TECHNICAL REPORT STANDARD TITLE PAGE

1. Report Number	2. Government Accession No.	3. Recipient's Catalog No.
FHWA/TX-92/1189-2F		

4. Title and Subtitle	5. Report Date
Development of Image Algorithms for Automated Pavement Distress Evaluation System	November 1992
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
P. Chan, A. Rao, L. Li, and R. L. Lytton	Research Report 1189-2F
9. Performing Organization Name and Address Texas Transportation Institute Texas A&M University College Station, Texas 77843	10. Work Unit No.
	11. Contract or Grant No. Study No. 2-18-89-1189
12. Sponsoring Agency Name and Address Texas Department of Transportation Transportation Planning Division P. O. Box 5051 Austin, Texas 78763	13. Type of Report and Period Covered Final Sept. 1988 - August 1992
	14. Sponsoring Agency Code

15. Supplementary Notes

Research performed in cooperation with DOT and FHWA
Research Study Title "Automatic Photo Interpretation System for the ARAN"

16. Abstract

The first step in the successful management of pavements is the location and identification of the distresses. This allows the pavement manager to identify candidates for maintenance and rehabilitation. This requires the collection of a large volume of distress data differentiated by type, extent, and severity. Visual methods of collection have proven to be too labor intensive, inconsistent, and hazardous because of exposure to traffic. Currently, videotapes of highway pavement surfaces are collected with the Automatic Road Analyzer (ARAN). The videotapes are then brought back to the lab for visual evaluation. Therefore, there exists a need to automatically determine the type and extent of cracking by computerized means.

This report describes the image analysis algorithm to classify the cracking of Asphalt-concrete Pavement (ACP) and Continuously Reinforced Concrete Pavement (CRCP). The image analysis software features a three-pass approach. The first pass detects crack segments from the analysis of the block projection histogram. The second pass re-examines the vicinity of the detected edge segments to locate the remaining thinner crack segments with less stringent rules. The third pass is to classify the cracking type based on the position alignment and orientation of the crack segments in the edge map. Summarized results of a 12-mile ACP survey section are included to illustrate the above algorithm. The frame percent accuracy is better than 70% and 60% for a Hot-mix Asphalt-concrete (HMAC) surface and a seal coat surface, respectively. Furthermore, the processing results of 0.5 miles of CRCP also show that the percent accuracy is better than 70%.

17. Key Words	18. Distribution Statement
Pavement Crackings, Pavement Surface Distress, Image Processing, Pattern Recognition, Image Analysis	No Restrictions. Available through the National Technical Information Service 5285 Port Royal Road Springfield, Virginia 22161

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	117	

Form DOT F 1700.7 (8-69)

Development of Image Algorithms for Automated Pavement Distress Evaluation System

by

Paul Chan

Ashok Rao

Lan Li

Robert L. Lytton

Texas Transportation Institute

Research Report 1189-2F

Study Title: "Automatic Photo Interpretation System for the ARAN"
Project 2-18-89-1189

Sponsored by

Texas Department of Transportation

in cooperation with

U.S. Department of Transportation

Federal Highway Administration

November 1992

ABSTRACT

The first step in the successful management of pavements is the location and identification of the distresses. This allows the pavement manager to identify candidates for maintenance and rehabilitation. This requires the collection of a large volume of distress data differentiated by type, extent, and severity. Visual methods of collection have proven to be labor intensive, inconsistent, and hazardous because of exposure to traffic. Currently, videotapes of highway pavement surfaces are collected with the Automatic Road Analyzer (ARAN). The videotapes are then brought back to the lab for visual evaluation. Therefore, there exists a need to automatically determine the type and extent of cracking by computerized means.

This report describes the image analysis algorithm to classify the cracking of Asphalt-concrete Pavement (ACP) and Continuously Reinforced Concrete Pavement (CRCP). The image analysis software features a three-pass approach. The first pass detects crack segments from the analysis of the block projection histogram. The second pass re-examines the vicinity of the detected edge segments to locate the remaining thinner crack segments with less stringent rules. The third pass is to classify the cracking type based on the position alignment and orientation of the crack segments in the edge map. Summarized results of a 12-mile ACP survey section are included to illustrate the above algorithm. The frame percent accuracy is better than 70% and 60% for a Hot-mix Asphalt-concrete (HMAC) surface and a seal coat surface, respectively. Furthermore, the processing results of 0.5 miles of CRCP also show that the percent accuracy is better than 70%.

DISCLAIMER

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official view or policies of the Texas Department of Transportation or the Federal Highway Administration. This report is not intended for construction, bidding, or permit purposes. The principal investigator of this research study is Dr. Robert L. Lytton who is a registered Professional Engineer of Texas (No. 27657).

IMPLEMENTATION STATEMENT

The developed cracking classification software can assist the Texas Department of Transportation (TxDOT) in collecting pavement surface distress data of Asphalt-concrete Pavement (ACP) and Continuously Reinforced Concrete Pavement (CRCP) for the Pavement Management Information System (PMIS). Along with the improvements in the video image quality and computer controllable Video Cassette Recorder (VCR), this developed system can routinely process large numbers of pavement videotapes and extract distress data automatically.

ACKNOWLEDGEMENTS

The authors wish to thank Mr. Thomas Scullion, program manager of Pavement Systems in TTI, with his enthusiastic participation, insightful advice and expertise in the Pavement Management Information system, he has contributed much to the success of this study. The authors also wish to thank the TxDOT project panel chairman, Mr. David Fink for his conceptual contributions at every stage of the study, member Mr. Carl Bertrand for his helpful advice in electronic instrumentation and member Mr. Al Rubio for his recommendations related to the video camera and video cassette recorder.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

Highway systems form the backbone of the transportation infrastructure of this nation. These resources are used for trade and travel. In this context, the importance of having safe and sound pavements cannot be emphasized enough. To keep pavements in good working condition, planning, regular maintenance, and efficient use of allocated funds are required. Hence a maintenance scheme that optimizes available resources for maintenance is a prime necessity.

The primary contributing factors to pavement wear and tear are environmental and loading, which can cause stress on the pavement surface or within the pavement layers. The stress manifests itself in the forms of cracks, spalling, and rutting. Without regular maintenance, the pavement will break up resulting in high costs of reconstruction. However, if maintenance treatments are applied at the critical time, the usable life of the roadway will be extended.

The Texas Department of Transportation (TxDOT) has implemented a Pavement Management Information System (PMIS) to assist in its management of the highway network. PMIS collects four types of road data to monitor the road condition. The four types of data are pavement surface distress, ride quality, deflection, and skid resistance. The pavement surface distress data are collected by trained rating teams or the Automatic Road Analyzer (ARAN). The ARAN records the pavement surface condition on videotapes which are reviewed to determine distress.

Sending rating teams out to the field, expose the personnel to hazardous traffic. The labor costs that are associated with this visual survey are also very high. 150 to 200 raters are trained annually at an estimated cost of $500,000.

Besides rating teams, the current distress survey also deploys the Automatic Road Analyzer (ARAN) vehicle to survey high traffic roads in urban areas such as Houston and Dallas. The recorded video tape is brought back to the office for data reduction. A number of trained technicians review and count and measure the different distresses recorded on the video tape.

However, this visual evaluation is very tedious work, and fatigue reduces accuracy.

TxDOT has been searching for ways to improve the evaluation process. With the pavement surface condition recorded on videotape, the next logical step is the automatic analysis of the video images. Development in integrated circuit design and fabrication techniques has grown to a point where very complex circuits will only take up one single chip. Growth in surface mount technology and multi-layer circuit board design result in more computer power on a single board.

Digital image processing techniques have long been considered as valuable contributions to the science and engineering arena because they enable a professional to comprehend a two-dimensional view or even a three-dimensional object rather than the conventional one-dimensional signal. Despite the tremendous potential of digital image processing (DIP) techniques, the applications of (DIP) remained locked in the research institutions, R&D departments of large corporations and the military due to the expensive hardware requirements. The advancements in Very Large Scale Integrated Circuit (VLSI) have provided an affordable hardware platform. Therefore, only in recent years have most of the disciplines started applying image processing technology in their areas.

The goal of this project is to develop a comprehensive image processing system to assist the TxDOT in distress data collection. Currently the videotapes of pavement surface condition are brought to the office for manual visual evaluation. The processing system developed in this project can analyze these video tapes automatically. The system determines the distress by types and extent. The image algorithms were designed to meet three criteria outlined by the TxDOT project panel. The criteria were (a) the percent accuracy must be above 70%, (b) the processing speed must be at least 4 miles per hour, and (c) the resolution for crack width is 1/8". These three requirements were based on the comparison with subjective visual rating.

Chapter II describes the common surface distress in Asphalt-concrete Pavement (ACP) and Continuously Reinforced Concrete Pavement (CRCP). The report proceeds to describe the development efforts of the image processing algorithms to meet the three project criteria. The image feature selection

is the most critical link in the classification process. Chapter III describes the image features and the utilization of the image features in the classification of different distress types. Chapter IV presents the classification rules for ACP and CRCP along with processing results. Chapter V states the conclusions and recommendations for future work.

CHAPTER II

PAVEMENT SURFACE DISTRESS

TxDOT Pavement Management Information System (PMIS) considers many aspects of the pavement to provide input to its maintenance and rehabilitation programs. An important aspect of the PMIS is surface distress which is provided by a visual evaluation survey. The visual evaluation survey looks at three major pavement types.

1. Asphalt-concrete Pavement (ACP)
2. Continuously Reinforced Concrete Pavement (CRCP)
3. Jointed Concrete Pavement (JCP)

Each pavement type has different surface distresses. TxDOT currently uses rating teams and video to collect visual evaluation data. The video recordings are rated manually. The problems in manual rating of the video tapes are listed below.

1. Accuracy
2. Operator Fatigue
3. Repeatability

To overcome the problems, TxDOT started research to automate the video tape rating process. The primary objective of this research was to develop procedures for automated rating of the pavement using image processing. It was desired that the automated rating system detects all types of surface distress that occur and generates a rating of the pavement comparable to that of a human observer.

Some of the problems faced in this research were:

1. The presence of different types of cracking on pavement surface.
2. Features like oilstains, tire marks, and paint markings on pavement surface are mistaken for cracking.
3. Varying color and texture of the pavement.

TxDOT annually publishes a manual that defines distresses by pavement type and describes how to rate these distress. This manual is called the "TxDOT Pavement Management Information System Rater's Manual".

2.1 ACP

TxDOT rates the following distresses on ACP.

1. Rutting.
2. Patching.
3. Failures.
4. Block Cracking.
5. Alligator Cracking.
6. Longitudinal Cracking.
7. Transverse Cracking.

However, in this research project, only four cracking distress types are evaluated with the automatic method. They are Longitudinal Cracking, Transverse Cracking, Alligator Cracking and Block Cracking. Rutting is collected with the ultrasonic rut bar. The number of Patches and Failures will be keyed in by the operator during ARAN's video survey. The following sections are excerpts from the 1992 Pavement Management Information System Rater's Manual.

Raters travel along the side of the road (with traffic, on the roadbed being rated) at no more than 15 miles per hour, rating the most severely distressed lane, and stopping at least once every 0.5 miles. The first stop is made at the beginning of the section. At this 200 foot stop, the raters observe all distress types visible. The purpose of the stop is to "calibrate" the rater's vision as to which distress types exist within the section. Additional stops are made where major changes occur. At the end of the section, raters enter their overall section ratings for all of the flexible pavement distress types.

A. Rutting

A rut is a surface depression in a wheelpath. Rutting in the rated lane may be observed in one or both of the wheelpaths. Rutting is caused by consolidation or lateral movement of the pavement materials due to traffic loads. Significant amounts of rutting indicate that one or more of the pavement layers is inadequate. Rutting is indicative of a structural problem and may lead to the onset of serious structural failures.

B. Patching

Patches are repairs made to pavement distress. The presence of patching indicates prior maintenance activity, and is thus used as a general measure of maintenance cost.

C. Failures

A Failure is a localized section of pavement where the surface has been severely eroded, badly cracked, or depressed. Failures are important to rate because they identify specific structural deficiencies which may pose safety hazards.

D. Block Cracking

Block cracking consists of interconnecting cracks that divide the pavement surface into approximately rectangular pieces, varying in size from 1 foot by 1 foot up to 10 feet by 10 feet. Although similar in appearance to alligator cracking, block cracks are larger. Block cracking is not load-associated. Instead, it is commonly caused by shrinkage of the asphalt concrete or by shrinkage of cement or lime stabilized based courses.

E. Alligator Cracking

Alligator cracking consists of interconnecting cracks which form small, irregularly-shaped blocks resembling the patterns found on an alligator's skin (Figure 1). Blocks formed by alligator cracks are less than 1 foot by 1 foot (0.3 meters by 0.3 meters). Larger blocks are rated as block cracking.

Alligator cracks are formed whenever the pavement surface is repeatedly flexed under traffic loads. As a result, alligator cracking may indicate improper design or weak structural layers. Alligator cracking may also be caused by heavily-loaded vehicles.

F. Longitudinal Cracking

Longitudinal cracking consists of cracks or breaks which run approximately parallel to the pavement centerline (Figure 2). Edge cracks, joint or slab cracks, and reflective cracking on composite pavement (i.e. overlaid concrete pavement) may all be rated as longitudinal cracking.

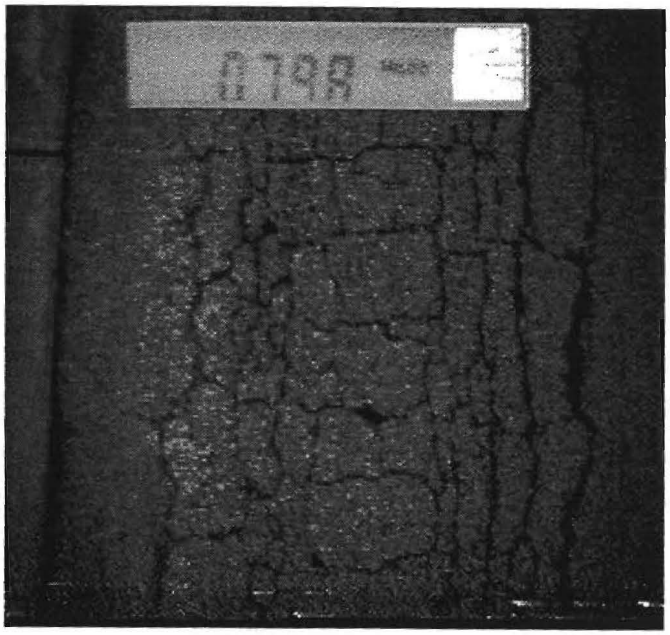

Figure 1. Alligator Cracking.

Figure 2. Longitudinal Cracking.

Differential movement beneath the surface is the primary cause of longitudinal cracking.

G. Transverse Cracking

Transverse cracking consists of cracks or breaks which travel at right angles to the pavement centerline (Figure 3). Joint cracks and reflective cracks may also be rated as transverse cracking.

Transverse cracks are usually caused by differential movement beneath the pavement surface. They may also be caused by surface shrinkage due to extreme temperature variations.

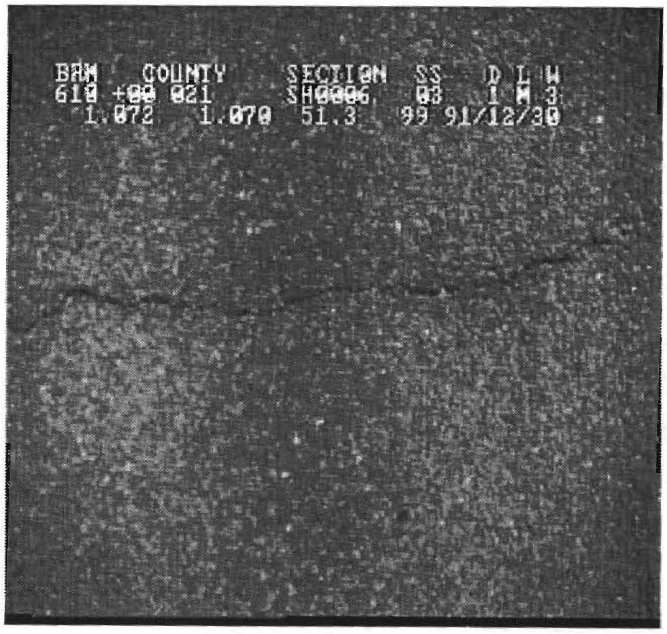

Figure 3. Transverse Cracking.

2.2 CRCP

In order to distinguish various types of distress, their physical shape characteristics are examined. According to the TxDOT rater's manual for CRCP (Continuously Reinforced Concrete Pavement), there are basically five types of distress. They include:

1. Spalled Cracks
2. Punchouts
3. Asphalt Patches

4. Concrete Patches

5. Average Crack Spacing

Among the five distress types, the frequency of occurrence of Punchouts, Asphalt Patches and Concrete Patches are relatively small when compared with Spalled Cracks and Transverse Cracks. In addition, the developed algorithm was aimed at the recognition of cracks. After a project meeting with the TxDOT project panel, the research team decided to employ the automatic evaluation techniques only for Spalled Cracks and Average Crack Spacing while the Punchouts and Patches are entered into the computer system during the survey by operators.

Rigid pavement sections (CRCP or JCP) are rated according to a different procedure from the ACP. Raters should begin counting distress occurrences at one end of the section, travelling along the edge of the road (with traffic, on the roadbed being rated) at no more than 15 miles per hour. The only stop required is at the end of the survey section, to enter the evaluation data.

A. Spalled Cracks

A spalled crack is a crack which has widened, showing signs of chipping on either side, along some or all of its length (Figure 4).

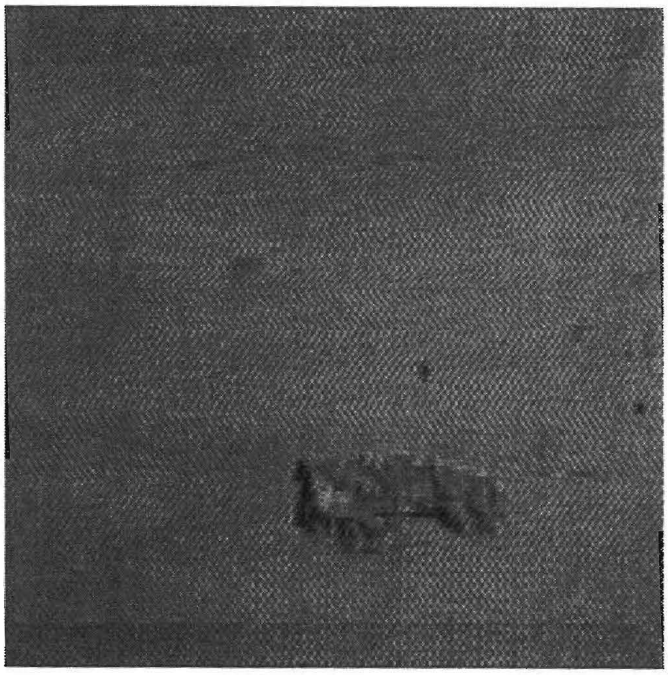

Figure 4. Spalled Crack.

B. Punchouts

A punchout is a block of pavement formed when one longitudinal crack crosses two transverse cracks. Although usually rectangular in shape, some punchouts may appear in other shapes.

C. Asphalt Patches

An asphalt patch is a localized area of asphalt concrete which has been placed to the full depth of the surrounding concrete slab, as a temporary method of correcting surface or structural defects.

D. Concrete Patches

A concrete patch is a localized area of newer concrete which has been placed to the full depth of the existing slab as a method of correcting surface or structural defects.

E. Average Crack Spacing

Average crack spacing is not, in itself, a pavement distress type. It is rated as a method of obtaining the percentage of transverse cracks that are spalled. However, average crack spacing is valuable as a measure of whether or not the CRCP slab is behaving as designed. A CRCP section with a small average crack spacing may deteriorate rapidly into a series of small punchouts if the proper corrective procedures are not applied.

2.3 JCP

According to the Rater's manual, the following distress types are to be rated on JCP sections.

1. Failed Joints and Cracks
2. Failures
3. Shattered Slabs
4. Slabs with Longitudinal Cracks
5. Concrete Patches
6. Apparent Joint Spacing

Of the six distress types, only the Failed Joints and Cracks and Slabs with Longitudinal Cracks are potential candidates for automatic evaluation. Distress types such as Failures and Shattered Slabs which include a number of different distress types. For example, Failures include the following five distress types:

1. Corner Breaks
2. Punchouts
3. Asphalt Patches
4. Failed Concrete Patches
5. D - Cracking

Each distress type shows a different form of pavement surface distortion or disintegration. These varieties in the physical shape and pattern can become so complex that the computer evaluation will be a very slow process. The process time that is required to sort out each possible pattern will exceed the time criterion set up at the beginning of the project. Because of the relative small number of miles of existing JCP, and most new construction in rigid pavement will be of CRCP, the TTI research team, after consulting with the TxDOT project panel, has decided to postpone the algorithm development for JCP. The following two sections are excerpts from the 1992 PMIS Rater's Manual.

A. Failed Joints and Cracks

The distress type Failed Joints and Cracks covers two major items: spalled joints or transverse cracks, and asphalt patches of spalled joints or transverse cracks.

B. Failures

Failures are localized areas in which traffic loads do not appear to be transferred across the reinforcing bars. Failures are typically areas of surface distortion or disintegration.

C. Shattered Slabs

A shattered slab is a slab which is so badly cracked that it warrants complete replacement.

D. Slabs with Longitudinal Cracks

A longitudinal crack is a crack which roughly parallels the roadbed centerline.

E. Concrete Patches

A concrete patch is a localized area of newer concrete which has been placed to the full depth of the existing slab as a method of correcting surface or structural defects.

F. Apparent Joint Spacing

Some transverse cracks may become so wide that they look and act like joints. The crack must be at least 1/2" wide all the way across the lane. These "apparent" joints are important because they serve to divide the original slab into smaller units.

CHAPTER III
IMAGE ANALYSIS AND IMAGE FEATURES

This section describes the image features used for image analysis in the classification of cracking on ACP and CRCP. At the beginning, it was decided to employ spatial domain image features rather than spatial frequency domain image features. The primary reasons for employing spatial domain images features are to meet the processing time requirement and to employ less expensive imaging hardware. The designed algorithm locates crack segments in small square blocks of 48 pixels. A pixel (picture element) is the basic unit of information in a digitized image. A large number of pixels arranged in a 2 dimensional array can reproduce the complete picture. The quality with which a picture is reproduced is called the resolution of the image and is a function of the number of pixels used in the 2 dimensional image array. The arrangement of pixels in a digitized video image array of 512 pixels in width by 480 pixels in depth provides good resolution and is an accepted industry standard.

Each pixel has an attribute associated with it, called the greylevel. The greylevel is the discrete value that each pixel holds and is a representation of the intensity of the picture at that point. Since a pixel is represented by an 8 bit integer in the computer memory, the number of discrete values it can take range from 0 to 255. Hence, a pixel describes 256 levels of grey in the image. A greylevel image is one in which each of the 512 by 480 pixels represents the intensity or greylevel of the picture at that point. Color images have pixels that carry extra information, namely Red, Green, Blue (RGB) colors. However, since pavements are a shade of grey, greylevel images are adequate to describe the picture completely. In an image, the dark pixels take on values in the lower end of the 0 - 255 range while light pixels take on values in the upper end of the 0 - 255 range. Typically, dark pixels can have a value such as 10 while light pixels can have a value of 200. Cracks on pavements show up as dark pixels on a digitized image having a value between 40 to 60 whereas the background has a value between 80 to 120. Thus, a crack detection scheme will locate pixels with lower greylevels by differentiating them from the background, then compute the severity, extent and orientation of these pixels to determine the cracking type.

The main function block diagram in Figure 5 depicts the functional blocks used in the algorithm to determine the cracking type. The first phase involves digitizing the image from videotape to create a 512 by 480 greyscale image. The digitized image is then divided into 63 blocks of size 48 by 48 pixels in width and depth. If the image contains a crack, then each block in the cracking section of the image now contains only a small part of the total crack called an edge. The goal of the algorithm is to pick out all the blocks containing edges and the edge orientations in each of these blocks. Division into blocks is necessary since differentiation of the edge from the background can be performed accurately only within small blocks.

Within each of the 48 by 48 blocks, a series of tests are run to determine the likelihood of finding an edge. This analysis within each of the 63 blocks is a series of steps that includes a variance and a mean computation. If these two tests indicate the presence of an edge, a projection histogram analysis is performed within the selected blocks. The presence of an edge is precisely determined by the projection histogram. These tests will be described in more detail in later sections.

Once an edge is discovered, its orientation is determined and the edge map is updated. The edge map is a simple array that holds the orientations of the edges discovered and is used in classification of the image. Besides edges detected as part of a transverse or longitudinal crack, marks on pavements such as oilstains, tire markings, and lane markings are picked up as edges too. Hence, clearing the edge map of extraneous or false edges is important to remove the effects of noise. The edge clear section performs this function by a comparison of the mean of each block to the surrounding blocks. To detect thin edges a second pass of the edge map is used. The second pass picks up thin edges which helps in classifying thin cracks more accurately.

Finally, the image type is determined using the information available in the edge map. A classification of the image into one of the image type categories is made. The classification section uses the edge map as its main input in order to classify the image. Once all images have been classified, the final result is printed out in tabular form showing the total number of cracks detected for each image category.

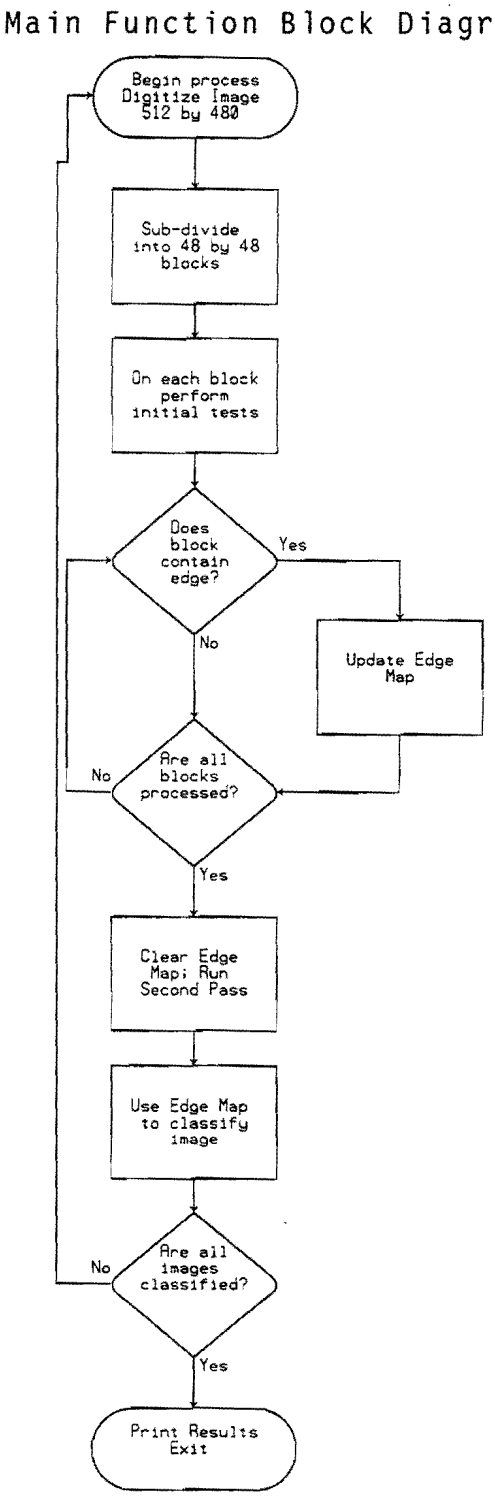

Figure 5. Main Function Block Diagram for Crack Classification.

3.1 ITEX Software

The Imaging Technology Series 151 image processor is a software configurable processor that can run all standard and special image processing routines on its own specialized hardware at speeds exceeding those of regular processors. It achieves this speed by coding all routines in low-level assembly language and executing them on its specialized hardware. These routines are in the form of C language callable library functions which requires a host C program to call these functions to execute. There are two main modes of software operation. In the first mode, the main program runs on the host computer, usually a Personal Computer (PC) or a desktop workstation (SUN). On a function call by the host computer, the execution of the function passes on to the Series 151 image processor which then executes the desired function. Control returns to the host after execution. Besides running all standard image processing functions like convolution and fast fourier transforms, the Series 151 can also run specialized image processing routines like morphological functions. A vast array of image processing functions are available to the host computer with many of these functions running at near real time rates.

In the second mode of operation, a program development system compiles the program in a way to make it possible to download the entire program to the Series 151 for execution on the Image Processor Accelerator. The Image Processor Accelerator (IPA) is a fast and dedicated image processor that is part of the Series 151 system. Entire programs are downloaded to it for execution whereas in the previous case, only the specific functions are downloaded. Downloading entire programs implies that the actual program is executed on the image processor leaving the host computer free to perform other tasks. Since the operational methods for the two cases differ, there are some advantages and disadvantages that need to be considered. Downloading the entire program to the IPA can make some programs run faster because the IPA has smaller overheads. Unfortunately, very few of the available library functions can be called using this mode. Most library functions can only be called from the host computer in the first mode of operation.

The boards that are present in the unit are capable of digitization, frame storage, arithmetic operations, real time convolution, histogram

feature extraction, and executing IPA functions. Some of the digitization functions are grab, snap, and freeze. By invoking grab in a C program it is possible to digitize the video image in real time. Frame operations typically include fb_zoom that allows the image to be zoomed out or in. Also, via the frame operations it is possible to read and write pixels into the frame buffer. The frame buffer provides storage for the image in the form of pixels. The function that can read pixels from the frame buffer is fb_rpixel. ALU functions can perform multiplication, shifting operations, addition, and subtraction of pixel data. The function that can perform multiplication is alu_multmode. The real time convolver can convolve an image in the frame buffer with a preset mask. Histogram extraction can be performed by calling one of the many software routines existing in the function library. The IPA functions allow for downloading of entire programs to it as well as execution of these programs at speeds comparable to that of the host computer without overheads. Overheads are caused by the transfer of pixel data from the image processor to the host computer through the S-BUS. The cracking program runs in the second mode of operation where the entire program is downloaded to the IPA for execution so as to avoid the overhead caused by transfer of pixel data. The unavailability of many library functions is not a cause for concern since only basic image processing functions are used in the program.

3.2 Projection Histogram

The projection histogram is a projection of the mean of the grey level values of an image along an axis. In general, a projection P(x) on the x-axis is defined as

$$P(x) = \frac{1}{N}\sum_{y=1}^{N} f(x,y) \qquad \text{where} \quad 1 \le x \le N$$

N is the block size and f(x,y) is the pixel grey level value of the image.

Figure 6 illustrates the projection histogram process for an idealized transverse cracking image block. It depicts the sequence of events taking place within a 4 by 4 pixel block. The actual algorithm has a 48 by 48 block. The histogram data is first computed by projecting the values of the

Illustration of the Projection Histogram for a Transverse Crack

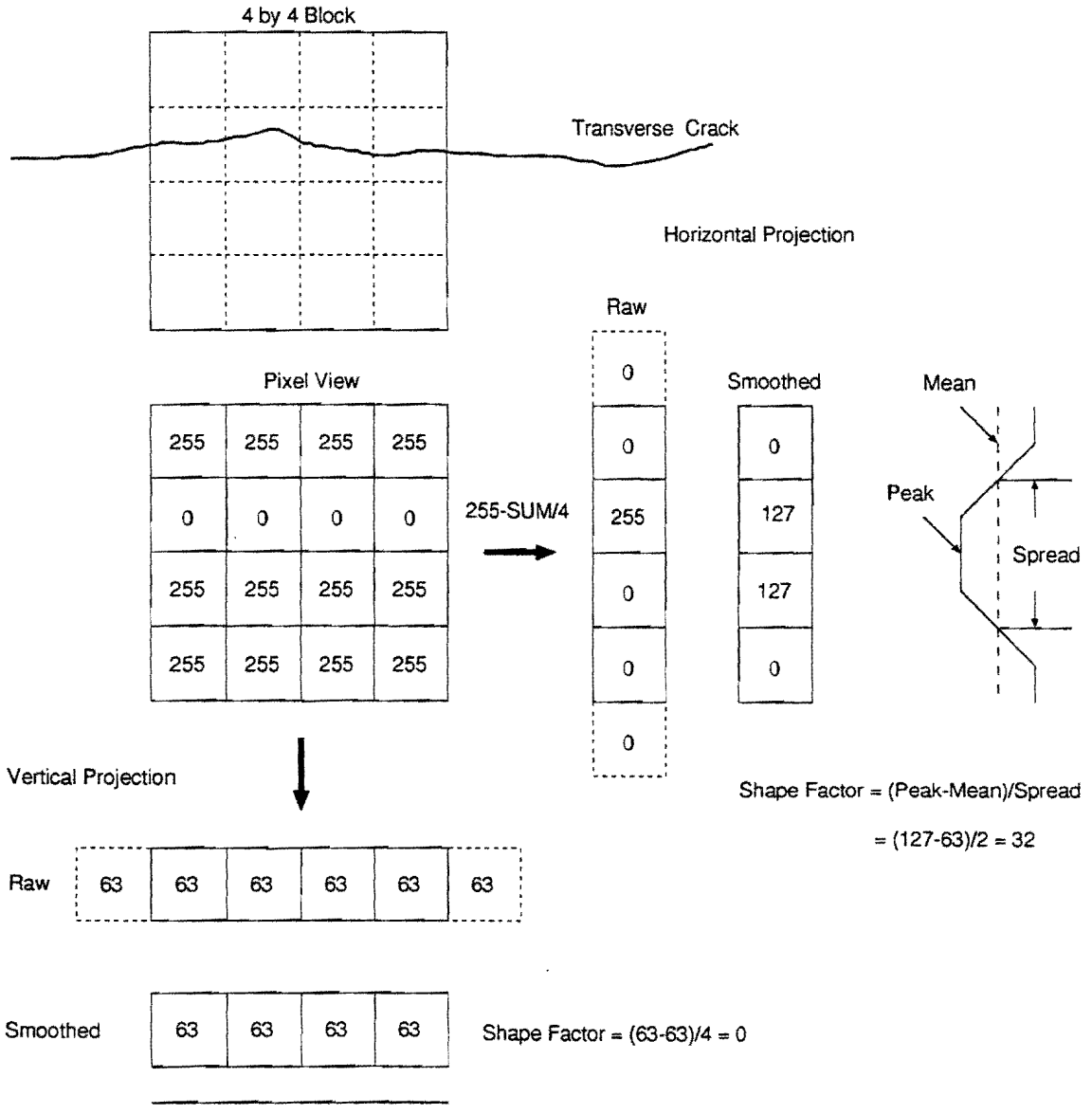

Figure 6. Illustration of the Projection Histogram for a Transverse Crack

pixel greylevels along the horizontal and vertical axis. In Figure 6, the vertical projection results in values of 63 in the vertical projection array because the values in each column are summed up and divided by the number of rows. Hence 255+0+255+255 sums up to 765 which when divided by 4 results in 191. This is reversed to obtain 63 (255-191). Reversal is performed so that edges which have a low greylevel appear as peaks in the projected array. Similarly, the horizontal projection results in the horizontal projection array. The values at either end of the projected arrays are mirrored so as to facilitate smoothing. In the horizontal projection array, the values at either end are 0 which are mirrored or duplicated on the end of the array. Smoothing in this case is performed with a 2 element running average, in the actual algorithm a 7 element running average is performed. The first value and the mirrored value are the first inputs to the smoothing routine. In the horizontal projection case the average of these two inputs is (0+0)/2=0. The second pair of inputs are 0 and 255. The average of these two inputs is 127 and this is reflected in the data in the smoothed array. Finally when the data in the two arrays are plotted, it is noticed that a peak exists for the horizontal projection whereas no peak exists for the vertical projection. The existence of a peak in the horizontal projection array signifies that a transverse edge is present in that block.

The next step is to compute the shape factor for the two projections to quantitatively decide whether an edge is present.
The shape factor is computed as

$$\text{Shape factor} = \frac{\text{peak - mean}}{\text{spread}}$$

Peak is the greylevel value of the largest peak. For the horizontal projection case it is 127. Mean is the average of the grey level values which is 63. Spread is the width from one point of the projected curve intersecting the mean to the other. It is 2 for the example. Therefore the shape factor = (127-63)/2=32. For the vertical case the shape factor is 0 because the value of the peak and the mean are the same. The large shape factor obtained for the horizontal projection is quantitative proof that an edge is present. The position of the peak on the projected axis denotes the

position of the edge element. In the example, the peak position is at horizontal array location 2. In the actual algorithm, the shape factor obtained is compared with a shape factor threshold to decide if an edge is present in the block. If the vertical shape factor dominates, the orientation of the edge is 90 else if the horizontal shape factor dominates, the orientation is 0.

Oilstains, dark spots, and shadows on the pavement create a noisy and dirty image that can contribute to the presence of a peak in the Projection Histogram curve. These edges detected are termed "false edges". Usually noise cause the peaks to be sharp and small. To avoid misinterpreting peaks generated by noise as real edges, the extracted curve is smoothed. The curve smoothing is a running average of the previous three values and the next three values of the extracted curve. Curve smoothing eliminates most of the random peaks generated by noise.

Figure 8 shows the horizontal projection for the block framed by the white square for a real pavement image shown in Figure 7. Figure 9 shows the vertical projection histogram for the same image in Figure 7. The dotted curve depicts the unsmoothed or raw greylevel projection while the dark curve is the greylevel after smoothing. Smoothing reduces sharp edges and pronounces the peak better. A large peak is evident in the horizontal projection seeming to indicate the presence of a transverse edge. No peak can be seen for the vertical projection. To verify these results quantitatively, the shape factor is computed for each case.

For the horizontal projection in Figure 8, the peak for the projection has a value of 131. The mean of the block is 109. Hence the spread points are at positions 30 and 18 on the x-axis. This leads to a spread of 12. The shape factor computes to (131-109)/(30-18)=1.83.

For the vertical projection in Figure 9, the peak has a value 114. The mean is 109 for the block. It can be seen that the means for the vertical and horizontal projections are the same because the same set of data in the block is being considered. The spread points are at positions 48 and 35. Hence, the shape factor computes to (114-109)/(48-35)=0.38.

To make a quantitative decision based on the data available from the shape factors for the two projections, the shape factors are compared to a shape factor threshold chosen by empirical analysis. Since the shape factor

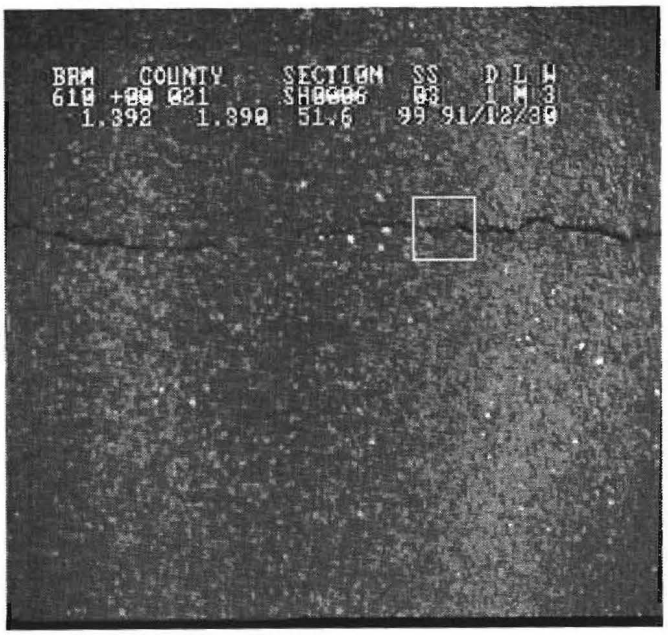

Figure 7. Image with Block on Transverse Edge.

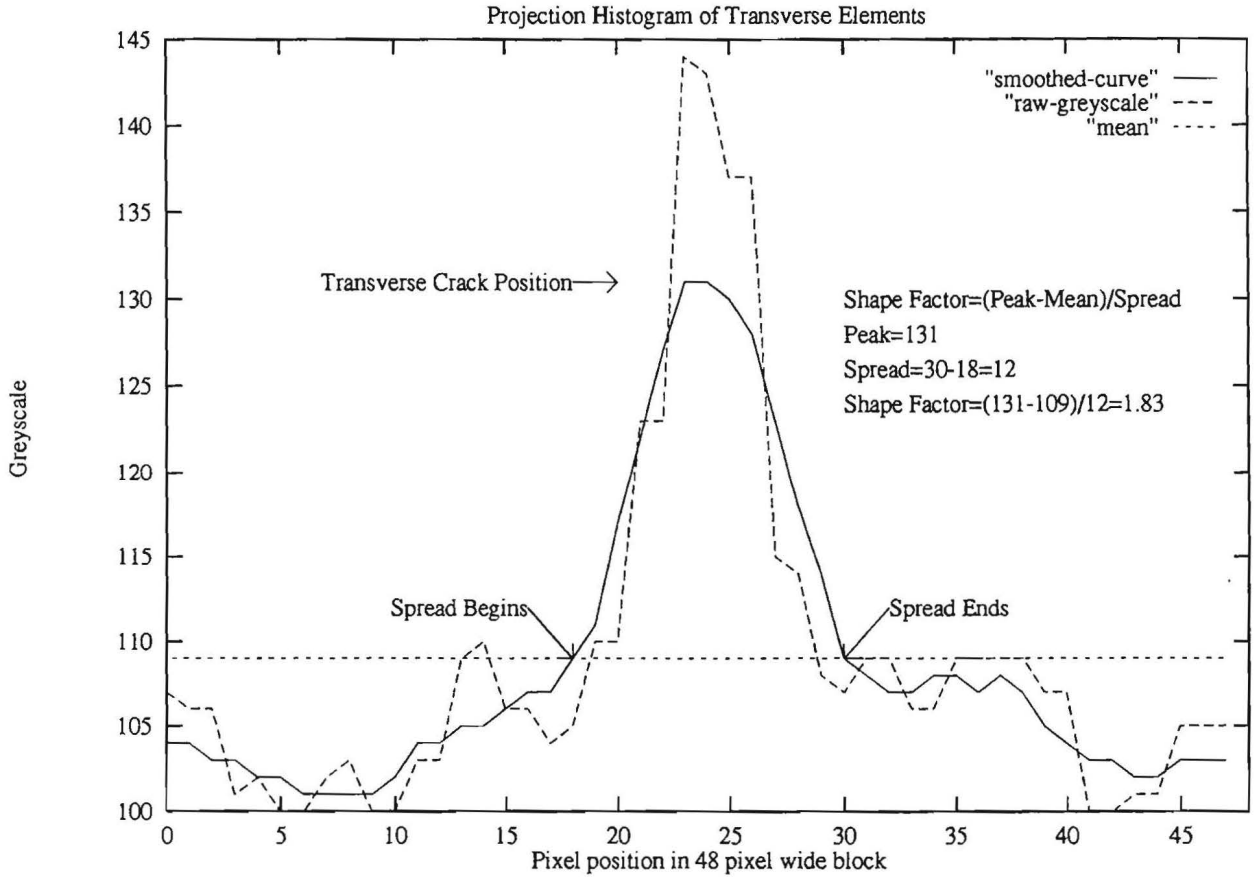

Figure 8. Horizontal Projection Histogram: Shape Factor = 1.83.

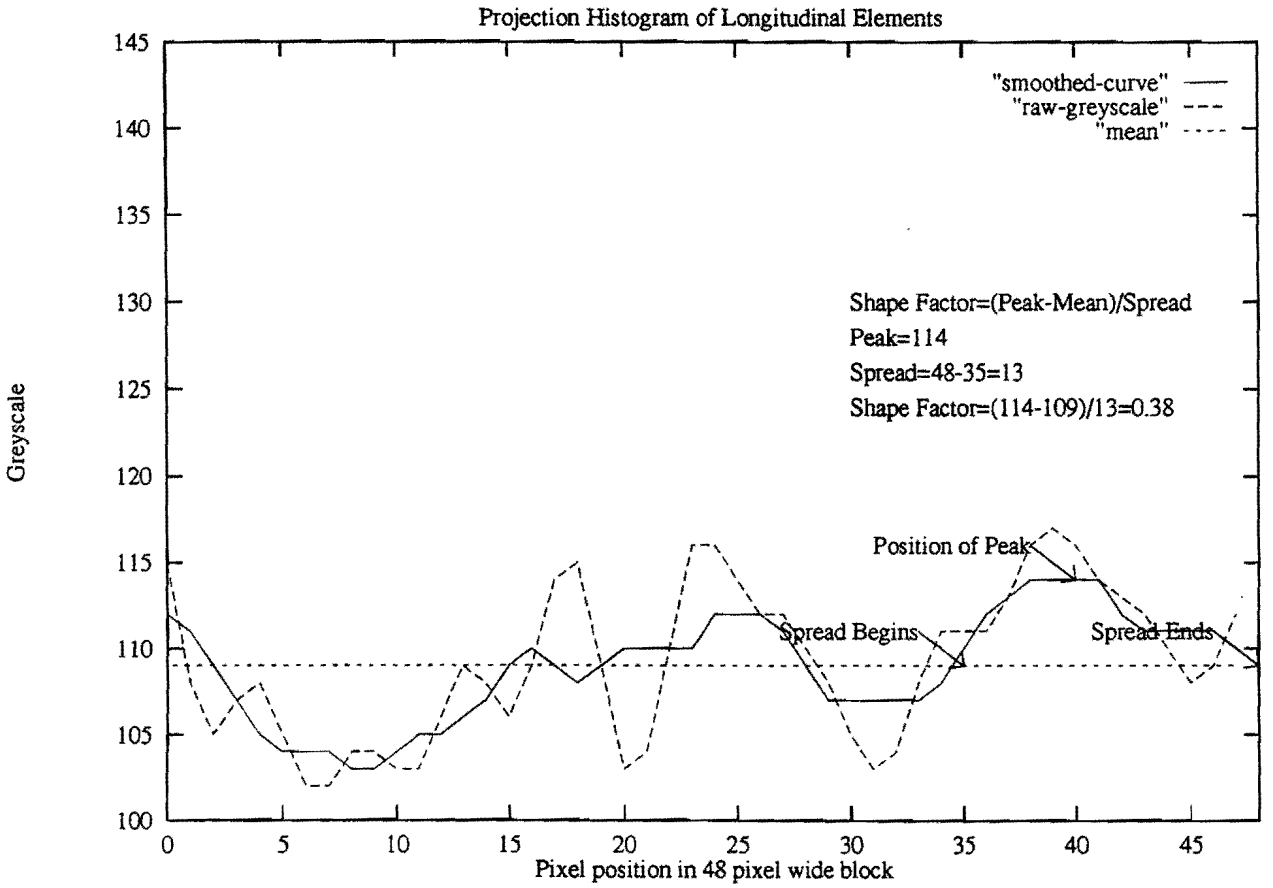

Figure 9. Vertical Projection Histogram: Shape Factor = 0.38.

for the horizontal case is much larger than the threshold and is also larger than the shape factor obtained for the vertical case, it is concluded that a transverse edge exists within the block whose orientation is "0".

3.3 Block size

The optimum block size chosen is 48 by 48 pixels square. Small block sizes like 16 by 16 prove to be more sensitive to noise since noise due to an oil-spot occupies the whole block and is detected as an edge. It also takes longer to scan the whole image. Large blocks 64 by 64 pixels square miss out on thin small cracks since the crack edge is then only a small fraction of the block. Table 1 compares the advantages and disadvantages of different sized blocks.

Table 1. Comparison of Block Size.

Block size	Advantages	Disadvantages
16 by 16	detects fine cracks	noisy
32 by 32	detects most cracks	slow
48 by 48	optimal speed	-
64 by 64	fast	misses fine cracks

3.4 Edge Map

Creation of the edge map is a pre-requisite to classifying the image because the classification algorithm uses the edge map as its main input. The creation of the edge map is a result of the initial processing done on each of the 48 by 48 blocks into which the image was divided. During the processing of each block, a variance, mean, and projection histogram test are carried out. The variance test is the first test that is carried out and differentiates between blocks having likely edges and those having none. If the variance test is successful, a projection histogram is performed to determine the orientation and position of the edge. It is not necessary for the projection histogram to return an edge. In fact many times it does not find a edge in the block. If the projection histogram does return a valid edge, the result is saved in an array called the edge map. Each block in the image is associated with an array location. Hence the results from each block are stored in this edge map. For example, if a transverse crack is input to the algorithm, then there is a strong likelihood that the edge map after processing all blocks will appear as in Figure 10.

1	1	1	1	1
1	1	1	1	1
0	0	0	1	0
1	1	1	0	1
1	1	1	1	1

Figure 10. Edge Map Showing the Presence of a Transverse Crack.

The 5 by 5 array in Figure 10 is the edge map of the complete image with each element of the array denoting the corresponding block in the image. The numeric character "1" in the array denotes that no edge was present in the block while the numeric character "0" denotes the presence of a transverse edge element and the numeric character "90" denotes the presence of a longitudinal element. As can be seen, there is a row of "0" in the edge map. This seems to indicate the presence of a transverse crack in the image. The decision to classify the complete image as a transverse crack is not made until the results of the edge map are input to the classification algorithm which then decides on the cracking type. The edge map is created from the analysis results on each block and is the key to the later classification process.

3.5 Edge Clear

The function edge clear removes noise from the edge map. This noise is usually caused by false edges picked up because of oilstains, shadows from passing vehicles, and shadows from trees by the roadside present in the image.

These foreign image features usually do not cover the whole image, instead they are spread over a small area with the darkest part being in the center of the feature as in the case of an oilstain. The lighter parts are usually around the oil stain and carry the maximum probability of being picked up as a false edge. The darker parts of features can be eliminated from good pavement by the "mean test". This test works in the following manner. If the block has a local mean much higher than the global mean (ie: mean of the whole image), the result from the block can be discarded. However, this test fails on blocks with light oilstains and where a false edge is detected. To eliminate these blocks an edge clear technique is employed. For example, if a part of the edge map appears as in Figure 11.

The numeric character "2" in the edge map denotes a block with a low mean caused by an oilstain or shadow. The numeric character "1" denotes no edge detected in the block while the numeric character "90" is a longitudinal edge detected in the block. The numeric character "90" in the edge map seems to indicate the presence of a longitudinal crack, however, the real reason for the presence of the "90" edge is because of the lighter

1	1	1	1	1
1	1	90	1	1
1	90	2	90	1
1	1	90	1	1
1	1	1	1	1

Figure 11. Edge Map with "False" 90° Edges Caused by Oilstain.

area around the main dark area. The edge clear routine works by clearing all "90" or "0" edges around the "2" in the edge map. All "90" or "0" edges that occur in the 4 directions around the "2" are set to "5". It is set to "5" and not "1" so as to keep track of the number of false edges. This technique ensures that false "90" edges are not used in the classification of the image type, instead they are cleared and set to "5". Hence the edge map appears as in Figure 12.

1	1	1	1	1
1	1	5	1	1
1	5	2	5	1
1	1	5	1	1
1	1	1	1	1

Figure 12. Edge Map with False Edges Removed.

3.6 Second Pass

The second pass is used to detect thin edges or curved edges that were not detected in the first pass. The mean, variance, and projection histogram analysis performed on each block determines the presence of strong edges and their respective orientations. This is called the first pass of the algorithm.

In the first pass, to offset the effects of noise, the comparison thresholds used with the mean, variance, and projection histogram analysis are set high. If these thresholds were set low for the first pass it would result in too many false edges being detected. The net result is that only strong edges present in the image are detected, while weak edges that might

be part of a crack are missed. There is a need for these weak edges to be picked up in cases where the number of strong edges picked up is not enough for classification into a image type but is still indicative of a crack being present.

If there are a sufficient number of strong edges picked up in the first pass, a second pass is performed. The values for the comparative thresholds are reduced and the edge map is scanned again. In the second pass, the edge map is searched with a decreased threshold only in the areas surrounding the locations of the strong edges to restrict the search to areas of the image having a greater probability of having weak edges. The row or column pair having the maximum number of strong edges is rescanned and new edges found in these areas are recorded in the edge map. The classification routine makes a more informed decision with the extra information available in the edge map. Typically the second pass helps in classifying thin transverse and thin longitudinal crackings, or cracks that lie in a direction deviating from the horizontal or the vertical direction.

CHAPTER IV
CLASSIFICATION RULES AND RESULTS

This chapter describes the developed classification rules in automatic distress evaluation. These rules employ the image features discussed in Chapter III to recognize various cracking types in both ACP and CRCP.

4.1 ACP Rules

The ACP crack classification process is divided into three passes. The first pass is called the "Initial Test on Each Block." In this pass, the digitized image is sub-divided into 7 by 9 (Row by Column) blocks of 48 pixels. Each block is tested to determine if there exists a segment of the cracking inside the block. The existence and the orientation of the crack segment data is coded into a 7 by 9 matrix called Edge Map. The second pass called "Edge Map Clear and Second Scan" detects and negates false edges caused by non-cracking objects on road surface such as oil stains. The second pass also re-examines the vicinity of the detected edge segments to locate remaining thinner crack segments with less stringent rules and updates the Edge Map accordingly. The third pass which is called "Final Classification" classifies the cracking type based on the number, the position, and the orientation of the edge segments in the modified Edge Map.

A. Initial Test on Each Block

The Projection Histogram Curve and its corresponding Shape Factor have been discussed in Chapter III in detail. The first test applied to the image block is to compute the variance of the Projection Histogram Curve (see Figure 13 and Table 2). A high variance value indicates that the Projection Histogram Curve is not level and flat, but rather a jagged curve, possibly with a peak or valley.

This computed variance of the Projection Histogram is compared with a threshold value. The determination of the threshold is empirical and based on the results computed from a number of image blocks where each one has a crack segment slightly different from the other in crack width, intensity contrast between crack and background, and shape of crack.

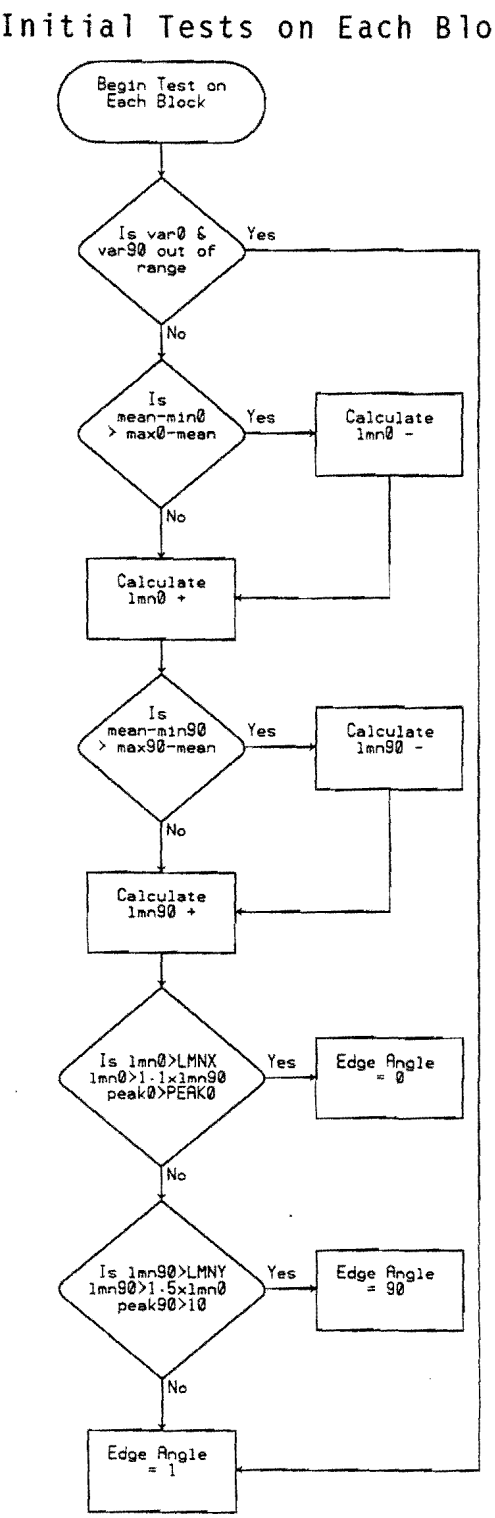

Figure 13. Flowchart for Initial Tests on Each Block.

Table 2. Notation Table for Classification Rules for ACP and CRCP.

SYMBOL	COMMENTS
ab_lo	number of blocks that are considered as oilstains
ab_hi	number of blocks that are considered as lane markings
t0	max number of horizontal edge in a pair of rows
t90	max number of vertical edge in a pair of rows
W0	Total number of horizontal edge in the image
W90	Total number of vertical edge in the image
sd0	Standard deviation of the position of the horizontal edges in t0
sd90	Standard deviation of the position of the vertical edges in t90
d-cnt	number of blocks that have both high horizontal and vertical projection variance
sd var	variance of the projection variance
T	Threshold value
K	Constant

Those image blocks that have Projection Histogram variances greater than the threshold are further analyzed. This analysis is called Shape Factor. A Shape Factor (lmn) is a measure of how "sharp" is the peak of the Projection Histogram. Most cracks that have been observed on the survey tape are darker than the background pavement. However, a light color crack which was filled with fine materials from the base course is also observed. A dark crack has a peak in the Projection Histogram Curve but a light color crack has a valley instead. Before the Shape Factor is calculated, the size of the peak is compared with the size of the valley in the Projection Histogram Curve to determine if the crack segment is darker or lighter than the background pavement.

The determination of a horizontal crack segment is based on the following three criteria computed from the Projection Histogram Curve.

1. Horizontal Shape Factor > Threshold
2. Horizontal Shape Factor > Vertical Shape Factor
3. Peak Size > Threshold

The first rule is utilized to ensure that the Projection Histogram Curve in the horizontal direction has a large Shape Factor. In other words, the quotient of the peak size divided by the peak base (spread) is large. The second rule was resulted when a single black spot inside an image block gave both a large horizontal Shape factor and a large Vertical Shape factor. The second rule is designed to avoid classifying the spot as a crack. The third rule emphasizes the amplitude of the peak because a large Shape Factor value could be caused by a small peak divided by a small peak base (spread). A small peak having a small peak base usually is associated with a spot or random noise in the image block.

The numeric character "0" will be assigned to the image block to denote the presence of a horizontal crack segment if the horizontal Projection Histogram Curve satisfies the above three criteria (rules). A similar set of rules are set up for vertical edge detection. A numeric character "90" will be assigned to those image blocks which have properties indicative of a vertical edge. All other image blocks will be assigned a numeric character of "1" for no crack found.

B. Edge Map Clear and Second Scan

At this point, the algorithm passes control to the Edge Map Clear subroutine as shown in Figure 14. The subroutine starts by defining image blocks that are much darker such as oilstains and shadows, and blocks that are much lighter, such as lane markings. From the study of a large number of different images, the TTI research team empirically determined that the threshold value for the difference between a global (whole) image mean and a local (block) image mean to be 20.

A numeric character "2" is assigned to an image block with mean greylevel larger than the global mean by 20. A numeric character "3" is assigned to an image block that is smaller than the global mean. As it was

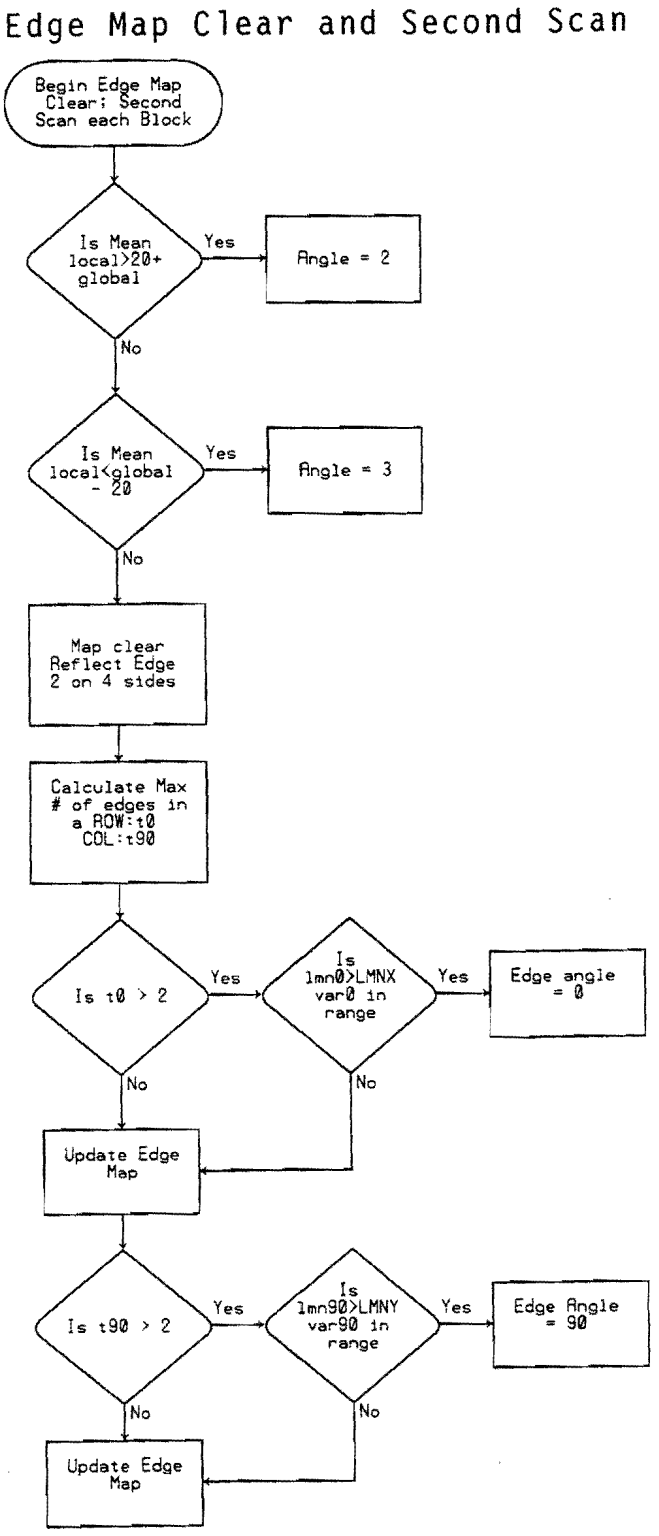

Figure 14. Flowchart for Edge Map Clear and Second Scan.

discussed in Chapter III, the pavement images are presented in reverse scale where a dark object has a high greylevel and a light object has a low greylevel. In other words, a numeric character "2" is assigned to dark image blocks while "3" is assigned to light image blocks.

Frequently, the images blocks that overlap with the oilstain would result in a Projection Histogram Curve similar to the one projected from a real crack segment. This Projection Histogram Curve resulting from partial overlapping with an oilstain, may even satisfy the three requirements to be considered as a crack segment. Therefore, the Edge Map Clear subroutine is used to check all the neighboring image blocks after the "2" dark image blocks are located. Any "false" edges that are detected in those four neighboring blocks will be removed. A numeric character "5" will replace the previous "0" or "90" for the false edge detected in these neighboring image blocks and the number of the "5" image blocks are tabulated to be used later in the classification process.

The Second Scan starts by computing the maximum number of horizontal edges within two neighboring rows of image blocks as shown in the lower half of the flowchart in Figure 14. If the maximum number of horizontal edges is greater than 2, the image blocks in the two rows will be rescanned. But the conditions needed to be met for the determination of a horizontal edge is less stringent than the First Pass. The Shape Factor threshold is reduced and the acceptable range for the variance of the Projection histogram Curve is widened. This Second Scan or Second Pass will pick up crack segments that are thin and curve shapes or crack segments that have a weak greylevel contrast with the background pavement. Similar procedures will be followed to pick up the thin vertical edges missed in the First Pass (Initial Test for Image Blocks).

C. Final Classification

Before the actual classification of various cracking types in ACP, a number of parameters are first calculated from the information in the edge map. Referring to Figure 15 and the notation explanation in Table 5, W0 and W90 are the total number of horizontal and vertical edges in the image respectively. t0 is the maximum number of horizontal edges in a pair of rows and t90 is the maximum number of vertical edges in a pair of columns. sd0 is the standard deviation of the location of the horizontal edges detected in the pair of rows with maximum number of edges. sd90 is the

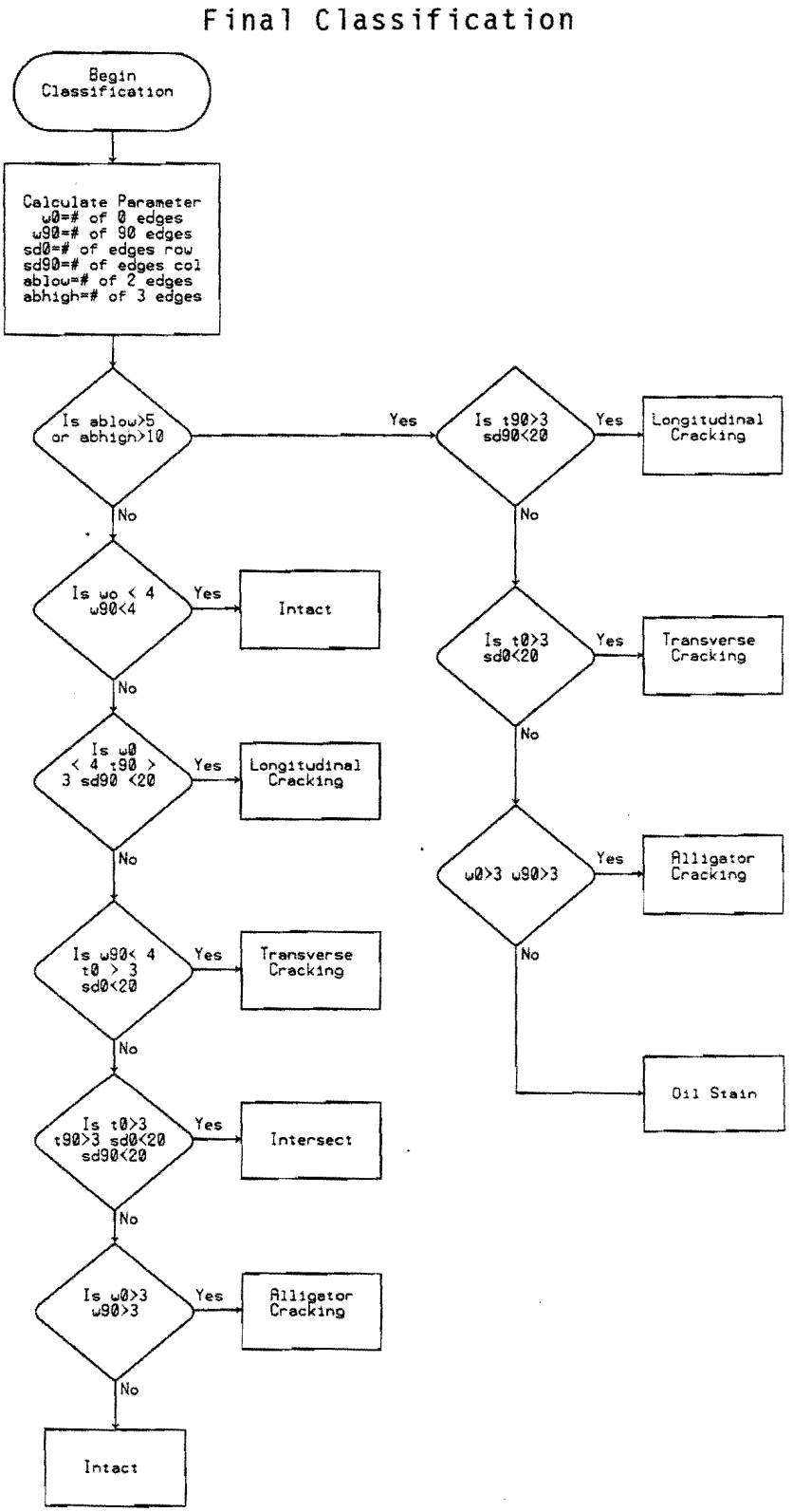

Figure 15. Flowchart for Final Classification of ACP.

standard deviation of the location of the vertical edges detected in the pair of columns with the maximum number of vertical edges. A small standard deviation value indicates that the crack segments closely line up either in a horizontal line or vertical line. This standard deviation measure is used to sort out false edges caused by random noise, because these false edges do not align. Lastly, ab_lo and ab_hi keeps track of the number of "dark" and "light" image blocks, respectively.

The counts of the dark and light image blocks (ab_lo and ab_hi) are used to divide the pavement images into two groups. The "Stained" images with non-cracking objects, such as oilstains, shadows and lane markings will have a high ablow or a high abhigh value. This group of "stained" images will follow the steps outlined in the right branch while the "clean" images will follow the steps outlined in the left branch of the flowchart.

The first step in the left branch is to isolate images without any form of cracking since the majority of the pavement images are intact. If the intact images are detected in the early stages of the classification process, the overall processing time will be shortened due to the fact that the following stages of the analysis algorithm are avoided. The next four stages involve the detection of four primary types of cracks, namely, Longitudinal Cracking, Transverse Cracking, Block (Intersect) Cracking, and Alligator Cracking. The rules for recognition of these cracks are designed based on the characteristic shape of each crack type. For example, in order to classify Longitudinal Cracking, three criteria must be met.

1) Total # of horizontal edge < 4
2) Total # of vertical edge in the maximum column pair > 3
3) The standard deviation of the vertical edges locations < 20

The criterion #1 is aimed at separating out both Block Cracking and Alligator Cracking because these two crack types have both horizontal and vertical edge segments.

Criterion #2 is based on the total count of the vertical edge segments within the pair of columns which has the maximum number of vertical edge segments. Criterion #3 is to check the alignment between this set of vertical edge segments. It is to see if all individual vertical edge segments are part of a meandering vertical crack.

An equivalent set of criteria are set up for the detection of Transverse Cracks. For the recognition of Block Cracking, it is assumed that a horizontal crack intersects with a vertical crack inside the image. The set of criteria used are to locate a horizontal line and a vertical line existing together. However, when there are no obvious alignments of these edge segments, then this cracking image is classified as the Alligator Cracking type.

The classification of the cracking types in "stained" images are shown on the right branch of the flowchart, for example, the rules for restricting the number of horizontal edge segments in the determination of Longitudinal Cracking is removed. The reason is that for the "stained" images there are a large number of edge segments in both directions, due to the presence of non-cracking objects on the road surface such as oilstains and skid marks.

4.2 ACP Results

To evaluate the video data and processing methodology, many miles of pavement were collected and processed. The results shown below are from SH-6 near Navasota, 12 miles of one wheel path videolog (reference marker 610 to 598 in the North bound direction). The videolog was collected with a special remote camera mounted at 7 feet high on a camera mount attached to the right side of the bumper. The videolog was collected on a sunny day during noon time where the lighting condition were almost ideal and there is no shadow projected from the videolog vehicle or from the trees on the roadside. The coverage of the camera is five feet wide and the camera shutter speed is 1/4000 sec. The videolog vehicle was travelling at 50 mph during the survey. The first half-mile section and the last half-mile section (section 19 to section 24) are sealcoat surfaces. Typical example images of Transverse Cracks and Longitudinal Cracks are illustrated in Figure 16 to 19.

It can be seen that most of the edges in the image are picked up by the algorithm. For the case of the 12 mile section, 900 frames were processed for each half mile section. Each frame covers an area 5 feet wide x 4 feet long. The actual processed area is about five feet wide and only three feet long because the top one foot is covered by information header. This processing is a 100% sampling (900 frames x 3 ft/frame = 2700 ft) and the image frames are contiguous and slightly overlapped. The approximate processing time for each half mile section is twelve minutes, which is

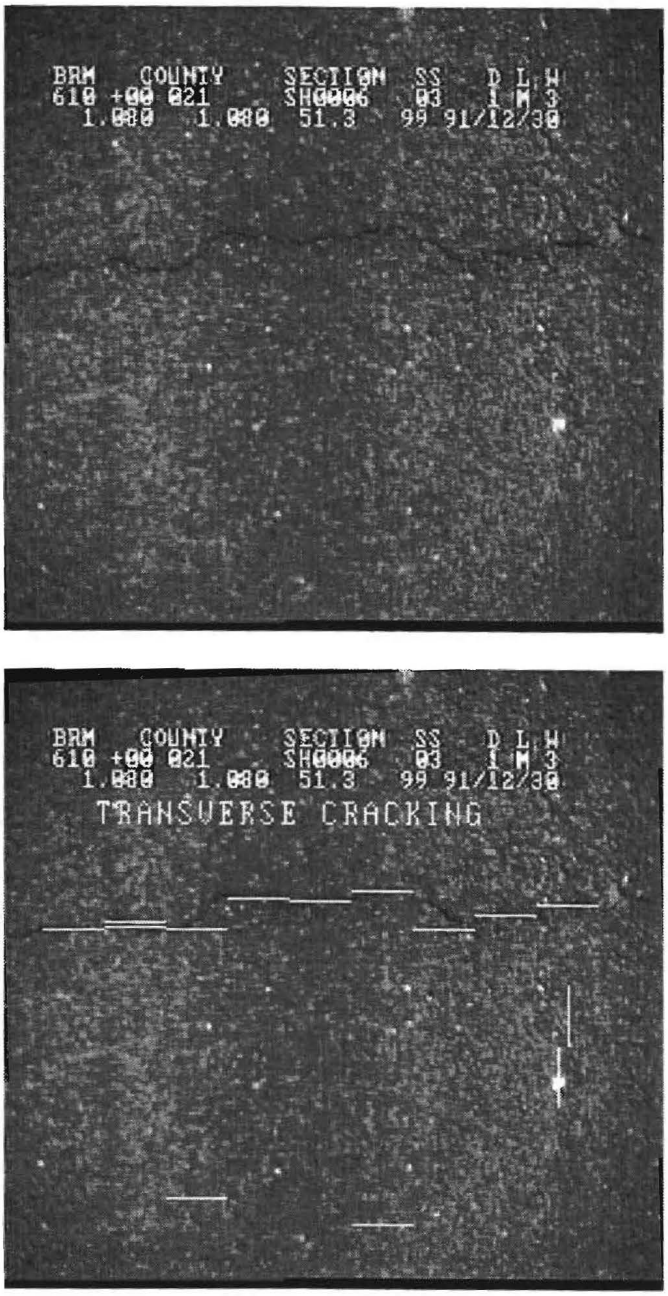

Figure 16. Transverse Cracking on HMAC Surface.

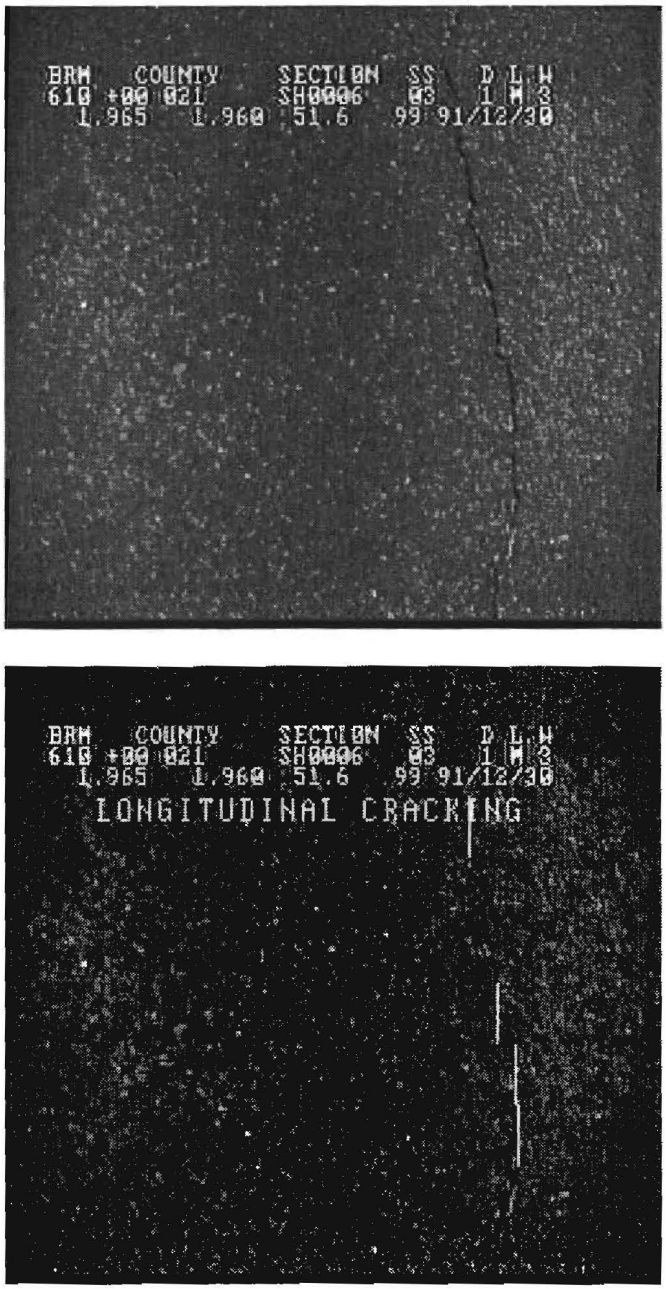

Figure 17. Longitudinal Cracking on HMAC Surface.

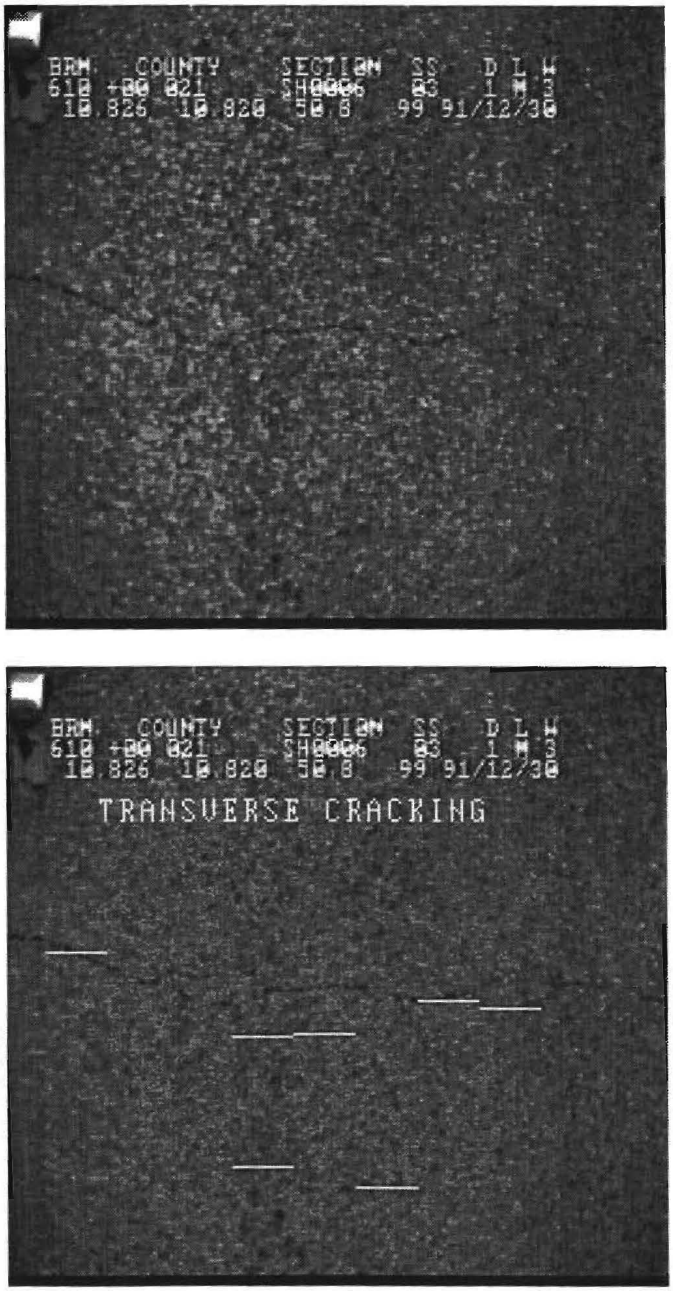

Figure 18. Transverse Cracking on Seal Coat Surface.

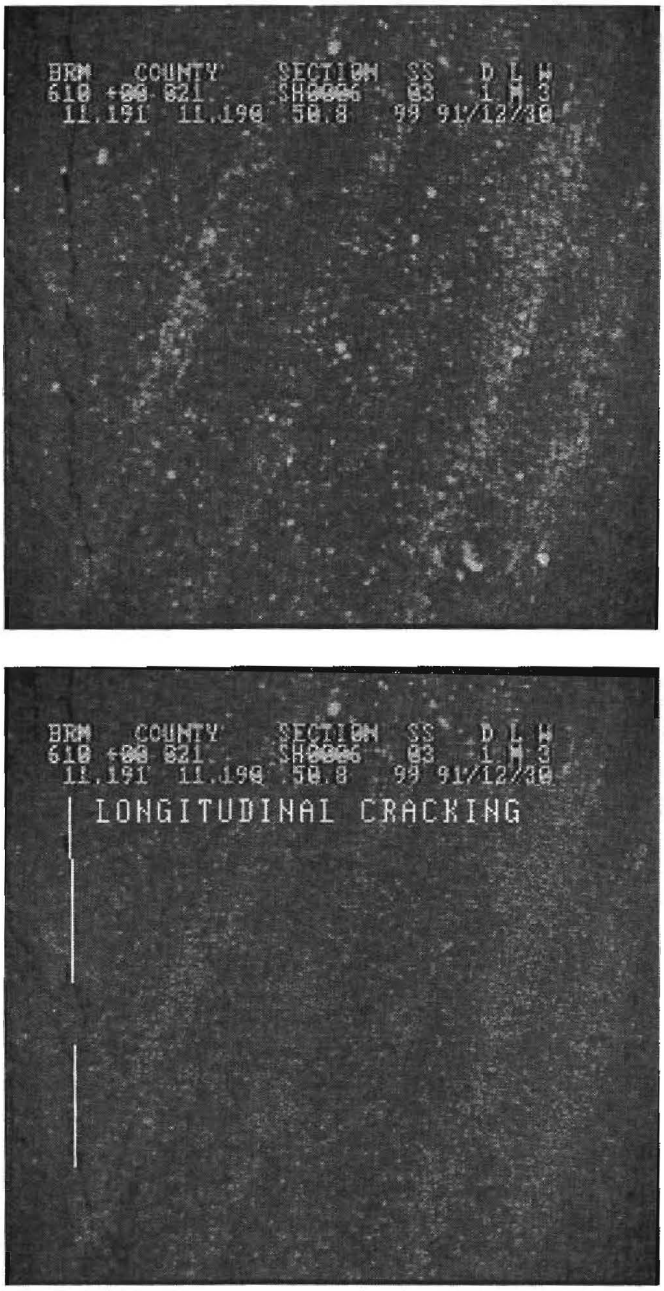

Figure 19. Longitudinal Cracking on Seal Coat Surface.

equivalent to 0.8 second per frame. Results were then tabulated every 100 frames and the program is terminated at 900 frames to report the results.

Table 3 lists the computer output of the crack analysis program for section #7. Four categories of cracking and an intact category are listed. Two or more categories which are oilstain and lane mark are omitted in the print out but were actually tabulated as well. The length of longitudinal cracking is equal to the number of frames multiplied by three feet. An intersecting crack depicts a horizontal crack intersecting with a vertical crack.

Tables 4 and 5 summarize the results of Transverse Cracking and Longitudinal Cracking for the 24 sections. Each half-mile long section is equivalent to 900 video frames. The computer processing results are listed in the "Raw Data - Computed" column. As the tape is being rated by the computer, a visual counting of distress was taken. These visual counts are entered in the "Raw Data - Actual" column. Both the visual counts from the tape and the computed counts are converted into PMIS ratings for comparison. The ratings used are the new multiple category system used in the TxDOT which is more accurate than the old category system.

The classification of transverse cracks is very stable. The "Computed" count for all sections agree well with the "Actual" count under PMIS rating. (Figure 20 and Figure 21). Figure 21 shows the TxDOT PMIS rating using the raw data from Figure 20. The computed rating categories for these sections are 0, 1, 2, and 4 cracks per station. The high accuracy of identifying transverse cracks is due to the fact that there are few non-distress items lying in the horizontal direction. When the calculated raw data was examined, it was found that more cracks were detected in some sections while fewer were detected in other sections. The sections that have higher computed counts than the actual counts are due to the double counting of the transverse cracks. The current setting of the VCR speed results in double counting if the crack appears at the bottom of a frame and then appears again at the top of the frame. A little overlap is desired so as to pick up most cracks. The results show that the algorithm adapts equally well for both HMAC and sealcoat surfaces.

For longitudinal cracks, (Table 4, Figure 22 and Figure 23) there are more non-distress items lying in vertical directions. A small number of

Table 3. Computer Program Output for Section 7
(Reference marker 607 + 0.0 to 606 + 0.5)

```
Enter start DMI for 1/2 mile section :3.0

Processing 1/2 mile section from  3.0 to  3.5


frames : intact, long, trans, alli, intersect
  100      91     7     2      0       0

frames : intact, long, trans, alli, intersect
  200     185     8     7      0       0

frames : intact, long, trans, alli, intersect
  300     254    34    12      0       0

frames : intact, long, trans, alli, intersect
  400     330    52    16      0       1

frames : intact, long, trans, alli, intersect
  500     424    53    21      0       1

frames : intact, long, trans, alli, intersect
  600     512    57    29      0       1

frames : intact, long, trans, alli, intersect
  700     594    59    45      0       1

frames : intact, long, trans, alli, intersect
  800     667    73    57      0       2

frames : intact, long, trans, alli, intersect
  900     751    81    65      0       2
```

Table 4. Summarized Transverse Cracking Data for 24 Half-Mile Section (Reference Marker 610 + 0.0 to 598 + 0.0).

Section #	Raw Data (# of cracks per 0.5 mile)		PMIS Rating (# per *station)	
	Actual	Computed	Actual	Computed
1	5	3	0	0
2	31	21	1	1
3	101	91	3	3
4	56	46	2	1
5	110	405	4	4
6	61	59	2	2
7	91	65	3	2
8	104	83	3	3
9	85	77	3	2
10	58	71	2	2
11	78	74	2	2
12	88	101	3	3
13	69	75	2	2
14	80	77	3	2
15	73	56	2	2
16	50	58	1	2
17	49	56	1	2
18	56	71	2	2
19	9	8	0	0
20	3	6	0	0
21	5	1	0	0
22	8	18	0	0
23	3	22	0	0
24	1	8	0	0

* station = 100 ft.

Table 5. Summarized Longitudinal Cracking Data for 24 Half-Mile Section (Reference Marker 610 + 0.0 to 598 + 0.0).

Section #	Raw Data (feet per 0.5 mile)		PMIS Rating (feet per *station)	
	Actual	Computed	Actual	Computed
1	75	153	2	4
2	90	150	2	4
3	60	126	2	4
4	48	210	0	6
5	60	99	2	2
6	54	108	2	4
7	126	243	4	8
8	18	30	0	0
9	60	24	2	0
10	45	75	0	2
11	15	96	0	2
12	180	111	6	4
13	39	66	0	2
14	9	21	0	0
15	60	72	2	2
16	360	222	12	8
17	246	204	8	6
18	45	102	0	2
19	0	33	0	0
20	3	408	0	14
21	0	273	0	10
22	0	306	0	10
23	30	105	0	2
24	18	78	0	2

* 1 station = 100 ft.

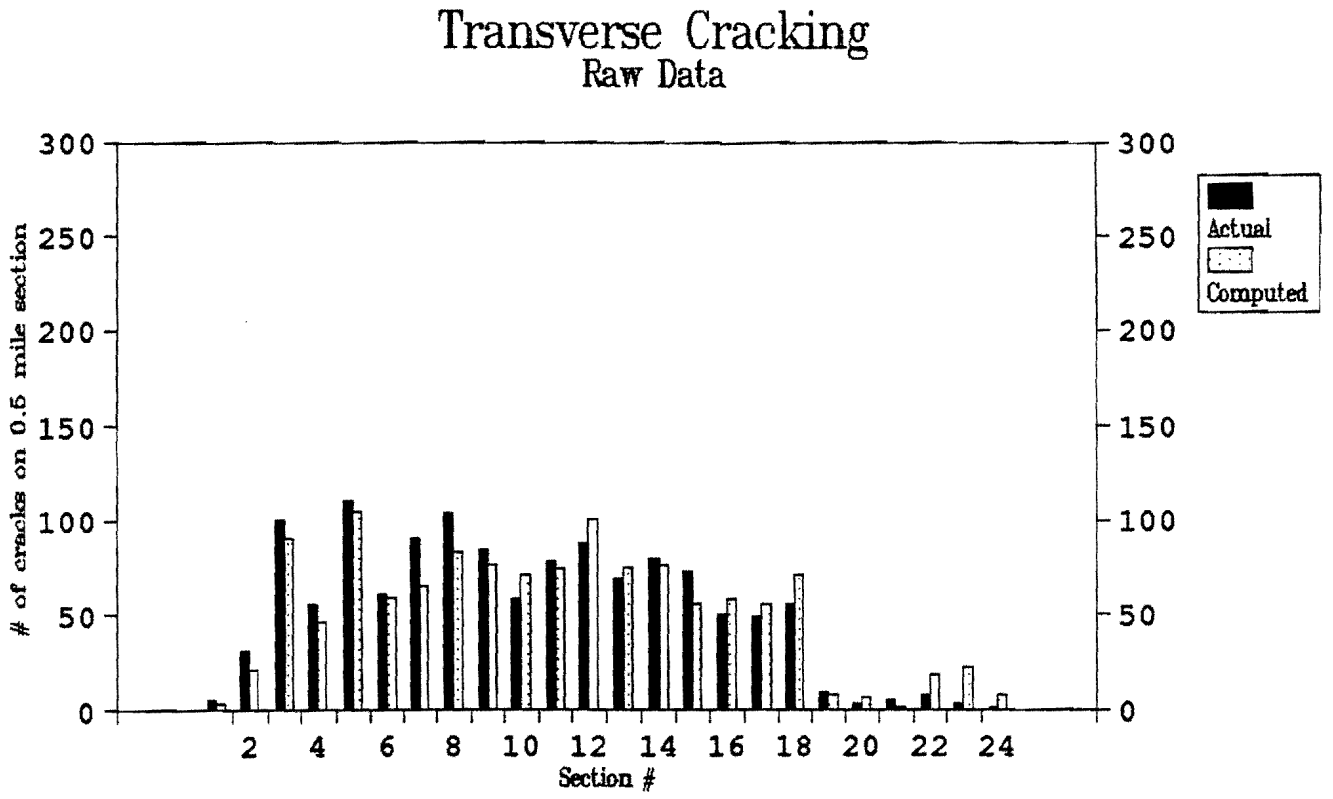

Figure 20. Number of Raw Transverse Cracking Count from Visual and Computer Evaluation.

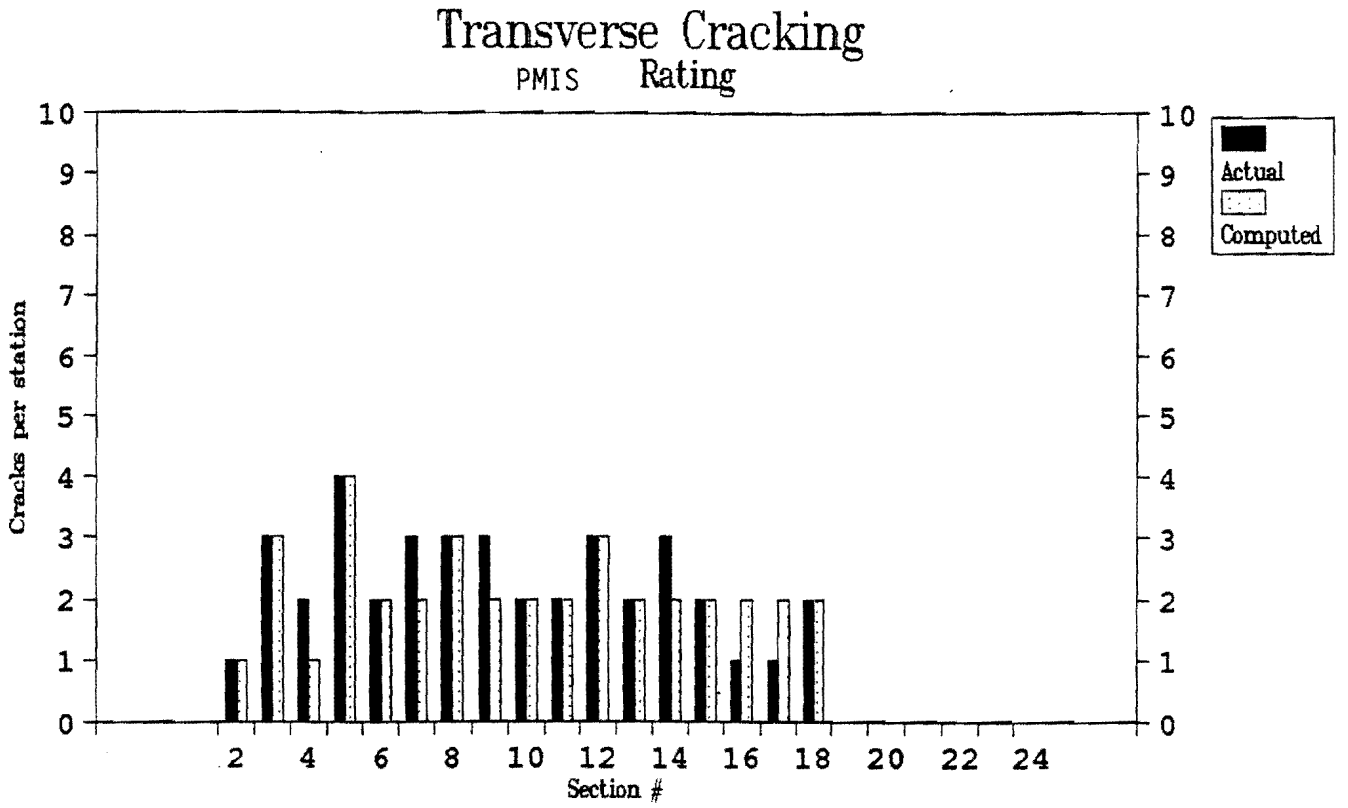

Figure 21. TXDOT PMIS Rating of Transverse Cracking.

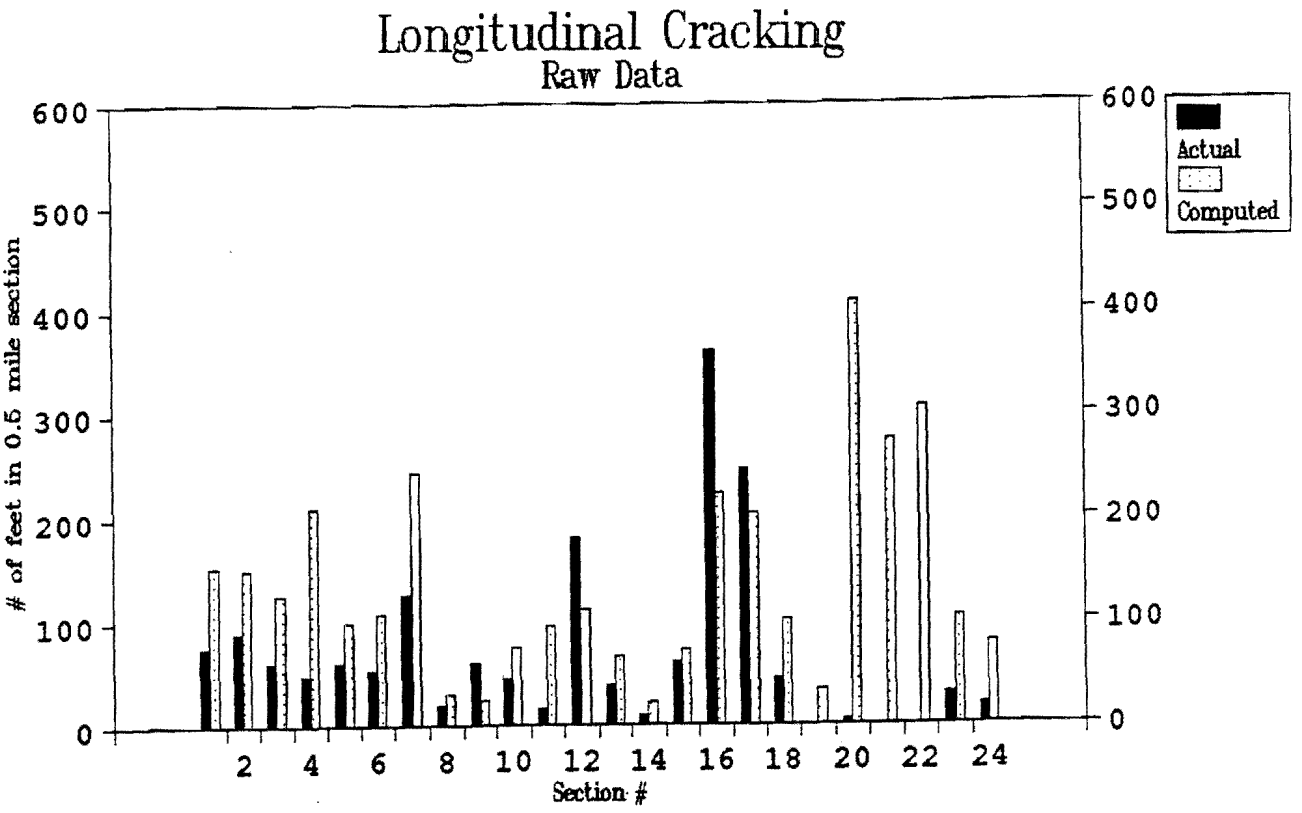

Figure 22. Raw Data for Longitudinal Cracking.

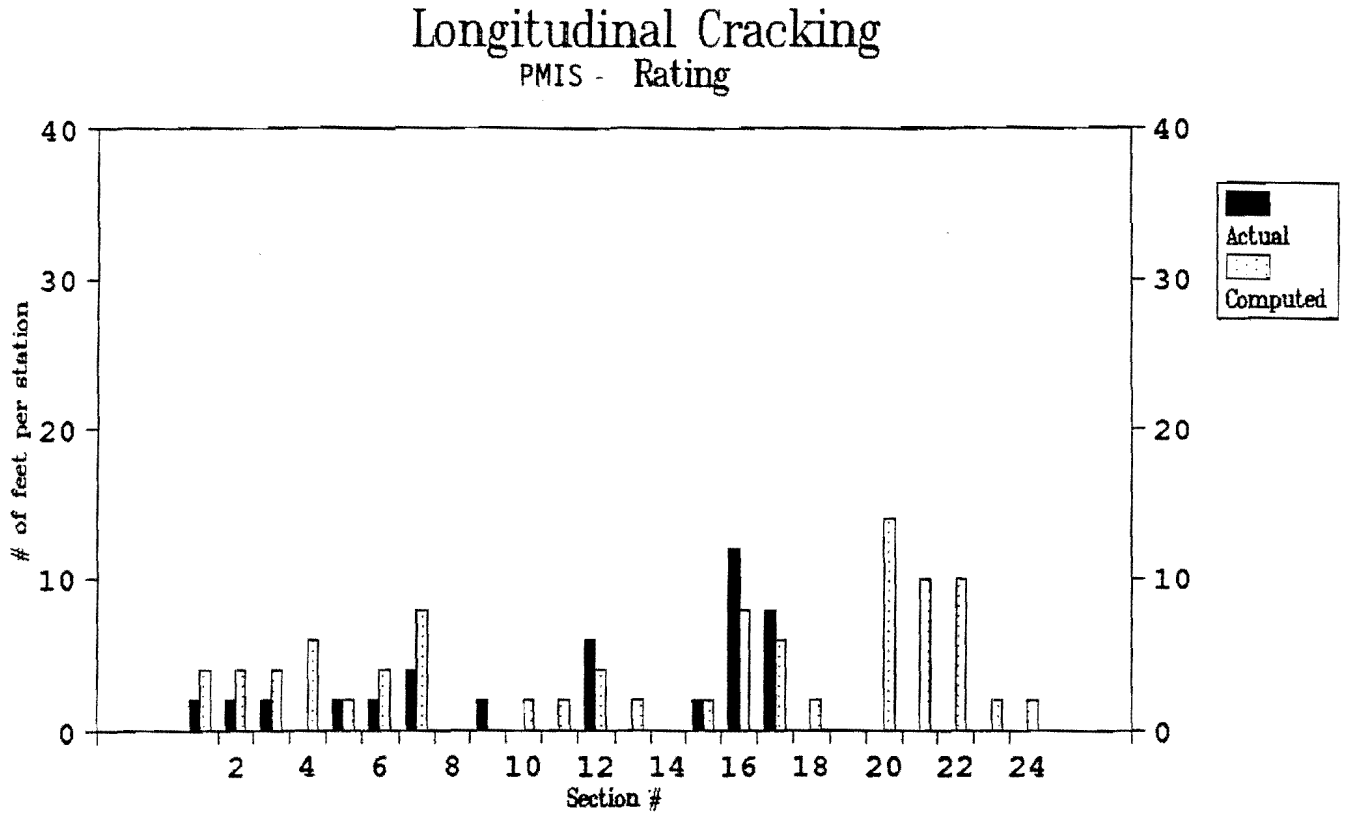

Figure 23. PMIS Rating for Longitudinal Cracking.

dark skid marks (Figure 24) oilstains, exposed aggregate, and ravelling are misclassified as cracks are observed since the algorithm has provisions for these non-distress items. As far as the PMIS ratings, most of the sections agree with the visual evaluation of the tape. But sections 20, 21, and 22 exhibit much higher computer counts when compared with the visual counts. This discrepancy is caused by the sealcoat treatment which covered only part of the original hot-mix surface as shown in Figure 25, hence a false edge is detected at the overlapped area.

In Table 3.4 it can be seen that under section 3 there were 101 actual transverse cracks that were visually rated. The computer picked up 91 cracks resulting in a classification % of 90. This is remarkably high and in fact can get higher given a VCR that can be controlled reliably in slow motion. A number of cracks were missed because the digitizer did not grab the frame properly or the VCR did not display the crack properly. A plot of these sections is also available. The PMIS rating for the actual number of cracks in section 3 is 3; the computed number is also 3.

4.3 CRCP Rules

The automatic pavement evaluation system concentrates on the analysis of surface cracks because they occur more frequently than non-cracking distresses. The two cracking types of CRCP currently required by PMIS are spalled cracks and average crack spacing. The average crack spacing is defined as the survey length divided by total number of transverse cracks and spalled cracks.

Therefore, the objective of the CRCP classification rules is to determine cracks that lie perpendicular to the pavement centerline. As shown in Figure 26 and Table 3.5, the flowchart has two main branches. The branch on the left shows steps for the analysis of relatively clean images, while the right branch shows the decision steps for the "stained" images. A "stained" image contains any combinations of the following contaminants: oilstains, skid marks, and lane markings.

The first step on the right branch is to decide if a longitudinal crack exists in the image. This feature is added because a number of longitudinal cracks on the CRCP surface video are observed. The decision of the presence of a longitudinal cracking is based on the following three conditions. The

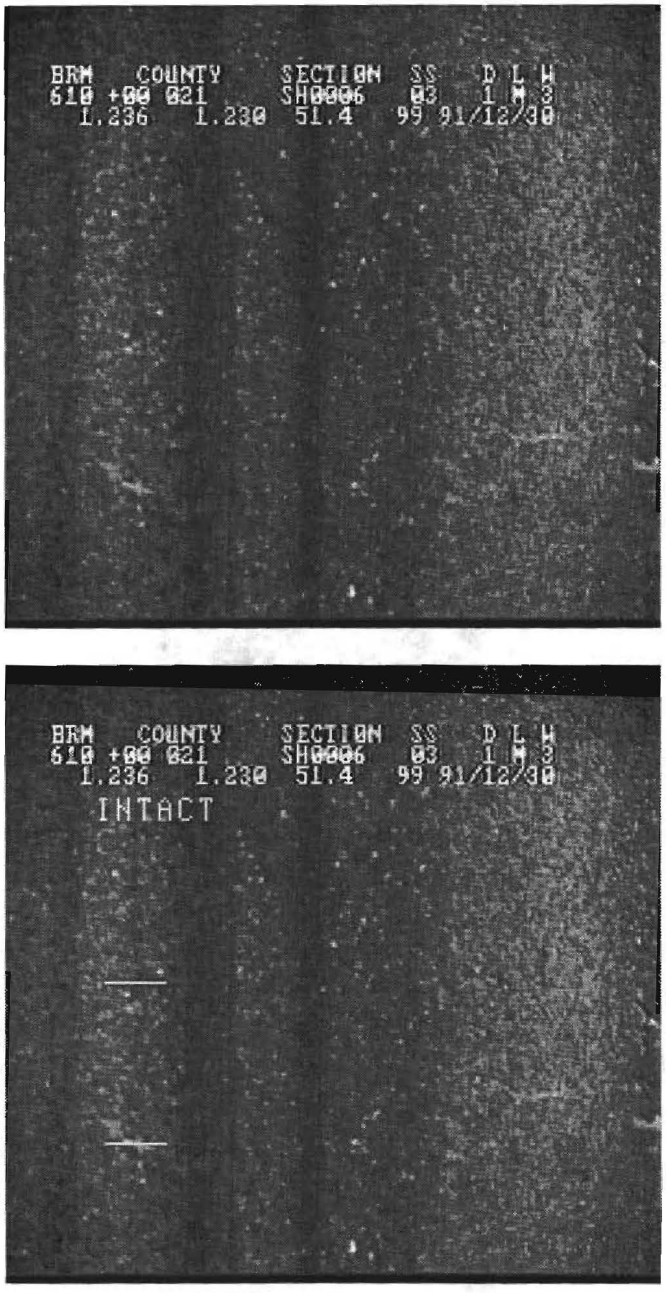

Figure 24. Skid Mark on HMAC Surface.

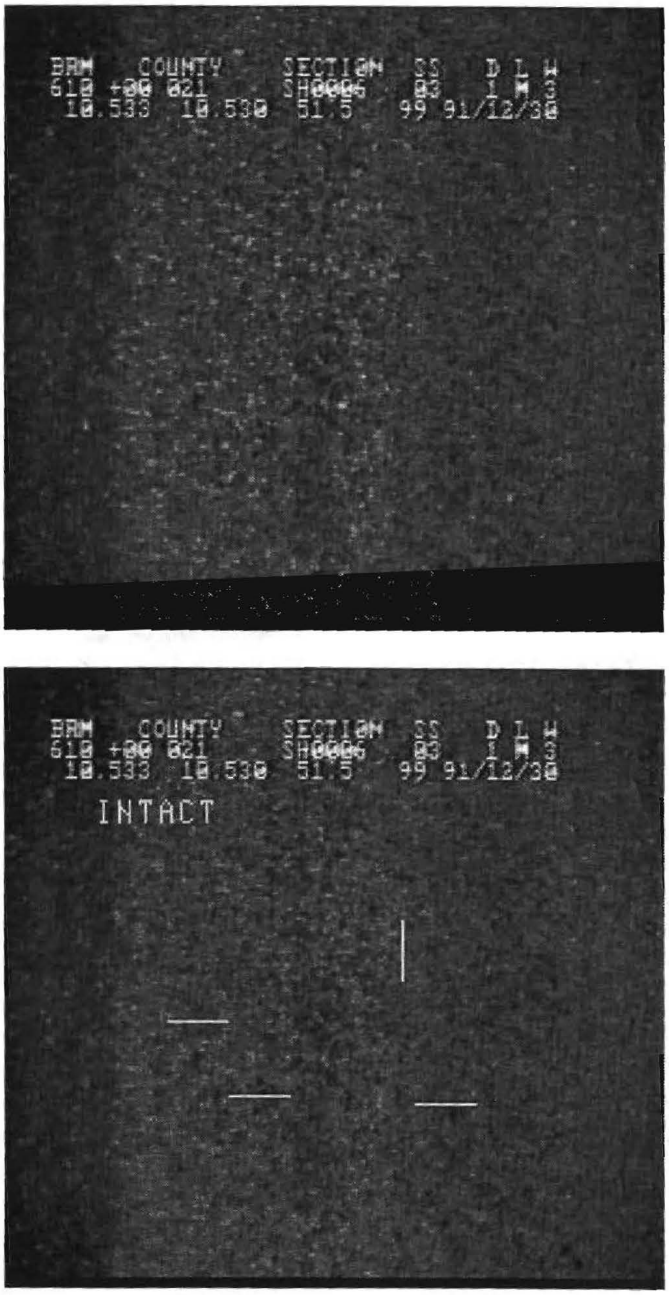

Figure 25. Intact Sealcoat Surface.

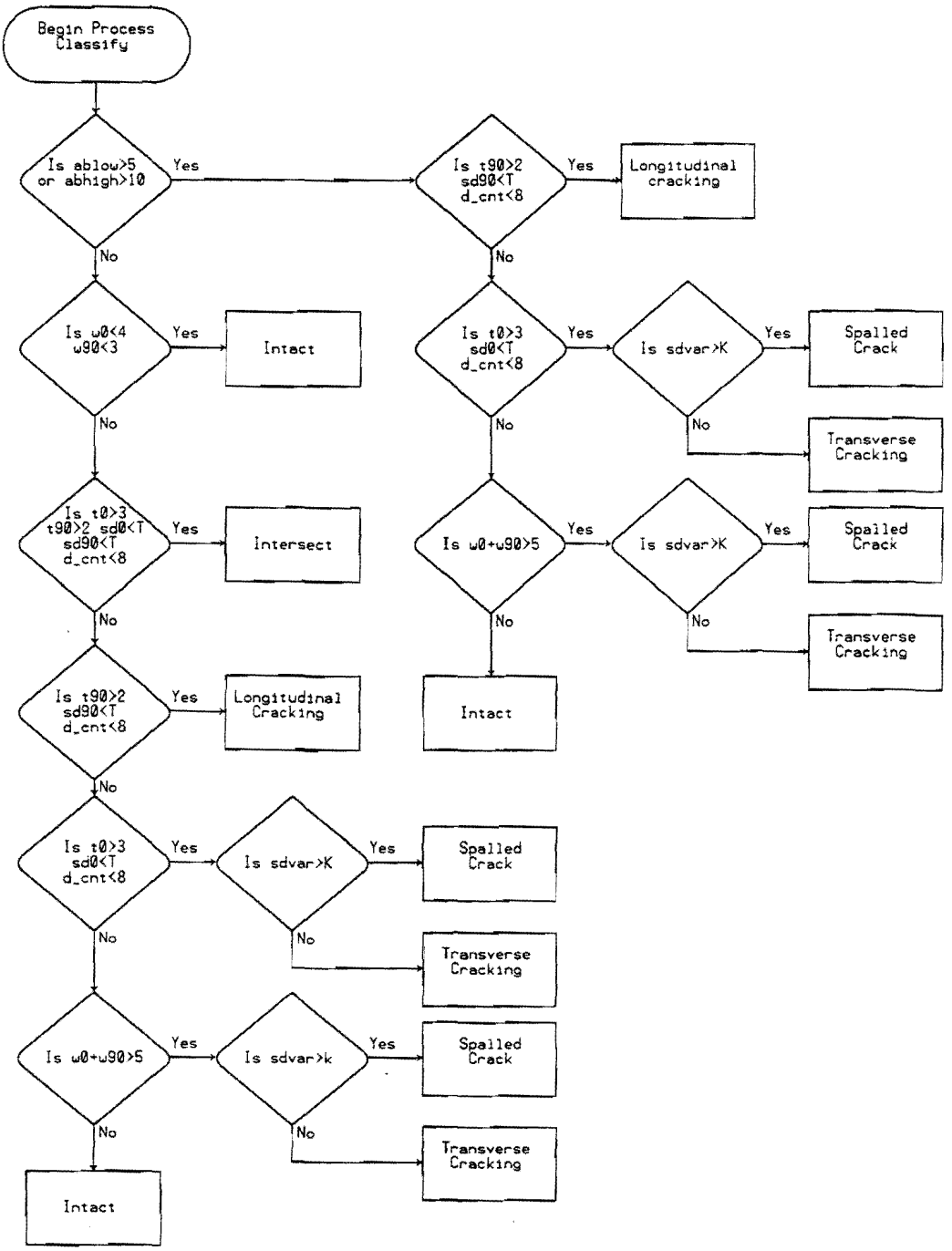

Figure 26. Flowchart for CRCP Distress Classification.

first is the max # of vertical edges in a pair of columns. The second is the degree of alignment of the edges along a straight line. The third condition is related to a parameter called "double count". A block labeled as a "double count" has high projection variance in both horizontal and vertical directions. This third condition helps to identify a heavily "stained" image to avoid misclassification.

The second step on the right branch employs a corresponding set of three conditions to determine the presence of transverse cracks. Once a transverse crack is classified, the next logical step is to determine if the detected transverse crack is spalled. Spalling describes an area where the cement paste and some of the aggregates have fallen out along the transverse crack. In the context of image analysis, the spalled area displays a high projection variance mostly in the horizontal direction and sometimes even in the vertical direction. The high projection variance is caused by the non-uniform pixel intensities of the spalled area. A parameter called the variance of the block projection variances is used to differentiate between a transverse crack and a spalled crack.

The third step examines the total number of horizontal and vertical edges. If this total number exceeds a certain threshold, there is a good chance that either a transverse or spalled crack is present. Once more, the variance of the block projection variances is used to differentiate the spalled cracks and transverse cracks.

The left branch is used to process relatively clean images and it has five steps. The first step in this decision branch isolates the intact image thus saving processing time. The second step actually can be eliminated as there are no intersecting cracks in CRCP. Steps 3, 4, and 5 are identical to the three steps in the right branch. These two branches in the flowchart are very effective in classifying spalled cracks and transverse cracks because there are provisions for identifying cracks in both clean and "stained" images.

4.4 CRCP Results

In order to verify the classification rules, a half mile section of US 59 was processed, a total number of 900 image frames were tested. Figure 27 to Figure 31 show representative images with the detection and classification of cracks following. Figure 27 and Figure 28 show two typical spalled cracks and their processed results. The white framed block

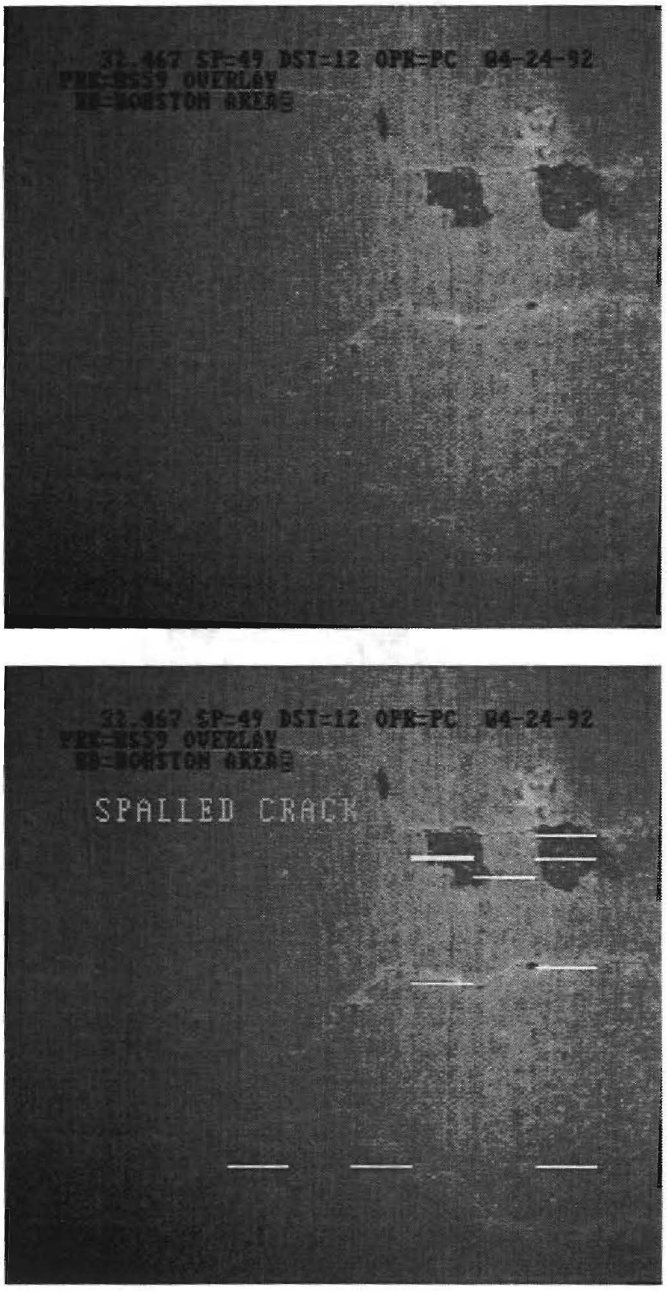

Figure 27. Spalled Crack on CRCP (Example 1).

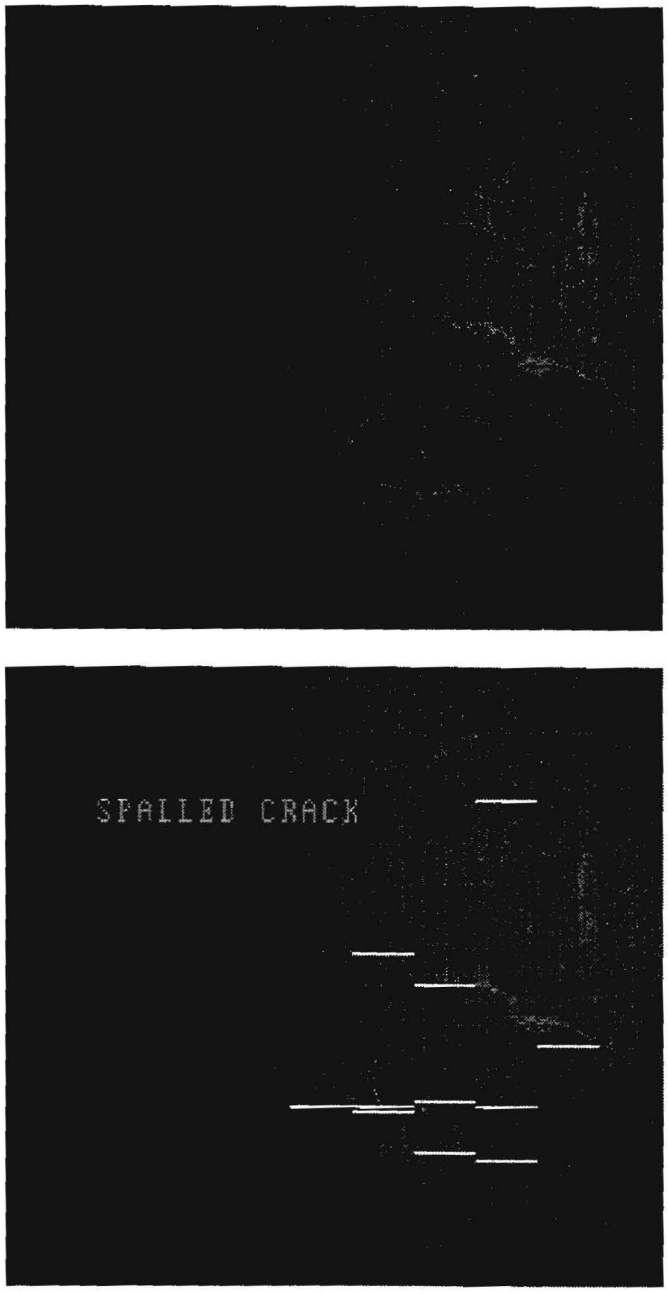

Figure 28. Spalled Crack on CRCP (Example 2).

indicates that the block has a high horizontal projection variance. When a crack segment was detected within the framed block, a line is placed on the position where the crack segment is detected. Figure 29 through Figure 30 show two transverse cracks being detected and classified. Figure 31 shows an image having no cracking being classified as intact.

The results are tabulated in Table 6 where the percent error in the two types of cracking is shown. The average crack spacing for this 0.5 mile section is 3.3 feet and this is calculated by visually counting the total number of transverse cracks and spalled cracks. Three observations can be made. The first is that some fine transverse cracks were missed. The solution to this problem is either to increase the resolution of the survey camera or employ multiple camera units. The second observation is that the algorithm can adapt well to the smear tiremarks and the water marks along the cracks. The third observation shows that only one crack is counted in a cracking image with multiple cracks. These missed counts contribute to the low percent accuracies. The image algorithms are needed to be modified to detect the presence of the multiple cracks.

Table 6. CRCP Processing Result of US 59 (DMI Reading 32.400 to 32.900).

Cracking Type	Actual Number	Computed Number	Percent Error
Spalled Crack	60	13	22%
Transverse crack	690	34	5%

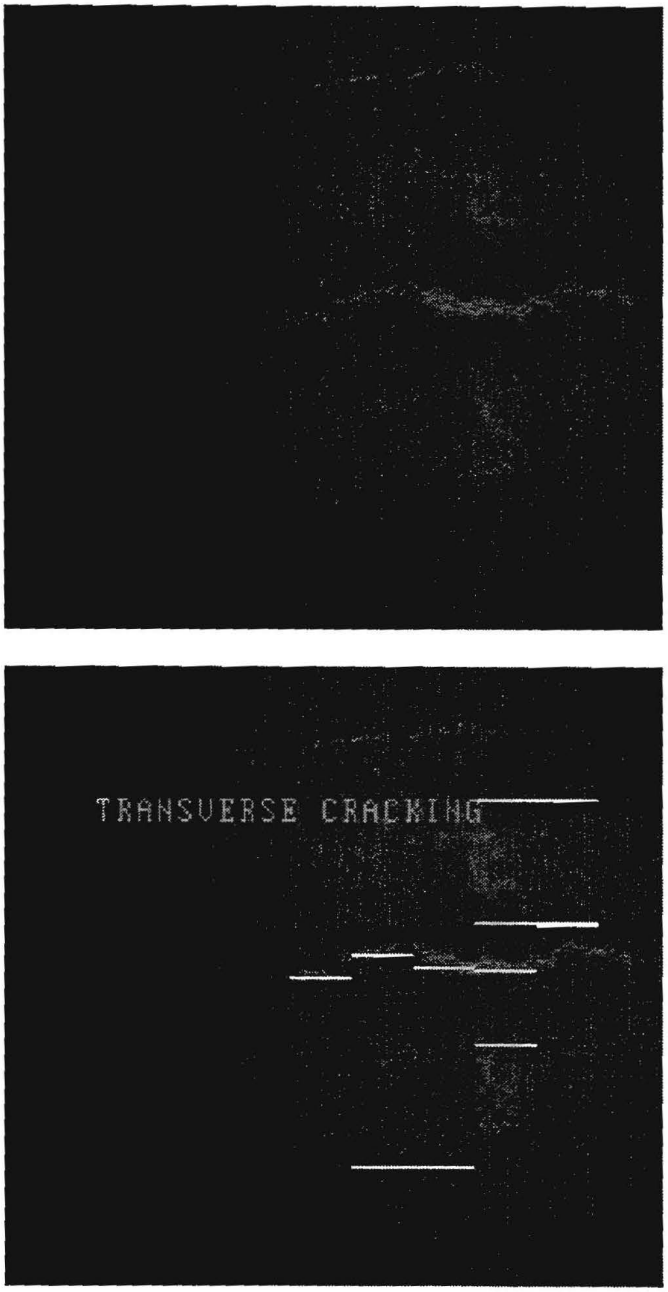

Figure 29. Transverse Crack on CRCP (Example 1).

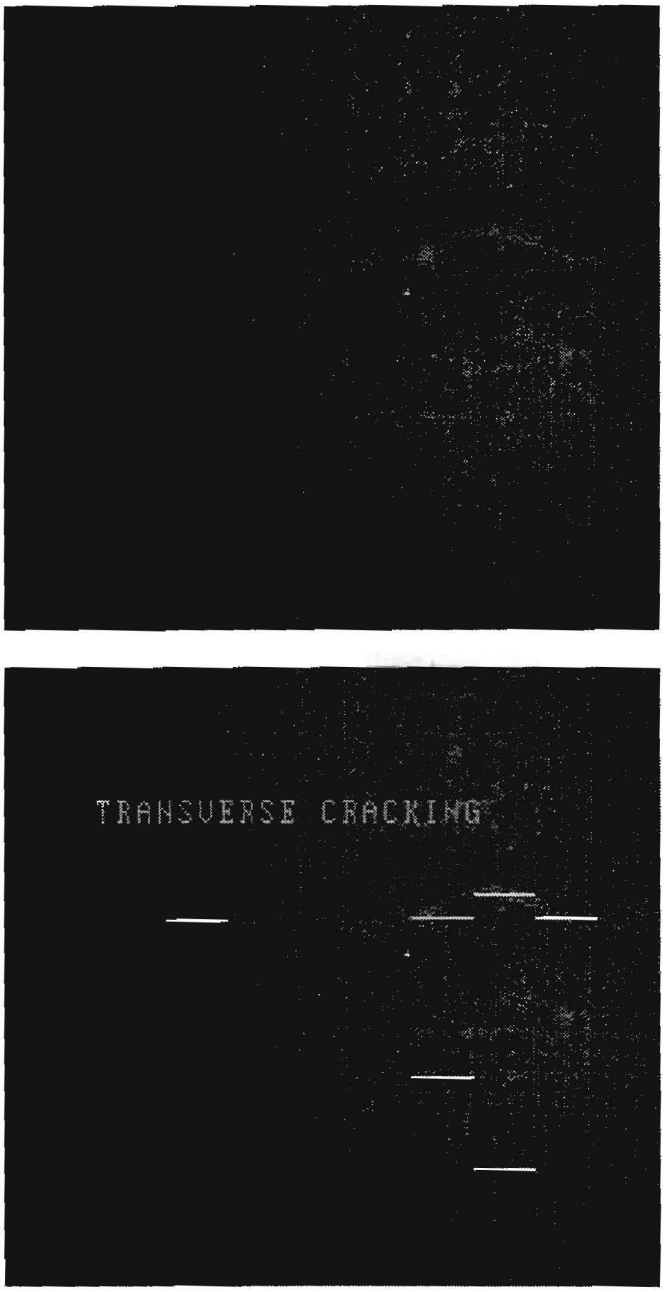

Figure 30. Transverse Crack on CRCP (Example 2).

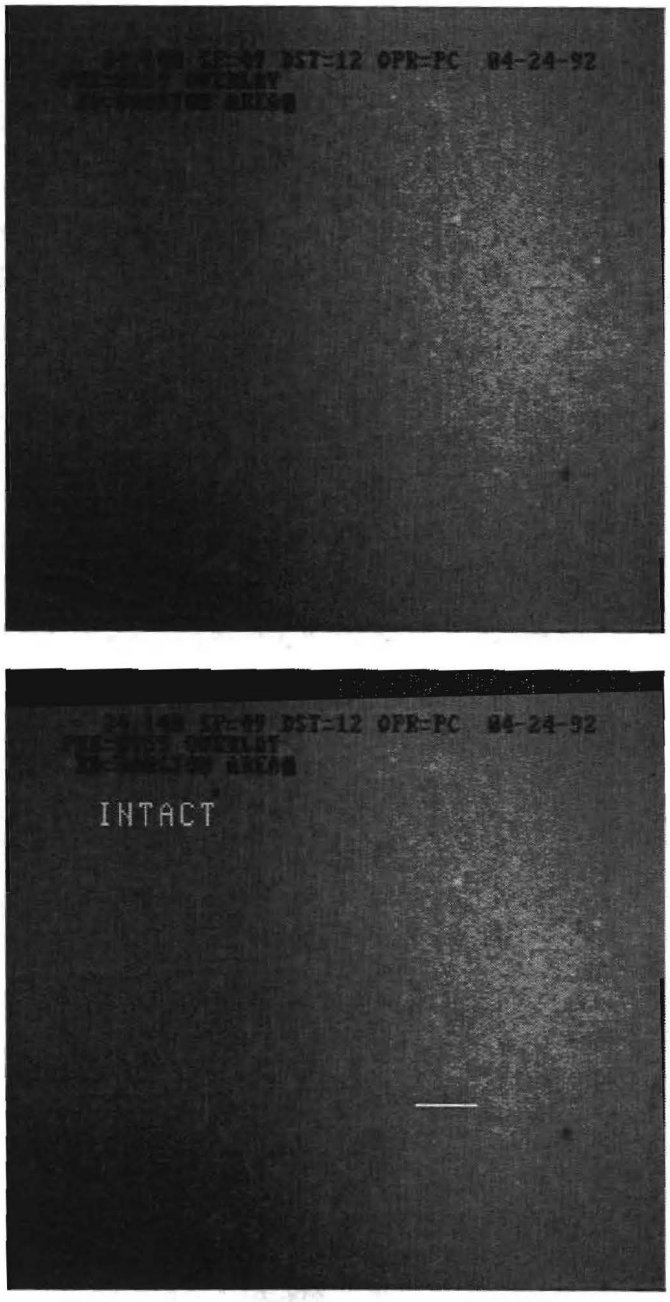

Figure 31. Intact CRCP.

CHAPTER V
CONCLUSIONS AND RECOMMENDATIONS

An image classification system based on the principle of the projection histogram for both ACP and CRCP are presented. The performance of this system is highly satisfactory and the project goals are achieved. The project goals are that the overall accuracy of the processing system must be over 70% and the minimum processing speed is 4 mph which is equivalent to a processing time of 1 frame per second.

The ACP algorithm results were over 70% accurate even in noisy images such as sealcoat surfaces or "stained" images such as oil stain images. The videolog was collected on a sunny day during noon time and there is no shadow projected from the videolog vehicle or from the trees on the roadside. This developed algorithm has been tested extensively with over 12 lane miles of pavement and the results are very consistent and reliable. Algorithms improvements are needed for the CRCP analysis program to account for the presence of multiple cracks.

Recommendations related to acquisition and enhancements of pavement distress videolog are addressed in 1189-1 entitled "System Hardware for Acquisition and Automatic Processing of Pavement Distress Videolog." These issues are:

1) Artificial Illumination system to eliminate shadows.
2) Proto-type videolog survey vehicle.
3) Camera type and camera lens selection.
4) Operator input of non-cracking distress count with keyboard during field survey.

The following recommendations are specifically for the office video image processing system and automatic evaluation software.

1) High resolution digitizer and frame buffer.
2) Variable speed tracking VCR.
3) Personal Computer platform imaging hardware.
4) Incorporate non-cracking distress data.
5) Sampling scheme for videolog processing.
6) Operator override and crosscheck of the processing result.

A high resolution frame grabber, which is a video digitizer and data buffer will enable the detection of early stage cracking. A conventional frame grabber has a pixel resolution of about 0.12 inch/pixel while a high resolution frame grabber will double the resolution to 0.06 inch. This will satisfy the PMIS requirement of 0.125 inch.

A time variable speed tracking VCR can overcome the video jitter, and the degradation in accuracy caused by this jitter. The VCR also allows computer access to individual frames with a time code for non-supervised processing.

Faster and denser imaging hardware has been manufactured by a number of vendors recently. This hardware is a potential candidate to convert the office processing system from a workstation to a PC platform. The advantages of using PC hardware are the user friendly library software, flexibility in PC programming because of larger user base and lower hardware cost. The disadvantage is the limitation in the overall processing speed. TTI is interested in evaluating some of this PC hardware in the follow-up study.

A subroutine will be developed to incorporate the non-cracking data such as the length of patching and number of failures for ACP. These data will be keyed in and stored in a file during the videolog survey with a specialized keyboard. The subroutine will read the data and integrate it with the automatic cracking evaluation to give comprehensive surface distress ratings.

Since the majority of the pavement surfaces are free of any form of distress, a sampling scheme should be developed when processing the pavement images. During the videolog survey, the operator will enter a condition score, from 1 to 10 to approximate the overall condition of the half mile section and this score will be stored in a file. At the time of post-processing, a sampling scheme will apply, for example, if the pavement score is above 9, only 10% of the pavement surface video is processed.

Options will be built into the processing software for the operator to override the computed results with manual input. The operator will also be able to visually double check those sections that show great differences between the field condition scores and the automatic evaluation results.

APPENDIX A

A.1

Program Listing of

Hotmix.c

```
/*****************************************************************
Program : hotmix.c

CopyRighted : Registration # TX 517 825

Author : TTI Div 2.

Date : Nov 92.

Task: Main program to detect cracking on ACP. Refer to
flowchart of the main block diagram to understand the program
better.
        The program initializes the image processor and then
digitizes a frame from video. Two versions of this program exist.
The first version gets its input from pre-acquired images from hard
disk while the second version gets its input from the VCR directly.
Both programs follow the same algorithm from this point onwards.

The image is first digitized from videotape and then sub-divided
into 48 by 48 blocks. Sub-division is employed to make it easier
to detect thin edges.  Within each of the 48 by 48 windows,
an analysis is performed to indicate the presence of an edge.
This block analysis is a series of steps
that includes a variance, mean and projection histogram analysis.

Once, an edge is discovered, its orientation is calculated and the
edge map is updated. The edge map is a simple array that holds the
orientations of the edges discovered and is used for later classification.
The edge clear section removes the effects of noise.
To detect thin edges a second pass of the edge map is used. The second pass
picks up thin edges which helps in classifying thin cracks.

Finally, the image is ready to be classified or recognized as  a crack
of a  certain type. The classification section uses the edge map as its main
input in order to classify the image. Once all images have been classified,
the results are printed out.
A number of variables are used in the program, most of which
are various thresholds used to distinguish edges.

*****************************************************************/

#include <stdio.h>
#include <itex150.h>
#include <strings.h>
#include <math.h>


#define X 30            /* Upper Right Horizontal Position of AOI */
#define Y 100           /* Upper Right Vertical Position of AOI */
#define BX 48           /* Horizontal Size of Window */
#define BY 48           /* Vertical Size of Window */
#define COL 9           /* Number of Columns */
#define ROW 7           /* Number of Rows */
#define WINDOW 7        /* Number of Pixels used in Smoothing Histogram */
#define OFFSET (int) (WINDOW/2) /* Number of Pixels used for fold back in Projection curve */
#define WING 12         /* Number of extra pixels on either side of window used */
/* in calculating actual spread */

/* Defines used in Decision function
   Refer to Decision rule         */

#define KLMNX 1.1
#define KLMNY 1.5
#define PEAK0 9
#define PEAK90 9
```

```
#define GTH0 20          /* Threshold Below Mean for White Paint Mark */
#define GTH90 10         /* Threshold Above Mean for Black Oil Stain */
#define SDTHR 21.0       /* Minimum SD of Detected Edges to be a Crack */


/* Global Variables accessed by most fuctions */

static int PK0[ROW][COL],PK90[ROW][COL],SPR0[ROW][COL],SPR90[ROW][COL],
LMN0_sign[ROW][COL],LMN90_sign[ROW][COL],wind[ROW][COL],
local_mean[ROW][COL],wind_var[ROW][COL],posi0[ROW][COL],posi90[ROW][COL];
float LMN0[ROW][COL],LMN90[ROW][COL],VAR0[ROW][COL],VAR90[ROW][COL];
int img[512][512];



/*
Function: Init
Inputs: -
Outputs: -
Task: Initializes the Image Processor.
*/
void init()
{
        load_cfg("/usr/ITEX/150/lib/std.cfg"); /* Load Standard Configuration */
        initsys();                             /* Clear Screen */
        adi_hbanksel(15);
        adi_hgroupsel(RED|GREEN|BLUE);
        adi_clearlut(255);
        adi_lutmode(DYNAMIC);                  /* Activate Dynamic Overlay */
        contour(ADI,GREEN,1,0,156,255);
        linlut(ADI,RED|GREEN|BLUE,0);
}


/*
Function: read_image
Inputs: -
Outputs:-
Task: Reads Image from disk, removes bright spots by averaging them, stores
result in buffer B1.
*/
void read_image()
{
        int i,j;
        int temp;
        BYTE buf[512];

        for(j=0;j<480;j++){                    /* 480 Horizontal lines */
                fb_rhline(B1,0,j,512,buf);     /* 512 Pixels in one line */
                for(i=0;i<512;i++){
                        temp=buf[i];
                        img[i][j]=temp;
                }
        }
}


/*
Function: hist_90
Inputs: -
        x:Upper left horizontal coordinate of window.
        y:Upper left vertical coordinate of window.
Outputs:-
        array: Array containing Projection Values.
        Returns mean of window.
```

```
Task: Calculate  Vertical Projection Curve
Sum Values of Gray Level along Vertical Axis
*/
float hist_90(x,y,array)
int x,y,array[BX+2*OFFSET+2*WING];
{
        int i,j,sum,sum0,temp;

        sum0=0;
        for (i=x-WING;i<x+BX+WING;i++){
                sum=0;
                for (j=y;j<y+BY;j++){
                        temp=255-img[i][j];  /* Reverse Gray level */
                        sum=sum+temp;
                }
                temp=sum/BX;
                array[i-x+OFFSET+WING]=sum/BY;     /* Store averaged values in array of size BY */
                if (i>=x && i< x+BX) sum0=sum0+temp;
        }
        for (i=0;i<=OFFSET;i++){
                array[OFFSET-i]=array[OFFSET+i]; /* Mirror values of array on each end */
                array[BX+2*WING+OFFSET-1+i]=array[BX+2*WING+OFFSET-1-i];
        }
        return((float) sum0/(float) BX);
}


/*
Function: hist_0
Inputs: -
        x:Upper left horizontal coordinate of window.
        y:Upper left vertical coordinate of window.
Outputs:-
        array: Array containing Projection Values.
        Returns mean of window.
Task: Calculate  Horizontal Projection Curve
Sum Values of Gray Level along Horizontal Axis
*/
float hist_0(x,y,array)
int x,y,array[BY+2*OFFSET+2*WING];
{
        int i,j,sum,sum0,temp;

        sum0=0;
        for (j=y-WING;j<y+BY+WING;j++){
                sum=0;
                for (i=x;i<x+BX;i++){
                        temp=255-img[i][j];  /* Reverse Gray level */
                        sum=sum+temp;
                }
                temp=sum/BX;
                array[j-y+OFFSET+WING]=temp;      /* Store averaged values in array of size BY */
                if (j>=y && j<y+BY) sum0=sum0+temp;
        }
        for (i=0;i<=OFFSET;i++){
                array[OFFSET-i]=array[OFFSET+i]; /* Mirror values of array on each end */
                array[BY+2*WING+OFFSET-1+i]=array[BY+2*WING+OFFSET-1-i];
        }
        return((float) sum0/(float) BY);
}


/*
Function: draw_hist
Inputs: -
        x: Coordinate x on screen where plot is desired
```

```
        y: Coordinate y on screen where plot is desired
        array: Array containing Projection Values.
        mean: Mean of array input.
Outputs: -
Task: Routine to Plot the Projection Histogram on the Screen.
Also Plots a line indicating mean.
*/
void draw_hist(x,y,array,mean)
int x,y,array[BX+2*WING];
{
        int i,j;
        for (i=0;i<BX+2*WING;i++){
                j=(array[i]*160)/160;
                line(B1,1,(x+i),y,(x+i),(y+j),255);
        }
        line(B1,1,x,y+mean,x+BX,y+mean,100);
}



/*
Function: clear_hist
Inputs: -
        x: Coordinate x on screen where plot is desired
        y: Coordinate y on screen where plot is desired
Outputs: -
Task: Routine to clear the Projection Histogram on the Screen.
so that the next plot does not overwrite on the previous one.
*/
void clear_hist(x,y)
int x,y;
{
        int i,j;
        for (i=0;i<BX;i++){
                j=160;
                line(B1,1,(x+i),y,(x+i),(y+j),0);
        }
}



/*
Function: line_90
Inputs: -
        x: Upper Left x coordinate of window.
        x: Upper Left y coordinate of window.
        pos90: Position of detected vertical edge.
Outputs: -
Task: Routine to Plot a line on the screen indicating the
position of detected edge.
*/
void line90(x,y,pos90)
int x,y,pos90;
{
        line(B1,0,(x+pos90),y,(x+pos90),y+BX-1,255);
}


/*
Function: line_0
Inputs: -
        x: Upper Left x coordinate of window.
        x: Upper Left y coordinate of window.
        pos0: Position of detected horizontal edge.
Outputs: -
Task: Routine to Plot a line on the screen indicating the
```

```
position of detected edge.
*/
void line0(x,y,pos0)
int x,y,pos0;
{
        line(B1,0,x,(y+pos0),(x+BX-1),(y+pos0),255);
}


/*
Function: curve_smooth
Inputs: -
        rawx: Unsmoothed array containing horizontal projection values.
        rawy: Unsmoothed array containing vertical projection values.
Outputs: -
        smoothx: Smoothed array containing horizontal projection values.
        smoothy: Smoothed array containing vertical projection values.
        maxx: Maximum value in x projection array.
        maxy: Maximum value in y projection array.
        minx: Minimum value in x projection array.
        miny: Minimum value in y projection array.
        pmaxx: Position of maxx in array.
        pmaxy: Position of maxy in array.
        pminx: Position of minx in array.
        pminy: Position of miny in array.
Task: Smooth both horizontal and vertical projection values
also calculate the max and min values in the array along with
their respective positions in the array.
*/
void curve_smooth(rawx,smoothx,rawy,smoothy,maxx,
minx,pmaxx,pminx,maxy,miny,pmaxy,pminy)
int rawx[BY+2*OFFSET+2*WING],smoothx[BY+2*WING],rawy[BY+2*OFFSET+2*WING],smoothy[BY+2*WING];
int *maxx,*minx,*pmaxx,*pminx,*maxy,*miny,*pmaxy,*pminy;
{
        int sumx,avgx,sumy,avgy;
        int Head,Tail,center,count;

        for(center=OFFSET;center<BY+2*WING+OFFSET;center++){
                sumx=0;
                sumy=0;
                avgx=0;
                avgy=0;
                Head=center-OFFSET;
                Tail=center+OFFSET;
                for(count=0;count<OFFSET;count++){
                        sumx +=rawx[Head];
                        sumx +=rawx[Tail];
                        sumy +=rawy[Head];
                        sumy +=rawy[Tail];
                        Head++;
                        Tail--;
                }
                sumx=sumx+rawx[center];
                sumy=sumy+rawy[center];
                avgx=sumx/WINDOW;
                avgy=sumy/WINDOW;
                smoothx[center-OFFSET]=avgx;
                smoothy[center-OFFSET]=avgy;
                if (center==OFFSET+WING){
                        *maxx=*minx=avgx;
                        *maxy=*miny=avgy;
                        *pmaxx=*pmaxy=center-OFFSET;
                        *pminx=*pminy=center-OFFSET;
                }
                if(center >= OFFSET+WING && center < BY+OFFSET+WING)
                {
```

```
                        if (avgx>=*maxx){
                                *maxx=avgx;
                                *pmaxx=center-OFFSET;
                        }
                        if (avgx<=*minx){
                                *minx=avgx;
                                *pminx=center-OFFSET;
                        }
                        if (avgy>=*maxy){
                                *maxy=avgy;
                                *pmaxy=center-OFFSET;
                        }
                        if (avgy<=*miny){
                                *miny=avgy;
                                *pminy=center-OFFSET;
                        }
                }
        }
}

/*
Function: lmnY
Inputs: -
        array:Smoothed array of vertical projection.
        max: Maximum vlaue in array.
        pmax: Position of max.
        mean: Mean of array.
Outputs: -
        peak: peak
        denom: Spread.
        posit: position of crack
        sign: positive
        Returns lmn
Task:
Calculate the lmn=peak/spread=(max-mean)/spread
for the vertical projection values for positive peaks.
*/
float lmnY(array,max,pmax,mean,peak,denom,posit,sign)
int mean,max,*peak,*denom,*posit,*sign,array[BY+2*WING];
{
        int i,maxi,mmi;
        float result;

        maxi=BX+2*WING-1;
        mmi=0;
        for (i=pmax;i<BX+2*WING;i++) /* Find spread to the right of position of maximum */
                if (array[i]<=mean){
                        maxi=i;
                        break;
                }
        for (i=pmax;i>=0;i--) /* Find spread to the left of position of maximum */
                if (array[i]<=mean){
                        mmi=i;
                        break;
                }
        *peak=max-mean;
        *denom=maxi-mmi;
        *posit=pmax;
        *sign=1;

        result= (float)(max-mean)/(float)(*denom);
        if (*denom>30 || *denom<7) result=0.0;          /* If spread<7 or spread> 30 set lmn=0 */

        return(result);
}
```

```
/*
Function: lmnYY
Inputs: -
        array:Smoothed array of vertical projection.
        min: Minimum value in array.
        pmin: Position of min.
        mean: Mean of array.
Outputs: -
        peak: peak
        denom: Spread.
        posit: position of peak
        sign: negative
        Returns lmn
Task:
Calculate the lmn=peak/spread=(mean-min)/spread
for the vertical projection values for negative peaks.
*/
float lmnYY(array,min,pmin,mean,peak,denom,posit,sign)
int mean,min,pmin,*peak,*denom,*posit,*sign,array[BY+2*WING];
{
        int i,mini,mni;
        float result;

        mini=BX+2*WING-1;
        mni=0;
        for (i=pmin;i<BX+2*WING;i++)  /* Find spread to the right of position of minimum */
                if (array[i]>=mean){
                        mini=i;
                        break;
                }
        for (i=pmin;i>=0;i--)  /* Find spread to the left of position of minimum */
                if (array[i]>=mean){
                        mni=i;
                        break;
                }

        *peak=mean-min;
        *denom=mini-mni;
        *posit=pmin;
        *sign=-1;

        result= (float)(mean-min)/(float)(*denom);      /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;

        return(result);
}


/*
Function: lmnX
Inputs: -
        array:Smoothed array of horizontal projection.
        max: Maximum value in array.
        pmax: Position of max.
        mean: Mean of array.
Outputs: -
        peak: Peak
        denom: Spread.
        Posit: Position of peak.
        sign: Positive
        Returns lmn
Task:
Calculate the lmn=peak/spread=(max-mean)/spread
for the horizontal projection values for positive peaks.
*/
float lmnX(array,max,pmax,mean,peak,denom,posit,sign)
```

```
int mean,max,*peak,*denom,*posit,*sign,array[BX+2*WING];
{
        int i,maxi,mmi;
        float result;

        maxi=BX+2*WING-1;
        mmi=0;
        for (i=pmax;i<BX+2*WING;i++)  /* Find spread to the right of position of maximum */
                if (array[i]<=mean){
                        maxi=i;
                        break;
                }
        for (i=pmax;i>=0;i--)  /* Find spread to the left of position of maximum */
                if (array[i]<=mean){
                        mmi=i;
                        break;
                }
        *peak=max-mean;
        *denom=maxi-mmi;
        *posit=pmax;
        *sign=1;

        result= (float)(max-mean)/(float)(*denom);      /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;
        return(result);
}

/*
Function: lmnXX
Inputs: -
        array:Smoothed array of horizontal projection.
        min: Minimum value in array.
        pmin: Position of min.
        mean: Mean of array.
Outputs: -
        peak: Peak
        denom: Spread.
        posit: Position of peak.
        sign: Negative
        Returns lmn
Task:
Calculate the lmn=peak/spread=(mean-min)/spread
for the horizontal projection values for negative peaks.
*/
float lmnXX(array,min,pmin,mean,peak,denom,posit,sign)
int mean,min,pmin,*peak,*denom,*posit,*sign,array[BX+2*WING];
{
        int i,mini,mni;
        float result;

        mini=BX+2*WING-1;
        mni=0;
        for (i=pmin;i<BX+2*WING;i++) /* Find spread to the right of position of minimum */
                if (array[i]>=mean){
                        mini=i;
                        break;
                }
        for (i=pmin;i>=0;i--) /* Find spread to the left of position of minimum */
                if (array[i]>=mean){
                        mni=i;
                        break;
                }

        *peak=mean-min;
        *denom=mini-mni;
        *posit=pmin;
```

```
        *sign=-1;

        result= (float)(mean-min)/(float)(*denom);        /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;
        return(result);
}

/*
Function: varX
Inputs: -
        array:Smoothed array of horizontal projection.
        mean: Mean of array.
Outputs: -
        Returns variance of array
Task:
Calculate the variance of the horizontal projection.
*/
float varX(array,mean)
int array[BX+2*OFFSET+2*WING];
float mean;
{
        int i;
        float sum,temp;

        sum=0.0;
        for (i=0;i<BX;i++){
                temp=(float) array[i+OFFSET+WING]-mean;
                sum=sum+temp*temp;
        }
        return((float)(sum)/(float)(BX-1));
}

/*
Function: varY
Inputs: -
        array:Smoothed array of vertical projection.
        mean: Mean of array.
Outputs: -
        Returns variance of array
Task:
Calculate the variance of the vertical projection.
*/
float varY(array,mean)
int array[BY+2*OFFSET+2*WING];
float mean;
{
        int i;
        float sum,temp;

        sum=0.0;
        for (i=0;i<BY;i++){
                temp=(float) array[i+OFFSET+WING]-mean;
                sum=sum+temp*temp;
        }
        return((float)(sum)/(float)(BY-1));
}


/*
Function: prints
Inputs: -
        type: pointer to string.
Outputs: -
Task:
Print string (ALLIGATOR etc) on screen.
```

```
*/
void prints(type)
char *type;
{
        text(B1,0,70,100,HORIZONTAL,2,255,type);
}



/*
Function: decision
Inputs: -
        lmn0:lmn for horizontal projection.
        lmn90: lmn for vertical projection.
        LMNX: Minimum threshold for horizontal projection.
        LMNY: Minimum threshold for vertical projection.
        i: Horizontal index of window.
        j: Vertical index of window.
        peak0:
        peak90:
        spread0:
        spread90:
        sign0:
        sign90:
Outputs: -
        Returns  angle
        angle 0: no edge
        angle 4:  horizontal edge
        angle 90: vertical edge
Task:
Decides if the window contains àn edge based on the
information available. Returns an angle.
*/
int decision(lmn0,lmn90,LMNX,LMNY,nLMNX,nLMNY,i,j,peak0,peak90,spread0,spread90,sign0,sign90)
float lmn0,lmn90,LMNX,LMNY,nLMNX,nLMNY;
int i,j,peak0,peak90,spread0,spread90,sign0,sign90;
{
        int angle;
        float temp_lmn0,temp_lmn90;

        angle=1;
        if (sign0==1) temp_lmn0=LMNX;
        else temp_lmn0=nLMNX;
        if (sign90==1) temp_lmn90=LMNY;
        else temp_lmn90=nLMNY;

        if (lmn0>temp_lmn0 && lmn0 > (KLMNX*lmn90) && (peak0 > PEAK0) &&
            (wind_var[i][j]==4 || wind_var[i][j]==49))angle=4*sign0;

        if (lmn90>temp_lmn90 && lmn90 > (KLMNY*lmn0) && (peak90 > PEAK90 && peak90<40) &&
            (wind_var[i][j]==90 || wind_var[i][j]==49) && peak90
            > 1.5*peak0 && spread90 > 5 ) angle=90*sign90;
        return(angle);
}



/*
Function: var2col
Inputs: -
Outputs: -
        Returns  smallest SD of edges in a pair of cols.
Task:
Determines if the edges detected fall in a straight line
indicating a longitudinal crack. Checks overlapping
pairs of cols calculating the SD of the edges. Returns SD.
```

```
If the number of edges in any col > 4, SD is calculated
for that col only.
*/
float var2col()
{
        int sum[COL-1];
        int i,j,k;
        int sumup,count1,count2;
        int var1,temp,temp1;
        static int data1[ROW],data2[ROW];
        float sd1,SD;

        SD=100.0;
        for (j=0;j<COL-1;j++){ /* Count the number of edges in every pair of cols */
                sumup=0;
                for (i=0;i<ROW;i++){
                        if (abs(wind[i][j])==90) sumup++;
                        if (abs(wind[i][j+1])==90) sumup++;
                }
                sum[j]=sumup;
        }


        for (j=0;j<COL-1;j++){
                count1=0;
                count2=0;
                if (sum[j]>2){       /* If the number of edges in the pair is >2 */
                        for (i=0;i<ROW;i++){
                                if (abs(wind[i][j])==90){ /* find number of edges in each row */
                                        data1[count1]=posi90[i][j];
                                        count1++;
                                }
                                if (abs(wind[i][j+1])==90){
                                        data2[count2]=BX+posi90[i][j+1];
                                        count2++;
                                }
                        }
                        if (count1>4){ /* If number of edges in row is > 4 */
                                temp=temp1=0; /* find SD for the row */
                                for (k=0;k<count1;k++){
                                        temp=temp+data1[k]*data1[k];
                                        temp1=temp1+data1[k];
                                }
                                var1=(temp-(temp1*temp1/count1))/(count1-1);
                                sd1=(float) sqrt((double) var1);
                                if (sd1<SD) SD=sd1;
                        }
                        if (count2>4){
                                temp=temp1=0;
                                for (k=0;k<count2;k++){
                                        temp=temp+data2[k]*data2[k];
                                        temp1=temp1+data2[k];
                                }
                                var1=(temp-(temp1*temp1/count2))/(count2-1);
                                sd1=(float) sqrt((double) var1);
                                if (sd1<SD) SD=sd1;
                        }
                        temp=temp1=0;
                        for (k=0;k<count1;k++){
                                temp=temp+data1[k]*data1[k];
                                temp1=temp1+data1[k];
                        }
                        for (k=0;k<count2;k++){
                                temp=temp+data2[k]*data2[k];
                                temp1=temp1+data2[k];
                        }
```

```
                        var1=(temp-(temp1*temp1/(count1+count2)))/((count1+count2)-1);
                        sd1=(float) sqrt((double) var1);
                        if (sd1<SD) SD=sd1;
                }
        }
        return(SD);
}


/*
Function: var2row
Inputs: -
Outputs: -
        Returns  smallest SD of edges in a pair of rows.
Task:
Determines if the edges detected fall in a straight line
indicating a transverse crack. Checks overlapping
pairs of rows calculating the SD of the edges. Returns SD.
*/
float var2row()
{
        int sum[ROW-1];
        int i,j,k;
        int sumup,count;
        int var1,temp,temp1;
        static int data[COL+COL];
        float sd1,SD;

        SD=100.0;
        for (i=0;i<ROW-1;i++){
                sumup=0;
                for (j=0;j<COL;j++){
                        if (abs(wind[i][j])==4) sumup++;
                        if (abs(wind[i+1][j])==4) sumup++;
                }
                sum[i]=sumup;
        }


        for (i=0;i<ROW-1;i++){
                count=0;
                if (sum[i]>3){
                        for (j=0;j<COL;j++){
                                if (abs(wind[i][j])==4){
                                        data[count]=posi0[i][j];
                                        count++;
                                }
                                if (abs(wind[i+1][j])==4){
                                        data[count]=BX+posi0[i+1][j];
                                        count++;
                                }
                        }
                        temp=temp1=0;
                        for (k=0;k<count;k++){
                                temp=temp+data[k]*data[k];
                                temp1=temp1+data[k];
                        }
                        var1=(temp-(temp1*temp1/count))/(count-1);
                        sd1=(float) sqrt((double) var1);
                        if (sd1<SD) SD=sd1;
                }
        }

        return(SD);
}
```

```
/*
Function: clear_edge
Inputs: -
Outputs: -
Task:
Clear false edges detected around oilstain in 4 directions
to prevent a misclassification.
*/
void clear_edge()
{
        int i,j;

        for (i=0;i<ROW;i++){
                for (j=0;j<COL;j++){
                        if (wind[i][j]==2){
                                /*if (i>0 && wind[i-1][j]!=2) wind[i-1][j]=5;
                                if (i<ROW-1 && wind[i+1][j]!=2) wind[i+1][j]=5;*/
                                if (j>0 && wind[i][j-1]!=2) wind[i][j-1]=5;
                                if (j<COL-1 && wind[i][j+1]!=2) wind[i][j+1]=5;
                        }
                }
        }
}


/*
Function: scan_trans
Inputs: -
        trans_index: Index of row where second scan needs to be done
        LMNX: LMN threshold for positive cracks.
        LMNY: LMN threshold for negative cracks.
Outputs: -
Task:
Perform a second scan using a different decision rule in order
to pick up edges missed in the first run for transverse cracks.
*/
void scan_trans(trans_index,LMNX,nLMNX)
int trans_index;
float LMNX,nLMNX;
{
        int i,j;
        float temp;

        for (i=trans_index;i<ROW && i<trans_index+2;i++){
                for (j=0;j<COL;j++){
                        if (LMN0_sign[i][j]==-1) temp=nLMNX;
                        else temp=LMNX;
                        if ((wind_var[i][j]==4 || wind_var[i][j]==49) && LMN0[i][j]>temp &&
                        wind[i][j]==1) wind[i][j]= 4;
                }
        }
}


/*
Function: scan_long
Inputs: -
        long_index: Index of col where second scan needs to be done
        LMNX: LMN threshold for positive cracks.
        LMNY: LMN threshold for negative cracks.
Outputs: -
Task:
Perform a second scan using a different decision rule in order
```

```
to pick up edges missed in the first run for longitudinal cracks.
*/
void scan_long(long_index,LMNY,nLMNY)
int long_index;
float LMNY,nLMNY;
{
        int i,j;
        float temp;

        for (j=long_index;j<COL && j<long_index+2;j++){
                for (i=0;i<ROW;i++){
                        if (LMN90_sign[i][j]==-1) temp=nLMNY;
                        temp=LMNY;
                        if ((wind_var[i][j]==90||wind_var[i][j]==49) && LMN90[i][j]>temp &&
                            wind[i][j]==1 && PK90[i][j] > 1.5*PK0[i][j] && SPR90[i][j]> 5) wind[i][j]= 90;
                }
        }
}


/*
Function: find_long
Inputs: -
Outputs: -
        index: Index of col having maximum number of 90  edges.
        Returns number of edges in the col.
Task:
Find index of col having maximum number of 90 edges from the edge map.
*/
int find_long(index)
int *index;
{
        int sum,i,j,MAX;

        MAX=0;
        *index=0;
        for (j=0;j<COL-1;j++){
                sum=0;
                for (i=0;i<ROW;i++){
                        if ((abs(wind[i][j]))==90) sum++;
                        if ((abs(wind[i][j+1]))==90) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                        *index=j;
                }
        }
        return(MAX);
}


/*
Function: find_long_var
Inputs: -
Outputs: -
        Returns index of col with max number of 90 var windows.
Task:
Find index of col having maximum number of black windows( 90 windows)
from the variance map.
*/
int find_long_var()
{
        int sum,i,j,MAX;

        MAX=0;
        for (j=0;j<COL;j++){
```

```
                sum=0;
                for (i=0;i<ROW;i++){
                        if ((abs(wind_var[i][j]))==90 || (abs(wind_var[i][j]))==49) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                }
        }
        return(MAX);
}


/*
Function: find_trans
Inputs: -
Outputs: -
        index: Index of row having maximum number of 4 ( horizontal )  edges.
        Returns number of edges in the row.
Task:
Find index of row having maximum number of 4 edges (horizontal) from the edge map.
*/
int find_trans(index)
int *index;
{
        int sum,i,j,MAX;

        MAX=0;
        *index=0;
        for (i=0;i<ROW-1;i++){
                sum=0;
                for (j=0;j<COL;j++){
                        if ((abs(wind[i][j]))==4) sum++;
                        if ((abs(wind[i+1][j]))==4) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                        *index=i;
                }
        }
        return(MAX);
}



/*
Function: get_dir
Inputs: - x: number of directory.
Outputs: -
        Returns pointer to name of directory.
Task:
Find directory name given a directory number.
*/
char *get_dir(x)
int x;
{
        char *dirname1 = "/export/acp_images/hotmix",
        *dirname2 = "/export/images2/canon",
        *dirname3 = "/export/images2/test",
        *dirname4 = "/export/images2/sh6",
        *dirname5 = "/export/images2/aran",
        *dirname6 = "/export/images3";
        char *dirname="";

        switch(x) {
        case '1':
                dirname=dirname1;
```

```
                break;
        case '2':
                dirname =dirname2;
                break;
        case '3':
                dirname =dirname3;
                break;
        case '4':
                dirname =dirname4;
                break;
        case '5':
                dirname =dirname5;
                break;
        case '6':
                dirname =dirname6;
                break;
        case '0':
                printf("New Directory :");
                scanf("%s",dirname);
                getchar();
        }
        return(dirname);
}


/*
Function: initialize lmn
Inputs: -
Outputs: -
        LMNX: Positive threshold for LMN0
        LMNY: Positive threshold for LMN90
        nLMNX: Negative  threshold for LMN0
        nLMNX: Negative  threshold for LMN90
Task:
Initialize values of LMN reading from file lmn-hotmix
*/
void initialize_lmn(LMNX,LMNY,nLMNX,nLMNY)
float *LMNX,*LMNY,*nLMNX,*nLMNY;
{

        FILE *fopen(),*shape;

        shape=fopen("/home/1189/ashok/itex/lmn-hotmix","r");
        fscanf(shape,"%f %f %f %f",LMNX,LMNY,nLMNX,nLMNY); /* Get values for lmn thresholds */
        fclose(shape);
}


/*
Function: classify edges
Inputs: -
        t0: max number of horizontal edges in a pair of rows.
        t90: max number of vertical edges in a pair of rows.
        ab_low:number of windows having paint mark in them.
        ab_high:number of windows having oil stain in them.
        double_count:number of windows having a high 0 and 90 variance.
        w0: number of windows having horizontal(4)  edges.
        w90:number of windows having vertical (90) edges.
        sd0:standard deviation of edges in t0
        sd90:standard deviation of edges in t90
Outputs: -
        class_num:number of classification
        Returns classification name
Task:
Initialize values of LMN reading from file lmn-hotmix
*/
```

```
char *classify(t0,t90,ab_low,ab_high,double_count,w0,w90,sd0,sd90,class_num)
int t0,t90,ab_low,ab_high,double_count,w0,w90,class_num;
float sd0,sd90;
{
        char *class="";


        if (ab_low > 5 || ab_high > 10){
                if (t90>2 && sd90<SDTHR && double_count <8){
                        class="LONGITUDINAL";
                        class_num=3;
                }
                else if (t0>3 && sd0<SDTHR && double_count <8){
                        class="TRANSVERSE CRACKING";
                        class_num=4;
                }
                else if ((w0+w90)>10){
                        class="ALLIGATOR CRACKING";
                        class_num=5;
                }
                else{
                        class="INTACT";
                        class_num=0;
                }
        }
        else if(w0<4 && w90<3){
                class="INTACT";
                class_num=0;
        }
        else if (t0>3 && t90>2 && sd0<SDTHR && sd90<SDTHR && double_count <8){
                class="INTERSECT";
                class_num=6;
        }
        else if (t90>2 && sd90<SDTHR && double_count <8){
                class="LONGITUDINAL CRACKING";
                class_num=3;
        }
        else if (t0>3 && sd0<SDTHR && double_count <8){
                class="TRANSVERSE CRACKING";
                class_num=4;
        }
        else if ((w0+w90)>10){
                class="ALLIGATOR CRACKING";
                class_num=5;
        }
        else{
                class="INTACT";
                class_num=0;
        }
        return(class);
}



void main()
{
        static int unsm0[BY+2*OFFSET+2*WING],unsm90[BX+2*OFFSET+2*WING],
        sm0[BY+2*WING],sm90[BX+2*WING];
        int max0,max90,mean0,mean90;
        int pmax0,pmax90,pmin0,pmin90,min0,min90;
        float lmn0,lmn90,var0,var90,mean1;
        char *dirname="",in;
        FILE *input_file,*fopen(),*shape;
        char fname[20],name[20];
        int len;
```

```
int r,s;
int i,j;
float nLMNX,nLMNY,LMNX,LMNY;
int row,col;
int angle,window90,window0;
char *class="";
int class_num;
float sd0,sd90;
int t0,t90,w0,w90,global_mean,ab_low,ab_high,posit;
static int SHADOW,PAINT,TRANS,LONG,ALLI,INTACT,INTER;
int sign0,sign90,trans_index,long_index;
int peak0,peak90;
int spread0,spread90,temp;
float tempp;
int double_count;


init();                 /* Initialize Image Processor */
fb_clf(B1,110);
printf("\nDirectory 0 - 6 :");
in=getchar();
dirname=get_dir(in);    /* Get handle of image directory */
getchar();
input_file = fopen("images","r");
chdir(dirname);
while (fgets(name,60,input_file)!=NULL){
        len = strlen(name);
        strcpy(fname,"");
        strncat(fname,name,(len-1));
        initialize_lmn(&LMNX,&LMNY,&nLMNX,&nLMNY); /* Get values for lmn */
        r=X;                                       /* Initialize starting x and y */
        s=Y;
        im_read(B1,0,0,512,512,fname);
        read_image();
        window0=window90=global_mean=ab_low=ab_high=0;
        for (row=0;row<ROW;row++){
                for (col=0;col<COL;col++){
                        rectangle(B2,0,r,s,BX,BY,15);
                        angle=1;
                        mean1=hist_0(r,s,unsm0);
                        var0=varX(unsm0,mean1);
                        mean0=(int) mean1;
                        mean1=hist_90(r,s,unsm90);
                        var90=varY(unsm90,mean1);
                        mean90=(int) mean1;
                        wind_var[row][col]=1;
                        VAR0[row][col]=var0;
                        VAR90[row][col]=var90;
                        lmn0=lmn90=0.0;
                        peak0=peak90=spread0=spread90=0;
                        if (var0 > 30.0 && var90< 300.0){
                                wind_var[row][col]=4;
                                rectangle(B1,0,r,s,BX,BY,200);
                        }
                        if (var90 >50.0 && var90 <350.0 && var0< 350.0){
                                if (wind_var[row][col]==4) wind_var[row][col]=49;
                                else wind_var[row][col]=90;
                                rectangle(B1,0,r,s,BX-5,BY-5,0);
                        }
                        if ((var0 > 30.0 && var90< 300.0)
                            || (var90 >50.0 && var90 <350.0 && var0<350.0)){
                                curve_smooth(unsm0,sm0,unsm90,sm90,&max0,&min0,
                                    &pmax0,&pmin0,&max90,&min90,&pmax90,&pmin90);
                                if ((mean0-min0) > (max0-mean0))
                                        lmn0=lmnYY(sm0,min0,pmin0,mean0,&peak0,
                                            &spread0,&posit,&sign0);
```

```
                        else
                                lmn0=lmnY(sm0,max0,pmax0,mean0,&peak0,
                                    &spread0,&posit,&sign0);
                        posi0[row][col]=posit-WING;
                        LMN0_sign[row][col]=sign0;

                        if ((mean90-min90) > (max90-mean90))
                                lmn90=lmnXX(sm90,min90,pmin90,mean90,&peak90,
                                    &spread90,&posit,&sign90);
                        else
                                lmn90=lmnX(sm90,max90,pmax90,mean90,&peak90,
                                    &spread90,&posit,&sign90);
                        posi90[row][col]=posit-WING;
                        LMN90_sign[row][col]=sign90;

                        angle=decision(lmn0,lmn90,LMNX,LMNY,nLMNX,nLMNY,
                            row,col,peak0,peak90,spread0,spread90,sign0,sign90);
                }
                local_mean[row][col]=mean0;
                global_mean=global_mean+mean0;
                LMN0[row][col]=lmn0;
                LMN90[row][col]=lmn90;
                PK0[row][col]=peak0;
                PK90[row][col]=peak90;
                SPR0[row][col]=spread0;
                SPR90[row][col]=spread90;

                wind[row][col]=angle;


                r=r+BX;
                fb_clf(B2,0);
        }
        s=s+BX;
        r=X;
}

r=X;
s=Y;
global_mean=global_mean/(ROW*COL);
for (row=0;row<ROW;row++){
        for (col=0;col<COL;col++){
                if ((local_mean[row][col]+GTH0)< global_mean) wind[row][col]=3;
                if (local_mean[row][col]>(GTH90+global_mean)&&
                     (wind[row][col]==90 || wind[row][col]== -90)) wind[row][col]= 2;
                if (local_mean[row][col] > (GTH0+global_mean) ) wind[row][col]=2;
        }
}
clear_edge();
t0=find_trans(&trans_index);
t90=find_long(&long_index);
if (t0>2) scan_trans(trans_index,LMNX,nLMNX);
if (t90>1) scan_long(long_index,LMNY,nLMNY);
t0=find_trans(&trans_index);
t90=find_long(&long_index);
r=X;
s=Y;
double_count=0;
for (row=0;row<ROW;row++){
        for (col=0;col<COL;col++){
                if (wind_var[row][col]==49) double_count++;
                angle=wind[row][col];
                switch(angle){
                case 4:
                        line0(r,s,posi0[row][col]);
                        window0++;
```

```
                        break;
                case -4:
                        line0(r,s,posi0[row][col]);
                        window0++;
                        break;
                case 90:
                        line90(r,s,posi90[row][col]);
                        window90++;
                        break;
                case -90:
                        line90(r,s,posi90[row][col]);
                        window90++;
                        break;
                case 2:
                        ab_low++;
                        break;
                case 3:
                        ab_high++;
                        break;
                default:
                        break;
                }
                r=r+BX;
        }
        s=s+BX;
        r=X;
}
w0=window0;
w90=window90;
sd0=var2row();
sd90=var2col();
printf("sd0=%f\n",sd0);
printf("sd90=%f\n",sd90);

class=classify(t0,t90,ab_low,ab_high,double_count,w0,w90,sd0,sd90,class_num);


printf("Image  mean  w0  w90   t0  t90   w<m   w>m       class\n");
printf("%s",fname);
printf("   %3d",global_mean);
printf("   %2d",w0);
printf("   %2d",w90);
printf("   %2d",t0);
printf("   %2d",t90);
printf("   %3d",ab_low);
printf("   %3d",ab_high);
printf("        %-s\n",class);
switch(class_num){
case 0:
        INTACT++;
        break;
case 1:
        SHADOW++;
        break;
case 2:
        PAINT++;
        break;
case 3:
        LONG++;
        break;
case 4:
        TRANS++;
        break;
case 5:
        ALLI++;
        break;
```

```
                case 6:
                        INTER++;
                        break;
                }
                printf("\n");
                prints(class);
        }
        printf("TRANSVERSE LONGITUDINAL ALLIGATOR INTACT SHADOW INTERSECT PAINT\n");
        printf("%9d %9d %9d %9d %9d %9d %9d\n",TRANS,LONG,ALLI,INTACT,SHADOW,INTER,PAINT);
}
```

A.2

Program Listing of

CRC.C

```
/*************************************************************
Program : crc.c

CopyRighted : Registration # TX 517 825

Author : TTI Div 2.

Date : Nov 92.

Task: Main program to detect cracking on CRC. Refer to
flowchart of the main block diagram to understand the program
better.

This program is adpated from the ACP program, hotmix.c.
A new image feature was used for classifiying
the spall and the transverse crack on CRC. It is based on the
local variance of the sub-windows that divided up the main image.
The program detects transverse and spall cracks and also
keeps a count of them.

*************************************************************/
#define X 30            /* Upper Right Horizontal Position of AOI */
#define Y 100           /* Upper Right Vertical Position of AOI */
#define BX 48           /* Horizontal Size of Window */
#define BY 48           /* Vertical Size of Window */
#define COL 9           /* Number of Columns */
#define ROW 7           /* Number of Rows */
#define WINDOW 7        /* Number of Pixels used in Smoothing Histogram */
#define OFFSET (int) (WINDOW/2) /* Number of Pixels used for fold back in Projection curve */
#define WING 12         /* Number of extra pixels on either side of window used */
/* in calculating actual spread */

/* Defines used in Decision function
   Refer to Decision rule         */

#define KLMNX 1.1
#define KLMNY 1.5
#define PEAK0 9
#define PEAK90 9

#define GTH0 20         /* Threshold Below Mean for White Paint Mark */
#define GTH90 10        /* Threshold Above Mean for Black Oil Stain */
#define SDTHR 21.0      /* Minimum SD of Detected Edges to be a Crack */
#define MAX_OBJECT COL*ROW
/*#define PRINT_VAR90
#define PRINT_PEAK_SPREAD
#define PRINT_VAR
#define PRINT_MEAN
#define PRINT_LMN*/

struct object{
        int col;
        int y;
};

static int PK0[ROW][COL],PK90[ROW][COL],SPR0[ROW][COL],SPR90[ROW][COL],
LMN0_sign[ROW][COL],LMN90_sign[ROW][COL],wind[ROW][COL],
local_mean[ROW][COL],wind_var[ROW][COL],posi0[ROW][COL],posi90[ROW][COL];
float LMN0[ROW][COL],LMN90[ROW][COL],VAR0[ROW][COL],VAR90[ROW][COL];

int img[512][512];

/* Added for position of the objects */
struct object object0[MAX_OBJECT],object90[MAX_OBJECT];
```

```
/*
Function: Init
Inputs: -
Outputs: -
Task: Initializes the Image Processor.
*/
void init()
{
        load_cfg("/usr/ITEX/150/lib/std.cfg"); /* Load Standard Configuration */
        initsys();                             /* Clear Screen */
        adi_hbanksel(15);
        adi_hgroupsel(RED|GREEN|BLUE);
        adi_clearlut(255);
        adi_lutmode(DYNAMIC);                  /* Activate Dynamic Overlay */
        contour(ADI,GREEN,1,0,156,255);
        linlut(ADI,RED|GREEN|BLUE,0);
}


/*
Function: read_image
Inputs: -
Outputs:-
Task: Reads Image from disk, removes bright spots by averaging them, stores
result in buffer B1.
*/
void read_image()
{
        int i,j;
        int temp;
        BYTE buf[512];

        for(j=0;j<480;j++){                    /* 480 Horizontal lines */
                fb_rhline(B1,0,j,512,buf);     /* 512 Pixels in one line */
                for(i=0;i<512;i++){
                        temp=buf[i];
                        img[i][j]=temp;
                }
        }
}


/*
Function: hist_90
Inputs: -
        x:Upper left horizontal coordinate of window.
        y:Upper left vertical coordinate of window.
Outputs:-
        array: Array containing Projection Values.
        Returns mean of window.
Task: Calculate  Vertical Projection Curve
Sum Values of Gray Level along Vertical Axis
*/
float hist_90(x,y,array)
int x,y,array[BX+2*OFFSET+2*WING];
{
        int i,j,sum,sum0,temp;

        sum0=0;
        for (i=x-WING;i<x+BX+WING;i++){
                sum=0;
                for (j=y;j<y+BY;j++){
                        temp=255-img[i][j];  /* Reverse Gray level */
                        sum=sum+temp;
                }
                temp=sum/BX;
```

```
                array[i-x+OFFSET+WING]=sum/BY;     /* Store averaged values in array of size BY */
                if (i>=x && i< x+BX) sum0=sum0+temp;
        }
        for (i=0;i<=OFFSET;i++){
                array[OFFSET-i]=array[OFFSET+i]; /* Mirror values of array on each end */
                array[BX+2*WING+OFFSET-1+i]=array[BX+2*WING+OFFSET-1-i];
        }
        return((float) sum0/(float) BX);
}


/*
Function: hist_0
Inputs: -
        x:Upper left horizontal coordinate of window.
        y:Upper left vertical coordinate of window.
Outputs:-
        array: Array containing Projection Values.
        Returns mean of window.
Task: Calculate  Horizontal Projection Curve
Sum Values of Gray Level along Horizontal Axis
*/
float hist_0(x,y,array)
int x,y,array[BY+2*OFFSET+2*WING];
{
        int i,j,sum,sum0,temp;

        sum0=0;
        for (j=y-WING;j<y+BY+WING;j++){
                sum=0;
                for (i=x;i<x+BX;i++){
                        temp=255-img[i][j];  /* Reverse Gray level */
                        sum=sum+temp;
                }
                temp=sum/BX;
                array[j-y+OFFSET+WING]=temp;      /* Store averaged values in array of size BY */
                if (j>=y && j<y+BY) sum0=sum0+temp;
        }
        for (i=0;i<=OFFSET;i++){
                array[OFFSET-i]=array[OFFSET+i]; /* Mirror values of array on each end */
                array[BY+2*WING+OFFSET-1+i]=array[BY+2*WING+OFFSET-1-i];
        }
        return((float) sum0/(float) BY);
}


/*
Function: draw_hist
Inputs: -
        x: Coordinate x on screen where plot is desired
        y: Coordinate y on screen where plot is desired
        array: Array containing Projection Values.
        mean: Mean of array input.
Outputs: -
Task: Routine to Plot the Projection Histogram on the Screen.
Also Plots a line indicating mean.
*/
void draw_hist(x,y,array,mean)
int x,y,array[BX+2*WING];
{
        int i,j;
        for (i=0;i<BX+2*WING;i++){
                j=(array[i]*160)/160;
                line(B1,1,(x+i),y,(x+i),(y+j),255);
        }
        line(B1,1,x,y+mean,x+BX,y+mean,100);
```

```
}


/*
Function: clear_hist
Inputs: -
        x: Coordinate x on screen where plot is desired
        y: Coordinate y on screen where plot is desired
Outputs: -
Task: Routine to clear the Projection Histogram on the Screen.
so that the next plot does not overwrite on the previous one.
*/
void clear_hist(x,y)
int x,y;
{
        int i,j;
        for (i=0;i<BX;i++){
                j=160;
                line(B1,1,(x+i),y,(x+i),(y+j),0);
        }
}



/*
Function: line_90
Inputs: -
        x: Upper Left x coordinate of window.
        x: Upper Left y coordinate of window.
        pos90: Position of detected vertical edge.
Outputs: -
Task: Routine to Plot a line on the screen indicating the
position of detected edge.
*/
void line90(x,y,pos90)
int x,y,pos90;
{
        line(B1,0,(x+pos90),y,(x+pos90),y+BX-1,255);
}


/*
Function: line_0
Inputs: -
        x: Upper Left x coordinate of window.
        x: Upper Left y coordinate of window.
        pos0: Position of detected horizontal edge.
Outputs: -
Task: Routine to Plot a line on the screen indicating the
position of detected edge.
*/
void line0(x,y,pos0)
int x,y,pos0;
{
        line(B1,0,x,(y+pos0),(x+BX-1),(y+pos0),255);
}

/*
Function: curve_smooth
Inputs: -
        rawx: Unsmoothed array containing horizontal projection values.
        rawy: Unsmoothed array containing vertical projection values.
Outputs: -
        smoothx: Smoothed array containing horizontal projection values.
        smoothy: Smoothed array containing vertical projection values.
```

```
        maxx: Maximum value in x projection array.
        maxy: Maximum value in y projection array.
        minx: Minimum value in x projection array.
        miny: Minimum value in y projection array.
        pmaxx: Position of maxx in array.
        pmaxy: Position of maxy in array.
        pminx: Position of minx in array.
        pminy: Position of miny in array.
Task: Smooth both horizontal and vertical projection values
also calculate the max and min values in the array along with
their respective positions in the array.
*/
void curve_smooth(rawx,smoothx,rawy,smoothy,maxx,
minx,pmaxx,pminx,maxy,miny,pmaxy,pminy)
int rawx[BY+2*OFFSET+2*WING],smoothx[BY+2*WING],rawy[BY+2*OFFSET+2*WING],smoothy[BY+2*WING];
int *maxx,*minx,*pmaxx,*pminx,*maxy,*miny,*pmaxy,*pminy;
{
        int sumx,avgx,sumy,avgy;
        int Head,Tail,center,count;

        for(center=OFFSET;center<BY+2*WING+OFFSET;center++){
                sumx=0;
                sumy=0;
                avgx=0;
                avgy=0;
                Head=center-OFFSET;
                Tail=center+OFFSET;
                for(count=0;count<OFFSET;count++){
                        sumx +=rawx[Head];
                        sumx +=rawx[Tail];
                        sumy +=rawy[Head];
                        sumy +=rawy[Tail];
                        Head++;
                        Tail--;
                }
                sumx=sumx+rawx[center];
                sumy=sumy+rawy[center];
                avgx=sumx/WINDOW;
                avgy=sumy/WINDOW;
                smoothx[center-OFFSET]=avgx;
                smoothy[center-OFFSET]=avgy;
                if (center==OFFSET+WING){
                        *maxx=*minx=avgx;
                        *maxy=*miny=avgy;
                        *pmaxx=*pmaxy=center-OFFSET;
                        *pminx=*pminy=center-OFFSET;
                }
                if(center >= OFFSET+WING && center < BY+OFFSET+WING)
                {
                        if (avgx>=*maxx){
                                *maxx=avgx;
                                *pmaxx=center-OFFSET;
                        }
                        if (avgx<=*minx){
                                *minx=avgx;
                                *pminx=center-OFFSET;
                        }
                        if (avgy>=*maxy){
                                *maxy=avgy;
                                *pmaxy=center-OFFSET;
                        }
                        if (avgy<=*miny){
                                *miny=avgy;
                                *pminy=center-OFFSET;
                        }
                }
```

```
        }
}


/*
Function: lmnY
Inputs: -
        array:Smoothed array of vertical projection.
        max: Maximum vlaue in array.
        pmax: Position of max.
        mean: Mean of array.
Outputs: -
        peak: peak
        denom: Spread.
        posit: position of crack
        sign: positive
        Returns lmn
Task:
Calculate the lmn=peak/spread=(max-mean)/spread
for the vertical projection values for positive peaks.
*/
float lmnY(array,max,pmax,mean,peak,denom,posit,sign)
int mean,max,*peak,*denom,*posit,*sign,array[BY+2*WING];
{
        int i,maxi,mmi;
        float result;

        maxi=BX+2*WING-1;
        mmi=0;
        for (i=pmax;i<BX+2*WING;i++) /* Find spread to the right of position of maximum */
                if (array[i]<=mean){
                        maxi=i;
                        break;
                }
        for (i=pmax;i>=0;i--) /* Find spread to the left of position of maximum */
                if (array[i]<=mean){
                        mmi=i;
                        break;
                }
        *peak=max-mean;
        *denom=maxi-mmi;
        *posit=pmax;
        *sign=1;

        result= (float)(max-mean)/(float)(*denom);
        if (*denom>30 || *denom<7) result=0.0;          /* If spread<7 or spread> 30 set lmn=0 */

        return(result);
}

/*
Function: lmnYY
Inputs: -
        array:Smoothed array of vertical projection.
        min: Minimum value in array.
        pmin: Position of min.
        mean: Mean of array.
Outputs: -
        peak: peak
        denom: Spread.
        posit: position of peak
        sign: negative
        Returns lmn
Task:
Calculate the lmn=peak/spread=(mean-min)/spread
```

```
for the vertical projection values for negative peaks.
*/
float lmnYY(array,min,pmin,mean,peak,denom,posit,sign)
int mean,min,pmin,*peak,*denom,*posit,*sign,array[BY+2*WING];
{
        int i,mini,mni;
        float result;

        mini=BX+2*WING-1;
        mni=0;
        for (i=pmin;i<BX+2*WING;i++)  /* Find spread to the right of position of minimum */
                if (array[i]>=mean){
                        mini=i;
                        break;
                }
        for (i=pmin;i>=0;i--)  /* Find spread to the left of position of minimum */
                if (array[i]>=mean){
                        mni=i;
                        break;
                }

        *peak=mean-min;
        *denom=mini-mni;
        *posit=pmin;
        *sign=-1;

        result= (float)(mean-min)/(float)(*denom);      /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;

        return(result);
}


/*
Function: lmnX
Inputs: -
        array:Smoothed array of horizontal projection.
        max: Maximum value in array.
        pmax: Position of max.
        mean: Mean of array.
Outputs: -
        peak: Peak
        denom: Spread.
        Posit: Position of peak.
        sign: Positive
        Returns lmn
Task:
Calculate the lmn=peak/spread=(max-mean)/spread
for the horizontal projection values for positive peaks.
*/
float lmnX(array,max,pmax,mean,peak,denom,posit,sign)
int mean,max,*peak,*denom,*posit,*sign,array[BX+2*WING];
{
        int i,maxi,mmi;
        float result;

        maxi=BX+2*WING-1;
        mmi=0;
        for (i=pmax;i<BX+2*WING;i++)  /* Find spread to the right of position of maximum */
                if (array[i]<=mean){
                        maxi=i;
                        break;
                }
        for (i=pmax;i>=0;i--)  /* Find spread to the left of position of maximum */
                if (array[i]<=mean){
                        mmi=i;
```

```
                        break;
                }
        *peak=max-mean;
        *denom=maxi-mmi;
        *posit=pmax;
        *sign=1;

        result= (float)(max-mean)/(float)(*denom);      /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;
        return(result);
}


/*
Function: lmnXX
Inputs: -
        array:Smoothed array of horizontal projection.
        min: Minimum value in array.
        pmin: Position of min.
        mean: Mean of array.
Outputs: -
        peak: Peak
        denom: Spread.
        posit: Position of peak.
        sign: Negative
        Returns lmn
Task:
Calculate the lmn=peak/spread=(mean-min)/spread
for the horizontal projection values for negative peaks.
*/
float lmnXX(array,min,pmin,mean,peak,denom,posit,sign)
int mean,min,pmin,*peak,*denom,*posit,*sign,array[BX+2*WING];
{
        int i,mini,mni;
        float result;

        mini=BX+2*WING-1;
        mni=0;
        for (i=pmin;i<BX+2*WING;i++) /* Find spread to the right of position of minimum */
                if (array[i]>=mean){
                        mini=i;
                        break;
                }
        for (i=pmin;i>=0;i--) /* Find spread to the left of position of minimum */
                if (array[i]>=mean){
                        mni=i;
                        break;
                }

        *peak=mean-min;
        *denom=mini-mni;
        *posit=pmin;
        *sign=-1;

        result= (float)(mean-min)/(float)(*denom);      /* If spread<7 or spread> 30 set lmn=0 */
        if (*denom>30 || *denom<7) result=0.0;
        return(result);
}

/*
Function: varX
Inputs: -
        array:Smoothed array of horizontal projection.
        mean: Mean of array.
Outputs: -
        Returns variance of array
```

```
Task:
Calculate the variance of the horizontal projection.
*/
float varX(array,mean)
int array[BX+2*OFFSET+2*WING];
float mean;
{
        int i;
        float sum,temp;

        sum=0.0;
        for (i=0;i<BX;i++){
                temp=(float) array[i+OFFSET+WING]-mean;
                sum=sum+temp*temp;
        }
        return((float)(sum)/(float)(BX-1));
}

/*
Function: varY
Inputs: -
        array:Smoothed array of vertical projection.
        mean: Mean of array.
Outputs: -
        Returns variance of array
Task:
Calculate the variance of the vertical projection.
*/
float varY(array,mean)
int array[BY+2*OFFSET+2*WING];
float mean;
{
        int i;
        float sum,temp;

        sum=0.0;
        for (i=0;i<BY;i++){
                temp=(float) array[i+OFFSET+WING]-mean;
                sum=sum+temp*temp;
        }
        return((float)(sum)/(float)(BY-1));
}



/*
Function: prints
Inputs: -
        type: pointer to string.
Outputs: -
Task:
Print string (ALLIGATOR etc) on screen.
*/
void prints(type)
char *type;
{
        text(B1,0,70,100,HORIZONTAL,2,255,type);
}


/*
Function: decision
Inputs: -
        lmn0:lmn for horizontal projection.
        lmn90: lmn for vertical projection.
        LMNX: Minimum threshold for horizontal projection.
```

```
        LMNY: Minimum threshold for vertical projection.
        i: Horizontal index of window.
        j: Vertical index of window.
        peak0:
        peak90:
        spread0:
        spread90:
        sign0:
        sign90:
Outputs: -
        Returns  angle
        angle 0: no edge
        angle 4:  horizontal edge
        angle 90: vertical edge
Task:
Decides if the window contains an edge based on the
information available. Returns an angle.
*/
int decision(lmn0,lmn90,LMNX,LMNY,nLMNX,nLMNY,i,j,peak0,peak90,spread0,spread90,sign0,sign90)
float lmn0,lmn90,LMNX,LMNY,nLMNX,nLMNY;
int i,j,peak0,peak90,spread0,spread90,sign0,sign90;
{
        int angle;
        float temp_lmn0,temp_lmn90;

        angle=1;
        if (sign0==1) temp_lmn0=LMNX;
        else temp_lmn0=nLMNX;
        if (sign90==1) temp_lmn90=LMNY;
        else temp_lmn90=nLMNY;

        if (lmn0>temp_lmn0 && lmn0 > (KLMNX*lmn90) && (peak0 > PEAK0) &&
            (wind_var[i][j]==4 || wind_var[i][j]==49))angle=4*sign0;

        if (lmn90>temp_lmn90 && lmn90 > (KLMNY*lmn0) && (peak90 > PEAK90 && peak90<40) &&
            (wind_var[i][j]==90 || wind_var[i][j]==49) && peak90
            > 1.5*peak0 && spread90 > 5 ) angle=90*sign90;
        return(angle);
}



/*
Function: var2col
Inputs: -
Outputs: -
        Returns  smallest SD of edges in a pair of cols.
Task:
Determines if the edges detected fall in a straight line
indicating a longitudinal crack. Checks overlapping
pairs of cols calculating the SD of the edges. Returns SD.
If the number of edges in any col > 4, SD is calculated
for that col only.
*/
float var2col()
{
        int sum[COL-1];
        int i,j,k;
        int sumup,count1,count2;
        int var1,temp,temp1;
        static int data1[ROW],data2[ROW];
        float sd1,SD;

        SD=100.0;
        for (j=0;j<COL-1;j++){ /* Count the number of edges in every pair of cols */
```

```
                sumup=0;
                for (i=0;i<ROW;i++){
                        if (abs(wind[i][j])==90) sumup++;
                        if (abs(wind[i][j+1])==90) sumup++;
                }
                sum[j]=sumup;
        }


        for (j=0;j<COL-1;j++){
                count1=0;
                count2=0;
                if (sum[j]>2){        /* If the number of edges in the pair is >2 */
                        for (i=0;i<ROW;i++){
                                if (abs(wind[i][j])==90){ /* find number of edges in each row */
                                        data1[count1]=posi90[i][j];
                                        count1++;
                                }
                                if (abs(wind[i][j+1])==90){
                                        data2[count2]=BX+posi90[i][j+1];
                                        count2++;
                                }
                        }
                        if (count1>4){ /* If number of edges in row is > 4 */
                                temp=temp1=0; /* find SD for the row */
                                for (k=0;k<count1;k++){
                                        temp=temp+data1[k]*data1[k];
                                        temp1=temp1+data1[k];
                                }
                                var1=(temp-(temp1*temp1/count1))/(count1-1);
                                sd1=(float) sqrt((double) var1);
                                if (sd1<SD) SD=sd1;
                        }
                        if (count2>4){
                                temp=temp1=0;
                                for (k=0;k<count2;k++){
                                        temp=temp+data2[k]*data2[k];
                                        temp1=temp1+data2[k];
                                }
                                var1=(temp-(temp1*temp1/count2))/(count2-1);
                                sd1=(float) sqrt((double) var1);
                                if (sd1<SD) SD=sd1;
                        }
                        temp=temp1=0;
                        for (k=0;k<count1;k++){
                                temp=temp+data1[k]*data1[k];
                                temp1=temp1+data1[k];
                        }
                        for (k=0;k<count2;k++){
                                temp=temp+data2[k]*data2[k];
                                temp1=temp1+data2[k];
                        }
                        var1=(temp-(temp1*temp1/(count1+count2)))/((count1+count2)-1);
                        sd1=(float) sqrt((double) var1);
                        if (sd1<SD) SD=sd1;
                }
        }
        return(SD);
}



/*
Function: var2row
Inputs: -
```

```
Outputs: -
        Returns  smallest SD of edges in a pair of rows.
Task:
Determines if the edges detected fall in a straight line
indicating a transverse crack. Checks overlapping
pairs of rows calculating the SD of the edges. Returns SD.
*/
float var2row()
{
        int sum[ROW-1];
        int i,j,k;
        int sumup,count;
        int var1,temp,temp1;
        static int data[COL+COL];
        float sd1,SD;

        SD=100.0;
        for (i=0;i<ROW-1;i++){
                sumup=0;
                for (j=0;j<COL;j++){
                        if (abs(wind[i][j])==4) sumup++;
                        if (abs(wind[i+1][j])==4) sumup++;
                }
                sum[i]=sumup;
        }


        for (i=0;i<ROW-1;i++){
                count=0;
                if (sum[i]>3){
                        for (j=0;j<COL;j++){
                                if (abs(wind[i][j])==4){
                                        data[count]=posi0[i][j];
                                        count++;
                                }
                                if (abs(wind[i+1][j])==4){
                                        data[count]=BX+posi0[i+1][j];
                                        count++;
                                }
                        }
                        temp=temp1=0;
                        for (k=0;k<count;k++){
                                temp=temp+data[k]*data[k];
                                temp1=temp1+data[k];
                        }
                        var1=(temp-(temp1*temp1/count))/(count-1);
                        sd1=(float) sqrt((double) var1);
                        if (sd1<SD) SD=sd1;
                }
        }

        return(SD);
}


/*
Function: clear_edge
Inputs: -
Outputs: -
Task:
Clear false edges detected around oilstain in 4 directions
to prevent a misclassification.
*/
void clear_edge()
{
```

```
        int i,j;

        for (i=0;i<ROW;i++){
                for (j=0;j<COL;j++){
                        if (wind[i][j]==2){
                                /*if (i>0 && wind[i-1][j]!=2) wind[i-1][j]=5;
                                if (i<ROW-1 && wind[i+1][j]!=2) wind[i+1][j]=5;*/
                                if (j>0 && wind[i][j-1]!=2) wind[i][j-1]=5;
                                if (j<COL-1 && wind[i][j+1]!=2) wind[i][j+1]=5;
                        }
                }
        }
}


/*
Function: scan_trans
Inputs: -
        trans_index: Index of row where second scan needs to be done
        LMNX: LMN threshold for positive cracks.
        LMNY: LMN threshold for negative cracks.
Outputs: -
Task:
Perform a second scan using a different decision rule in order
to pick up edges missed in the first run for transverse cracks.
*/
void scan_trans(trans_index,LMNX,nLMNX)
int trans_index;
float LMNX,nLMNX;
{
        int i,j;
        float temp;

        for (i=trans_index;i<ROW && i<trans_index+2;i++){
                for (j=0;j<COL;j++){
                        if (LMN0_sign[i][j]==-1) temp=nLMNX;
                        else temp=LMNX;
                        if ((wind_var[i][j]==4 || wind_var[i][j]==49) && LMN0[i][j]>temp &&
                            wind[i][j]==1) wind[i][j]= 4;
                }
        }
}



/*
Function: scan_long
Inputs: -
        long_index: Index of col where second scan needs to be done
        LMNX: LMN threshold for positive cracks.
        LMNY: LMN threshold for negative cracks.
Outputs: -
Task:
Perform a second scan using a different decision rule in order
to pick up edges missed in the first run for longitudinal cracks.
*/
void scan_long(long_index,LMNY,nLMNY)
int long_index;
float LMNY,nLMNY;
{
        int i,j;
        float temp;

        for (j=long_index;j<COL && j<long_index+2;j++){
                for (i=0;i<ROW;i++){
```

```
                        if (LMN90_sign[i][j]==-1) temp=nLMNY;
                        temp=LMNY;
                        if ((wind_var[i][j]==90||wind_var[i][j]==49) && LMN90[i][j]>temp &&
                            wind[i][j]==1 && PK90[i][j] > 1.5*PK0[i][j] &&
                            SPR90[i][j]> 5) wind[i][j]= 90;
                }
        }
}


/*
Function: find_long
Inputs: -
Outputs: -
        index: Index of col having maximum number of 90  edges.
        Returns number of edges in the col.
Task:
Find index of col having maximum number of 90 edges from the edge map.
*/
int find_long(index)
int *index;
{
        int sum,i,j,MAX;

        MAX=0;
        *index=0;
        for (j=0;j<COL-1;j++){
                sum=0;
                for (i=0;i<ROW;i++){
                        if ((abs(wind[i][j]))==90) sum++;
                        if ((abs(wind[i][j+1]))==90) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                        *index=j;
                }
        }
        return(MAX);
}


/*
Function: find_long_var
Inputs: -
Outputs: -
        Returns index of col with max number of 90 var windows.
Task:
Find index of col having maximum number of black windows( 90 windows)
from the variance map.
*/
int find_long_var()
{
        int sum,i,j,MAX;

        MAX=0;
        for (j=0;j<COL;j++){
                sum=0;
                for (i=0;i<ROW;i++){
                        if ((abs(wind_var[i][j]))==90 || (abs(wind_var[i][j]))==49) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                }
        }
        return(MAX);
}
```

```
/*
Function: find_trans
Inputs: -
Outputs: -
        index: Index of row having maximum number of 4 ( horizontal )  edges.
        Returns number of edges in the row.
Task:
Find index of row having maximum number of 4 edges (horizontal) from the edge map.
*/
int find_trans(index)
int *index;
{
        int sum,i,j,MAX;

        MAX=0;
        *index=0;
        for (i=0;i<ROW-1;i++){
                sum=0;
                for (j=0;j<COL;j++){
                        if ((abs(wind[i][j]))==4) sum++;
                        if ((abs(wind[i+1][j]))==4) sum++;
                }
                if (sum>MAX){
                        MAX=sum;
                        *index=i;
                }
        }
        return(MAX);
}



void main()
{
        long start, finish;
        static int unsm0[BY+offset2+2*WING],unsm90[BX+offset2+2*WING],
        sm0[BY+2*WING],sm90[BX+2*WING];
        int max0,max90,mean;
        int pmax0,pmax90,pmin0,pmin90,min0,min90;
        float lmn0,lmn90,var0,var90,mean1;
        char *dirname1 = "/export/CRC/us59",
        *dirname2 = "/export/images2/canon",
        *dirname3 = "/export/images2/test",
        *dirname4 = "/export/images2/sh6",
        *dirname5 = "/export/images2/aran",
        *dirname6 = "/export/images3";
        char *dirname="",in;
        FILE *input_file,*fopen(),*shape;
        char fname[20],name[20];
        int len;
        int r,s;
        int i,j;
        float nLMNX,nLMNY,LMNX,LMNY;
        int row,col;
        int angle,window90,window0;
        char *class="";
        int class_num;
        float sd0,sd90;
        int t0,t90,w0,w90,global_mean,ab_low,ab_high,posit;
        static int SHADOW,PAINT,TRANS,LONG,ALLI,INTACT,INTER,SPALL;
        int sign,trans_index,long_index;
        int peak0,peak90;
        int spread0,spread90,temp;
```

```
float tempp;
int double_count;
int var0count;
int var90count;
/* These variable is for calculating the variance of var90 and var0 */
double sum0=0.0, sum90=0.0, ssum0=0.0, ssum90=0.0;
double varvar0=0.0, varvar90=0.0, sdvar, local_var;
/* Dealing objects */
int objCount;

init();
fb_clf(B1,110);
text(B1,0,45,90,HORIZONTAL,3,200,"Distress Analysis");
text(B1,0,235,220,HORIZONTAL,2,200,"on");
text(B1,0,90,300,HORIZONTAL,2,200,"Concrete Pavements");
printf("\nDirectory 0 - 6 :");
in = getchar();
switch(in) {
case '1':
        dirname=dirname1;
        break;
case '2':
        dirname =dirname2;
        break;
case '3':
        dirname =dirname3;
        break;
case '4':
        dirname =dirname4;
        break;
case '5':
        dirname =dirname5;
        break;
case '6':
        dirname =dirname6;
        break;
case '0':
        printf("New Directory :");
        scanf("%s",dirname);
        getchar();
}
input_file = fopen("images","r");
chdir(dirname);
getchar();

start = clock();
while (fgets(name,60,input_file)!=NULL){
        getchar();
        objCount = 0;
        var0count = 0;
        var90count = 0;
        len = strlen(name);
        strcpy(fname,"");
        strncat(fname,name,(len-1));
        shape=fopen("/home/1189/pchan/crack/lmn","r");
        fscanf(shape,"%f %f %f %f",&LMNX,&LMNY,&nLMNX,&nLMNY);
        fclose(shape);
        r=X;
        s=Y;
        im_read(B1,0,0,512,512,fname);
        read_image();
        getchar();
        w0=w90=global_mean=ab_low=ab_high=0;
        sum0=0.0, sum90=0.0, ssum0=0.0, ssum90=0.0;
        for (row=0;row<ROW;row++){
                for (col=0;col<COL;col++){
```

```
                                rectangle(B2,0,r,s,BX,BY,15);
                                angle=1;
                                mean1=hist_0(r,s,unsm0);
                                var0=varX(unsm0,mean1);
                                hist_90(r,s,unsm90);
                                var90=varY(unsm90,mean1);
                                mean=(int) mean1;
                                wind_var[row][col]=1;
                                VAR0[row][col]=var0;
                                if (VAR0[row][col] > 150.0)
                                {
                                        var0count++;
                                        /*printf("VAR0 (%d,%d) = %f\n",row,col,VAR0[row][col]);*/
                                }
                                VAR90[row][col]=var90;
#ifdef PRINT_VAR90
                                if (VAR90[row][col] > 200.0)
                                {
                                        var90count++;
                                        /*printf("VAR90 (%d,%d) = %f\n",row,col,VAR90[row][col]);*/
                                }
#endif PRINT_VAR90
                                lmn0=lmn90=0.0;
                                peak0=peak90=spread0=spread90=0;
                                if (var0 > 15.0 && var90< 500.0){
                                        wind_var[row][col]=4;
                                        rectangle(B1,0,r,s,BX,BY,170);
                                }
                                if (var90 >80.0 && var90 <500.0 && var0< 500.0){
                                        if (wind_var[row][col]==4) wind_var[row][col]=49;
                                        else wind_var[row][col]=90;
                                        rectangle(B1,0,r,s,BX-5,BY-5,70);
                                }
                                if ((var0 > 15.0 && var90< 500.0)
                                    || (var90 >80.0 && var90 <500.0 && var0<500.0)){
                                        curve_smooth(unsm0,sm0,unsm90,sm90,&max0,&min0,
                                            &pmax0,&pmin0,&max90,&min90,&pmax90,&pmin90);
                                        if ((mean-min0) > (max0-mean)){
                                LMN0[row][col]=lmnYY(sm0,min0,pmin0,mean,&SPR0[row][col]);
                                                /*posit=pmin0-WING;*/
                                                posi0[row][col]=pmin0-WING;
                                                sign=-1;
                                                /*peak0=mean-min0;*/
                                                PK0[row][col]=mean-min0;
                                        /*printf("Local Variance     =   %f\n",LMN0[row][col]);
        printf("Negative peak lmnYY Spread0  = %d Peak0  = %d\n",SPR0[row][col],PK0[row][col]);*/
                                        }
                                        else{
                        LMN0[row][col]=lmnY(sm0,max0,pmax0,mean,&SPR0[row][col]);
                                                /*posit=pmax0-WING;*/
                                                posi0[row][col]=pmax0-WING;
                                                sign=1;
                                                /*peak0=max0-mean;*/
                                                PK0[row][col]=max0-mean;
                                        /*printf("Local Variance     =   %f\n",LMN0[row][col]);*/
        /*printf("Positive peak lmnY Spread0  = %d Peak0  = %d\n",SPR0[row][col],PK0[row][col]);*/
                                        }
                                        /*posi0[row][col]=posit;*/

                                        LMN0_sign[row][col]=sign;

                                        if ((mean-min90) > (max90-mean)){
                        LMN90[row][col]=lmnXX(sm90,min90,pmin90,mean,&SPR90[row][col]);
                                                /*posit=pmin90-WING;*/
                                                posi90[row][col]=pmin90-WING;
                                                sign= -1;
```

```
                                        /*peak90=mean-min90;*/
                                        PK90[row][col]=mean-min90;
                                /*printf("Local Variance     =   %f\n",LMN90[row][col]);*/
/*printf("Negative peak lmnXX Spread90 = %d Peak90 = %d\n",SPR90[row][col],PK90[row][col]);*/
                                        }
                                        else{
                        LMN90[row][col]=lmnX(sm90,max90,pmax90,mean,&SPR90[row][col]);
                                        /*posit=pmax90-WING;*/
                                        posi90[row][col]=pmax90-WING;
                                        sign=1;
                                        /*peak90=max90-mean;*/
                                        PK90[row][col]=max90-mean;
                        /*printf("Local Variance     =   %f\n",LMN90[row][col]);*/
/*printf("Positive peak lmnX Spread90 = %d Peak90 = %d\n",SPR90[row][col],PK90[row][col]);*/
                                        }
                                        /*posi90[row][col]=posit;*/
                                        objCount++;

                                        LMN90_sign[row][col]=sign;

                                        /*clear_hist(0,0);
                                        clear_hist(90,0);
                                        draw_hist(0,0,sm0,mean);
                                        draw_hist(90,0,sm90,mean);
                                        getchar();*/

                                        angle=decision(LMN0[row][col],LMN90[row][col],
                                            LMNX,LMNY,nLMNX,nLMNY,mean,
                                            max0,max90,min0,min90,
                                            PK0[row][col],PK90[row][col],
                                            SPR0[row][col],SPR90[row][col],row,col);
                                }
                                local_mean[row][col]=mean;
                                global_mean+=mean;
                                /*LMN0[row][col]=lmn0;*/
                                /*LMN90[row][col]=lmn90;*/
                                /*PK0[row][col]=peak0;*/
                                /*PK90[row][col]=peak90;*/
                                /*SPR0[row][col]=spread0;*/
                                /*SPR90[row][col]=spread90;*/

                                wind[row][col]=angle;


                                r+=BX;
                                fb_clf(B2,0);
                        }
                        s+=BX;
                        r=X;
                }

                for(col=0;col<COL;col++)
                        for(row=0;row<ROW;row++)
                        {
                                sum0 += (double)VAR0[row][col];
                                sum90 += (double)VAR90[row][col];
                                ssum0 += (double)VAR0[row][col] * VAR0[row][col];
                                ssum90 += (double)VAR90[row][col] * VAR90[row][col];
                        }
                sum0 /= (double)(col*row);
                sum90 /= (double)(col*row);
                ssum0 /= (double)(col*row);
                ssum90 /= (double)(col*row);
                varvar0 = ssum0 - sum0*sum0;
                varvar90 = ssum90 - sum90*sum90;
                sdvar = sqrt(varvar0*varvar0 + varvar90*varvar90);
```

```
r=X;
s=Y;
global_mean/=(ROW*COL);
for (row=0;row<ROW;row++){
        for (col=0;col<COL;col++){
                if ((local_mean[row][col]+GTH0)< global_mean) wind[row][col]=3;
                if (local_mean[row][col]>(GTH90+global_mean)&&
                    (wind[row][col]==90 || wind[row][col]== -90)) wind[row][col]= 2;
                if (local_mean[row][col] > (GTH0+global_mean) ) wind[row][col]=2;
        }
}
edge_2clear();
t0=find_trans(&trans_index);
t90=find_long(&long_index);
/*printf("\nLocal var0\n");
for (i=0;i<ROW;i++){
        for (j=0;j<COL;j++){
                printf("%5.1f  ",VAR0[i][j]);
        }
        printf("\n");
}
printf("\nLocal var90\n");
for (i=0;i<ROW;i++){
        for (j=0;j<COL;j++){
                printf("%5.1f  ",VAR90[i][j]);
        }
        printf("\n");
        }

printf("\nLOCAL SQRTVAR\n");
for (row=0;row<ROW;row++){
        for (col=0;col<COL;col++){
local_var=sqrt(VAR0[row][col]*VAR0[row][col]+VAR90[row][col]*VAR90[row][col]);
                printf("%5.1f  ",local_var);
        }
        printf("\n");
}*/
if (t0>2) scan_trans(trans_index,LMNX,nLMNX);
if (t90>1) scan_long(long_index,LMNY,nLMNY);
t0=find_trans(&trans_index);
t90=find_long(&long_index);
r=X;
s=Y;
double_count=0;
for (row=0;row<ROW;row++){
        for (col=0;col<COL;col++){
                if (wind_var[row][col]==49) double_count++;
                angle=wind[row][col];
                switch(angle){
                case 4:
                        line0(r,s,posi0[row][col]);
                        w0++;
                        break;
                case -4:
                        line0(r,s,posi0[row][col]);
                        w0++;
                        break;
                case 90:
                        line90(r,s,posi90[row][col]);
                        w90++;
                        break;
                case -90:
                        line90(r,s,posi90[row][col]);
                        w90++;
                        break;
```

```
                case 2:
                        ab_low++;
                        break;
                case 3:
                        ab_high++;
                        break;
                default:
                        break;
                }
                r+=BX;
        }
        s+=BX;
        r=X;
}
/*w0=window0;
w90=window90;*/
sd0=var2row();
sd90=var2col();
/*printf("sd0=%f\n",sd0);
printf("sd90=%f\n",sd90);*/

/* Classification */
if (ab_low > 5 || ab_high > 10){
        if (t90>2 && sd90<SDTHR && double_count <8){
                class="LONGITUDINAL";
                class_num=3;
        }
        else if (t0>3 && sd0<SDTHR && double_count <8){
                if(sdvar > 950){
                        class="SPALLED CRACK";
                        class_num=7;
                }
                else
                {
                        class="TRANSVERSE CRACKING";
                        class_num=4;
                }
        }
        else if ((w0+w90)>5){
                if(sdvar > 950){
                        class="SPALLED CRACK";
                        class_num=7;
                }
                else
                {
                        class="TRANSVERSE CRACKING";
                        class_num=4;
                }
                /*class="ALLIGATOR CRACKING";
                class_num=5;*/
        }
        else{
                class="INTACT";
                class_num=1;
        }
}
else if(w0<4 && w90<3){
        class="INTACT";
        class_num=0;
}
else if (t0>3 && t90>2 && sd0<SDTHR && sd90<SDTHR && double_count <8){
        class="INTERSECT";
        class_num=6;
}
else if (t90>2 && sd90<SDTHR && double_count <8){
        class="LONGITUDINAL CRACKING";
```

```
        class_num=3;
}
else if (t0>3 && sd0<SDTHR && double_count <8){
        if(sdvar > 950)
        {
                class="SPALLED CRACK";
                class_num=7;
        }
        else
        {
                class="TRANSVERSE CRACKING";
                class_num=4;
        }
}
else if ((w0+w90)>5){
        /*if (find_block()==1) class="BLOCK CRACKING";
                {   class="TRANSVERSE CRACKING";
                    class_num = 4;
                }*/
        if(sdvar > 950){
                class="SPALLED CRACK";
                class_num=7;
        }
        else
        {
                class="TRANSVERSE CRACKING";
                class_num=4;
        }
        /*class="ALLIGATOR CRACKING";
                class_num=5;*/
}
else{
        class="INTACT";
        class_num=0;
}


printf("Image        mean   w0  w90   t0  t90   w<m   w>m       class\n");
printf("%s",fname);
printf("   %3d",global_mean);
printf("   %2d",w0);
printf("   %2d",w90);
printf("   %2d",t0);
printf("   %2d",t90);
printf("   %3d",ab_low);
printf("   %3d",ab_high);
printf("        %-s\n",class);
switch(class_num){
case 0:
        INTACT++;
        break;
case 1:
        SHADOW++;
        break;
case 2:
        PAINT++;
        break;
case 3:
        LONG++;
        break;
case 4:
        TRANS++;
        break;
case 5:
        ALLI++;
        break;
```

```
                case 6:
                        INTER++;
                        break;
                case 7:
                        SPALL++;
                        break;
                }
                /*printf("Variance of VAR0  = %lf\n",varvar0);
                printf("Variance of VAR90 = %lf\n",varvar90);
                printf("Sqrt of the sum of vavarvar = %lf\n",sdvar);*/
#ifdef PRINT_MEAN
                printf("\nMEAN 0\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",local_mean[i][j]);
                        }
                        printf("\n");
                }
#endif PRINT_MEAN
#ifdef PRINT_VAR
                tempp=0.0;
                temp=0;
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%5.1f  ",VAR0[i][j]);
                                if (VAR0[i][j] < 100.0){
                                        tempp=tempp+VAR0[i][j];
                                        temp++;
                                }
                        }
                        printf("\n");
                }
#endif PRINT_VAR
#ifdef PRINT_VAR
                printf("var0=%f\n",tempp /(temp));
                tempp=0.0;
                temp=0;
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%5.1f  ",VAR90[i][j]);
                                if (VAR90[i][j] < 100.0){
                                        tempp=tempp+VAR90[i][j];
                                        temp++;
                                }
                        }
                        printf("\n");
                }
#endif PRINT_VAR
#ifdef PRINT_VAR
                printf(" var90=%f\n",tempp /(temp));
                printf("\n var0 + var90\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%5.1f  ",VAR0[i][j]+VAR90[i][j]);
                        }
                        printf("\n");
                }
#endif PRINT_VAR
#ifdef PRINT_LMN
                printf("II PASS \n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",wind[i][j]);
                        }
                        printf("\n");
                }
```

```
                printf("LMN 0 \n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%4.2f ",LMN0[i][j]);
                        }
                        printf("\n");
                }
                printf("LMN 90 \n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%4.2f ",LMN90[i][j]);
                        }
                        printf("\n");
                }
#endif PRINT_LMN
                /*printf("-----------------------------------\n");
                for (j=0;j<COL;j++){
                        tempp=0.0;
                        temp=0;
                        for (i=0;i<ROW;i++){
                                if (LMN90[i][j]!=0.0) temp++;
                                tempp=tempp+LMN90[i][j];
                        }
                        if (temp!=0) printf("%4.2f ",tempp/temp);
                        else printf("%4.2f ",tempp);
                }
                printf("\n");

                printf("VAR \n");
                temp=0;
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",wind_var[i][j]);
                                if (wind_var[i][j]==49) temp++;
                        }
                        printf("\n");
                }
                printf(" double = %d \n",temp);*/
#ifdef PRINT_PEAK_SPREAD
                printf("PEAK 0\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",PK0[i][j]);
                        }
                        printf("\n");
                }
                printf("SPREAD 0\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",SPR0[i][j]);
                        }
                        printf("\n");
                }
                printf("PEAK 90\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",PK90[i][j]);
                        }
                        printf("\n");
                }
                printf("SPREAD 90\n");
                for (i=0;i<ROW;i++){
                        for (j=0;j<COL;j++){
                                printf("%3d  ",SPR90[i][j]);
                        }
                        printf("\n");
```

```
                }
#endif PRINT_PEAK_SPREAD
                prints(class);
                printf("\n");
                /*printf("Count of VAR0[i][j] > 150.0 = %d\n",var0count);*/
#ifdef PRINT_VAR90
                printf("Count of VAR90[i][j] > 200.0 = %d\n\n",var90count);
#endif PRINT_VAR90
        }
        /*finish = clock();
        printf("Total time for execution = %ld seconds\n",(finish-start));*/
        printf("TRANSVERSE LONGITUDINAL INTACT     SHADOW   INTERSECT   PAINT     SPALL\n");
        printf("%9d %9d %9d %9d %9d %9d %9d\n",TRANS,LONG,INTACT,SHADOW,INTER,PAINT,SPALL);
}
```

REFERENCES

"1991 Pavement Evaluation System Rater's Manual," Texas State Department of Highways and Public Transportation, 1991.

D.R. Curphey, D.K. Fronek, J.H. Wilson, "Pavement Management System using Video Imaging Processing," Phase 1 Final Report, National Science Foundation Report 17,045-762, July 1984.

C. Chien, "Detection of Cracks in Highway Pavement from Aerial Photographs," Thesis, University of Texas, Austin, 1982.

T. Fukuhara, K. Terada, M. Nagao, A. Kasahara, and S. Ichihashi, "Automatic Pavement Distress Survey System," Proc. 1st Int. Conf. on Application of Advanced Technologies in Transportation Engineering, San Diego, Feb. 1989, pp. 33-38.

G. Caroff, P. Joubert, F. Purdhomme, and G. Soussain, "Classification of Pavement Distresses by Image Processing (MACADAM SYSTEM)," Proc. 1st Int. Conf. on Application of Advanced Technologies in Transportation Engineering, Feb. 1989, pp. 46-51.

K. B. Chan, S. Soetandio, and R. L. Lytton, "Distress Identification by an Automatic Thresholding Technique," Proc. 1st Int. Conf. on Application of Advanced Technologies in Transportation Engineering, Feb. 1989, pp. 468-473.

D. Mendelsohn, "Automated Pavement Crack Detection. An Assessment of Leading technologies," Proc. 1st Int. Conf. on Pavement Management, 1987.

R. E. Smith, M. I. Darter and S. M. Herrin, "Highway Pavement Distress Identification Manual," Report DOT-FH-11-9175/NCHRP 1-19. FHWA, U.S. Department of Transportation, 1979.

R. C. Gonzalez and P. Wintz. Digital Image Processing, 2nd Edition. Addison-Wesley, Inc, 1987.

Series 150/151 Overview 47-H15001-00 Imaging Technology Inc, Technical Publications Department, 600, West Cummings Park, Woburn, MA 01801.