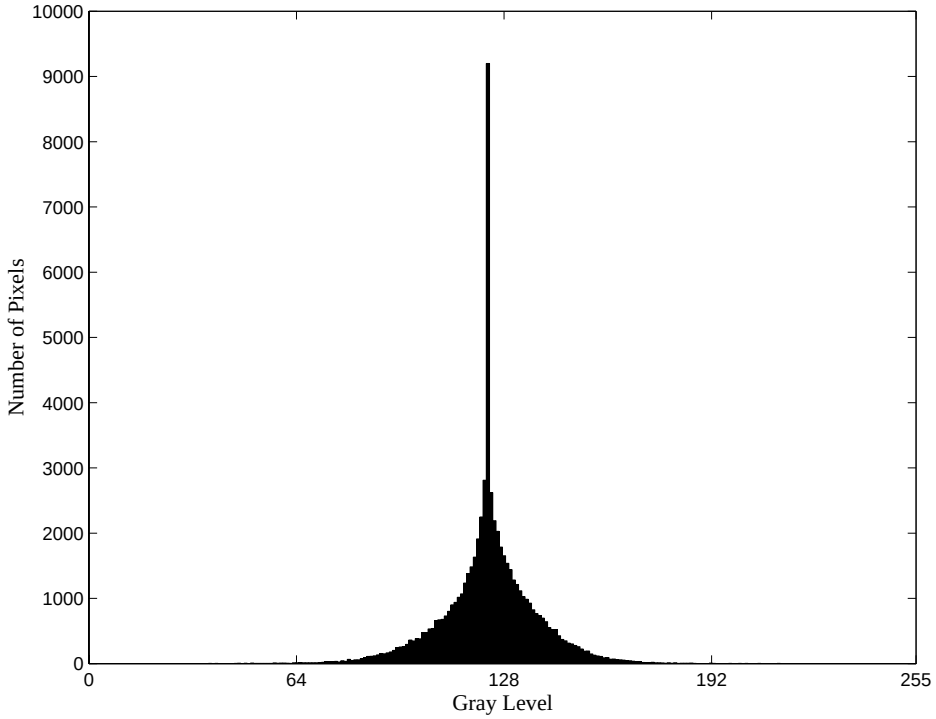


Figure 4.18: Histogram of pixel values in quadrant 4 of filtered image (weave)

4.5 Image Normalization

Though all of the defect detection methods that will be described later run solely on the numerical values of the images, it was important to be able to show what some of the images looked like visually. Due to the nature of the wavelet transform, it was possible for the resulting filtered image to contain values outside the range that is normal for an 8-bit gray scale image (0 to 255). In addition to that, the values contained in each of the four quadrants sometimes were very different relative to one another.

Because of these properties, simple image scaling was not adequate to provide a satisfying visual representation of the filtered data. Therefore, a normalization method was required that operated on each quadrant independent of the others. This algorithm worked by first finding the lowest value in the quadrant in question. After subtracting that value from

all pixels in the quadrant, the maximum value in the quadrant was found. Finally, each pixel was divided by a scale factor, which was the dividend of the maximum value and the maximum 8-bit gray value, 255. This algorithm performed the aforementioned operations on each quadrant separately. The resulting images can now be compared to other normalized images since all are shown on the same scale. Furthermore, comparisons between quadrants in the same image can be made.

An example of an un-normalized image is shown in Figure 4.19. Upon viewing that image, it was easy to see that some sort of normalization was needed because the data in quadrants two, three, and four was not visible due to the low values contained in the first quadrant relative to the other quadrants. By using the normalization algorithm, the image shown in Figure 4.20 was produced, making it much easier to view the visually represented data from the previously washed out quadrants.

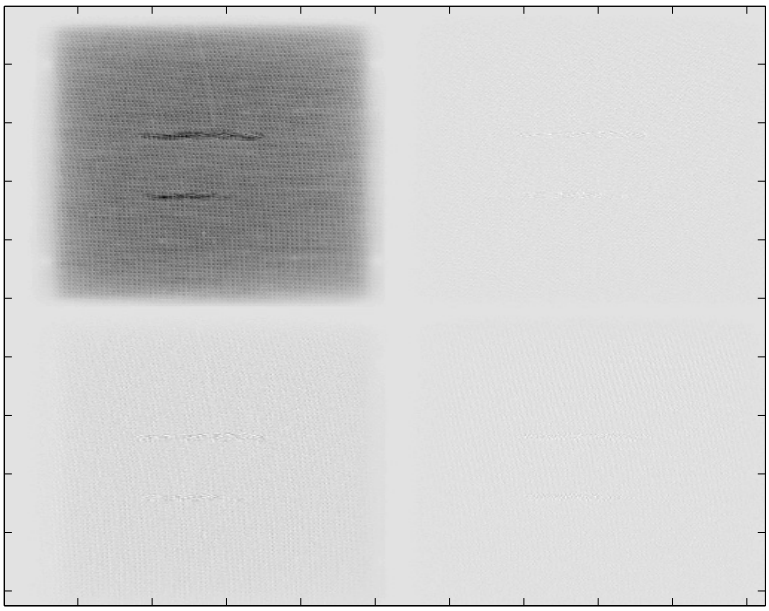


Figure 4.19: Filtered Weave image, no normalization

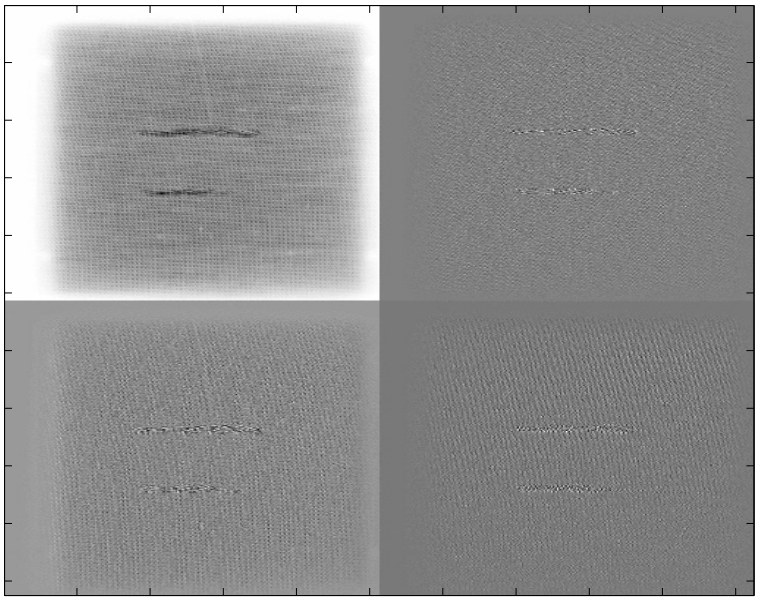
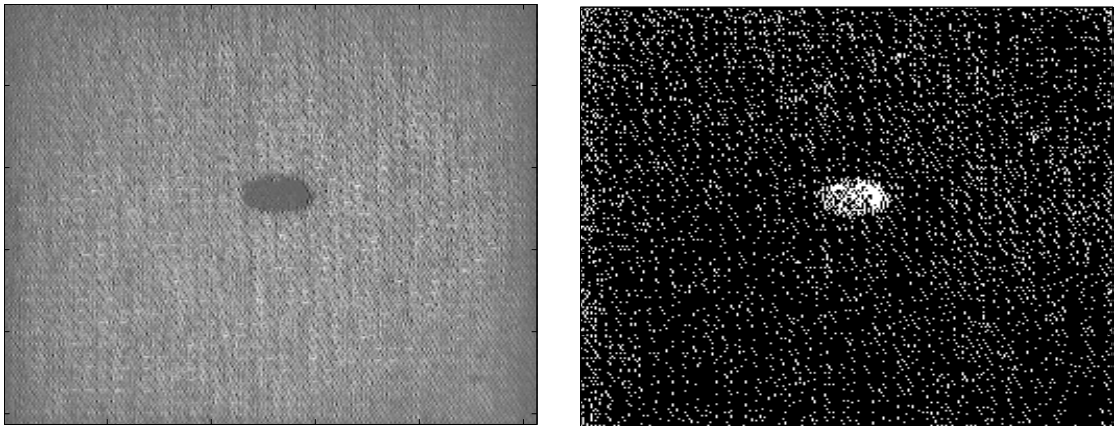


Figure 4.20: Filtered Weave image with normalization

4.6 Thresholding

The thresholding algorithm, while simple in nature, was a critical part of defect detection. While any defects present in the original images were now readily apparent to the naked eye in the filtered images, detecting them computationally was another matter. Because of the nature of the entropy function in terms of resulting gray level variances, it was possible to identify a threshold point from the gray level distribution (similar to Figure 4.18). This threshold point served as a cutoff point for the image processing, meaning that any pixel with a gray level below the threshold point (or above, for reverse thresholding) would be turned black, and any above would be turned white. Therefore, the resulting image would consist of only black and white pixels, with the idea being that pixels representing a defect would be turned white, and all others black.

Figure 4.21 illustrates how reverse thresholding works. The image before thresholding (Figure 4.21a) was the fourth quadrant of a filtered image. Because the defect shown in the image was darker than the rest of the image, reverse thresholding had to be used. In that particular instance, a threshold value of 105 was used, meaning that any pixel darker than gray level 105 was turned white and all others black. The resulting image is shown in Figure 4.21b. The oil stain stands out in the image.



(a) Before Thresholding (b) After Thresholding

Figure 4.21: Quadrant 4 Image before and after Thresholding

Though the threshold value for the filtered images was inherently large (above 200) for normal thresholding and relatively small (below 150) for reverse thresholding, random noise and other naturally occurring inconsistencies in the fabric allowed for some non-defect pixels to fall above or below the threshold point, resulting in additional white pixel blocks in the thresholded image (as seen in Figure 4.21b). Before any defect detection could take place, these non-defect pixel blocks needed to be identified and removed.

4.7 Noise Removal

Upon viewing numerous thresholded images, with and without defects, it was noticed that noise blocks tended to be small in comparison with true defect blocks. Therefore, the algorithm for removing noise needed to be able to determine the size of the block to decide whether or not to erase it. Figure 4.22 depicts the flow of the noise removal process.

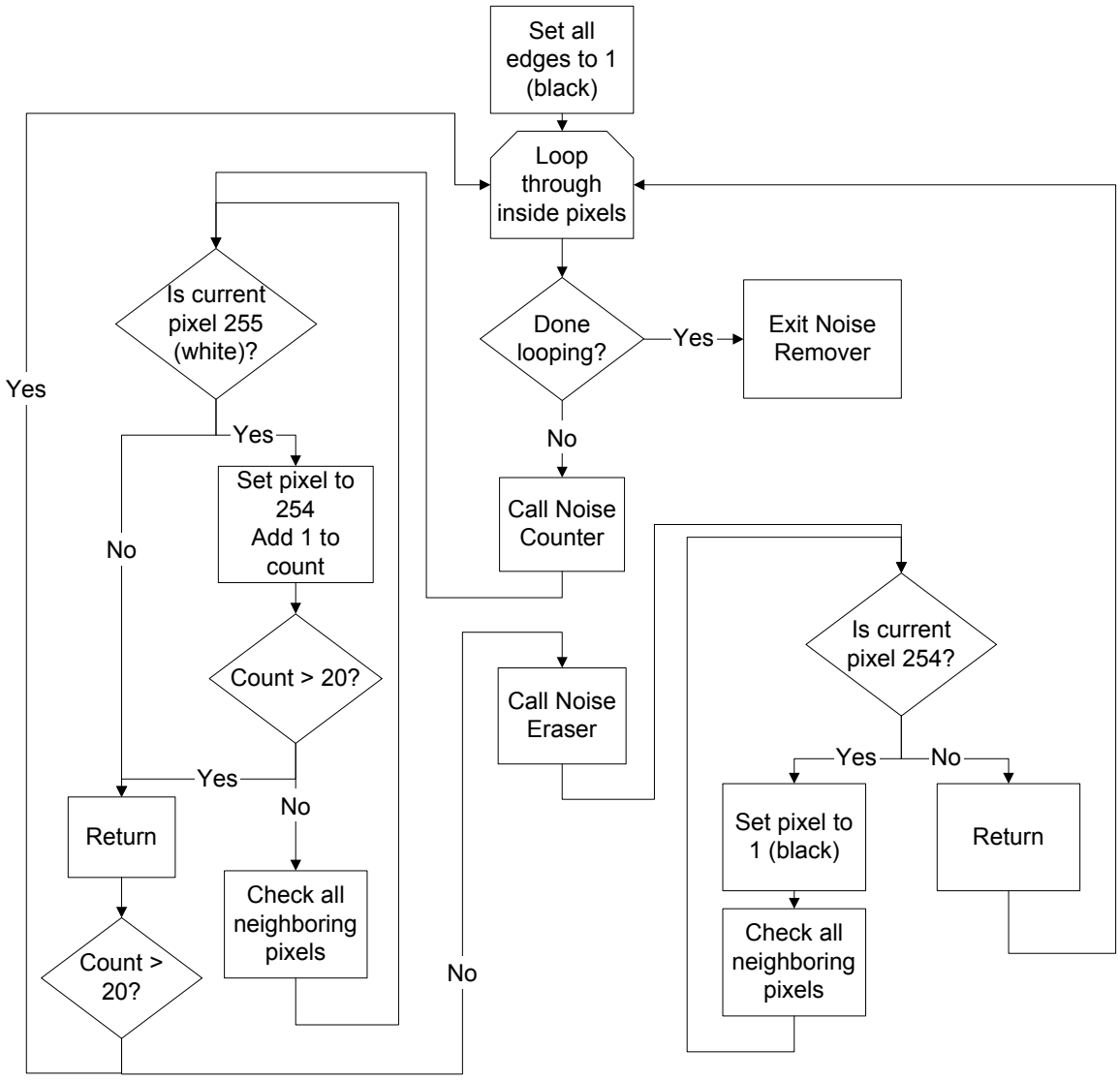


Figure 4.22: Flow Chart of the Noise Removal Process

The noise removal process was broken into three separate algorithms. The first algorithm constructed was a pixel block counter that was capable of searching the image for white pixels. Once a white pixel had been identified (a value of 255), a noise counting routine was used to determine the size of the pixel block. This routine worked by making recursive function calls, passing the location of all eight neighbors to the original pixel (Figure 4.23), one at a time. Once the routine had counted a pixel, it changed the pixel value to 254 such that the counter would not visit that pixel anymore. This was done because the counting routine only looked for white pixels (i.e., those having a gray level of 255). Once no more white pixels could be found in the current block, the routine would exit and return the number of pixels that made up the block in question.

1	2	3
4		5
6	7	8

Figure 4.23: Neighbors of a Pixel

Next, the main noise removing routine would compare the size of the block to the predetermined size criteria. That size criteria was made to be variable to allow for reusability in any number of cases. If the size of the pixel block was greater than the set size criterion, then the block would be ignored, and the routine would continue searching for white pixel blocks. Otherwise, the routine would call its other child algorithm, the noise eraser.

The noise erasing routine was also a recursive function. Starting with the initial pixel of value 254, the routine would branch out to all other 254-value pixels in the same block, turning them all black. Once the block had been removed from the image, the function

returned, and the noise search continued until all other noise blocks had been identified and removed.

Once searching was complete, the routine would go through the image and change all remaining 254-value pixels back to value 255 for any further processing that was required. The reason that this was done last was so that the noise counter would not visit the same pixel block twice during its successive row scans of the image since it was only searching for pixels of value 255. The final result from these algorithms was an image that contained only pixel blocks of size greater than the predetermined cutoff.

An example of an image that required noise removal was depicted in Figure 4.21. The defect showed up as the large white pixel block in the center of the image surrounded by a fair amount of small pixel blocks representing noise. Because of the small size of these pixel blocks, the noise removing algorithm could be run on this image with a fairly low noise-size setting; in this case, the setting was 15, meaning that any pixel blocks below 15 pixels in size would be removed from the image. The image that resulted from that noise removal procedure is shown in Figure 4.24.

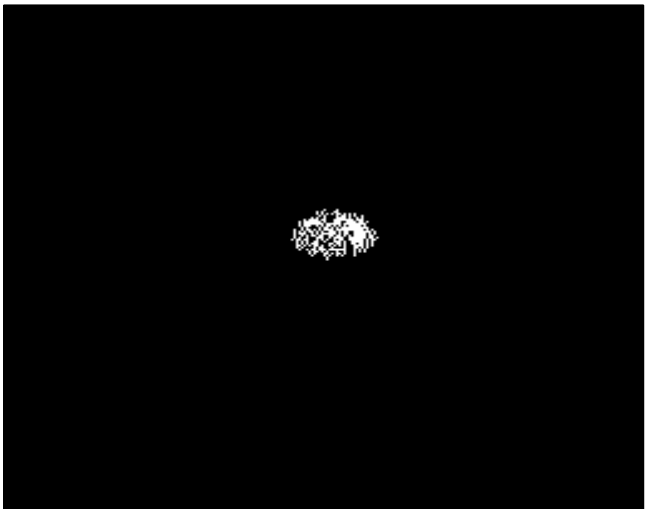


Figure 4.24: Thresholded Image after Noise Removal

4.8 Defect Segmentation

Now that the required preprocessing of the filtered images had been completed, defect segmentation could begin. Because of the properties of the wavelet filter, linear defects of a vertical or horizontal nature would show up in quadrants two and three, respectively. Examples of these types of defects would be mispicks, missing or broken ends, and other single yarn defects. Other defects of a more planar nature would show up best in the fourth quadrant. This defect class includes holes, tears, oil spots, and any other defect that is planar in nature.

Experimentation with defect detection began with quadrant four. Fabric images containing tears and oil spots of varying degrees of severity were subjected to the filtering process. Subsequently, those images were thresholded and subjected to the noise removal algorithm with a size criterion of 3 and 15 pixels, respectively. The resulting images contained only the defect, so no further processing was needed as the defect had been both

identified and located. An example of a tear defect is shown in Figure 4.25, and an example of a stain defect was shown in Figure 4.24.

Figure 4.25: Plain Weave defect, thresholded and cleaned

Defect detection in quadrants two and three consisted of analyzing images that contained mispicks. Initially, the filtered images were subjected to the thresholding routine, but it was noticed that the pixel blocks that represented the mispick were of insufficient size to be adequately distinguished from noise in the image. Increasing the threshold value did serve to reduce the amount of noise present and the size of the remaining noise blocks, but doing so also had an adverse effect on the size of the defect blocks. An example of this phenomenon is shown in Figure 4.26. The image shown was quadrant three of a filtered image containing a mispick. As evident in the thresholded image of the mispick, it was impossible to set the size criterion for the noise removing algorithm so that only noise pixels would be removed since there was no significant size differential between defect pixel blocks

and noise pixel blocks. Therefore, another method for detection in quadrants two and three had to be researched.

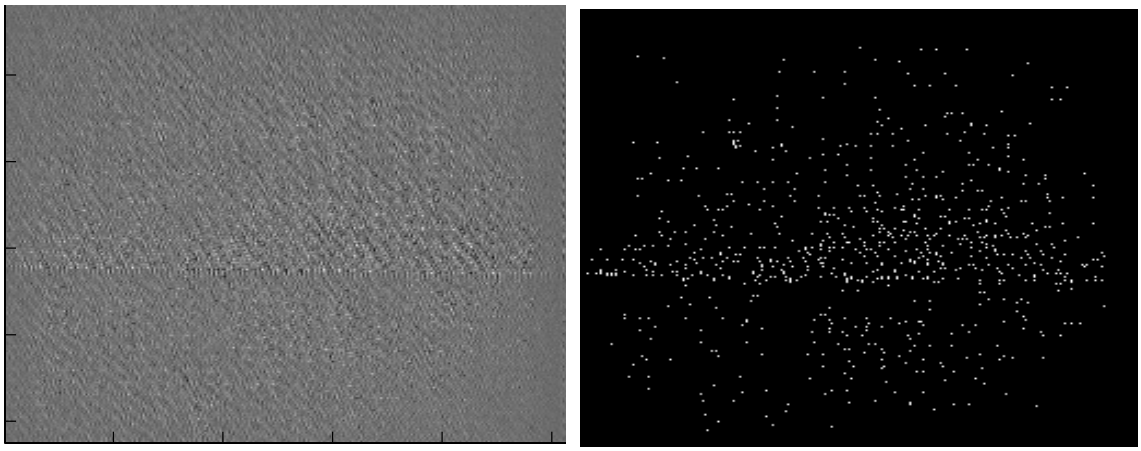


Figure 4.26: Filtered Mispick Image and resulting Thresholded Image

4.9 Sobel Edge Detection

The first method that was investigated to find defects in quadrants two and three was the Sobel edge detector. This method worked by performing a two dimensional gradient measurement on the image, which would emphasize areas of high gradient differential (Stanger 1991). The detector was constructed by creating a convolution matrix, shown below in Figure 4.27, and applying it to all pixels in the image with the exception of those pixels located at the edge of the image.

+1	+2	+1
0	0	0
-1	-2	-1

Figure 4.27: Convolution Matrix

The application of this convolution matrix involved moving through the inner *n*-2**m*-2 pixels in the image, where *n* is the number of rows and *m* is the number of columns. At each pixel in the aforementioned range, the convolution matrix was applied to all of the reference pixel's neighbors, and the sum of the resulting values was calculated. That sum was stored into the same row and column position as the reference pixel, only this time into a new array of identical size to the original image. The resulting output image could then be thresholded in the same manner as described above. The output of this detector did indeed emphasize areas of high gradient differential; however, due to the numerical noisiness of the original image, the detector also emphasized transitions that did not represent any sort of defect.

Because of these emphasized noise pixel blocks, there was no significant difference between the defect and the noise (shown in Figure 4.28), making the processes of thresholding and noise removal moot. Further research into techniques to find linear inconsistencies in an image were researched, and one such method that showed promise was the Sliding Window technique.

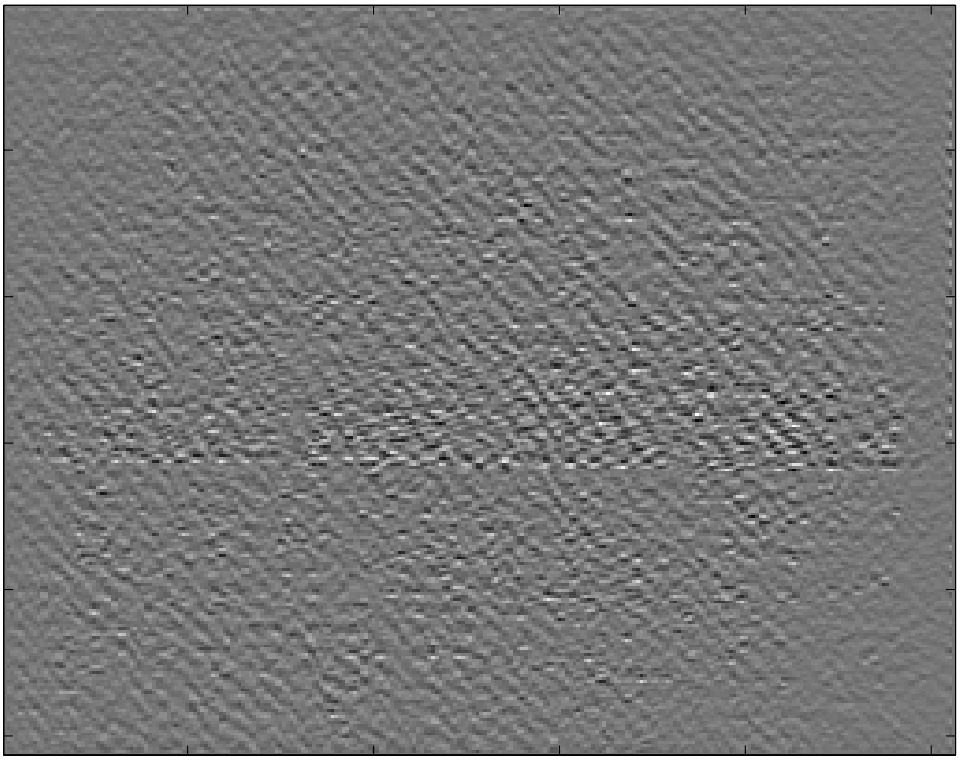


Figure 4.28: Mispick defect after Sobel Edge Detection

4.10 Sliding Window Technique

The final method that was used for defect detection in quadrants two and three was the Sliding Window technique. This technique involves constructing two windows around the reference pixel and calculating the standard deviation of the pixel values in each window. The first window constructed was 17 pixels square (eight pixels on either side of the reference pixel) and the second window was 9 pixels square (four on either side). As with the Sobel Edge Detector, the sliding window method worked with a subset of the pixels; only the inner *n*-16***m*-16 pixels were operated on by this technique to account for the size of the window, where *n* is the number of rows and *m* the number of columns. Once the standard deviations of the windows have been found, the ratio of the smaller window to the larger was

compared to a predefined threshold value. If the resulting ratio was less than the threshold, the reference pixel was set to one (black); otherwise, it was set to 255 (white).

Though the resulting image contains a great deal of noise blocks (see Figure 4.29), it was noticed that the size difference between the noise and the defect blocks was great enough to perform noise removal with an appropriate noise size (size >= 120). Once the noise blocks had been removed from the image, only blocks representing the mispick were still present as shown in Figure 4.30. The vertical line components that show up in the right hand sides of both the original sliding window application and the subsequent noise filtered images were due to the high gradient shift seen on that side of the image. These blocks represented no defects in the image, and therefore could be ignored. Therefore, no further processing was required.

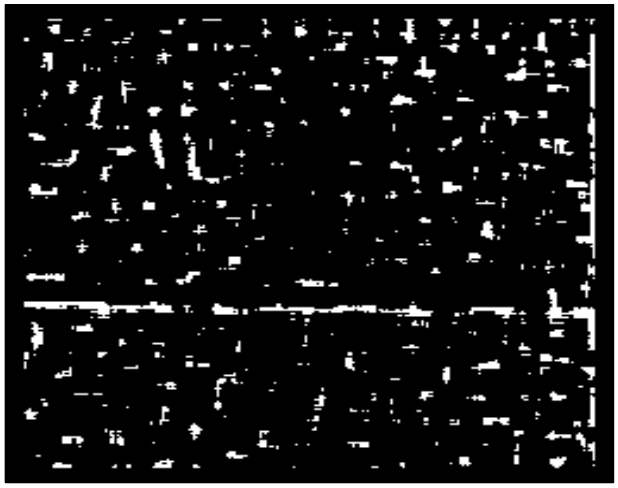


Figure 4.29: Mispick image after Sliding Window application

Figure 4.30: Sliding Window Image after Noise Removal

4.11 Robustness

In order to determine the robustness of the GA in terms of finding the "optimal" solutions, a set of experiments were run utilizing two different fabric images, a plain weave (Figure 4.13) and a 3x1 twill (Figure 4.31).

Figure 4.31: Twill Image

Full optimization runs were performed for each fabric image, meaning that each of quadrants two, three, and four were optimized separately. Ten full optimization runs were performed for each fabric image. For each trial run, a different random seed was used (one trial run denotes a separate optimization on each quadrant) so that the results between runs could be compared. Therefore, since all three quadrants were optimized with the same random seed and each run used a different random seed, this experiment tested the ability of the GA to find similar solutions using different starting points to determine the repeatability (robustness) of the optimization scheme. The results of this experiment are detailed in Section 5.3.

5.0 Results and Discussion

5.1 GA Development

The GA used during the course of this study underwent several phases of development before the final phase was deemed acceptable. The capability of the GA at each phase of development was evaluated in two different ways. First, the GA was given a simple two by two black and white checkerboard case to which the optimal solution was known (see Figure 4.1). Because the solution to this problem was known, this evaluation served as a benchmark for all phases of GA development (see Table 5.1). Furthermore, due to the symmetry of the problem, there were multiple permutations of the solution that were all deemed acceptable. For example, terms one and two could be swapped with terms three and four, but the solution retains its validity due to the symmetry present in the image (See Table 5.2).

Table 5.1: Simple Case Evaluation Results

GA Phase	Solution Found	Generations needed
Simple Float GA	Yes	≈1000
Float GA with Penalty	Yes	<100
Float GA with Gradient Search	Yes	<10
Float GA with Entropy	Yes	1

Table 5.2: Simple Case Solutions

Solution Num.	C_1	C_2	C_3	C_4
1	$\pm 1/\sqrt{2}$	$\pm 1/\sqrt{2}$	0	0
2	0	0	$\pm 1/\sqrt{2}$	$\pm 1/\sqrt{2}$
3	$\pm 1/\sqrt{2}$	0	0	$\pm 1/\sqrt{2}$
4	0	$\pm 1/\sqrt{2}$	$\pm 1/\sqrt{2}$	0

As shown in Table 5.1, all phases of the GA were capable of finding the optimal solution to the simple test case. However, it can be seen that subsequent phases of the GA required fewer generations to find that solution. Table 5.2 shows the four different combinations of solution coefficients that yield the optimal solution. The reason that there are four combinations results from the symmetry of the problem as well as the constraint equations that must be satisfied for the solution to be feasible. These equations are given by Equation 5.1. Note that any of the solution permutations shown in Table 5.2 solve Equation 5.1.

$$\begin{aligned} c_1^{\,2} + c_2^{\,2} + c_3^{\,2} + c_4^{\,2} &= 1 \\ c_1 c_3 + c_2 c_4 &= 0 \end{aligned} \tag{5.1}$$

Secondly, real fabric images were used to test the capabilities of the GA. In this case, since the optimal solution is unknown, the GA was tested for its ability to find a good solution, number of generations needed, and elapsed time (see Table 5.3). In this case, a good solution is defined as one that is feasible (satisfies all constraint equations) and that the GA is unable to improve upon after 50 or more subsequent generations (denotes the location

of a local optimal solution since the GA is unable to improve upon the current best solution). The evaluations presented here represent time and efficiency statistics only; actual optimization capability was ignored at this point in the GA evaluation.

Table 5.3: Real Fabric Evaluation Results

GA Phase	Solution Found	# Generations	Time	Time/Gen.
Simple Float	No	>40,000	16 hrs.	1.5 sec.
Float with Penalty	No	>40,000	16 hrs.	1.5 sec.
Float with Gradient	Yes	500	42 hrs.	300 sec.
Float with user Gradient	Yes	400	16.6 hrs.	150 sec.
Float with Entropy	Yes	400	1 hr.	9 sec.

Clearly, subsequent phases of the GA exhibited tremendous improvements over previous phases. The first improvement was incorporating a gradient descent search into the GA's objective function. This allowed the GA more local exploitation of promising areas in the solution space, thus allowing it to locate local optimal solutions. As shown in Table 5.3, this represented a tremendous improvement over the first two phases of the GA because it was then possible to find a fitting solution to the problem at hand. The first two phases of the GA were not equipped with any local improvement procedures, such as the gradient descent search, and this prevented those phases from being able to totally exploit promising areas in the solution space, which resulted in their long run time and inability to find a good solution.

Secondly, by supplying the objective function with a pre-calculated gradient, the run time was significantly improved. During the third phase of GA development, the gradient

search was added; however, the search algorithm was not supplied with an analytic form of the objective function gradient, and therefore had to calculate that gradient itself. The gradient search algorithm used a perturbation method to find the gradient. This method involved perturbing the function at various points and evaluating the response, thus allowing the calculation of the gradient. To improve the efficiency and speed of this search algorithm, the gradient was calculated symbolically ahead of time and supplied to the gradient search. Thus, time-savings were realized since the search algorithm simply had to plug in the wavelet coefficient values to the supplied gradient rather than having to solve for the gradient directly.

Finally, by using entropy as the objective function, a vast improvement was realized in the generational time and the number of generations required to find a local optimal solution. The gradient search that was used in the third and fourth phases of GA development represented very complex calculations due to the form of the objective function being used. That objective function was the sum of the squares of the pixel values after filter application (Equation 4.3), which resulted in a very complex gradient. The new objective function based on entropy still evaluates the image after filter application (first line of Equation 4.3), but then applies the entropy calculation to it (Equation 4.6). Trial runs showed that the new objective function was calculated faster than the previous functions, resulting in time savings during that phase of evaluation.

Based on these results alone, it was apparent that the first two phases of GA development represented algorithms that were incapable of solving for the wavelet coefficients for real fabric images. The other three phases, however, were capable of finding

adequate solutions and needed further evaluation to determine which phase was the best in terms of coefficient optimization.

The other main step to evaluating the various phases of GA development with respect to optimization capability was to study the uniformity of the filtered images. This was done by comparing histograms of pixel values in quadrant four generated from filtered images based on the different objective functions. The histograms for the sum of squares objective function and the entropy objective function are shown in Figures 5.1 and 5.2, respectively. Both plots exhibit the same basic shape, which is characteristic of this type of optimization. However, the distribution of pixel values for the sum of squares objective function is flatter and more spread out along its base than is the plot for the entropy objective function. Because the histogram for entropy exhibits a tighter distribution than does sum of squares, it was determined that the entropy objective function was better able to optimize the filtered image with respect to background uniformity. Furthermore, entropy as an objective function optimizes the coefficients based on the uniformity of the image, since a lower entropy value translates into a more uniform image; the sum of squares objective function is optimizing the image by minimizing pixel values and does nothing towards improving image uniformity. Combining these findings shows that entropy was able to filter out more of the fabric texture, resulting in a more uniform gray level in the filtered image.

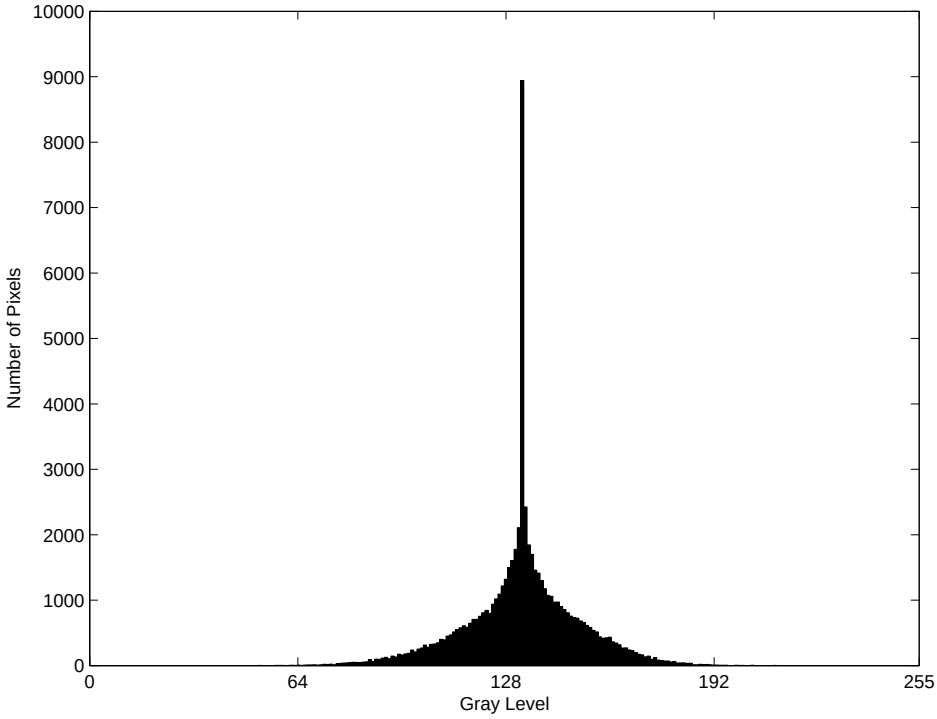


Figure 5.1: Quadrant 4 histogram using Sum of Squares

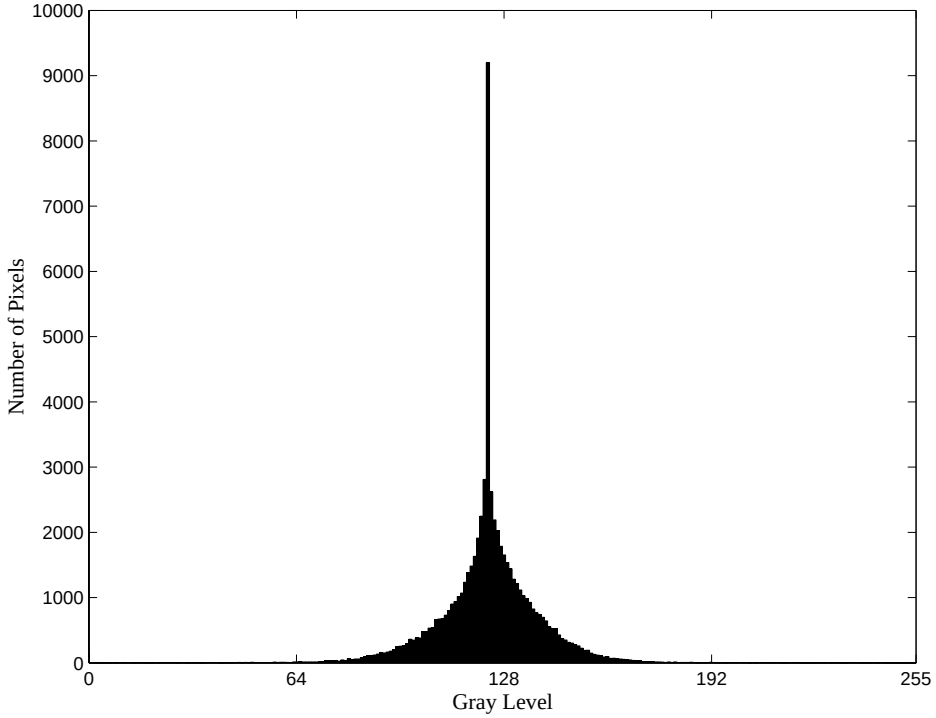


Figure 5.2: Quadrant 4 histogram using Entropy

Based on the findings from these evaluations of the GA development phases, it was determined that using entropy as the objective function was a better method for optimizing the wavelet filter coefficients with respect to texture removal. Therefore, this method would serve as the evaluation function for the GA throughout the remainder of this study.

5.2 Defect Segmentation

5.2.1 Quadrant 4 Defect Segmentation

Due to the nature of wavelet filters and their application to real fabric images, defects that are two-dimensional in nature, such as spots, holes, or tears, are emphasized most in quadrant four. This is due to the setup of the filter, which applies a high pass filter to quadrant four from both the horizontal and vertical directions. Figure 5.3 illustrates such an occurrence. Because the spot defect present represents a high frequency occurrence in both directions, it is visually apparent in quadrant four. However, the image is still expressed in terms of an 8-bit grayscale image, which precludes the use of a feature extractor due to the similarities of neighboring pixels in the 256-color scheme. This process begins with the use of a thresholding algorithm.

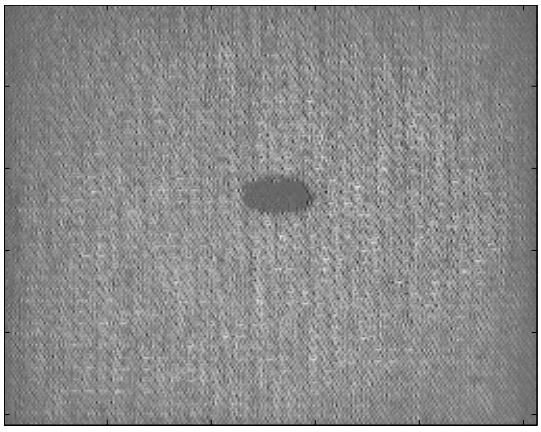


Figure 5.3: Quadrant four image with defect

The thresholding algorithm that was applied in this study was global thresholding. The algorithm is supplied with a thresholding point, and once applied, any pixel whose value is less than the threshold value is changed to black (gray level 0); any pixel above the setting is changed to white (gray level 255). In this manner, the image is converted to a black and white binary image, meaning that only two gray levels are present. Furthermore, for images such as that shown in Figure 5.3 where the defect is of a darker shade than the background, reverse thresholding is applied. This method is identical to normal thresholding except that pixel values below the threshold setting are changed to white, and all others are changed to black. The thresholding function constructed in this study can perform both actions.

Once thresholding is completed, the image has been converted to a binary image, which is suitable for feature extraction. Figure 5.4 illustrates the results of the threshold algorithm after application to the image shown in Figure 5.3. Again, reverse thresholding was applied since the defect was darker than the background. The image in Figure 5.4 also contains a large amount of small white pixel blocks. These pixel blocks represent noise,

which is a typical effect of global thresholding. Before feature extraction can take place, these blocks must be removed from the image.

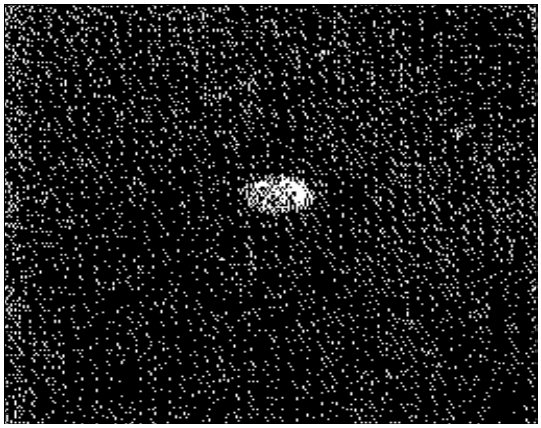


Figure 5.4: Quadrant four image after reverse thresholding

To remove the small white pixel blocks from the thresholded image, a noise removal algorithm was constructed. As detailed in Section 4.7, this algorithm is comprised of two functions: a pixel counter and a shade changer. The noise remover moves through the image and when it encounters a white pixel, it calls the pixel counter, which begins a series of recursive function calls that serve to map out the entire pixel block. Upon completion of the mapping process, the pixel counter returns a count of the number of pixels contained in that block to the noise remover. Should that number exceed a user-defined setting, the shade changer is called. This algorithm functions as does the pixel counter, but instead of counting the white pixels it encounters in the block, the shade changer changes those white pixels to black pixels, effectively removing them from the image. The end result of the noise removal algorithm is an image that contains only blocks of pixels larger than the user-defined size

setting. Figure 5.5 shows the image depicted in Figure 5.4 after it has been subjected to noise removal.

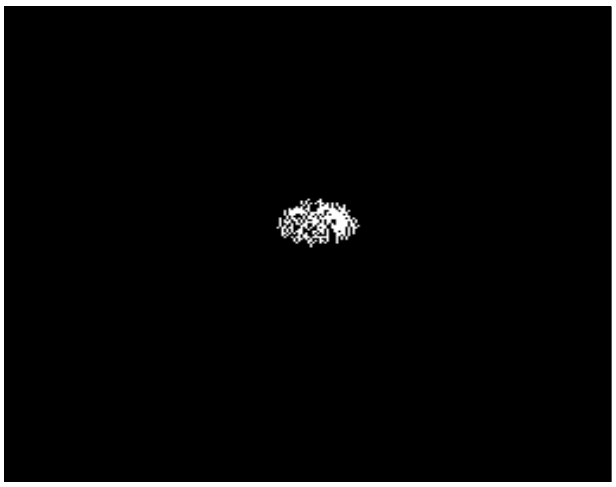


Figure 5.5: Quadrant four image after noise removal

Because the image has been converted to a binary image, and all noise pixel blocks have been removed, a feature extractor can now be employed to classify any defect that may be present in the image.

5.2.2 Quadrants 2 and 3 Defect Segmentation

Because the wavelet transform applies a low pass and a high pass filter to both quadrants two and three, defects that are linear in nature, such as mispicks or other yarn defects, will be emphasized in these quadrants depending on the orientation of the defect in question. Vertical defects are emphasized the most in quadrant two since they represent a high frequency event in the vertical direction and since the wavelet transform applies the high pas filter in this same direction. The transform applies the high pass filter to quadrant three in the horizontal direction, thus emphasizing the high frequency event that characterizes

a horizontal defect. Figure 5.6 depicts a horizontal defect, a mispick that was shown in quadrant three of the filtered image.

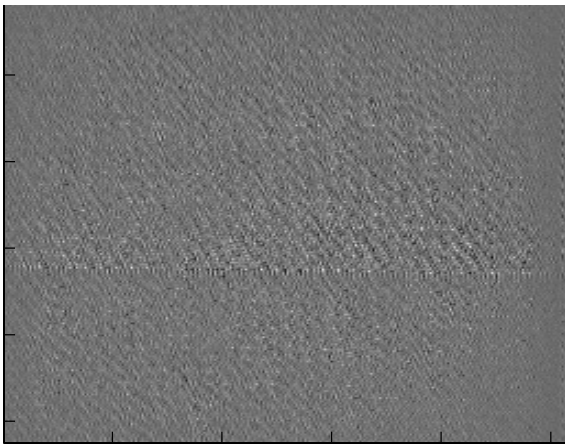


Figure 5.6: Horizontal defect in quadrant three

Attempts at segmenting defects in these quadrants began with the method used for quadrant four defects, thresholding and noise removal. However, because defects in these quadrants do not represent large deviations in gray scale values from those of the background, thresholding and noise removal proves incapable of segmenting those defects. Figure 5.7 shows such a defect after thresholding. Noise removal could not be performed on this image since the pixels representing the defect were computationally indistinguishable from those of noise. Therefore, other algorithms for segmenting defects in quadrants two and three needed to be found.

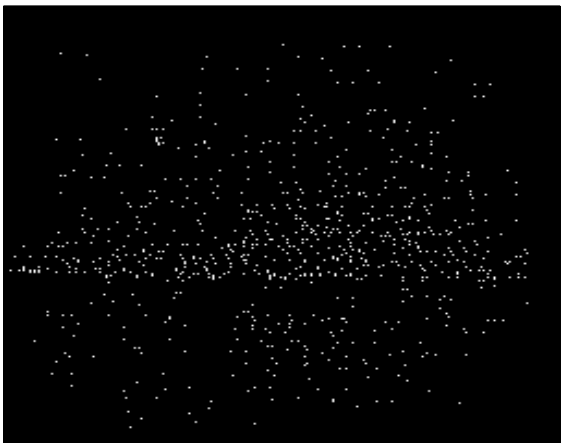


Figure 5.7: Horizontal defect after thresholding

The next attempt at segmenting these defects utilized a Sobel edge detector. These detectors emphasize areas of high gradient differential; in other words, areas of large shifts in gray scale values. Figure 5.8 depicts the mispick image after application of the Sobel algorithm. While the outside areas of the image have been made more uniform, the area surrounding the defect and the area of the defect itself are virtually identical in nature. This is due to the fact that there was enough texture information still present in the image that represented areas of high gradient differential, effectively masking the presence of any defects. Furthermore, because the regular repeating pattern of the fabric structure is made up of several yarns crossing each other, there exists a prevalence of edges that confound the Sobel algorithm, making it unsuitable for defect segmentation in quadrants two and three.

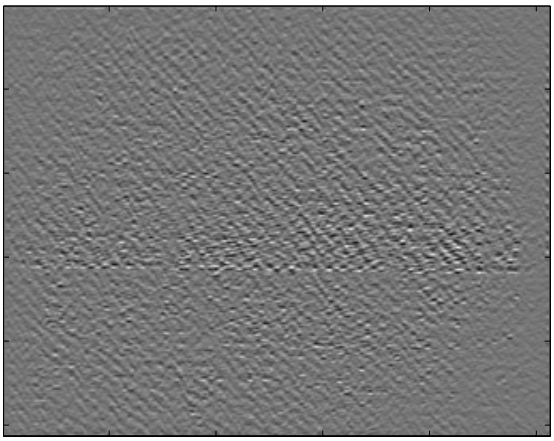


Figure 5.8: Horizontal defect after Sobel edge detection

The next attempt for defect segmentation made use of the sliding window technique. This method constructs two windows around each reference pixel, one large and one small. It then calculates the variation present in both windows and then determines the ratio of variation in the smaller window to that in the larger window. If this ratio is greater than a preset threshold, the reference pixel is changed to white; otherwise it is changed to black. The idea behind this algorithm is that if the reference pixel lies in an area that departs from the normal background of the image, the variation in the smaller window will be greater than that of the larger window since the larger window encompasses more of the normal background texture than does the smaller window. This serves to lessen the variation in the larger window, which in turn increases the ratio of the two. The mispick image after the sliding window algorithm has been applied is shown in Figure 5.9. The presence of the smaller white blocks throughout the image represents noise that must be removed to allow the use of a feature extractor. Figure 5.10 shows the image after the noise has been removed.

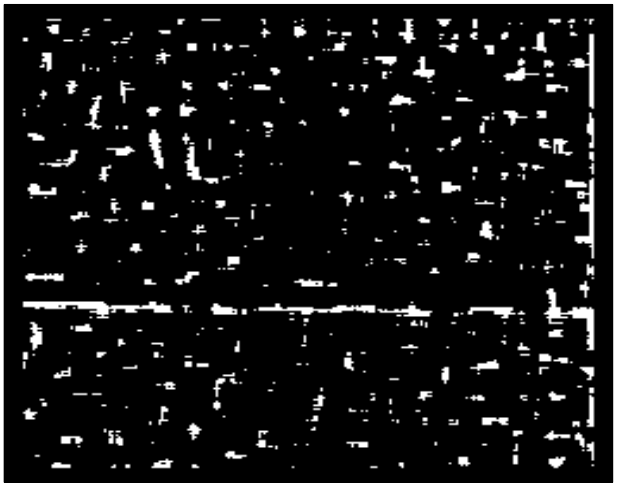


Figure 5.9: Horizontal defect after sliding window application

Figure 5.10: Horizontal defect after sliding window and cleaning

As shown in Figure 5.10 as compared to Figure 5.9, the image must undergo a significant level of noise removal due to the size of the noise blocks. However, because the defect was present across the entire width of the image, the pixel blocks representing it were significantly larger than those of noise. Therefore, noise removal can be successfully applied to this image without removing the defect itself. Now that the image has been converted to a

binary image and all noise has been removed, a feature extractor can be used to identify the defect present in the image.

5.3 Robustness

As detailed in Section 4.11, an experiment was run to determine the repeatability, or robustness, of the GA's optimization scheme. Table 5.4 lists the results from this experiment, giving the mean entropy value from all ten replications for each quadrant, as well as the standard deviation (Note: the data for quadrant two of the twill image was calculated from only 9 runs due to conflicts that arose in the GA for that specific random seed on that particular quadrant).

Table 5.4: Robustness Experiment Results

Fabric	Quadrant	Mean	Standard Deviation
Weave	2	5.9208	0.1092
Weave	3	6.0158	0.1286
Weave	4	5.9466	0.1297
Twill	2	5.7401	0.1607
Twill	3	5.8144	0.1707
Twill	4	5.7530	0.1356

These results show that the $\pm 2\sigma$ interval around each mean is within $\pm$ 5.8 % of that mean (the smallest $\pm 2\sigma$ interval was $\pm$ 3.6 % for quadrant 2 of the weave image). This indicates that the system will generate similar solutions regardless of the random seed setting, signifying that the system is indeed robust.

6.0 Conclusions

Experimentation showed that a GA tuned wavelet filter was capable of removing texture information from a fabric image, thus expediting the effectiveness of defect segmentation algorithms applied to the filtered images. Because the wavelet filter equations were couched as a non-convex, non-linear optimization problem, a hybrid GA was required to find the "optimal" solutions to these equations. Hybridizing the GA with a gradient descent search allowed the GA to employ its substantial global searching capabilities in a manner that, when combined with the local exploitation ability provided by the gradient search, proved sufficient to find the "optimal" set of wavelet coefficients. Furthermore, it was found that using a floating point GA representation as the basis for the optimization scheme proved better than a classical, or binary, GA representation since a floating-point representation operated on the floating-point coefficients without the need for a binary conversion, or discretization.

Image scaling and lighting were also shown to be critical factors to the fitness of a wavelet filter to its corresponding texture. If the fabric image exhibited too high a zoom level, the resulting wavelet filter would be incapable of filtering out enough texture to allow for effective defect segmentation. Accordingly, too low a zoom level resulted in a wavelet filter that removed not only the texture from the image, but also any defects present in that image due to the low contrast between defect and background texture. It was concluded that achieving a zoom level that fit one unit of the fabric repeat into an eight by eight pixel block made for optimal results; the images used contained 64 repeat units (8 units by 8 units, or 64x64 pixels). Since eight pixels represented the edge of a repeat unit, the corresponding wavelet filter was defined by a set of eight wavelet coefficients, one per pixel in the repeat

unit. In addition, the backlighting level was important since this factor had a direct impact on the contrast levels seen in the images. Should the backlighting be too low, not enough contrast would exist in the images, thereby resulting in a wavelet filter that could not adequately map out the texture. Furthermore, if the backlighting was too high, the texture would be washed out, again resulting in an inadequate wavelet filter. It was found that the best backlighting level gave an image where the yarns would show up in the 1-50 gray level range (black), whereas the spaces between the yarns would appear in the 220-255 range (white).

Experimentation also revealed that because real fabric images lack true symmetry, a wavelet filter defined by a single coefficient set could not remove enough fabric texture. Because the wavelet transform was applied to both the rows and columns of the fabric image, it was possible to define the row and column filters separately, each with a different set of wavelet coefficients. Each set was optimized for the direction of application, resulting in a level of filtering that was not possible with a universal coefficient set.

Because the wavelet transform applies a different combination of high and low pass filters to each of the four resulting quadrants, the size and orientation of the defects proved important. It was found that two dimensional defects, such as holes, tears, spots, or stains were emphasized most in quadrant four. Simple thresholding and noise removal proved sufficient to segment these defects for classification. However, quadrants two and three presented difficulties. Line defects, such as mispicks or broken ends, were emphasized in either quadrant two (vertical orientation) or quadrant three (horizontal orientation). Because these defects did not represent a considerable deviation from the background texture, as would a two-dimensional defect, simple thresholding and noise removal were not capable of

defect segmentation. It was found that application of the sliding window algorithm was needed to adequately segment this class of defects to allow for classification.

In summary, though a real-time automated system has not been constructed, the results of this study should provide a strong foundation upon which to carry out further research into the use of wavelet filters for automated inspection.

7.0 Recommendations for Future Research

This research showed that using a GA tuned wavelet filter for defect segmentation provided images suitable for feature classification, and therefore an apt method for online defect inspection. Because technological advancements are the main driving force of image processing applications, the utilization of faster computers would decrease analysis time. The computer used during this research was based upon a 2.16 GHz processor running on a 333 MHz front side bus (FSB) with an internal cache size of 256 Kb. Processors and the associated platforms are already on the market and sport FSB speeds of 400 MHz, processor speeds above 2.16 GHz, and internal caches exceeding 512 Kb. These processors also include more internal cache which has been shown to improve processor performance when running CPU intensive applications, such as image processing algorithms.

Aside from using better technology, the first recommendation would be to build the image processing algorithms using C rather than Matlab. Because Matlab executes uncompiled code, significant time is lost during runtime to create the executable code. By using the algorithms built in C, significant amounts of time will be saved, thus providing faster analysis times. Matlab is still recommended as the medium for algorithm creation as it allows for quick and easy construction. Furthermore, code written in Matlab can be easily translated into C, allowing for seamless transition from one language to the other.

The effective use of the system described in this research is dependent on the level of automation present. While this research represents strides towards efficient defect detection, such a system is impossible to implement on-loom at such time due to a lack of automation. First, the program needs to be able to automatically capture fabric images for analysis. This would need to be synchronized with the analysis routine to provide efficient image analysis.

Second, the wavelet filter and defect segmentation routines need to be automatically applied to all incoming images. This would require that appropriate threshold settings and noise removal settings be determined automatically according to some set of decision criteria. Once complete, these program constructs could be tied together through use of a controller program that would oversee the collection of images as well as the analysis itself. Finally, a feature classifier is required by the system to be able to identify what defects, if any, were present in a given image.

8.0 References

Baykut, A., Atalay, A., Ercil, A., Guler, M. Real-time defect inspection of textured surfaces. *Real-time Imaging*, 6:17-27, 2000.

Baykut, A., Ozdemir, S., Meylani, R., Ercil, A., Ertuzun, A. Comparative Evaluation of Texture Analysis Algorithms for Defect Inspection of Textile Products. Bogazici University Research Report, FBE-IE-08/97-12, 1997.

Chan, C. and Pang, G. Fabric defect detection by fourier analysis. *IEEE Transactions on Industry Applications*, 36(5):1267-1276, October 2000.

Coifman, R. R. and Wickerhauser, M. V. Entropy based algorithms for best basis selection. *IEEE Transactions on Information Theory*, 38(2 special issue pt II):713-718, 1992.

Houck, C. R., Joines, J. A., Kay, M. G. Empirical investigation of the benefits of partial lamarckianism. *Evolutionary Computation*, 5(1):31-60, 1997.

Houck, C. R., Joines, J. A., Kay, M. G. Comparison of genetic algorithms, random restart, and two-opt switching for solving large location-allocation problems. *Computers & Operations Research*, 23(6):587-596, 1996.

Jasper, W. J., Garnier, S., Potlapalli, H. Texture Characterization and defect detection using adaptive wavelets. *Optical Engineering*, 35(11):3140-3149, 1996.

Jasper, W. J., Joines, J. A., Brenzovich, J. A. Fabric Defect Detection Using a GA Tuned Wavelet Filter. 2002.

Joines, J. A. and Houck, C. R. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's. *Proceedings of the First IEEE Conference on Evolutionary Computation*, 2:579-584, 1994.

Joines, J. A., Houck, C. R., Kay, M. G. A genetic algorithm for function optimization: A Matlab implementation. Technical Report NCSU-IE Technical Report 95-09, North Carolina State University, 1996.

Joines, J. A. and Kay, M. G. *Utilizing Hybrid Genetic Algorithms*. Kluwer Academic Publishers, 2002.

Kumar, A. and Pang, G. Fabric defect segmentation using multichannel blob detectors. *Optical Engineering*, 39(12):3176-3190, December 2000.

Kumar, A. and Pang, G. Defect detection in textured materials using gabor filters. *IEEE Transactions on Industry Applications*, 38(2):425-440, April 2002.

Leung, L. W., King, B., Vohora, V. Comparison of image data fusion techniques using entropy and INI. *22nd Asian Conference on Remote Sensing*, November 2001.

Leung, L. W., Lam, F. K., To, J. T. P. Entropy-based multiscale image segmentation with edge refinement. *ICITA 2002 Conference Proceedings*, 2002.

Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. AI Series. Springer-Verlag, New York, 3rd edition, 1996.

Press, W., Teukolosky, W., Vetterling, W., Rlanner, B. *Numerical Recipes in C: the art of scientific computing*. Cambridge University Press, 2nd edition, 1992.

Recknagel, R., Kowarschik, R., Notni, G. High-resolution defect detection and noise reduction using wavelet methods for surface measurement. *Journal Of Optics A: Pure And Applied Optics*, 2(6):538-545, November 2000.

Renders, J. M. and Flasse, S. Hybrid methods using genetic algorithms for global optimization. *IEEE Transactions on Systems, Man, and Cybernetics Part B:*, 26(2):243-258, April 1996.

Sari-Sarraf, H. and Goddard, J. Jr. Vision system for on-loom fabric inspection. *IEEE Transactions on Industry Applications*, 36(6):1252-1259, November 1999.

Stanger, V. J. Edge-based combination algorithm applied to multiple restored images. *Electronics Letters*, 27(18):1638-1640, 1991.

Tsai, D. and Huang, T. Automated surface inspection for statistical textures. *Image and Vision Computing*, 21:307-323, 2003.

Yang, X. Z., Pang, G., Yung, N. Discriminative fabric defect detection using adaptive wavelets. *Optical Engineering*, 41(12):3116-3126, December 2002.

9.0 Appendices

9.1 System Usage Instructions

9.1.1 Image Gathering

Connect the camera to the PC via the camera link cable supplied by the image capture board manufacturer. Connect the light source to the power supply and turn on the power supply. Set the power supply to internal strobing; the light will appear to be constantly on even though it is being strobed continuously. Log into the PC and open the PDVshow program (camera monitor program supplied by the board manufacturer). Set the program for continuous image gathering; the program will gather images at a rate of 15 frames per second. Center the camera over the light source and place the fabric sample on top of the light source. Adjust the height of the camera over the fabric sample until approximately one fabric repeat unit fits into a 8x8 pixel square of the image. If you cannot determine the size at this time due to focus problems or light issues, follow the next steps and try again.

To tune the light source and camera for optimal images, begin by turning the light intensity on the power supply to its highest setting. Adjust the aperture setting on the camera lens until the gray levels near the center of the image are around 150-160. If adjusting the lens aperture cannot get the gray levels down into this range, slowly decrease the light intensity until the gray levels are in the desired range. Note that a lower aperture setting on the camera lens (more closed) will result in a greater depth of focus, which is desirable. Once the gray levels are in the desired range (150-160), adjust the focus level of the camera lens until the image appears sharp and crisp.

Next, set the camera monitor program to single image capture. Notice that when the program is in continuous capture, images that appear darker or lighter than normal will occasionally appear (the images will flicker a bit). This is normal and is due to the continuous strobing by the power supply. Continue to grab single images until an image is

captured with gray levels in the desired range (150-160). This may need to be done several times due to the flicker. Once a good image is captured, save the image as a TIFF file.

9.1.2 Image Conversion

Now that the image has been stored to a TIFF file, it needs to be converted into a Matlab M-file for training purposes. Open the TIFF file in any viewing program and find a good 64x64 pixel region that is free of defects. Note the row and column number of the region. Compile the extract.c program (see Appendix 9.2) and run it. Enter the starting row and number of rows (64 rows in this case) and the starting column and number of columns (64 columns in this case). The program will output the selected region to a Matlab M-file named picture.m. This step will need to be repeated again to convert the entire image to an M-file; for this, select a 512x512 pixel region (will usually contain some of the black image border around the light source). You may rename the M-file(s) to a more descriptive name (i.e., weave2.m, twill1.m, etc.) Move the M-file(s) into the directory containing the GA files and open Matlab. NOTE: the M-file(s) MUST be moved into the GA directory BEFORE Matlab is opened; otherwise, the program will not be able to find the files.

9.1.3 GA Training

Before the GA training phase can be started, some settings will need to be changed. Open the convert_pic2.m file. Add a line of code for the new file into the if then structure. Assign the file a number other than those already in use. Save the convert_pic2.m file.

Open the fconbean2cEq*.m file, where * is either 2, 3, or 4 (quadrant numbers). You may change the number of generations (termOps = [400], where 400 is the maximum number of generations), the mutation settings (denoted by mOpts), or the crossover settings (xOpts).

It is recommended that these settings be left unchanged as they represent the optimal settings found during experimentation. Since the image will have one repeat unit per 8x8 pixel block, you may leave the coefficient setting alone (evalOps, currently set to 16; two sets of coefficients, one coefficient per pixel in the repeat unit).

Once the desired settings have been changed, go to the Matlab command window and type in runAllQuads and press enter. This script is setup to run an optimization on each of quadrants 2, 3, and 4. After optimizing each quadrant, the script will save the coefficient data into individual MAT files, named q2.mat, q3.mat, and q4.mat. These files may be uploaded at a later time by typing load 'q*.mat' in the command window, where the * is the quadrant number you wish to load. Once the script has completed its run (takes roughly 3 hours on the computer detailed in section 3.0 of this paper), you may begin the defect segmentation process.

9.1.4 Applying the Wavelet filter

Make sure that the coefficients are all loaded into Matlab (if you did not shut down Matlab after the optimization runs, the coefficients are loaded; otherwise, load them with the command given above). At this time, the composite filtered image must be constructed. Run the setupPics script, which will do this for you. Type in [p pn] = setupPics(q2, q3, q4, ##) in the Matlab command window, where *p* is the variable you wish to store the un-normalized composite image into, *pn* is the variable you wish to store the normalized composite image into, q2, q3, and q4 are the variables containing the coefficients for those quadrants (may be other names) and ## is the picture number for the image you wish to work with (check convert_pic2.m for the appropriate image number). The script will create the wavelet filters for each quadrant, apply them to the image, and construct the composite image. This

composite image contains the fully optimized version of each quadrant combined into one image (one image is normalized, the other is not). The images are now ready for thresholding. Because quadrant 4 requires a different thresholding algorithm than quadrants 2 and 3, we will begin with quadrant 4.

9.1.5 Quadrant 4 Defect Segmentation

Create a variable named q4 and set it equal to quadrant 4 of the composite image by typing q4 = pn(33:64, 33:64) into the Matlab command window. Note that this command is for a 64x64 pixel image; should your image be a different size, replace 33 with half the picture size +1 and 64 with the max picture size (257:512 is for a 512x512 pixel image). Now that quadrant 4 has been isolated, it needs to be thresholded. Run the thresholding program by typing thresh = thresholder(pic_name, tval, ttype) into the command window. The variable pic_name is the variable that quadrant 4 is stored in; tval is the threshold value you wish to use; ttype sets which type of thresholding to use, 0 for normal, 1 for reverse thresholding. Normal thresholding will set all pixels whose value is larger than tval to white, and all other to black. Reverse thresholding will do the opposite. Look at the image by typing image(pic_name) into the command window, followed by colormap(gray). Pic_name is the variable containing the thresholded image. You may need to try different tval settings to achieve optimal thresholding. Typically, values above 180 work for normal thresholding, and values below 70 work for reverse thresholding.

Now that the image has been thresholded, the noise must be removed. Look at the image by typing image(pic_name) into the command window, followed by colormap(gray). Pic_name is the variable containing the thresholded image. Setting the colormap to gray allows you to view the image in black and white. Look at the noise in the image and

determine the approximate size of the noise pixel blocks. Then, run the noise remover program. Do this by typing cleaned = remove_noise(pic_name, nsize) into the command window, where pic_name is the variable containing the thresholded image and nsize the approximate size of the noise blocks (MUST be a positive integer). The variable cleaned will contain the cleaned image once the program finishes running. Again, you may need to adjust the nsize setting for optimal noise removal.

Now that the image has been thresholded and cleaned, nothing should remain other than the defect itself. This concludes defect segmentation for quadrant 4.

9.1.6 Quadrant 2 and 3 Defect Segmentation

Quadrants 2 and 3 use a different thresholding algorithm due to the nature of the defects that are emphasized in these quadrants. Capture the appropriate quadrant as you did for quadrant 4, but this time, the row and column numbers will change for the different quadrants. For example, on a 64x64 pixel image, quadrant 2 is (1:32, 33:64) and quadrant 3 is (33:64, 1:32). Adjust these numbers according to your image size. These two quadrants utilize the sliding window method for thresholding. Run the sliding window program by typing thresh = slidingWindow(pic_name), where pic_name is the variable containing the quadrant. You may need to adjust the threshold setting for optimal thresholding. This can be done by opening the slidingWindow.m file and changing the thresh variable setting. A larger number will translate into a stricter thresholding, since the ratio of the larger window to the smaller window must be larger. A smaller threshold setting will result in a weaker thresholding, since the ratio setting is smaller.

Apply noise removal in the same way as for quadrant 4, only this time, the nsize setting may need to be larger since the sliding window thresholding method usually results in substantially larger blocks of noise (the setting was around 120 during experimentation).

9.2 Image Extraction Program

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>


int main()
{
  int numrows, numcols, sRow, sCol;
  char infile[80];
  char str[80];
  unsigned char image[1018][1000];
  FILE *fd_out;
  int fd;
  int i, j;

  printf("Enter a file name: ");
  scanf("%s", infile);

  printf("Enter the starting row and number of rows: ");
  scanf("%d %d", &sRow, &numrows);

  printf("Enter the starting col and number of cols: ");
  scanf("%d %d", &sCol, &numcols);

  fd = open(infile, O_RDONLY);
  lseek(fd, 0x8, 0x0 );

  for ( i = 0; i < 1018; i++ ) {
    read(fd, &image[i][0], 1000);
  }

  close(fd);

  fd_out = fopen("picture.m", "w");
  strcpy(str, "global pic_image\n");
  fprintf(fd_out, "%s", str);
  strcpy(str, "pic_image=[\n");
  fprintf(fd_out, "%s", str);
```

9.3 Genetic Algorithm Code

fconbean2cEq2.m

```
global imatrix
echo on
% This script shows how to use the ga using a float representation.
% You should see the demos for
% more information as well. gademo1, gademo2, gademo3

% Setting the seed to the same for binary
%rand('seed',149)
disp('Yes')
% Crossover Operators
xFns = 'arithXoverm heuristicXoverm simpleXoverm';
%xFns = 'simpleXoverm'
xOpts = [2 0; 2 3; 2 0];

% Mutation Operators
mFns = ['boundaryMutationm multiNonUnifMutationm nonUnifMutationm
unifMutationm'];
%mFns = ['unifMutationm'];% multiNonUnifMutationm nonUnifMutationm
unifMutationm'];

% Termination Operators
termFns = 'maxGenTerm';
termOps = [400]; % 200 Generations

mOpts = [4 0 0;6 termOps(1) 3;6 termOps(1) 6;4 0 0];
% Selection Function
selectFn = 'normGeomSelectm';
selectOps = [0.08];

% Evaluation Function
evalFn = 'grad2cE_evalQ2';
evalOps = [16];
imatrix = convert_pic2(13);
%type grad2cE_evalQ3;

% Bounds on the variables
bounds = ones(evalOps(1),1)*[-1 1];
global penalty
penalty = [5];
penaltyFN='penaltybean';
penaltyOps = [2.8 1.2 15];%[alph1 alpha2 Nf]
% GA Options [psilon float/binar display]
gaOpts=[1e-6 1 1];

% Generate an intialize population of size 20
startPop = initializegam(100,bounds,evalFn,[evalOps penalty]);

% Lets run the GA
% Hit a return to continue
%pause
[xc endPop bestPop trace]=gac(bounds,evalFn,evalOps,startPop,gaOpts,...

termFns,termOps,selectFn,selectOps,xFns,xOpts,mFns,mOpts,penaltyFN,penalty
Ops);
```

```
% x is the best solution found
%xc{:}
% Hit a return to continue
a = xc{1};

bnd=ones(1,size(a,2));
options = optimset('LargeScale', 'off', 'Display', 'off');
options = optimset(options, 'GradObj','off','GradConstr','on'); %,
'DerivativeCheck','on');
options = optimset(options, 'MaxIter',500,'MaxFunEvals',5000);

[x, fval, exitflag, output] = fmincon('objfungrad2cEq2',a,[],[],[],[],-
bnd,bnd,'confungrad2c',options);

if exitflag > 0
    quad2c = x;
    [c, ceq, dc, dceq] = confungrad2c(quad2c);
    a;
    ceq;
else
    xc{:};
    quad2c = xc{1};
end

%pause

echo off
```

fconbean2cEq3.m

```
global imatrix
echo on
% This script shows how to use the ga using a float representation.
% You should see the demos for
% more information as well. gademo1, gademo2, gademo3

% Setting the seed to the same for binary
%rand('seed',149)
disp('Yes')
% Crossover Operators
xFns = 'arithXoverm heuristicXoverm simpleXoverm';
%xFns = 'simpleXoverm'
xOpts = [2 0; 2 3; 2 0];

% Mutation Operators
mFns = ['boundaryMutationm multiNonUnifMutationm nonUnifMutationm
unifMutationm'];
%mFns = ['unifMutationm'];% multiNonUnifMutationm nonUnifMutationm
unifMutationm'];

% Termination Operators
termFns = 'maxGenTerm';
termOps = [400]; % 200 Generations

mOpts = [4 0 0;6 termOps(1) 3;6 termOps(1) 6;4 0 0];
% Selection Function
selectFn = 'normGeomSelectm';
selectOps = [0.08];

% Evaluation Function
evalFn = 'grad2cE_evalQ3';
evalOps = [16];
imatrix = convert_pic2(13);
%type grad2cE_evalQ3;

% Bounds on the variables
bounds = ones(evalOps(1),1)*[-1 1];
global penalty
penalty = [5];
penaltyFN='penaltybean';
penaltyOps = [2.8 1.2 15];%[alph1 alpha2 Nf]
% GA Options [psilon float/binar display]
gaOpts=[1e-6 1 1];

% Generate an intialize population of size 20
startPop = initializegam(100,bounds,evalFn,[evalOps penalty]);

% Lets run the GA
% Hit a return to continue
%pause
[xc endPop bestPop trace]=gac(bounds,evalFn,evalOps,startPop,gaOpts,...

termFns,termOps,selectFn,selectOps,xFns,xOpts,mFns,mOpts,penaltyFN,penalty
Ops);
```

```
% x is the best solution found
%xc{:}
% Hit a return to continue
a = xc{1};

bnd=ones(1,size(a,2));
options = optimset('LargeScale', 'off', 'Display', 'off');
options = optimset(options, 'GradObj','off','GradConstr','on'); %,
'DerivativeCheck','on');
options = optimset(options, 'MaxIter',500,'MaxFunEvals',5000);

[x, fval, exitflag, output] = fmincon('objfungrad2cEq3',a,[],[],[],[],-
bnd,bnd,'confungrad2c',options);

if exitflag > 0
    quad3c = x;
    [c, ceq, dc, dceq] = confungrad2c(quad3c);
    a;
    ceq;
else
    xc{:};
    quad3c = xc{1};
end

echo off
```

fconbean2cEq4.m

```
global imatrix
echo on
% This script shows how to use the ga using a float representation.
% You should see the demos for
% more information as well. gademo1, gademo2, gademo3

% Setting the seed to the same for binary
%rand('seed',149)
disp('Yes')
% Crossover Operators
xFns = 'arithXoverm heuristicXoverm simpleXoverm';
%xFns = 'simpleXoverm'
xOpts = [2 0; 2 3; 2 0];

% Mutation Operators
mFns = ['boundaryMutationm multiNonUnifMutationm nonUnifMutationm
unifMutationm'];
%mFns = ['unifMutationm'];% multiNonUnifMutationm nonUnifMutationm
unifMutationm'];

% Termination Operators
termFns = 'maxGenTerm';
termOps = [400]; % 200 Generations

mOpts = [4 0 0;6 termOps(1) 3;6 termOps(1) 6;4 0 0];
% Selection Function
selectFn = 'normGeomSelectm';
selectOps = [0.08];

% Evaluation Function
evalFn = 'grad2cE_evalQ4';
evalOps = [16];
imatrix = convert_pic2(13);
type grad2cE_evalQ4;

% Bounds on the variables
bounds = ones(evalOps(1),1)*[-1 1];
global penalty
penalty = [5];
penaltyFN='penaltybean';
penaltyOps = [2.8 1.2 15];%[alph1 alpha2 Nf]
% GA Options [psilon float/binar display]
gaOpts=[1e-6 1 1];

% Generate an intialize population of size 20
startPop = initializegam(100,bounds,evalFn,[evalOps penalty]);

% Lets run the GA
% Hit a return to continue
%pause
[xc endPop bestPop trace]=gac(bounds,evalFn,evalOps,startPop,gaOpts,...

termFns,termOps,selectFn,selectOps,xFns,xOpts,mFns,mOpts,penaltyFN,penalty
Ops);
```

```
% x is the best solution found
%xc{:}
% Hit a return to continue
a = xc{1};

bnd=ones(1,size(a,2));
options = optimset('LargeScale', 'off', 'Display', 'off');
options = optimset(options, 'GradObj','off','GradConstr','on'); %,
'DerivativeCheck','on');
options = optimset(options, 'MaxIter',500,'MaxFunEvals',5000);

[x, fval, exitflag, output] = fmincon('objfungrad2cEq4',a,[],[],[],[],-
bnd,bnd,'confungrad2c',options);

if exitflag > 0
    quad4c = x;
    [c, ceq, dc, dceq] = confungrad2c(quad4c);
    a;
    ceq;
else
    xc{:};
    quad4c = xc{1};
end

echo off
```

objfungrad2cEq2.m

```
function [f] = objfungradjoe(c)
global imatrix
%imatrix = convert_pic2(13);

psize = size(imatrix,1)/2;
psize2 = size(c,2)/2;
sum = 0;
%c
%pmatrix

cmatrix1 = build_cmatrix(c(1:psize2),psize*2);
cmatrix2 = build_cmatrix(c(psize2+1:end), psize*2);

pic = cmatrix1*imatrix*cmatrix2';

q2 = pic(1:psize, psize+1:end);

%f = svds(q3, 1);
[row col] = size(q2);
q2 = reshape(q2', 1, row*col);
Hi = hist(q2,256);
Hip = nonzeros(Hi);
L = sumjoe(Hip);
Entropy = ((-sumjoe(Hip .* log2(Hip)) / L) + log2(L));

f = Entropy;
```

objfungrad2cEq3.m

```
function [f] = objfungradjoe(c)
global imatrix
%imatrix = convert_pic2(13);

psize = size(imatrix,1)/2;
psize2 = size(c,2)/2;
sum = 0;
%c
%pmatrix

cmatrix1 = build_cmatrix(c(1:psize2),psize*2);
cmatrix2 = build_cmatrix(c(psize2+1:end), psize*2);

pic = cmatrix1*imatrix*cmatrix2';

q3 = pic(psize+1:end, 1:psize);

%f = svds(q3, 1);
[row col] = size(q3);
q3 = reshape(q3', 1, row*col);
Hi = hist(q3,256);
Hip = nonzeros(Hi);
L = sumjoe(Hip);
Entropy = ((-sumjoe(Hip .* log2(Hip)) / L) + log2(L));

f = Entropy;
```

objfungrad2cEq4.m

```
function [f] = objfungradjoe(c)
global imatrix
%imatrix = convert_pic2(13);

psize = size(imatrix,1)/2;
psize2 = size(c,2)/2;
sum = 0;
%c
%pmatrix

cmatrix1 = build_cmatrix(c(1:psize2),psize*2);
cmatrix2 = build_cmatrix(c(psize2+1:end), psize*2);

pic = cmatrix1*imatrix*cmatrix2';

q4 = pic(psize+1:end, psize+1:end);

%f = svds(q3, 1);
[row col] = size(q4);
q4 = reshape(q4', 1, row*col);
Hi = hist(q4,256);
Hip = nonzeros(Hi);
L = sumjoe(Hip);
Entropy = ((-sumjoe(Hip .* log2(Hip)) / L) + log2(L));

f = Entropy;
```

confungrad2c.m

```
function [c, ceq, dc, dceq] = confungradjoe(coeff)

constr1 = zeros(1,size(coeff, 2)/4);
dconstr1 = zeros(size(coeff, 2)/2,size(coeff, 2)/4);

constr2 = zeros(1,size(coeff, 2)/4);
dconstr2 = zeros(size(coeff, 2)/2,size(coeff, 2)/4);

constr = zeros(1,size(coeff,2)/2);
dconstr = zeros(size(coeff,2),size(coeff,2)/2);

for j = 1:size(coeff, 2)/4,
   for i = 1:size(coeff, 2)/2-2*(j-1),
      constr1(j) = constr1(j) + (coeff(i)*coeff(i+2*(j-1)));
      constr2(j) = constr2(j) +
(coeff(i+size(coeff,2)/2)*coeff(i+size(coeff,2)/2+2*(j-1)));
   end
end

constr1(1) = constr1(1) - 1;
constr2(1) = constr2(1) - 1;

for i = 1:size(constr1,2),
    constr(i) = constr1(i);
    constr(i+size(constr2,2)) = constr2(i);
end

%constr = [coeff(1)^2+coeff(2)^2+coeff(3)^2+coeff(4)^2-1
      coeff(1)*coeff(3)+coeff(2)*coeff(4)];
coeff;
c = [];
ceq = constr;

for k = 1:size(coeff, 2)/2,
   for j = 1:size(coeff, 2)/4,
      if (k+2*(j-1) > size(coeff, 2)/2) & (k-2*(j-1) > 0)
          dconstr1(k, j) = coeff(k-2*(j-1));
          dconstr2(k, j) = coeff(k-2*(j-1)+size(coeff,2)/2);
      elseif (k-2*(j-1) < 1) & (k+2*(j-1) <= size(coeff, 2)/2)
          dconstr1(k, j) = coeff(k+2*(j-1));
          dconstr2(k, j) = coeff(k+2*(j-1)+size(coeff,2)/2);
      elseif (k-2*(j-1) > 0) & (k+2*(j-1) <= size(coeff, 2)/2)
          dconstr1(k, j) = coeff(k+2*(j-1)) + coeff(k-2*(j-1));
          dconstr2(k, j) = coeff(k+2*(j-1)+size(coeff,2)/2) + coeff(k-
            2*(j-1)+size(coeff,2)/2);
      else
          dconstr1(k, j) = 0;
          dconstr2(k, j) = 0;
      end
   end
end

for i = 1:size(coeff,2)/2,
    for j = 1:size(coeff,2)/4,
```

```
        dconstr(i,j) = dconstr1(i,j);
        dconstr(i+size(coeff,2)/2, j+size(coeff,2)/4) = dconstr2(i,j);
    end
end

%dconstr = [2*coeff(1) coeff(3); 2*coeff(2) coeff(4); 2*coeff(3) coeff(1);
      2*coeff(4) coeff(2)]; %coeff(3) coeff(4) coeff(1) coeff(2)];

dc = [];
dceq = dconstr;
```

9.4 Defect Segmentation Utility Code

build_cmatrix.m

```
function [cmatrix] = build_cmatrix(coeff, picsize)
%global picture
%picture;
%picsize = size(pic_image, 1);
cmatrix = zeros(picsize, picsize);
%pmatrixtemp = zeros(picsize, numcoeff);
for i = 1:(picsize)/2,
   for j = 1:size(coeff,2),
      cmatrix(i, mod(j-1+(i-1)*2,picsize)+1) = coeff(j);
   end
end
for j = 1:picsize,
   cmatrix((picsize/2)+1,j) = cmatrix(picsize/2,picsize+1-j);
   if(mod(j,2)==1)
      cmatrix((picsize/2)+1,j) = -cmatrix((picsize/2)+1,j);
   end
end
for i = 1:(picsize/2)-1,
   for j = 1:size(coeff,2),
      cmatrix(i+(picsize/2)+1,mod(j-1+(i-1)*2,picsize)+1) =
            coeff(size(coeff,2)-j+1);
      if(mod(j,2)==1)
         cmatrix(i+(picsize/2)+1,mod(j-1+(i-1)*2,picsize)+1) = -
            cmatrix(i+(picsize/2)+1,mod(j-1+(i-1)*2,picsize)+1);
      end
   end
end
```

convert_pic.m

```
function [pmatrix] = convert_pic(numcoeff, pic_image)
%global pic_image
%picture;
picsize = size(pic_image, 1);
pmatrix = zeros(picsize/2, numcoeff);
%pmatrixtemp = zeros(picsize, numcoeff);
%for i = 1:(picsize/2),
%   for j = 1:numcoeff,
%      pmatrix(i, j) = pic_image(mod(2*(i-1)+(j-1), picsize)+1);
%   end
%end

for i = 1:(picsize/2),
   for j = 1:numcoeff,
      pos = i*2-j+1;
      if(pos < 1)
          pos = pos + picsize;
      end
      %pmatrix(i+picsize/2,j) = pic_image(pos);
      pmatrix(i,j) = pic_image(pos);
      if(mod(j,2)==0)
          %pmatrix(i+picsize/2,j) = -pmatrix(i+picsize/2,j);
          pmatrix(i,j) = -pmatrix(i,j);
      end
   end
end
```

convert_pic2.m

```
function [squareimage] = convert_pic2(filenum)
%global picture
%picture;

if filenum == 1
    weave1;
elseif filenum == 2
    weave2;
elseif filenum == 3
    weave3;
elseif filenum == 4
    weave4;
elseif filenum == 5
    weave5;
elseif filenum == 6
    weave5d;
elseif filenum == 7
    weaveBig;
elseif filenum == 8
    twill1;
elseif filenum == 9
    twill1d;
elseif filenum == 10
    twill1d2;
elseif filenum == 11
    twill1d3;
elseif filenum == 12
    twill1Big;
elseif filenum == 13
    twill1s;
elseif filenum == 14
    general4;
elseif filenum == 22
    general2;
elseif filenum == 23
    general2defect;
elseif filenum == 24
    i2x2d1;
elseif filenum == 25
    i2x2r1;
elseif filenum == 26
    i2x2hdef;
elseif filenum == 27
    i2x2d1hdef;
elseif filenum == 28
    i2x2r1hdef;
elseif filenum == 29
    i2x2cdef;
elseif filenum == 30
    i2x2d1cdef;
elseif filenum == 31
    i2x2r1cdef;
else
    weave1defect;
```

```
end

picsize = size(pic_image, 1);
rownum = sqrt(picsize);
squareimage = zeros(rownum, rownum);
for i=1:rownum,
   for j=1:rownum,
      squareimage(i, j) = pic_image((i-1)*rownum + j);
   end
end
```

edgeDetect.m

```
function [sobel] = edgeDetect(opic)

[row col] = size(opic);
xval   = [-1  0  1 1  1  0 -1 -1];
yval   = [-1 -1 -1 0  1  1  1  0];
sobArr = [ 1  2  1 0 -1 -2 -1  0];
sobel = ones(row, col);

for i = 2:row-1,
    for j = 2:col-1,
        sum = 0;
        for k = 1:8,
            sum = sum + sobArr(k)*opic(i+yval(k),j+xval(k));
        end
        sobel(i,j) = sum;
    end
end
```

find_minMax.m

```
function [minX, maxX] = find_minMax(myArray)

minX = 1000;
maxX = -1000;

for i = 1:size(myArray, 1),
    for j = 1:size(myArray, 2),
        if myArray(i,j) > maxX
            maxX = myArray(i,j);
        end
        if myArray(i,j) < minX
            minX = myArray(i,j);
        end
    end
end
```

noise_count.m

```
function [nimage] = noise_count(image, i, j)

   global fcount;
   %image
   %i
   %j
   if(image(i,j) ~= 255)
       nimage = image;
       return;
   end
   image(i,j) = 254;
   fcount = fcount + 1;
   if (fcount >= 20)
         nimage = image;
       return;
   end
   image = noise_count(image, i-1, j-1);
   image = noise_count(image, i-1, j);
   image = noise_count(image, i-1, j+1);
   image = noise_count(image, i, j-1);
   image = noise_count(image, i, j+1);
   image = noise_count(image, i+1, j-1);
   image = noise_count(image, i+1, j);
   image = noise_count(image, i+1, j+1);

   nimage = image;
   return;
```

noise_erase.m

```
function [nimage] = noise_erase(image, i, j)

   if(image(i,j) ~= 254)
       nimage = image;
       return;
   end
   image(i,j) = 1;
   image = noise_erase(image, i-1, j-1);
   image = noise_erase(image, i-1, j);
   image = noise_erase(image, i-1, j+1);
   image = noise_erase(image, i, j-1);
   image = noise_erase(image, i, j+1);
   image = noise_erase(image, i+1, j-1);
   image = noise_erase(image, i+1, j);
   image = noise_erase(image, i+1, j+1);
   nimage = image;
   return;
```

normPic.m

```
function [pic] = normPic(image)

iSize = size(image,2);
pic = zeros(iSize, iSize);
%first quadrant
[minX maxX] = find_minMax(image(1:iSize/2, 1:iSize/2));
for i = 1:iSize/2,
    for j = 1:iSize/2,
        pic(i,j) = image(i,j)-minX;
    end
end
%[minX maxX] = find_minMax(pic(1:iSize/2, 1:iSize/2))
scaleFactor = (maxX-minX) / 255;
for i = 1:iSize/2,
    for j = 1:iSize/2,
        pic(i,j) = pic(i,j)/scaleFactor;
    end
end

%second quadrant
[minX maxX] = find_minMax(image(1:iSize/2, (iSize/2)+1:iSize));
for i = 1:iSize/2,
    for j = (iSize/2)+1:iSize,
        pic(i,j) = image(i,j)-minX;
    end
end
%[minX maxX] = find_minMax(pic(1:iSize/2, (iSize/2)+1:iSize))
scaleFactor = (maxX-minX) / 255;
for i = 1:iSize/2,
    for j = (iSize/2)+1:iSize,
        pic(i,j) = pic(i,j)/scaleFactor;
    end
end

%third quadrant
[minX maxX] = find_minMax(image((iSize/2)+1:iSize, 1:iSize/2));
for i = (iSize/2)+1:iSize,
    for j = 1:iSize/2,
        pic(i,j) = image(i,j)-minX;
    end
end
%[minX maxX] = find_minMax(pic((iSize/2)+1:iSize, 1:iSize/2))
scaleFactor = (maxX-minX) / 255;
for i = (iSize/2)+1:iSize,
    for j = 1:iSize/2,
        pic(i,j) = pic(i,j)/scaleFactor;
    end
end

%fourth quadrant
[minX maxX] = find_minMax(image((iSize/2)+1:iSize, (iSize/2)+1:iSize));
for i = (iSize/2)+1:iSize,
    for j = (iSize/2)+1:iSize,
        pic(i,j) = image(i,j)-minX;
```

```
    end
end
%[minX maxX] = find_minMax(pic((iSize/2)+1:iSize, (iSize/2)+1:iSize))
scaleFactor = (maxX-minX) / 255;
for i = (iSize/2)+1:iSize,
    for j = (iSize/2)+1:iSize,
        pic(i,j) = pic(i,j)/scaleFactor;
    end
end
```

remove_noise.m

```
function [cleaned] = remove_noise(image, pnum)

   global fcount;
   [row col] = size(image);
   for i = 1:row,
       image(i,1) = 1;
       image(i,row) = 1;
       image(1,i) = 1;
       image(row,i) = 1;
   end
   set(0,'RecursionLimit',1000)
   for i = 2:row-1,
      for j = 2:col-1,
         fcount = 0;
             if(image(i,j) == 255)
            image = noise_count(image, i, j);
            pixel_count = fcount;
            %sprintf('count %d, %d = %d',i,j,pixel_count)
                  if(pixel_count <= pnum)
               image = noise_erase(image, i, j);
               %sprintf('erase %d, %d = %d',i,j,pixel_count)
            end
         end
      end
   end

   for i = 1:row,
      for j = 1:col,
         if(image(i,j) == 254)
            image(i,j) = 255;
         end
      end
   end
   cleaned = image;
```

runAllQuads.m

```
fconbean2cEq2
save 'q2.mat' quad2c

fconbean2cEq3
save 'q3.mat' quad3c

fconbean2cEq4
save 'q4.mat' quad4c
```

setupPics.m

```
function [p, pn] = setupPics(c2,c3,c4, picnum)
global imatrix
imatrix = convert_pic2(picnum);
%imatrix = picnum;
psize = size(imatrix,1);
psize2 = size(c2,2)/2;

p = zeros(psize,psize);
pn = p;
cmatrix1 = build_cmatrix(c2(1:psize2),psize);
cmatrix2 = build_cmatrix(c2(psize2+1:psize2*2),psize);
cmatrix3 = build_cmatrix(c3(1:psize2),psize);
cmatrix4 = build_cmatrix(c3(psize2+1:psize2*2),psize);
cmatrix5 = build_cmatrix(c4(1:psize2),psize);
cmatrix6 = build_cmatrix(c4(psize2+1:psize2*2),psize);

o2 = cmatrix1*imatrix*cmatrix2';
o3 = cmatrix3*imatrix*cmatrix4';
o4 = cmatrix5*imatrix*cmatrix6';

o2n = normPic(o2);
o3n = normPic(o3);
o4n = normPic(o4);

for i = 1:psize/2,
    for j = 1:psize/2,
        %original output
        p(i,j) = o4(i,j);                                       %q1
        p(i,j+psize/2) = o2(i,j+psize/2);                       %q2
        p(i+psize/2,j) = o3(i+psize/2,j);                       %q3
        p(i+psize/2,j+psize/2) = o4(i+psize/2,j+psize/2);       %q4
        %normalized output
        pn(i,j) = o4n(i,j);                                     %q1
        pn(i,j+psize/2) = o2n(i,j+psize/2);                     %q2
        pn(i+psize/2,j) = o3n(i+psize/2,j);                     %q3
        pn(i+psize/2,j+psize/2) = o4n(i+psize/2,j+psize/2);     %q4
    end
end
```

slidingWindow.m

```
function [SLW] = slidingWindow(pic)

[row col] = size(pic);
SLW = ones(row, col);
arrA = zeros(289,1);
arrB = zeros(81,1);
stdA = 0;
stdB = 0;
thresh = 1.2;

for i = 9:row-8,
    for j = 9:col-8,
        qB = pic((i-8):(i+8),(j-8):(j+8));
        qA = pic((i-4):(i+4),(j-4):(j+4));
        stdB = std(reshape(qB,1,289));
        stdA = std(reshape(qA,1,81));
        if (stdB/stdA) <= thresh
            SLW(i,j) = 1;
        else
            SLW(i,j) = 255;
        end
    end
end
```

sumjoe.m

```
function [total] = sumjoe(myvec)

total = 0;
for i=1:size(myvec,1),
    total = total + myvec(i);
end
```

thresholder.m

```
function [thresh] = thresholder(opic,tval,ttype)
%ttype is 0 for normal thresholding, 1 for reverse thresholding

[row col] = size(opic);
thresh1 = zeros(row,col);
for i = 1:row,
    for j = 1:col,
        if(xor((opic(i,j) <= tval), ttype))
            thresh1(i,j) = 1;
        else
            thresh1(i,j) = 255;
        end
    end
end
thresh = thresh1;
```

writePic.m

```
function [] = writePic(filename, data)

fid = fopen(filename, 'w');
fprintf(fid, 'pic_image=[\n');
for i = 1:size(data,1),
    for j = 1:size(data,2),
        fprintf(fid, '%d\n', data(i,j));
    end
end
fprintf(fid, '];\n');
fclose(fid);
```

9.5: P-matrix Definition

$$
\mathrm{P} = \begin{bmatrix}
P_0 & P_1 & P_2 & P_3 & \cdots & P_{n-2} & P_{n-1} \\
P_2 & P_3 & P_4 & P_5 & \cdots & P_n & P_{n+1} \\
P_4 & P_5 & P_6 & P_7 & \cdots & P_{n+2} & P_{n+3} \\
P_6 & P_7 & P_8 & P_9 & \cdots & P_{n+4} & P_{n+5} \\
\vdots & & & & & & \\
P_{m-n} & P_{m-n+1} & P_{m-n+2} & P_{m-n+3} & \cdots & P_{m-2} & P_{m-1} \\
P_{m-2} & P_{m-1} & P_0 & P_1 & & P_{n-4} & P_{n-3} \\
 & & & & & & \\
P_1 & -P_0 & P_{m-1} & -P_{m-2} & \cdots & P_{m-n+3} & -P_{m-n+2} \\
P_3 & -P_2 & P_1 & -P_0 & \cdots & P_{m-n+5} & -P_{m-n+7} \\
\vdots & & & & & & \\
P_{n-1} & -P_{n-2} & P_{n-3} & -P_{n-4} & \cdots & P_1 & -P_0 \\
P_{n+1} & -P_n & P_{n-1} & -P_{n-2} & \cdots & P_3 & -P_2 \\
\vdots & & & & & & \\
P_{m-1} & -P_{m-2} & P_{m-3} & -P_{m-4} & \cdots & P_{m-n+1} & -P_{m-n}
\end{bmatrix}
$$

9.6: Wavelet Filter Matrix Definition

$$\mathrm{W} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & \\ \vdots & & & & & & \ddots & & & \\ & & & & & & & c_0 & c_1 & c_2 & c_3 \\ c_2 & c_3 & & & & & & & c_0 & c_1 \\ & & & & & & & & & \\ -c_1 & c_0 & & & & & \cdots & & -c_3 & c_2 \\ -c_3 & c_2 & -c_1 & c_0 & & & & & & \\ & & -c_3 & c_2 & -c_1 & c_0 & & & & \\ \vdots & & & & & & \ddots & & & \\ & & & & & & & -c_3 & c_2 & -c_1 & c_0 \end{bmatrix} = \begin{bmatrix} H \\ \\ L \end{bmatrix}$$