10-1-1994

Automatic visual inspection of placement of bare dies in multichip modules

Mahesh Chheda

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
AUTOMATIC VISUAL INSPECTION
OF
PLACEMENT OF BARE DIES
IN
MULTICHIp MODULES

by

Mahesh Chheda

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electrical Engineering

Approved by:

Prof. P. R. Mukund
(Thesis Advisor)

Prof. 

Prof. 

Prof. 

Prof. 

(Departmental Head)

DEPARTMENT OF ELECTRICAL ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
OCTOBER, 1994
Automatic Visual Inspection
Of
Placement Of Bare Dies
In
Multichip Modules

By
Mahesh Chheda

I, Mahesh Chheda, hereby grant permission to the Wallace Memorial Library of the Rochester Institute Of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 9/7/94 Signature Of Author: ________________
(Mahesh Chheda)
ACKNOWLEDGMENT

This thesis was required in partial fulfillment for the Master's Of Science degree in Electrical Engineering by Rochester Institute Of Technology. I am grateful to my advisor, Dr. P. R. Mukund without whose invaluable help, it would have been difficult to complete this thesis, and I would like to thank him for that. Also I would like to thank Dr. Salem, Mr. P. V. Gupta, Mr. R. Rao and Dr. Mathew, who are professors at RIT, for their advise and support to help me finish my thesis.

I would like to thank Dr. Unnikrishnan, who is Head Of The Department of Electrical Engineering at RIT, for his kindness and full cooperation in allowing me to use the facilities in EE Department. I would also like to appreciate some of my friends at RIT, namely Mr. Anand Shah and Mr. Mathew Nazareth for their great help and kind support during the writing of this thesis.
ABSTRACT

Multichip Modules are gaining lot of popularity in today's IC technology, as they are good solutions for high density packaging. This thesis presents a method for checking the placement of bare dies on a common substrate of an MCM. This testing is done using Automatic Visual Inspection (AVI), which is better and more reliable, compared to manual inspection. Comparison is the basis in this thesis to detect faults in an MCM. The MCM to be tested is compared with a known good ideal MCM using image processing techniques. The mismatches, if any, between these two images, i.e. image of an MCM which is being tested and image of known good reference MCM, are evaluated to find the exact location and nature of the fault.

This AVI is implemented completely in software using 'C' language. Test cases and their results are presented.
# CONTENTS

Acknowledgment i
Abstract ii
List Of Figures vi

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Basic MCM Test Flow</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Automatic Visual Inspection</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Image Acquisition</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Image Processing</td>
<td>7</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Preprocessing</td>
<td>7</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Defect Detection</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>RELATED DIGITAL IMAGE PROCESSING CONCEPTS</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simple Arithmetic Image Processing Operations</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Image Addition</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Image Subtraction</td>
<td>13</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Image Multiplication</td>
<td>14</td>
</tr>
</tbody>
</table>
Chapter 2

2.2 Image Transforms 14

2.1.1 Translation 14

2.1.2 Scaling 15

2.1.3 Rotation 16

2.1.4 Hough Transform 18

2.3 Image Enhancement 26

2.4 Image Analysis 27

2.4.1 Edge Detection 29

2.4.2 Correlation 32

2.4.3 Thinning 33

Chapter 3

RELATED IMAGE PROCESSING ALGORITHMS 35

3.1 Sobel Operator 36

3.1.1 Algorithm for Sobel Operator 37

3.2 Alignment Of Images 38

3.2.1 Image Rotation 39

3.2.2 Algorithm For Alignment Of Images 41

3.3 Thinning 44

3.3.1 Algorithm For Thinning 49

3.4 Hough Transform 54

3.4.1 Algorithm For Hough Transform 54

3.5 Inverse Hough Transform 57

3.5.1 Algorithm For Thresholding The Parameter Matrix 58

3.5.2 Algorithm For Inverse Hough Transform 58
3.6 Detection Of Coordinates Of A Line
3.7 Error Detection

Chapter 4 RESULTS
4.1 TEST (1)
4.2 TEST (2)
4.3 TEST (3)
4.4 TEST (4)
4.5 TEST (5)
4.6 TEST (6)

Chapter 5 CONCLUSIONS
5.1 General
5.2 Future Work

References

Appendix A C PROGRAMS
A1 C Program For Edge Detection Of Test Image
A2 C Program For Alignment Of Two Images
A3 C Program For Finding And Isolating The Location Of Fault
A4 C Program For Error Detection
# LIST OF FIGURES

**Figures:**

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Basic MCM Test Flow</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Block Diagram For Automatic Visual Inspection Of An MCM</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Rotation Of A Point In The Image</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Representation Of Line In XY Plane</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Representation Of Line In mc Parameter Space</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Quantization Of mc Parameter Space</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Normal Representation Of Line In XY Plane</td>
<td>23</td>
</tr>
<tr>
<td>2.6 Quantization Of ρθ Parameter Space</td>
<td>24</td>
</tr>
<tr>
<td>2.7 ρθ Parameter Space</td>
<td>25</td>
</tr>
<tr>
<td>2.8 3x3 Image Window Region</td>
<td>31</td>
</tr>
<tr>
<td>2.9(a) Sobel Operator Mask Used To Compute Gx At Center Point Of 3x3 Region</td>
<td>31</td>
</tr>
<tr>
<td>2.9(b) Sobel Operator Mask Used To Compute Gy At Center Point Of 3x3 Region</td>
<td>32</td>
</tr>
<tr>
<td>2.10 Medial Axes Of Two Simple Regions</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 3

3.1(a) 3x3 Image Window Region 36
3.1(b) Sobel Operator Mask Used To Compute $G_x$ At Center Point Of 3x3 Region 36
3.1(c) Sobel Operator Mask Used To Compute $G_y$ At Center Point Of 3x3 Region 37
3.2 Neighborhood Arrangement Of Contour Point P1 Used By Thinning Algorithm 45
3.3 Example Of 3x3 Region Of An Image For Thinning Algorithm 46

Chapter 4

4.1(a) Original Image Of Known Good MCM For Test 1 66
4.1(b) Original Image Of Test1 MCM 66
4.1(c) Edge Detected Image Of Good MCM 67
4.1(d) Edge Detected Image Of Test1 MCM 67
4.1(e) Isolated Differential Area Of Good MCM From Test1 MCM 68
4.1(f) Isolated Differential Area Of Test1 MCM From Good MCM 68
4.1(g) Output Image Of Test1 MCM With Faulty Area Highlighted 69
4.2(a) Original Image Of Test2 MCM 72
4.2(b)  Edge Detected Image Of Test2 MCM
4.2(c)  Isolated Differential Area Of Good MCM From Test2 MCM
4.2(d)  Isolated Differential Area Of Test2 MCM From Good MCM
4.2(e)  Output Image Of Test1 MCM With Faulty Area Highlighted
4.3(a)  Original Image Of Test3 MCM
4.3(b)  Edge Detected Image Of Test3 MCM
4.3(c)  Isolated Differential Area Of Good MCM From Test3 MCM
4.3(d)  Isolated Differential Area Of Test3 MCM From Good MCM
4.3(e)  Output Image Of Test3 MCM With Faulty Area Highlighted
4.4(a)  Original Image Of Test4 MCM
4.4(b)  Edge Detected Image Of Test4 MCM
4.4(c)  Isolated Differential Area Of Good MCM From Test4 MCM
4.4(d)  Isolated Differential Area Of Test4 MCM From Good MCM
4.4(e)  Output Image Of Test4 MCM With Faulty Area Highlighted
4.5(a)  Original Image Of Known Good MCM For Test 1
4.5(b)  Original Image Of Test1 MCM
4.5(c) Edge Detected Image Of Good MCM

4.5(d) Edge Detected Image Of Test1 MCM

4.5(e) Isolated Differential Area Of Good MCM
       From Test1 MCM

4.5(f) Isolated Differential Area Of Test1 MCM
       From Good MCM

4.5(g) Output Image Of Test1 MCM With Faulty Area
       Highlighted
CHAPTER 1

INTRODUCTION

Multichip Modules (MCMs) are basically two or more bare chips interconnected together on individual substrates. MCMs completely eliminate one level of packaging, namely the single chip package. Direct die mounting places bare die directly on a module. Multichip Modules can be constructed using a number of different thick or thin film processes and assembly schemes. Multichip Modules are categorized as MCM-L : Laminated interconnect (fine line printed circuit board), MCM-C : Ceramic interconnect (low temperature cofired ceramic-LTCC, High temperature ceramic), and MCM-D : Deposited interconnect (thin and medium film). Some of the advantages of MCMs over single chip package IC's are increased speed, reduced size, reduced weight, improved reliability [1].

The MCM enjoys increasing popularity as more and more company efforts are being devoted to this approach. As research resolves technological uncertainties, new challenging issues continuously arise pertaining to both
IC TEST

SUBSTRATE TEST

MCM ASSEMBLY

PRE BURN_IN TEST - 1 (T1)

FAIL

DIAGNOSE 1

PASS

BURN_IN

REWORK

POST BURN_IN TEST - 2 (T2)

FAIL

DIAGNOSE 2

PASS

ENCAPSULATION

FINAL TEST-3 (T3)

PASS

SHIP

FIGURE 1.1: Basic MCM Test Flow [2]
the development and the manufacturing stages. Testing is one of the important issues, since it depends heavily on the important concern of process yield. Defects could be introduced into the MCM in most of the manufacturing stages (mask, photo-resist, metal, wire bonding, die attachment, etc.). Good testing equipment should be able to inspect all of these materials to allow inspection in most critical stages, and detect all possible faults so that they can be repaired in early stages and increase the processes yield, thereby decreasing the cost of MCM manufacturing.

1.1 : Basic MCM Test Flow

The basic MCM test flow, shown in fig. 1.1[2], has been applied to both simple and complex MCMs. There may be variations in each step and what equipment is used, but the basic test flow remains the same. Defects can be introduced in the bare IC die, substrate and module manufacturing processes. The test processes should screen out the defective parts at strategically placed locations in the manufacturing flow. Fresh lot test yield (TEST-1) provides a good indication of the health of the process and determines how much effort is needed to diagnose and repair a module.

The fresh lot module test-1 yield is the product of the module component yields, assembly yield, substrate yield and the product of each die location yield. Assembly yield can then be broken down into wire bond yield, die attachment yield, die placement yield and other major assembly steps.
Individual die location yield is a function of yield due to defects induced in assembly and yield loss due to wafer sort (die testing) escapes [2].

Post burn-in test yield (test-2) is dominated by die failures. Hence fresh lot yield after burn-in is the product of individual die burn-in yield. Prior to shipment the module is encapsulated. This process is a relatively simple operation resulting in high final test yield (test-3) [2].

1.2: Automatic Visual Inspection

The fast emerging multichip module technology presents problems and challenges for in-process inspection which is essential to achieving good yields. One defect in the substrate of an MCM can lead to the loss of an entire MCM. For this reason, in-process visual inspection, either manually or using machine vision (Automatic visual inspection), is essential. The visual inspection process can be defined as observing the same object repeatedly to detect any defects. This type of inspection can be used in process control as well as in the repair of a defective part. Both manual and Automatic visual inspection (AVI) have advantages and disadvantages. Generally, manual inspection is three dimensional and can sort images more easily than machine vision systems. But, on the other hand, humans are much slower, prone to errors due to fatigue.

AVI is used over manual inspection because it is non-damaging, fast and
FIGURE 1.2: Block Diagram For Automatic Visual Inspection Of An MCM
reliable. It can not only detect critical and potentially critical flaws, but also serve as an invaluable process control tool. In addition, defect type and location can also be stored, which can be used later during the time of repair.

The operation of an AVI system for an MCM can be summarized as scanning an MCM substrate which is to be tested, collecting a binarized image of that substrate, and analyzing the image via a set of logic algorithms so as to locate the defects, if any exist. The module can be repaired or scrapped depending on the type of defect, thus avoiding the added expense associated with further processing, and thereby improving the final product yield.

The block diagram, shown in fig. 1.2 [3], can be best understood by separating their two functional steps: Image acquisition and Image processing.

1.3: Image acquisition:

In implementing an AVI system, image acquisition plays an important part. Any deficiencies of the initial images, due to insufficient contrast or poor focusing of the camera, can cause great problems with image analysis and interpretation for later part of an AVI system. If the initial image is poor, the purpose of vision system cannot be fulfilled. Image acquisition unit acquires the modules and transforms it into electronic data which can be transferred to the Image processor. The purpose of the Image acquisition subsystem is to produce high contrast image at an adequate resolution, at highest possible
speed. Intensive light can be projected on the module under inspection by an illuminated light source for acquiring good images. The camera captures the module image and divides the image to picture elements called "pixels". The analog stream of pixels is converted to digital image and sent to the image processor for processing of that image[4].

1.4: Image Processing:

The purpose of this part of the AVI system is to reliably and accurately detect all the defects on the inspected object under test. This subsystem contains all the 'intelligence' of the tool i.e., the different algorithms that are used to detect the defects, implemented in either hardware or software or both. The input to this stage is a digital image that is represented by numbers in a pre-defined range, for e.g. 0 to 255, termed as 'gray levels'. Every pixel is assigned a value with 0 being the darkest and 255 the brightest. The image processing function generally consists of two levels. In first pre-processing stage, low-level operations are carried out such as filtering, feature extraction, etc. In the second defect detection stage, the analysis and interpretation of the image is required. The two stages, pre-processing and defect detection, both contain sophisticated algorithms that are applied to the image, resulting in a reliable and accurate report of the flaws[5].

1.4.1: Pre-Processing:

The digital image from the camera needs to be preprocessed. For example,
image segmentation or edge detection is done to the input image in order to turn this multi-level image into a binary one containing only pure black and white pixels (i.e. having a gray level of 0(black) generally for background and 255 (white) generally for object outlines). This is done in order to distinguish the different edges of the modules in MCMs i.e. identifying the die edges from the substrate. In order to simplify system operation and maintain high reliability, this process should be automatic with minimum user intervention. Another pre-processing function usually performed at this stage is the enhancement of the image. This is done by applying various algorithms (filtering algorithms, smoothing algorithms and others) that together produce a finer, cleaner image on which the defect detection algorithms will be applied.

1.4.2 : Defect Detection :

There are different defect detection methods for AVI which are used in today's industry[6]. Two basic approaches to detect defects in an AVI system are - Design Rule Check(DRC) and Reference Comparison. Design Rule Check inspects the image and considers an image pattern defective if it does not confirm with the pre-defined rules specified in its design. It effectively handles defects such as line and space width violations, pinholes, etc. It does not compare image under test with a reference image directly on a pixel by pixel basis and hence it cannot detect any flaws that do not violate any of the rules such as completely missing parts, perfect shorts and others. These flaws can be detected by template matching technique where the image pattern
under test is compared with a known perfect reference image pattern. Any difference between these two image patterns is considered a defect. This technique is called Reference Comparison method. A precise alignment between the two images is required before comparison for this method[7].

Two basic concepts used for Reference inspection are: Raster and Vector. The Raster, known as Bit-map which uses the image subtraction approach, compares images by comparing their pixels. Pixel-to-pixel comparison requires huge memory space and is also very slow. The Vector method, also known as critical Features, which is more complex and hard to develop, is much more efficient. It uses sophisticated artificial intelligence algorithms to compare entire features (i.e. pads, lines). This implies that the computer first accurately locates and extracts this features, thus compressing the information, and then, with a manageable body of information, processes the image. Sophisticated compression of information is imperative to allow efficient processing and reasonable storage space. Representing the image by its topology drastically reduces required memory space, but requires more complex and sophisticated algorithms to ensure that all the meaningful data on the module will be recognized[8].

In MCMs, die can be placed inaccurately or there may be opens, shorts or errors in interconnects between die and substrate during its manufacturing stage. This thesis presents the design and implementation of an automatic visual inspection system for multichip modules, which will detect any placement errors of the bare die. The raster reference inspection is used for
detection of defects and is implemented completely on software. Subsequent chapters gives the design and implementation of various algorithms used for this AVI system.
CHAPTER 2

RELATED DIGITAL IMAGE PROCESSING CONCEPTS

The term, Digital Image Processing generally refers to a series of operations performed on a numerically represented two-dimensional image in order to obtain a desired result for some specific application. Processing of digital image is done using computers, and since computer needs digital data, the image is converted to a digital form before it is processed. This conversion process is called Digitization. The image is divided into small picture elements, which is called Pixels. The whole image is represented by a rectangular array (rows and columns) of pixels, i.e. horizontal lines made up of adjacent pixels. Each pixel location represents the brightness or darkness of the image at that point. Thus each pixel contains the pictorial information of the image at that particular point. Each pixels have an integer value called the gray level. Generally pixel values represented in the range of 0 to 255, have 0 as representing white, 128 as gray, and 255 as black. This pixel values are stored in computers as binary format[9].

The different kinds of image processing other than digital image processing
are [10]:

(1) Optical and photographic, which uses lenses, enlargers, and many photographic techniques such as dodging and unsharp masking to scale, vary colors, reduce blur and so on in pictures, and

(2) Electrical analog, in which images are converted into analog electrical signals, transmitted, received, and reconstructed as pictures like standard television.

The some of the advantages of digital image processing over the other two kinds of image processing are[10]:

(1) Precision of image: There is a loss of image quality for each generation of photographic process, while in electrical, signals are degraded by physical limitations. But digital image processing can maintain exact precision.

(2) Extremely flexible: Enlarger may magnify an image, but in digital image processing, one part or the whole image can be magnified, reduced, rotated, etc. The contrast and brightness of a television picture can be adjusted, but with digital image processing, many different adjustments can be made.

The disadvantages of digital image processing are its speed and cost.

Digital Image processing has applications in many areas, such as remote sensing via satellites and other spacecraft’s, medical, printing and publishing, astronomy, robotics, and automated inspection of industrial parts.
The remaining part of this chapter explains some image processing techniques which are related to this thesis.

2.1: Simple Arithmetic Image Processing Operations

Some of the simple arithmetic operations performed between two images are image addition, subtraction and multiplication. In order to perform these operations between the two images, both the images should be of same size, i.e. if one image has size of 256 x 256 pixels, other image should also be of size 256 x 256 pixels. These images are represented in two-dimensional arrays.

2.1.1: Image Addition

To add two images A and B, so as to form a new image C, the following equation is used:

\[ C[x][y] = A[x][y] + B[x][y] \] ........ (2.1)

Where, 
A[x][y] is the gray level of image A at coordinates x & y,
B[x][y] is the gray level of image B at coordinates x and y, and
C[x][y] is the gray level of the new image at coordinates x and y.
formed due to addition of A and B.

2.1.2: Image Subtraction

To subtract the two images A and B, same operation is performed as addition, except that instead of adding the gray levels of the images, subtract them, as shown in the equation below:
\[ C[x][y] = A[x][y] - B[x][y] \quad \text{......... (2.2)} \]

### 2.1.3: Image Multiplication

To multiply the two images A and B, same operation is performed as addition, except that instead of adding the gray levels of the images, multiply them, as shown in the equation below:

\[ C[x][y] = A[x][y] \times B[x][y] \quad \text{......... (2.3)} \]

### 2.2: Image Transforms

Image transforms generally refers to a class of unitary matrices used for representing images. Some of the basic transformations that is performed in images are translation, scaling, rotation. This section describes these transformations. The transformations are expressed in two-dimensional Cartesian coordinate system. coordinates of the points of an image are denoted by \((x, y)\).

#### 2.2.1: Translation

To translate or to shift a point in an image with coordinates \((x, y)\) to a new location \((X, Y)\) with displacement of \((x', y')\), the following equation is used:

\[
X = x + x' \\
Y = y + y' \quad \text{......... (2.4)}
\]
Equation (2.4) can be represented in matrix form as shown below:

\[
\begin{bmatrix}
X \\
Y
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & x' \\
0 & 1 & y'
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

\[\text{...... (2.5)}\]

In the above equation (2.5), the transformation matrix used for image translation for a two-dimensional image is:

\[
T =
\begin{bmatrix}
1 & 0 & x' \\
0 & 1 & y'
\end{bmatrix}
\]

\[\text{...... (2.6)}\]

Hence the unified matrix can be represented as

\[
A = Ta
\]

\[\text{...... (2.7)}\]

Where, \(A\) is a single column matrix whose components are the required transformed coordinates, \(a\) is also a single column matrix whose components are the original coordinates and \(T\) is the transformation matrix which is shown above in equation (2.6).

2.2.2 : Scaling

To scale the image, i.e. to enlarge or reduce the original image, the following
Equation (2.8) can be represented in matrix form as shown below:

\[
\begin{bmatrix}
X_s \\
Y_s
\end{bmatrix} =
\begin{bmatrix}
S_x & 0 \\
0 & S_y
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
\]

where the transformation matrix for scaling is:

\[
T =
\begin{bmatrix}
S_x & 0 \\
0 & S_y
\end{bmatrix}
\]

**2.2.3: Rotation**

The simplest form of transformation for two-dimensional rotation is performed by rotating the point about the coordinate axes. There are three transformations required to rotate an arbitrary point in an image: The first translates the arbitrary point to the origin, Second is rotation and third is translating the point back to its position. To rotate a point at coordinates \(x\) and \(y\) by angle \(\varphi\) degrees to a new point at coordinates \(X\) and \(Y\), the following equation is used:

\[
X = x \cos \varphi - y \sin \varphi
\]

\[
Y = x \sin \varphi + y \cos \varphi
\]
Equation (2.11) can be represented in the matrix form as follows:

\[
\begin{bmatrix}
X \\
Y
\end{bmatrix} = \begin{bmatrix}
\cos\varphi & -\sin\varphi \\
\sin\varphi & \cos\varphi
\end{bmatrix} \begin{bmatrix}
x \\
y
\end{bmatrix} \quad \ldots \ldots \text{(2.12)}
\]

and the transformation matrix for rotation is given by:

\[
T = \begin{bmatrix}
\cos\varphi & -\sin\varphi \\
\sin\varphi & \cos\varphi
\end{bmatrix} \quad \ldots \ldots \text{(2.13)}
\]

Image rotation is depicted in fig. 2.1 [13] shown below.

\[\text{Fig. 2.1} \quad \text{Rotation Of A Point In The Image}\]
Thus the rotated image $B[x][y]$ of $A[x][y]$ is given by:

$$A[x][y] = B[x][y] = B[x \cos \phi - y \sin \phi][x \sin \phi + y \cos \phi]$$  \hspace{1cm} \ldots (2.14)

### 2.2.4: Hough Transform

As far as this thesis is concerned, *Hough Transform* is the most important and difficult transform from the transforms discussed so far. It is used as one of the methods to detect shapes in an image. It is mainly used for detecting straight lines (straight edges of shapes). It has become a popular tool for image processing, computer vision, and scene understanding. This method was first introduced by Paul Hough in 1962 (Hough, 1962). He developed this method in connection with his study of particle tracks through the viewing field of a bubble chamber. This method has been modified and improved by different people since then (for further reference see [14]). This section describes simple Hough transform and its use for detecting straight lines in images, in detail.

If there are $N$ points in an image, then we can find straight lines in that image by finding the subsets of these $N$ points that lie on straight lines. One of the simple method is to find every pair of points for each and every line, and from that find all subsets of points that are close to particular lines. This method needs to first find $N(N-1)/2 = N^2$ lines and then execute $(N)(N(N-1))/2 = N^3$ comparisons of every point to all lines. This method is not a practical solution for detecting lines due to its rigorous and huge amount of calculations required[12].
Hough[1962] came out with much better method, which is commonly referred to as Hough transform. He used the slope-intercept parametric representation of a line. The equation of a straight line passing through a point \((x_i, y_i)\) coordinates of an image in slope-intercept form is given by

\[ y_i = mx_i + c \]

...... (2.15)

Where: \(m\) is the slope of the line, and \(c\) is the intercept on the y axis.

Fig. 2.2 shows a line represented in xy plane.

![Figure 2.2](image)

**Figure 2.2**

Representation Of Line In XY Plane
There will be infinite numbers of straight lines that passes through point at \((x_i, y_i)\) coordinates, with different values of \(m\) and \(c\). Each line will have a unique value of \(m\) and \(c\) in the above equation (2.15).

Equation (2.15) can be re-written as:

\[
c = -mx_i + y_i \tag{2.16}
\]

If we consider the \(mc\) parameter space, then equation (2.16) represents single line for a fixed pair \((x_i, y_i)\). Consider a second point \((x_j, y_j)\) in an image, which will also have a line in \(mc\) parameter space associated with it. If this line intersects the line associated with \((x_i, y_i)\) in the parameter space at coordinate \((m', c')\), then the two points lie on a straight line in \(xy\) plane, which has a slope of \(m'\), and \(c'\) as the intercept on \(y\) axes. Similarly, all points that lie on this line, will have lines in the \(mc\) parameter space intersecting at \((m', c')\). This is shown in fig 2.3.

![Fig. 2.3: Representation Of Line In mc Parameter Space](image)
This mc parameter space can be sub-divided (quantized) into so-called *accumulator cells*. This is shown in fig. 2.4 [12].

![Quantization Of mc Parameter Space](image)

**Fig. 2.4**
**Quantization Of mc Parameter Space**

\( m_{\text{max}} \) and \( m_{\text{min}} \) are the maximum and minimum values of the slope, and \( c_{\text{max}} \) and \( c_{\text{min}} \) are the maximum and minimum values of the y-intercept respectively. The cell at coordinates \((m_i, c_j)\) in the quantized mc parameter space will have an accumulator value \( A(m_i, c_j) \). Following steps are used to detect lines in an image using Hough transform:

1. For every edge point \((x, y)\) in the image plane, we substitute every possible value of parameter \( m \) on the \( m \) axes, and find the corresponding values of \( c \) using equation \( c = -mx_i + y_i \) (eq. 2.16). All calculated values of \( c \)'s will be
floating point numbers. Round these values to the nearest possible cell value in the Hough transform table.

(2) For every resulting integer value of \( c \) for a corresponding value of \( m \), the accumulator value of cell at that \((m, c)\) coordinate, is incremented by one. For example, if \( c_j \) is the resulting value of \( c \) after solving the equation for a value of parameter \( m \) as \( m_i \), then the accumulator of the cell \((m_i, c_j)\) is incremented by one i.e.

\[
A(m_i, c_j) = A(m_i, c_j) + 1 \quad \text{(2.17)}
\]

(3) In the end, after computing for each and every point in the image, if accumulator \( A(m_i, c_j) \) has a value of \( X \), then that indicates that there are \( X \) points in the \( xy \) plane that lie on the line having an equation \( y = m_i x + c_j \).

If \( m \) axes is divided into \( A \) increments, then for each and every point \((x, y)\), we have \( A \) values of \( c \) corresponding to \( A \) possible values of \( m \). Thus, if there \( N \) points in an image, then there will be \( N \times A \) computations required for this method. If \( A \) is not a large number, then number of computations required for this method won’t be large too. Also the number of sub-division in the \( mc \) parameter space determines the accuracy of the collinearity of the points in the \( mc \) parameter space.

The big disadvantage of using the \( mc \) parameter space, is that both the slope and the \( y \)-intercept approach infinity as the line approaches a position parallel to \( y \) axes, which leads to an unbounded parameter space. To overcome this difficulty of an unbounded parameter space, the second parametrization
called the normal parametrization, which was introduced by Duda and Hart [Duda et al 1972] [14], is used. The normal representation of a line is given by:

\[ x \cos \theta + y \sin \theta = \rho \] ...... (2.18)

In this type of representation, the line is defined by the \((\rho, \theta)\) parameters, where the \(\rho\) is the algebraic length of the normal to the line in \(xy\) plane, which also passes through the origin, and \(\theta\) is the angle that the normal makes with the \(x\) axes. Normal representation of line is shown in fig. 2.5.

![Normal Representation Of line In XY Plane](image)

**Fig. 2.5**  
Normal Representation Of line In XY Plane

The method for constructing a table of accumulators using this representation is similar to the one discussed before for the slope-intercept representation. This is shown in fig. 2.6 [12]:

23
But instead of straight lines in the mc parameter space, each point \((x_i,y_i)\) in the image space generates a loci sinusoidal curve in the \(\rho \theta\) parameter space. Each point \((\rho_i, \theta_j)\) on that sinusoidal curve codes the \((\rho, \theta)\) parameters of straight lines, which passes through \((x_i, y_i)\). Thus for \(X\) number of collinear points that lie on a line represented by equation \(xcos\theta_j + ysin\theta_j = \rho_i\) will have \(X\) sinusoidal curves that will intersect at \((\rho_i, \theta_j)\) in the \(\rho \theta\) parameter space. Similarly, as in slope-intercept method, every possible value of \(\theta\) will result in a corresponding value of \(\rho\), and value of accumulator \(A(\rho_i, \theta_j)\) will be \(X\). \(\rho \theta\) parameter space is shown in fig. 2.7 [14].
The possible range of $\theta$ measured with respect to $x$ axes is $\pm 90^\circ$. Thus, a horizontal line will have the value of $\theta = 0^\circ$ and value of $\rho$ equal to the positive $x$ intercept, and similarly, a vertical line will have either $\theta = 90^\circ$, with $\rho$ being equal to the positive $y$-intercept, or $\theta = -90^\circ$, with $\rho$ being equal to the negative $y$-intercept. For an image of size $A \times B$, the range of rho is given by $-(A^2 + B^2)^{1/2}$ to $(A^2 + B^2)^{1/2}$. Thus the normal parametrization overcomes the problems associated with an unbounded parameter space found in slope-intercept parametrization. This method is used instead of slope-intercept parametrization for this thesis.
There are many other kinds of image transforms, such as, Discrete Fourier Transform (DFT), Fast Fourier Transform (FFT), Walsh transform, Hadamand transform, discrete cosine transform, Haar transform, slant transform, etc., which are not used for this thesis and hence are not explained. For further reference on this transforms, refer to [11] and [12].

2.3: Image Enhancement

Image enhancement techniques are used to enhance the image, so as to make it more useful for display and analysis. It enhances the image by sharpening the image features such as edges, boundaries, etc. The inherent information content in the data is not increased by the enhancement process. Different methods are used for enhancing an image such as gray level and contrast manipulation, noise reduction, edge crispening, and sharpening, filtering, interpolation and magnification, etc. Many Image processing techniques are empirical and require interactive procedures to obtain satisfactory results. Image enhancement is very useful in virtually all image processing applications. Some of the common image enhancement techniques used are shown below[11]:

A: Point operations

(1) Contrast stretching
(2) Noise clipping
(3) Window slicing
(4) Histogram modeling
B: Spatial Operations
(1) Noise smoothing
(2) Median Filtering
(3) Unsharp masking
(4) Low-pass, band-pass, high-pass filtering
(5) Zooming

C: Transform Operations
(1) Linear filtering
(2) Root filtering
(3) Homomorphic filtering

D: Pseudo coloring
(1) False coloring
(2) Pseudo coloring

For further reference on this techniques, refer to [11] and [12].

2.4: Image Analysis

The purpose of image analysis is to extract important features from image data, from which a description, interpretation, or understanding of the scene can be provided to applications, such as computer vision systems. Image analysis is quite different from other image processing operations, such as
transformation, restoration, and enhancement. In this operations the output is another image, which is different from input image. Different image analysis techniques used are listed below [11]:

**A : Feature Extraction**

(1) Spatial features
(2) Transform features
(3) Edges and boundaries
(4) Shape features
(5) Moments
(6) Texture

**B : Segmentation**

(1) Template Matching
(2) Thresholding
(3) Boundary Detection
(4) Clustering
(5) Quad-trees
(6) Texture matching

**C : Classification**

(1) Statistical
(2) Decision Trees
(3) Similarity measures
(4) Minimum spanning trees
2.4.1: Edge Detection

Since edges characterize object boundaries, edge detection can be used to segment an image into different regions or for identification of objects for various applications. Therefore extraction of edges has become one of the most essential techniques in image processing and computer vision. Generally, edges are defined as boundaries of regions, where there is a sharp and relatively distinct change of some characteristic, like gray level or color. Edge detectors enhances the change of such characteristics. For example, it will highlight the edges, wherever there is a remarkable change of gray levels between two regions, while keeping the rest of the background dark.

There are different ways to detect edges. Most of the edge detectors use spatial differentiation technique. Edges are detected by scanning the entire image with a local operator window, mostly of size 3x3. It has been shown, that the magnitude of the first derivative can be used to detect the presence of an edge, while the sign of the second derivative can be used to determine on which side the edge pixel lies on, i.e. whether it is on the background side or on the object side[12]. Thus, first order derivatives are frequently used for detection of edges in a specified direction. Gradient operators (explained in further detail in next section) are used for detecting edges irrespective of direction. Laplacian operators (second order derivatives) are sensitive to edges as well as to corner points. It is mainly used to determine if the given pixel is on the background side or on the object side.
2.4.1.1: Gradient Operators

Gradient of an image \( f(x,y) \) at coordinates \( (x,y) \) can be represented in two-dimensional vector form as shown below[12]:

\[
\mathbf{G}[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \quad \text{........ (2.19)}
\]

Where, vector \( \mathbf{G} \) points in the direction of maximum change of function \( f \) at coordinates \( (x,y) \). As the magnitude of the first derivative is used to detect edges, the magnitude of vector \( \mathbf{G} \), commonly referred to as the gradient, is more important, and is given by [12]:

\[
\mathbf{G}[f(x,y)] = [G_x^2 + G_y^2]^{1/2} = |G_x| + |G_y| \quad \text{........ (2.20)}
\]

There are different kinds of gradient operators, which are based on performing partial derivatives \( \partial f/\partial x \) and \( \partial f/\partial y \) at every pixel location, namely Roberts gradient operator, which uses 2x2 operator window mask, Prewitt operator, which uses 3x3 operator window mask, and Sobel operator, which also uses 3x3 operator window mask.
**Sobel Operator**:  

Sobel operators uses differences on a 3x3 neighborhood region about pixel located at \((x,y)\). The 3x3 image region is shown in fig. 2.8[12], where \(x_5\) represents the gray level at location \((x,y)\) whose gradient is to be calculated, and \(x_1, x_2, x_3, x_4, x_6, x_7, x_8, x_9\) are the gray levels of the 8-neighbors of \((x,y)\).

\[
\begin{array}{ccc}
  x_1 & x_2 & x_3 \\
  x_4 & x_5 & x_6 \\
  x_7 & x_8 & x_9 \\
\end{array}
\]

**Fig. 2.8**  
3x3 Image Window Region

The components of the gradient in the x direction is given by:

\[
G_x = (x_7 + 2x_8 + x_9) - (x_1 + 2x_2 + x_3) \quad \ldots \ldots (2.20)
\]

and in the y direction is given by:

\[
G_y = (x_3 + 2x_6 + x_9) - (x_1 + 2x_4 + x_7) \quad \ldots \ldots (2.21)
\]

The two 3x3 Sobel operator masks are shown in fig. 2.9 (a) and 2.9 (b).

\[
\begin{array}{ccc}
  -1 & -2 & -1 \\
  0 & 0 & 0 \\
  1 & 2 & 1 \\
\end{array}
\]

**Fig. 2.9 (a)**  
Sobel Operator Mask Used To Compute \(G_x\) At Center Point Of The 3x3 Region
This masks are convolved with the image \( f(x,y) \), to get the gradient at all points in the image. The output result of this convolution is the extracted edges of the image, which is commonly referred to as the gradient image.

The advantage of using Sobel operators over the other operators is that the derivative operations using 3x3 masks are less sensitive to noise, and also weighing the pixels closest to the center by 2 gives additional smoothing of the image.

**2.4.2: Correlation**

The correlation between two, two-dimensional digital image, \( f(x,y) \) and \( g(x,y) \) of size \( A \times B \) is given by [12]:

\[
C(a,b) = \sum_{x} \sum_{y} f(x,y) \cdot g(x-a, y-b) \quad \text{...... (2.22)}
\]

Where \( a = 0, 1, 2, 3, \ldots \ldots, A-1 \),

and \( b = 0, 1, 2, 3, \ldots \ldots, B-1 \)
One of the main application of correlation is template matching. Template matching is used to find the closest match between a given unknown image with a set of known images. It can also be used to align the two same images, but with different orientation. One of the approach to do this is spatial technique, where the correlation between the perfectly aligned image $A$ and mis-aligned image $B$ is calculated, for each shift of the mis-aligned image. The correlation that gives the largest value, gives the closest match between the aligned image and shifted mis-aligned image. This is done by implementing the above equation (2.22), for every value of $(m,n)$ to obtain corresponding value of $C$. As, $m$ and $n$ are varied, $B(x,y)$ moves around the image $A(x,y)$ and the corresponding value of $C$ is calculated for every value of $m$ and $n$. The maximum value of $C(m,n)$ then indicates the position where $B(x,y)$ best matched $A(x,y)$. The second approach used is frequency domain technique, which uses Fast Fourier Transform (FFT). FFT is applied to the two images, take the complex conjugate of one image and multiply with the other, and then perform the inverse FFT on the result obtained from multiplication.

2.4.3: Thinning

After edge detection, generally the edges might be thick. When Hough transform is used to detect these thick lines (lines which are more then one pixel wide) for applications, such as AVI, it may produce erratic results. This image can further be improved by thinning the thick edges in such a way that they become lines of one pixels wide only.
This is often accomplished by obtaining the *skeleton* of the edges by using thinning (also called skeletonizing) algorithm. This algorithm transforms a thick line to a thin single line, which lies roughly along its medial axes. The medial axes of two simple regions is shown in fig. 2.10 [12]. The thinning algorithm is explained in further detail in next chapter.
CHAPTER 3

RELATED IMAGE PROCESSING ALGORITHMS

This chapter describes the methodology used in this work. It also explains related image processing algorithms used for this thesis. The purpose of this thesis is to automate the testing of Multichip Modules (MCMs) using image processing techniques. It looks for any misplacement of bare dies and missing or broken wire bonds in an MCM. This is carried out by comparing the image of test MCM with the an image of a known good MCM.

First the image of a known good MCM is captured using black & white camera. The resolution of the pixels depend on the magnifying power of the camera. After the image is captured and converted into digital form, edge detection is performed on this digital image. The resulting edge detected image is then stored in the memory for comparison. Now the image of the MCM, which is to be tested, is captured using the same camera, so as to maintain the same resolution. This test image is also edge
detected. Sobel operator is used for detecting edges. The algorithm for this operator is explained below.

### 3.1: Sobel Operator

The sobel operator mask which are shown in previous chapter are again represented below:

<table>
<thead>
<tr>
<th>i-1, j-1</th>
<th>i-1, j</th>
<th>i-1, j+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j-1</td>
<td>i, j</td>
<td>i, j+1</td>
</tr>
<tr>
<td>i+1, j-1</td>
<td>i+1, j</td>
<td>i+1, j+1</td>
</tr>
</tbody>
</table>

**Figure 3.1 (a)**
3x3 Image Window Region

<table>
<thead>
<tr>
<th>-1</th>
<th>-2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 3.1 (b)**
Sobel Operator Mask Used To Compute Gx At Center Point Of The 3x3 Region
and the equations used are:

\[ G_x = |a(x_7 + 2x_8 + x_9) - (x_1 + 2x_2 + x_3)| \] ..........(3.1)

\[ G_y = |a(x_3 + 2x_6 + x_9) - (x_1 + 2x_4 + x_7)| \] ..........(3.2)

\[ b[i][j] = G_x + G_y \] ..........(3.3)

3.1.1: Algorithm

Let:

\[ a[i][j] \] be the input two dimensional array used to store the
input image whose edges are to be detected.

\[ b[i][j] \] be the output two dimensional array used to store the
extracted edges of input image.

\[ \text{imin and jmin} \] represent the starting pixel of the image,

\[ \text{imax and jmax} \] represent the end pixel of the image.

If size of the input image is 256x256 pixels, then
the range of values of \( i \) and \( j \) will be: \( 0 \leq i < 256 \) and \( 0 \leq j < 256 \).
imin = jmin = 0.
imax = jmax = 255.

Steps:
(1) Initialize output array b[i][j] to 0.
(2) For each value of i from (imin+1) to (imax-1) in increments of +1 do:
    Begin
    (3) For each value of j from (jmin+1) to (jmax-1) in increments of +1 do:
        Begin
        (4) $gx = a[i+1][j-1] + 2 * (a[i+1][j]) + a[i+1][j+1] - a[i-1][j-1] - 2 * (a[i-1][j]) - a[i-1][j+1].$
        (5) $GX = \text{Absolute value of } gx.$
        (6) $gy = a[i-1][j+1] + 2 * (a[i][j+1]) + a[i+1][j+1] - a[i-1][j-1] - 2 * (a[i][j-1]) - a[i+1][j-1].$
        (7) $GY = \text{Absolute value of } gy.$
        (8) $b[i][j] = GX + GY$
        (9) If $b[i][j]$ is greater then threshold $T$ Then
        (10) $b[i][j] = 255.$
        End (3)
    End (2)

3.2: Alignment of Images:
To compare the image of known good MCM with the one of test MCM, both the images needs to be perfectly aligned together, i.e. one needs to be on top of the other. While capturing the image of test MCM, the image might be shifted or rotated slightly, hence it needs to be aligned with the image of known good MCM, whose image is already stored in the memory. So after edge detection of the test image, the two edge detected images, i.e. the known good image and the test image, need to be aligned together. The algorithm for alignment of images uses the principles of template matching and co-relation to do this alignment. While aligning the two images, the test image needs to be shifted and rotated. Shifting of an image is very simple to implement, but rotating the image is not. Algorithm for rotating an image is presented below.

3.2.1 : Image Rotation :

Equation used for rotating an image in clockwise direction is given by :

\[ b[i][j] = a[i \cdot \cos \theta + j \cdot \sin \theta][ - i \cdot \sin \theta + j \cdot \cos \theta] \]

for each and every value of \( i \) and \( j \), where

\( b[i][j] \) is the rotated version of image \( a[i][j] \), rotated by angle \( \theta \) (theta).

3.2.1.1 : Algorithm For Image Rotation

Let :

\( r_{ic} \) and \( r_{jc} \) be the rotation center coordinates of the input image.

\( r_{id} \) and \( r_{jd} \) be the center coordinates of the output rotated image.
$i_{\text{min}}$ and $j_{\text{min}}$ represent the starting pixel of the image, $i_{\text{max}}$ and $j_{\text{max}}$ represent the end pixel of the image.

If whole 256x256 image is to be rotated by angle $\theta$, then

$r_i = r_j = r_i = r_j = 128,$

$i_{\text{min}} = j_{\text{min}} = 0,$ and

$i_{\text{max}} = j_{\text{max}} = 255.$

**Steps:**

1. Convert $\theta$ from degrees to radians. This is necessary, only if the software used for writing the code (for e.g., C) must have theta in radians in order to perform math functions like sine or cosine.

   The equation for conversion is:
   
   $\theta \text{ in radians} = ( \theta \text{ in degrees} ) \times \pi / 180 = \theta(\text{rads})$  
   ........................................(3.5)

2. Find enclosing rectangle in the rotated image plane using following equations:

   $i_{\text{rec}} = (i_{\text{min}} - i_{\text{max}}) / 2$

   $j_{\text{rec}} = (j_{\text{min}} - j_{\text{max}}) / 2$

   $E_{\text{R}} = ( (i_{\text{rec}})^2 + (j_{\text{rec}})^2 )^{1/2} + 1$  
   ........................................(3.6)

3. **For** each value of $i$ from $-E_{\text{R}}$ to $(E_{\text{R}}-1)$ in increments of $+1$ **do**:

   **Begin**

4. **For** each value of $j$ from $-E_{\text{R}}$ to $(E_{\text{R}}-1)$ in increments of $+1$ **do**:

   **Begin**

5. $I = ( i \times \cos\theta + j \times \sin\theta ) + r_i$

6. $J = ( -i \times \sin\theta + j \times \cos\theta ) + r_j$
Now the actual algorithm for aligning the good image and the test image is presented below.

**3.2.2: Algorithm For Alignment Of Two Images**

Let:

- \(a[i][j]\) be the two dimensional array to store the known good MCM image.
- \(b[i][j]\) be the two dimensional array to store the test MCM image.
- \(X_{\text{min}}\) be the minimum possible \(x\) shift in the test image.
- \(Y_{\text{min}}\) be the minimum possible \(y\) shift in the test image.
- \(X_{\text{max}}\) be the maximum possible \(x\) shift in the test image.
- \(Y_{\text{max}}\) be the maximum possible \(y\) shift in the test image.
- \(\theta_{\text{min}}\) be the minimum possible angle by which test image can be rotated while the image was grabbed.
- \(\theta_{\text{max}}\) be the maximum possible angle by which test image can be rotated while the image was grabbed.
prod1  be the maximum value of the product of the two images.
prod2  be the current product value of the two images.
xshift be the final value of the x shift required to shift the test image on x axes.
yshift be the final value of the y shift required to shift the test image on y axes.
θshift be the final value of the q required to rotate the test image by θ degrees.

Steps:
(1) Initialize prod1 to 0.
(2) For each value of a from Xmin to Xmax in increments of +1 do:
    Begin
(3) For each value of b from Ymin to Ymax in increments of +1 do:
        Begin
(4) Shift input test image b[i][j] on x axes by a.
(5) Shift input test image b[i][j] on y axes by b.
        Lets denote this shifted test image by simg[i][j].
(6) For each value of θ from θmin to θmax in increments of 1°
        do:
            Begin
(7) Initialize prod2 to 0.
(8) Rotate the shifted image simg[i][j] by θ° using the rotation algorithm previously described.
        Lets denote this shifted image by rotsimg[i][j].
Multiply this rotated-shifted image \( \text{rotsimg}[i][j] \) with the known good image \( a[i][j] \) and let that resulting value be equal to \( \text{prod2} \).

If \( \text{prod2} \) is greater than \( \text{prod1} \), Then do:

**Begin**

/**
Copy the values of \( a \), \( b \), and \( \theta \)  **/

(11) \( \text{xshift} = a \)

(12) \( \text{yshift} = b \)

(13) \( \theta \text{shift} = \theta \)

**End** (10)

**End** (6)

**End** (3)

**End** (2)

(12) Shift the input test image \( b[i][j] \) on x axes by \( \text{xshift} \) and on y axes by \( \text{yshift} \).

(13) Rotate that shifted image by \( \theta \text{shift} \).

This rotated-shifted test image will be perfectly aligned with the known good MCM image.

After the images are aligned together, the location of the fault is detected by comparing each pixel by pixel between the two images. If there is a mismatch, then the coordinates of the point of mismatch is noted down. After the comparison is over, the whole mismatched region is highlighted and isolated for finding the degree of mismatch. This is done by detecting
the start and end coordinates of each edge lines formed either by bare dies or by wire bonds for both the images. Then these coordinates of the two images are compared respectively, and accordingly determine the degree of mismatch, i.e. by how much the bare die is shifted or rotated, or whether wire bond is missing or not. In order to find the coordinates of the lines, we need to operate hough transform on the images, and then perform inverse hough transform so as to get the exact location of the lines. But these edge lines will be more than one pixel thick. So performing hough transform on these thick lines will result in having more quantized parametric cells, with some valid value in it. This will produce erratic data, and will also be time consuming. To avoid this problem, *Thinning* operation is first performed on these thick edges in mismatched region of the images which is isolated, and then hough transform and inverse hough transform. The algorithms for thinning, hough transform and inverse hough transform are presented below.

### 3.3: *Thinning*:

Various thinning algorithms, like classical thinning algorithm, asynchronous thinning algorithm, etc. are used for finding medial axis (skeleton) of a region depending on their speed and efficiency [15]. These thinning algorithms delete edge points of a region in such a way that it does not remove the corner points of the region, nor does it break the connectivity of the segment of the skeleton.
This section describes a simple thinning algorithm for thinning binary regions, where values 1 are assumed to represent region points, and values 0 represents the background points. This method employs two basic steps, which are applied to any pixel having a value 1, and at least one of it’s eight neighboring pixels having value 0. This pixel is called the contour point. These two steps are successively passed over the given region, until the desired result is achieved. The 3x3 neighborhood arrangement of contour point p1, used by the thinning is shown in fig. 3.2 [12]:

<table>
<thead>
<tr>
<th>P9</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8</td>
<td>P1</td>
<td>P4</td>
</tr>
<tr>
<td>P7</td>
<td>P6</td>
<td>P5</td>
</tr>
</tbody>
</table>

**Figure 3.2**

**Neighborhood Arrangement Of Contour Point p1**

**Used By Thinning Algorithm**

Step 1, which is applied to every border pixel in the considered binary region, checks for the following 4 conditions [12]:

(a) \( 2 \leq A(p1) \geq 6 \) \hspace{1cm} .......(3.7)

(b) \( B(p1) = 1 \) \hspace{1cm} .......(3.8)

(c) \( p2 \cdot p4 \cdot p6 = 0 \) \hspace{1cm} .......(3.9)

(d) \( p4 \cdot p6 \cdot p8 = 0 \) \hspace{1cm} .......(3.10)
Where $A(p_1)$ is the number of neighboring pixels of $p_1$ having value 0,
i.e. \[ A(p_1) = p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8 + p_9 \] (3.11)
and $B(p_1)$ is the number of 0-1 transitions in the ordered sequence of $p_2, p_3, p_4, p_5, p_6, p_7, p_8, \text{and } p_9$.

If these conditions are satisfied, then the contour point $p_1$ is flagged for deletion. These conditions can better be understood by example shown in fig. 3.3.

\[
\begin{array}{ccc}
0 & 1 & 0 \\
0 & p_1 = 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

**Figure 3.3**
Example Of 3x3 Region Of An Image For Thinning Algorithm

In the above figure, the results of the above 4 conditions are:

(a) \[ A(p_1) = 1 + 0 + 1 + 1 + 1 + 0 + 0 = 5 \] (3.12)
(b) \[ B(p_1) = 2 \] (3.13)
(c) \[ 1 \times 1 \times 1 = 1 \] (3.14)
(d) \[ 1 \times 1 \times 0 = 0 \] (3.15)

From the four conditions for step 1, if one or more conditions are violated, the value of the contour point in question is left unchanged, i.e.
it is not flagged for deletion. But if all of the above four conditions are satisfied, then the point in question is flagged for deletion. However, the point is unchanged until all border points have been processed, so as to keep the structure of data same during the execution of this steps. After step 1 has been applied to all points in the region, the contour points that were flagged for deletion, are deleted (i.e. Their values are changed from 1 to 0). For the above example, conditions (a) and (d) are satisfied, but conditions (b) and (c) are violated, and hence contour point P1 is not deleted.

After step 1 is executed, step 2 is applied to the resulting data in exactly the same way as step 1, with conditions (a) and (b) remaining the same, but conditions (c) and (d) being different. They are:

\[(c2) \quad p2 \times p4 \times p8 = 0 \quad \ldots \ldots (3.16)\]
\[(d2) \quad p2 \times p6 \times p8 = 0 \quad \ldots \ldots (3.17)\]

For above example, the result of conditions (c2) and (d2) are:

\[(c2) \quad 1 \times 1 \times 0 = 0 \quad \ldots \ldots (3.18)\]
\[(d2) \quad 1 \times 1 \times 0 = 0 \quad \ldots \ldots (3.19)\]

Here in step 2, conditions (c2) and (d2) are satisfied. But still condition (b) is not satisfied, and hence contour point P1 is not deleted after executing step 2.

**Reasons for these conditions:**
Condition (a): This condition is satisfied when contour point P1 has more than one and less than seven of the 8 neighboring
points have values 1. When only one of the 8 neighboring points has a value 1, it is not flagged for deletion, because it implies that p1 is the end point of a skeleton segment. And it would cause erosion into the region if contour point p1, having more than 6 out of 8 neighboring points of value 1, is deleted.

Condition (b): Condition (b) prevents breaking of segments of skeleton during the thinning operation, because this condition is not satisfied when it is applied to points on a stroke 1 pixel thick.

Conditions (c) and (d):

A point that satisfies conditions (c) and (d) ( i.e. either p4 = 0 or p6 = 0 or p2 and p8 = 0 ), as well as conditions (a) and (b), is an south or east boundary point or a northwest corner point in the boundary. This means that point p1 is not a part of the skeleton and hence should be deleted.

Conditions (c2) and (d2):

Similar to above conditions, a point that satisfies these conditions ( i.e. p2 = 0 or p8 = 0 or p4 and p6 = 0 ), signifies that it is north or west boundary point, or a southwest corner point, and hence should be deleted.

Note: The northeast corner points of the binary region have p2 and p4 = 0, and southwest corner points have p6 and
\[ p_8 = 0. \] Hence they satisfy all four conditions, i.e. conditions (c) and (d), as well as conditions (c2) and (d2).

The above thinning algorithm can be summarized as executing following steps:

(a) Step 1 is applied to flag border points for deletion.
(b) Deleting all the flagged points, which are flagged by step 1.
(c) Step 2 is applied to flag the remaining border points for deletion.
(d) Deleting all the flagged points, flagged by step 2.

This 4 steps are applied successively until no further points can be deleted, thus yielding the skeleton of the region. After this, there is no point in applying this algorithm and hence is terminated. The implementation of the above thinning algorithm is given below:

3.3.1: Algorithm for thinning:

Let:

- \( a[i][j] \) be the two dimensional array which stores the input image on which the thinning operation is to be performed.
- \( x[i], y[i] \) be the one dimensional arrays that stores the x-y coordinates of the points that needs to be deleted after execution of step 1 and step 2.
- \( \text{pass} \) be the number of times step 1 and step 2 are to be executed on the image.
- \( p[i] \) be the one dimensional array used to store the values of 8 neighboring points of the contour point being processed.
b[i] be the one dimensional array, which stores the binary value of either 0 or 1 for pixel having a gray value of 0 or 255 respectively.
sum signify A(p1) of condition (a).
trans signify B(p1) of condition (b).
xmin, ymin be the starting x-y coordinates of the region in the image on which thinning operation is to be performed.
xmax, ymax be the ending x-y coordinates of the region in the image on which thinning operation is to be performed.

Steps:
(1) For each value of temp1 from 0 to pass in increments of +1 do :
    Begin
(2)   For each value of step from 1 to 2 in increments of +1 do :
       Begin
(3)       temp2 = 0
(4)       temp3 = temp4 = 0
(5)       For each value of i from (xmin+1) to xmax in increments of +1 do :
           Begin
(6)               For each value of j from (ymin+1) to ymax in increments of +1 do :
                   Begin
/** Perform the thinning algorithm, if and only if the point of question has a value 1 **/
(7)                   If a[i][j] is equal to 255 Then do :
Begin

(8) \text{temp5} = 0

/\*\* Convert the gray values 0 and 255 to respective binary values 0 and 1 \*/\n
(9) \textbf{For each} value of \(k\) from -1 to 1 in increments of +1 \textbf{do:}

Begin

(10) \textbf{For each} value of \(l\) from -1 to 1 in increments of +1 \textbf{do:}

Begin

(11) \textbf{If} \(a[i+k][j+l]\) is equal to 255 \textbf{Then do:}

(12) \(b[\text{temp5}] = 1\)

\textbf{Else do:}

(13) \(b[\text{temp5}] = 0\)

(14) \(\text{temp5} = \text{temp5} + 1\)

End (11)

End (10)

End (9)

/\*\* Check for condition (a) \*/\n

(16) \(\text{trans} = 0\)

(17) \textbf{If sum} is between 2 and 6 \textbf{Then do:}

Begin

(18) \text{Arrange the array} \(p\) \text{in right order as shown in fig. 3.2 for checking condition (b) i.e.}

\(p[2] = b[1]\)

\(p[3] = b[2]\)

51
\[ p[9] = b[0] \]

(19) For each value of \( k \) from 2 to 9 in increments of +1 do:

Begin

(20) If \( p[k] \) is equal to 0 And \( p[k+1] \) is equal to 1 Then do:

(21) \( \text{trans} = \text{trans} + 1 \)

End (19)

/** Check for condition (b) **/

(22) If \( \text{trans} \) is equal to 1 And \( \text{step} \) is equal to 1 Then do:

Begin


/** Check for conditions (c) and (d). If satisfied, then flag the point for deletion **/

(25) If \( \text{temp3} \) is equal to 0 And \( \text{temp4} \) is equal to 0

Then do:

Begin

(26) \( x[\text{temp2}] = i \)

(27) \( y[\text{temp2}] = j \)

52
temp2 = temp2 + 1

End (25)

End (22)

Else If trans is equal to 1 And step is equal to 2 Then do:

Begin


/** Check for conditions (c2) and (d2). If satisfied, then flag the point for deletion **/

If temp3 is equal to 0 And temp4 is equal to 0 Then do:

Begin

x[temp2] = i
y[temp2] = j
temp2 = temp2 + 1

End (32)

End (29)

End (17)

End (7)

End (6)

End (5)

/** Delete the flagged points **/

For each value of temp6 from 0 to (temp2-1) in increments of +1 do:
Begin
(37) \( i = x[temp6] \)
(38) \( j = y[temp6] \)
(39) \( a[i][j] = 0 \)
End (36)
End (2)
End (1)

3.4 : **Hough Transform** :

Hough transform is performed after thinning on the fault isolated image. Normal representation technique is used to do hough transform. Normal representation of a line is given by
\[
\rho = x\cos\theta + y\sin\theta
\] 
\----------(3.20)

3.4.1 : **Algorithm For Hough Transform**

Let:
- \( a[i][j] \) be the matrix for the input image.
- \( b[i][j] \) be the matrix for the output image.
- \( p\_mat \) be the parametric matrix in rho-theta space.
- \( p\_row \) be the number of rows of \( p\_mat \) which represents angle theta.
p_col be the number of columns of p_mat which represents rho.

COS[a] be the look up table for cosine function which stores the respective cosine value of cell a.

SIN[a] be the look up table for sine function which stores the respective sine value of cell a.

imax and jmax represent the end pixel of the image.

Range of ρ and θ for an image of size imax x jmax are

\[-(imax^2 + jmax^2)^{1/2} \leq \rho \geq (imax^2 + jmax^2)^{1/2}\]
\[-\pi/2 \leq \theta \geq \pi/2\]  

...(3.21)

For 256 x 256 image, the ranges are

\[-(2*(256)^2)^{1/2} \leq \rho \geq (2*(256)^2)^{1/2}\]
\[-\pi/2 \leq \theta \geq \pi/2\]

imax = jmax = 255

If we keep the resolution of ρ in the quantized parameter space to be 1 and of θ to be 2, then

p_row = \left(2*(256)^2\right)^{1/2} \times 2

p_col = \pi \times 2

**Steps to generate the look up table for sine and cosine functions:**

(1) deg_to_rad = \pi / 180
(2) For each value of i from 0 to p_row in increments of +1 do:

Begin

(3) theta = i \times 180 / (p_row-1) - 90.
(4) \( \theta = \theta \ast \text{deg}_\text{to_rad} \).
(5) \( \text{COS}[i] = \cos(\theta) \)
(6) \( \text{SIN}[i] = \sin(\theta) \)

End (2)

Steps For Hough Transform:

(1) Initialize all the cells of the parametric space \( p\_\text{mat} \) to 0.
(2) \( \rho\_\text{range} = (\text{imax}^2 + \text{jmax}^2)^{1/2} \)
(3) **For** each value of \( i \) from 0 to (imax-1) in increments of +1 **do** :

Begin
(4) **For** each value of \( j \) from 0 to (jmax-1) in increments of +1 **do** :

Begin
(5) **If** value of \( a[i][j] \) is equal to 255 **Then do** :

Begin
(6) **For** each value of \( a \) from 0 to \( \text{p\_row} \) in increments of +1 **do** :

Begin
(7) \( \rho = i \ast \text{COS}[a] + j \ast \text{SIN}[a] \)
(8) \( \rho = (\rho\_\text{range} + \rho) / (\rho\_\text{range} \ast 2) \)
(9) \( \rho = \rho \ast (\text{p\_col} - 1) \)
(10) \( \text{b} = \text{Nearest integer value of } \rho. \)
(11) Increment the integer value of parametric cell \( p\_\text{mat}[a][b] \) by 1 i.e. \( p\_\text{mat}[a][b] = p\_\text{mat}[a][b] + 1. \)

End (6)

End (5)

End (4)
Now the values contained in parameter cell $p_{\text{mat}[a][b]}$ contain the number of points that lie on a line given by the equation

$$\rho_b = i \cdot \cos \theta_a + j \cdot \sin \theta_a$$

\hspace{1em} ......(3.22)

where, value of $\theta_a$ is the respective value of $\theta$ for the corresponding integer value of $a$ on $\theta$ axes in quantized parameter space, and value of $\rho_b$ is the respective value of $\rho$ for the corresponding integer value of $b$ on $\rho$ axes in quantized parameter space.

### 3.5 : **Inverse Hough Transform** :

In order to detect lines and their coordinates, in the thinned fault isolated image, we need to reconstruct the lines from the values contained in the parameter matrix $p_{\text{mat}}$. For that we need to use inverse hough transform. But first, we need to threshold the parameter matrix in order to remove the noise or lines which are actually not present in the image, but still have some value in the parameter matrix. Thresholding is done in order to binarize the parameter matrix, i.e. If the value contained in cell $p_{\text{mat}[a][b]}$ is greater than some threshold value, then the value of that cell is replaced by value 1. If not, then it is changed to value 0. The algorithm for thresholding the parametric space is given on next page.
3.5.1: Algorithm for thresholding the parameter matrix \( p_{mat} \)

**Steps:**

1. For each value of \( a \) from 0 to \( p_{row} \) in increments of +1 do:
   
   Begin

2. For each value of \( b \) from 0 to \( p_{col} \) in increments of +1 do:
   
   Begin

3. If the value of \( p_{mat}[a][b] \) is greater then threshold value \( T \) Then do:

4. \( p_{mat}[a][b] = 1. \)

   Else do:

5. \( p_{mat}[a][b] = 0. \)

   End (2)

End (1)

3.5.2: Algorithm for Inverse Hough Transform

**Steps:**

1. Initialize all the elements in the output matrix \( b[i][j] \) to 0.

2. For each value of \( a \) from 0 to \( p_{row} \) in increments of +1 do:
   
   Begin

3. For each value of \( b \) from 0 to \( p_{col} \) in increments of +1 do:
   
   Begin

4. Initialize temp to 0.

5. If value of cell \( p_{mat}[a][b] \) is equal to 1 Then do:
Begin

(6) For each value of i from 0 to imax in increments of +1 do :

    Begin

(7) \[ \rho = b \times 2 \times \rho_{\text{range}} / (p_{\text{col}} - 1) - \rho_{\text{range}}. \]

(8) If Value of \( \sin[a] \) is equal to 0 Then do :

(9) Increment j by 1.

Else do :

Begin

(10) \[ \text{temp} = (\rho - (i \times \cos[a])) / \sin[a]. \]

(11) \( j = \) nearest possible integer value of \text{temp}.

End (8)

(12) If value of j is between 0 and jmax Then do:

Begin

(13) If value of \( a[i][j] \) is equal to 255 Then do:

(14) \( b[i][j] = 255. \)

End (12)

End (6)

End (5)

End (3)

End (2)

3.6: Detection of coordinates of a line

The start and end coordinates of a line are found by using the inverse hough transform. This is implemented by adding the following steps after step (13) in the above algorithm for inverse hough transform.
Steps:

(1) Note down this starting coordinate of the line.

(2) Check for the value of next pixel. If it is 255 Then do:

(3) Repeat the steps for inverse hough transform until the value of next pixel is 0.

(4) Note down the current values of i and j. This gives the end coordinates of the line.

(5) Note down the values of a and b. This gives the value of rho and theta for that line.

There will be many lines detected by this procedure. We need to filter out all unwanted lines. Many lines which are not horizontal or vertical, will be broken. For e.g. if a line is at angle 45° with actual starting coordinates as (100, 100) and ending coordinates as (150,150), inverse hough transform will produce many lines, which are actually broken segments of this one line, having slightly different rho and theta. For e.g. it may detect the above line as two lines, one line having starting coordinates as (100, 100) and ending coordinates as (125,125), and the other line having starting coordinates as (126, 126) and ending coordinates as (150, 150). We need to join this lines. The lines which are duplicated are eliminated first by comparing the start and end pixels. If they are same then we delete the duplicate line. Broken lines are joined by comparing the rho and theta of each line. If these values are within some allowable range, then the two lines are assumed to be same. The coordinates of both the lines are
accordingly compared, and the best value, giving the start and end coordinate of the two lines combined, is stored. This procedure is applied to both the images (i.e. the faulty area of the test image and the known good image). This way, the end result will have the coordinates of the line which are actually present in the fault detected area of the test image and known good image.

3.7: **Error Detection**

The lines detected in the fault detected area of the test image and the known good image are compared. First the actual rho and theta of these detected lines are calculated using the equations derived below:

\[
\rho = x_1 \cos \theta + y_1 \sin \theta \\
\rho = x_2 \cos \theta + y_2 \sin \theta 
\] .......(3.23)

Where \( x_1, y_1 \) are the starting coordinates of the line, and \( x_2, y_2 \) are the end coordinates of the line.

Equation (3.17) is derived from the fact that coordinates lying on the same line will have same values of \( \rho \) and \( \theta \).

Solving the simultaneous equation for eq. (3.17), we get

\[
\tan \theta = \frac{(x_1 - x_2)}{(y_2 - y_1)} 
\] .......(3.24)

Therefore, \( \theta = \tan^{-1} \left( \frac{(x_1 - x_2)}{(y_2 - y_1)} \right) \) .......(3.25)
Substituting the value of $\theta$ in above equation (3.17), we get the corresponding value of $p$. Using the above equations, the corresponding value of rho and theta are calculated and stored, for each and every line.

The actual values of rho and theta of the two images are now compared. The closest match between the line in the fault isolated test image and known good image, is found from this comparison. The line found in the test image is assumed to be the corresponding line of the good image. This way, the corresponding match of each and every line in the known good image is found in the test image. The difference in the values of coordinates of the matched lines, gives the shift in X and Y axes of the line in the test image. That is, the line in a known good image is shifted in the test image by the differential value of the coordinates of X and Y axes. Similarly the difference in the value of theta, gives the shift in the angle of line in the test image with respect to line in good image. The lines, which are often mentioned above, actually represent the edges of the bare dies or the wire bondings. Using the above procedure, the exact location and amount of the fault can be calculated.

This fault detecting procedure can be summarized as follows:
Edge detection is performed on the two images, the known image of the good MCM and the input image of the test MCM. The edges of the bare dies and the wire bondings are represented as white lines and the background as black in the images, after the edge detection. The test image
is then perfectly aligned with the known good image in order to perform the comparison. These perfectly aligned images are then compared pixel by pixel. If there are any mismatches, then the mismatched area is isolated for both the images. Thinning operation is then performed on this fault isolated images. Lines and their coordinates in the fault isolated images are then detected using hough transform and inverse hough transform. Finally from these calculated values of coordinates of the lines, the location and degree of mismatch between the two images are found.
CHAPTER 4

RESULTS

The algorithms presented in the previous chapter were implemented using 'C' language. The software was divided into four different parts that perform following operations:

(1) Edge detection
(2) Alignment of images
(3) Finding the area of fault
(4) Finding the exact location and degree of fault

For testing the software, the image of an MCM of size 1.7 x 2.25 cm² was captured using black & white camera on Apple Macintosh computer. The software used for capturing the image was "PIXMAC" which is developed by RIT. The same image was then altered in different ways using "Adobe Photoshop" software to test out the software. These different images were converted from different kinds of format to "RAW" image format. The graylevel images were of the size 256x256 pixels. The entire C code was written in unix environment, because of its speed and memory capability. Different tests and their results which were carried out are shown from next page.
4.1: TEST (1):

Fig. 4.1(a) shows the image of the original known good reference MCM. Fig. 4.1(b) shows the image of the test MCM which is to be tested. It was seen that the actual area occupied by the MCM from the whole 256 x 256 image was 200 X 151 pixels. For the MCM of size 1.7 x 2.25 cm², the resolution was 112.5 μm/pixel. In other words, area of 1 x 1 pixel in the image represent the area of 112.5 x 112.5 μm² on the MCM. For much better resolution, we need to increase the image size by 4 times, i.e. increase the size of the image to 1024 x 1024 pixels. This would give the resolution of approximately 30 μm/pixel. This could be easily implemented with some minor changes in the program. But image of this size would require a large amount of memory. The reason for selecting the image of size 256 x 256 pixels was because of the memory constraints. The whole image of test MCM (called Test1 MCM) is shifted by 8 pixels on x and y axes. The fault is in the lower left end corner of the image, where the bare die is shifted by 6 pixels each, on x axes and y axes. Fig. 4.1(c) and 4.1(d) show the resulting edge detected image of both the original images. These two images were aligned and then compared to find the difference. The differential area on both the images were isolated so as to speed the process and more accurately detect the location of the fault. This images are shown in fig. 4.1(e) and 4.1(f). Hough transform and inverse hough transform were performed on these images. After that, error detection algorithm was ran, and accordingly the location and nature of the fault were determined. Fig. 4.1(g) shows the faulty area, which was highlighted by the black box generated by the software. Final output results showing the coordinates (pixel locations) of each line misplaced are also given.
Fig. 4.1 (a)
Original Image Of Known Good MCM For Test1

Fig. 4.1 (b)
Original Image Of Test1 MCM
Fig. 4.1 (c)
Edge Detected Image Of Good MCM

Fig. 4.1 (d)
Edge Detected Image Of Test1 MCM
Fig. 4.1 (e)
Isolated Differential Area For Good MCM

Fig. 4.1 (f)
Isolated Differential Area For Test1 MCM
The output results which were displayed on the computer screen are shown below:

/** RESULTS OF Test1_MCM_RAW */

Please enter the input image file name : Test1_MCM.raw

***** PERFORMING EDGE DETECTION ON THE INPUT IMAGE

***** EDGE DETECTION DONE. DISPLAYING THE IMAGES

***** ALIGNING THE TWO IMAGES
For aligning the image, the shift in x axis is 8 & y axis is 8 and rotation angle is 0.000000

***** DONE ALIGNING. DISPLAYING THE ALIGNED TEST IMAGE *****

***** FINDING THE LOCATION OF THE FAULT *****

***** LOCATION OF THE FAULT IS DETECTED *****

***** PERFORMING THINNING OPERATION ON THE FAULTY AREA *****

***** DISPLAYING THE IMAGE WITH FAULTY AREA HIGHLIGHTED *****

/** COORDINATES OF THE LINE FOR TEST IMAGE **/

THE COR. OF LINE 1 ARE X1=129 AND Y1=81
THE END COR. OF LINE 1 ARE X2=172 AND Y2=81

THE COR. OF LINE 2 ARE X3=130 AND Y3=104
THE END COR. OF LINE 2 ARE X4=172 AND Y4=104

THE COR. OF LINE 3 ARE X5=129 AND Y5=82
THE END COR. OF LINE 3 ARE X6=129 AND Y6=103

THE COR. OF LINE 4 ARE X7=172 AND Y7=81
THE END COR. OF LINE 4 ARE X8=173 AND Y8=103

/** COORDINATES OF THE LINE FOR GOOD IMAGE **/

THE COR. OF LINE 1 ARE X1=124 AND Y1=75
THE END COR. OF LINE 1 ARE X2=166 AND Y2=75

THE COR. OF LINE 2 ARE X3=124 AND Y3=98
THE END COR. OF LINE 2 ARE X4=166 AND Y4=98
THE COR. OF LINE 3 ARE X5=123 AND Y5=76
THE END COR. OF LINE 3 ARE X6=123 AND Y6=97

THE COR. OF LINE 4 ARE X7=166 AND Y7=75
THE END COR. OF LINE 4 ARE X8=167 AND Y8=97

The shift in xcor and ycor of point 1 is 5 AND 6
The shift in xcor and ycor of point 2 is 6 AND 6
The shift in theta of line1 is 0.0000
The shift in xcor and ycor of point 3 is 6 AND 6
The shift in xcor and ycor of point 4 is 6 AND 6
The shift in theta of line2 is 0.0000
The shift in xcor and ycor of point 5 is 6 AND 6
The shift in xcor and ycor of point 6 is 6 AND 6
The shift in theta of line3 is 0.0000
The shift in xcor and ycor of point 7 is 6 AND 6
The shift in xcor and ycor of point 8 is 6 AND 6
The shift in theta of line4 is 0.0000

From the output results, it is seen that except for the location of point 1, all values were correct. Since 1 pixel denotes 112.5 μm, the only error in the output result for test 1 was that the pixel location of one point was off by 112.5 μm.
4.2: TEST (2):

Same images as in test 1 are used for test 2, but with gaussian noise added to the test image of MCM. This is called Test2 MCM. After comparing edge detected images of test2 MCM and known good reference MCM, big difference between the two images was found, which is shown in fig. 4.2(c) and 4.2(d) of the two isolated differential area of these two images. We can see the effect of adding gaussian noise, by looking at the difference in images of test 1 which are shown in fig. 4.1(e) and 4.1(f), and images of test 2 which are shown in fig. 4.2(c) and 4.2(d). These noises, which are seen as small lines or dots, are removed by thresholding the parameter matrix of hough transform and removing any lines of length less than 4 pixels in the error detection algorithm.

Fig. 4.2 (a)
Original Image Of The Test2 MCM
Fig. 4.2(b)
Edge Detected Image Of The Test2 MCM

Fig. 4.2 (c)
Isolated Differential Area For Good MCM
fig. 4.2(d)
Isolated Differential Area For Test2 MCM

Fig. 4.2 (e)
Output Image Of Test2 MCM With Faulty Area Highlighted
The output results of the test2 are shown below:

/** RESULTS OF Test2_MCM.RAW **/

/** COORDINATES OF THE LINES IN TEST IMAGE **/

THE COR. OF LINE 1 ARE X1=130 AND Y1=81
THE END COR. OF LINE 1 ARE X2=172 AND Y2=81

THE COR. OF LINE 2 ARE X3=129 AND Y3=104
THE END COR. OF LINE 2 ARE X4=173 AND Y4=104

THE COR. OF LINE 3 ARE X5=172 AND Y5=81
THE END COR. OF LINE 3 ARE X6=173 AND Y6=103

THE COR. OF LINE 4 ARE X7=129 AND Y7=82
THE END COR. OF LINE 4 ARE X8=129 AND Y8=104

/** COORDINATES OF THE LINES IN GOOD IMAGE **/

THE COR. OF LINE 1 ARE X1=124 AND Y1=75
THE END COR. OF LINE 1 ARE X2=166 AND Y2=75

THE COR. OF LINE 2 ARE X3=124 AND Y3=98
THE END COR. OF LINE 2 ARE X4=166 AND Y4=98

THE COR. OF LINE 3 ARE X5=166 AND Y5=75
THE END COR. OF LINE 3 ARE X6=167 AND Y6=97

THE COR. OF LINE 4 ARE X7=123 AND Y7=76
THE END COR. OF LINE 4 ARE X8=124 AND Y8=97

The shift in xcor and ycor of point 1 is 6 AND 6
The shift in xcor and ycor of point 2 is 6 AND 6
The shift in theta of line1 is 0.0000
The shift in xcor and ycor of point 3 is 5 AND 6
The shift in xcor and ycor of point 4 is 7 AND 6
The shift in theta of line2 is 0.0000
The shift in xcor and ycor of point 5 is 6 AND 6
The shift in xcor and ycor of point 6 is 6 AND 6
The shift in theta of line3 is 0.0000
The shift in xcor and ycor of point 7 is 6 AND 6
The shift in xcor and ycor of point 8 is 5 AND 7
The shift in theta of line4 is 2.7263

There were 5 errors found, with a difference of 1 pixel in four different point locations and error in the angle of line 4 which should be 0.0 instead of 2.7263. This errors were due to the added noise.

4.3: TEST (3):

This test is similar to test 1 and test 2, with difference being that gaussian noise is added to the test image of MCM. This is called Test3 MCM. After comparing edge detected images of test3 MCM and known good reference MCM, similar differences between the two images were found as in test 2. These differences are shown in fig. 4.3(c) and 4.3(d), the two isolated differential area of these two images. We can see the effect of adding gaussian noise, by looking at the difference in images of test 1 which are shown in fig. 4.1(e) and 4.1(f), and images of test 3 which are shown in fig. 4.3(c) and 4.3(d). These noises are removed in the same manner as in test 2, i.e by thresholding the parameter matrix, which is formed by applying hough transform to each and every point in the two isolated differential images, and deleting all the lines having length of less than 4 pixels while performing inverse hough transform.
Fig. 4.3(a)
Original Image Of Test3 MCM

Fig. 4.3 (b)
Edge Detected Image Of Test3 MCM
Fig. 4.3 (c)
Isolated Differential Area For Good MCM

Fig. 4.3 (d)
Isolated Differential Area For Test 3 MCM
The output results of the test3 are shown below:

/** RESULTS FOR Test3_MCM.RAW **/

/** COORDINATES OF THE LINES IN TEST IMAGE **/

THE COR. OF LINE 1 ARE X1=130 AND Y1=81
THE END COR. OF LINE 1 ARE X2=172 AND Y2=81

THE COR. OF LINE 2 ARE X3=129 AND Y3=104
THE END COR. OF LINE 2 ARE X4=172 AND Y4=104

THE COR. OF LINE 3 ARE X5=171 AND Y5=81
THE END COR. OF LINE 3 ARE X6=173 AND Y6=103

THE COR. OF LINE 4 ARE X7=129 AND Y7=82
THE END COR. OF LINE 4 ARE X8=129 AND Y8=104
** COORDINATES OF THE LINES IN GOOD IMAGE **

The coordinates of line 1 are X1=123 AND Y1=75
The end coordinates of line 1 are X2=166 AND Y2=75

The coordinates of line 2 are X3=124 AND Y3=98
The end coordinates of line 2 are X4=166 AND Y4=98

The coordinates of line 3 are X5=165 AND Y5=75
The end coordinates of line 3 are X6=167 AND Y6=97

The coordinates of line 4 are X7=123 AND Y7=76
The end coordinates of line 4 are X8=123 AND Y8=97

The shift in xcor and ycor of point 1 is 7 AND 6
The shift in xcor and ycor of point 2 is 6 AND 6
The shift in theta of line1 is 0.0000
The shift in xcor and ycor of point 3 is 5 AND 6
The shift in xcor and ycor of point 4 is 6 AND 6
The shift in theta of line2 is 0.0000
The shift in xcor and ycor of point 5 is 6 AND 6
The shift in xcor and ycor of point 6 is 6 AND 6
The shift in theta of line3 is 0.0000
The shift in xcor and ycor of point 7 is 6 AND 6
The shift in xcor and ycor of point 8 is 6 AND 7
The shift in theta of line4 is 0.0000

From the above results, there were only 2 errors found, with a difference of 1 pixel at two different point locations: One for starting point of line 2 and the other for ending point of line 4. It was seen from test 2 and test3, that by thresholding the parameter matrix of hough transform and modifying
inverse hough transform, we can remove the noises.

4.4: TEST (4):

In test 4, same known good image as in test 1 is used, but the test image is changed. The image of the bare die, which is on the left below the center axes of the image of whole MCM, is shifted and also rotated slightly. This was done to check the program for lines which are at some angle other than horizontal and vertical.

Fig. 4.4 (a)
Original Image Of Test4 MCM
Fig. 4.4 (b)
Edge Detected Image Of Test4 MCM

Fig. 4.4 (c)
Isolated Differential Image For Good MCM
Fig. 4.4 (d)
Isolated Differential Area For Test4 MCM

Fig. 4.4 (e)
Output Image Of Test4 MCM With Faulty Area Highlighted
The output results of the test4 are shown below:

/** RESULTS OF Test4_MCM.RAW **/

/** COORDINATES OF THE LINES IN TEST IMAGE **/

THE COR. OF LINE 1 ARE X1=131 AND Y1=78
THE END COR. OF LINE 1 ARE X2=172 AND Y2=82

THE COR. OF LINE 2 ARE X3=131 AND Y3=102
THE END COR. OF LINE 2 ARE X4=171 AND Y4=104

THE COR. OF LINE 3 ARE X5=127 AND Y5=80
THE END COR. OF LINE 3 ARE X6=129 AND Y6=101

THE COR. OF LINE 4 ARE X7=171 AND Y7=82
THE END COR. OF LINE 4 ARE X8=172 AND Y8=104

/** COORDINATES OF THE LINES IN GOOD IMAGE **/

THE COR. OF LINE 1 ARE X1=124 AND Y1=75
THE END COR. OF LINE 1 ARE X2=166 AND Y2=75

THE COR. OF LINE 2 ARE X3=124 AND Y3=98
THE END COR. OF LINE 2 ARE X4=166 AND Y4=98

THE COR. OF LINE 3 ARE X5=123 AND Y5=76
THE END COR. OF LINE 3 ARE X6=123 AND Y6=97

THE COR. OF LINE 4 ARE X7=166 AND Y7=75
THE END COR. OF LINE 4 ARE X8=167 AND Y8=97

The shift in xcor and ycor of point 1 is 7 AND 3
The shift in xcor and ycor of point 2 is 6 AND 7
The shift in theta of line1 is 5.5722
The shift in xcor and ycor of point 3 is 7 AND 4
The shift in xcor and ycor of point 4 is 5 AND 6
The shift in theta of line2 is 2.8624
The shift in xcor and ycor of point 5 is 4 AND 4
The shift in xcor and ycor of point 6 is 6 AND 4
The shift in theta of line3 is -5.4403
The shift in xcor and ycor of point 7 is 5 AND 7
The shift in xcor and ycor of point 8 is 5 AND 7
The shift in theta of line4 is 0.0000

The output results could not be compared with the true values, since actual
values of the coordinates of the lines could not be known as the whole square
representing the bare die was rotated by some angle. But it did detect 4
misplaced lines which were actually the sides of the misplaced bare die.

4.5: TEST (5):

This test was carried out to examine the case where no faults are present. For
this, the test image used is same as the image of known good reference MCM.
The output results of the test5 are shown below:

/** RESULTS OF Test5_MCM.RAW **/
****** PERFORMING EDGE DETECTION ON THE INPUT IMAGE ******

****** EDGE DETECTION DONE. DISPLAYING THE IMAGES ******

****** ALIGNING THE TWO IMAGES ******

the shift in x axis is 0 & y axis is 0 and rotation angle is 0.000000

****** DONE ALIGNING. DISPLAYING THE ALIGNED TEST IMAGE ******

****** FINDING THE LOCATION OF THE FAULT ******

****** THERE IS NO FAULT IN THE TEST MCM. TEST MCM IS GOOD ******

It can be seen from the output results that there was no fault found in the test MCM as expected.

4.6 : TEST (6) :

Different images of test MCM and good reference MCM are used for this test. Fig. 4.6(a) shows the image of the known good reference MCM, and Fig. 4.6(b) shows the image of the test MCM which is to be tested. The fault is on the upper right end corner of the image of MCM, where the bare die is shifted and rotated. Fig. 4.6(c) and 4.6(d) show the resulting edge detected image of both the images. Fig. 4.6(g) shows the faulty area, which is highlighted by the black box.
Fig. 4.6 (a)
Original Image Of Known Good MCM For Test6

Fig. 4.6 (b)
Original Image Of The Test6 MCM
Fig. 4.6 (c)
Edge Detected Image Of The Good MCM

Fig. 4.6 (d)
Edge Detected Image Of The Test6 MCM
The output results of the test6 are shown below:

/** RESULTS OF Test6_MCM.RAW **/

/** COORDINATES OF THE LINES IN TEST IMAGE **/

THE COR. OF LINE 1 ARE \( X_1=38 \) AND \( Y_1=117 \)
THE END COR. OF LINE 1 ARE \( X_2=80 \) AND \( Y_2=120 \)

THE COR. OF LINE 2 ARE \( X_3=42 \) AND \( Y_3=143 \)
THE END COR. OF LINE 2 ARE \( X_4=76 \) AND \( Y_4=145 \)

THE COR. OF LINE 3 ARE \( X_5=35 \) AND \( Y_5=119 \)
THE END COR. OF LINE 3 ARE \( X_6=36 \) AND \( Y_6=142 \)

THE COR. OF LINE 4 ARE \( X_7=80 \) AND \( Y_7=120 \)
THE END COR. OF LINE 4 ARE \( X_8=80 \) AND \( Y_8=140 \)
/** COORDINATES OF THE LINES IN GOOD IMAGE **/

THE COR. OF LINE 1 ARE X1=41 AND Y1=123
THE END COR. OF LINE 1 ARE X2=84 AND Y2=124

THE COR. OF LINE 2 ARE X3=42 AND Y3=149
THE END COR. OF LINE 2 ARE X4=84 AND Y4=150

THE COR. OF LINE 3 ARE X5=40 AND Y5=125
THE END COR. OF LINE 3 ARE X6=41 AND Y6=149

THE COR. OF LINE 4 ARE X7=85 AND Y7=124
THE END COR. OF LINE 4 ARE X8=85 AND Y8=148

The shift in xcor and ycor of point 1 is -3 AND -6
The shift in xcor and ycor of point 2 is -4 AND -4
The shift in theta of line1 is 2.7534
The shift in xcor and ycor of point 3 is 0 AND -6
The shift in xcor and ycor of point 4 is -8 AND -5
The shift in theta of line2 is 2.0025
The shift in xcor and ycor of point 5 is -5 AND -6
The shift in xcor and ycor of point 6 is -5 AND -7
The shift in theta of line3 is -0.1036
The shift in xcor and ycor of point 7 is -5 AND -4
The shift in xcor and ycor of point 8 is -5 AND -8
The shift in theta of line4 is 0.0000

The output results could not be compared with the true values just like test 4.
But it detected 4 misplaced lines which were actually the sides of the misplaced bare die.
5.1: General

The images used for the tests were of size 256 x 256 pixels. If an MCM is of size 2 x 2 cm$^2$ and if that occupies an area of 250 x 250 pixels of the whole image, which is grabbed from an ideal camera having perfect magnifying power and focus, then the area occupied by 1x1 pixel$^2$ of the image represents 80 x 80 μm$^2$ of area on MCM. Since the minimum possible misplacement of the bare die detected by the program is 1 pixel, the minimum misplacement that can possibly be detected for the above case would be 80μm. Anything less than that would not be detected. Also the minimum possible error that this program can create for the image of this size is 1 pixel. While displaying the coordinates of the line (edges of some object in an MCM) at the end, any error of magnitude 1 pixel which is committed by the execution of the code, will show the coordinates which are actually off by 80μm on the MCM. If the size of the MCM is 4 x 4 cm$^2$ then that would increase the error by twofold. This can be improved by increasing the sizes of the images, for e.g. double the size of the image, i.e.
change them from a size of 256 x 256 pixels to 512 x 512 pixels. This will bring down the minimum possible error by half. This shows that larger the size of the images, better the resolution and errors of lesser magnitude.

It can also be concluded from the tests, that using hough transform and inverse hough transform, we could find the closest possible location and type of fault in an MCM. It can show the horizontal, vertical and even angular misplacements. The images used for test were black and white, but if colored images were to be used then the program could be changed by changing the image processing algorithms in such a way that it incorporates all three levels of image colors (red, green, and yellow) instead of just one gray level.

Thus, we can conclude that detecting misplacement errors in an MCM using image processing techniques works out to be a good solution. This is carried out by first detecting edges, then aligning the images for comparison, then doing the actual comparison, then performing hough transform and inverse hough transforms on these images, and at the end evaluating the results of the transforms.

The only drawback of this method is the speed and resolution. The resolution can be improved by selecting the appropriate sizes for the images. And the speed can be improved by finding a better way to align the two images other then using correlation.
5.2: *Future Work*

More research work can be done in trying to improve the speed of this automatic visual inspection system. For e.g. the time required to align the images could be reduced by using Fast Fourier Transforms rather than using spatial techniques. This work could be extended by also including the inspection of wire bonds in wire bonded MCMs to detect any breakage or shorts in wire bonds. Also further improvements in this research work could be carried out by finding a much better and efficient way of trying to locate and determine the nature of fault.
REFERENCES


APPENDIX A

C PROGRAMS
A1: C Program For Edge Detection Of Test Image

/**************************************************************************/
/*** PROGRAM FOR EDGE DETECTION OF TEST IMAGE    ***/
/**************************************************************************/

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

/************  Function for SOBEL OPERATOR  ************/

int sobel(a, b)
/** a is the two dimensional array to store input image whose edges are to
be detected **/
/** b is the two dimensional array to store the ouput edge detected image
**/

unsigned char *a[256];
unsigned char *b[256];
{
    int i, j, l, m, n, o;  /** Temporary variables **/
    int G;  /** Magnitude of the gradient of the image **/
    int gx, gy;  /** Gradient vectors in x-y direction respectively **/
    int GX, GY;  /** Magnitude of the gradient vectors in
x-y direction respectively **/
    int imin,imax,jmin,jmax;  /** Variables for the size of
the image **/

    imin = 0;
imax = 255;

    jmin = 0;
jmax = 255;
for (i = imin; i <= imax; i++) {
    for (j = jmin; j <= jmax; j++) {
        l = i - 1;
        m = j - 1;
        n = j + 1;
        o = i + 1;
        if (l < 0)
            l = l + 256;
        if (o > 255)
            o = o - 256;
        if (m < 0)
            m = m + 256;
        if (n > 255)
            n = n - 256;

        /* Sobel equation used to calculate the components of the gradient in x direction for pixel at location a[i][j] **/
        gx = -(int) *(a[l] + m) - 2 * (int) (*(a[l] + j)) - (int) *(a[l] + n);
        gx += (int) *(a[o] + m) + 2 * (int) (*(a[o] + j)) + (int) *(a[o] + n);
        GX = abs(gx);

        /* Sobel equation used to calculate the components of the gradient in y direction for pixel at location a[i][j] **/
        gy = -(int) *(a[l] + m) - 2 * (int) *(a[i] + m) - (int) *(a[o] + n);
        gy += (int) *(a[l] + n) + 2 * (int) *(a[i] + n)) + (int) *(a[o] + n);
        GY = abs(gy);

        G = GX + GY;
        /* Thresholding the value of gradient G **/
        if (G > 120)
            G = 255;
        else
            G = 0;
        *(b[i] + j) = (unsigned char) G;
    }
}
return (0);

/** End of the function **/
/** MAIN **/

main()
{
    unsigned char *inimage[256]; /**< Two dimensional array
        inimage is used to store the input image **/
    unsigned char *outimage[256]; /**< Two dimensional array
        outimage is used to store the output image **/

    int i, j;
    char infile[30], outfile[30];
    FILE *inptr, *outptr, *tempptr;

    for (i = 0; i < 256; i++) {
        /** Assign memory to the pointers **/
        outimage[i] = (unsigned char *) malloc(256, sizeof(unsigned char));
        inimage[i] = (unsigned char *) malloc(256, sizeof(unsigned char));
        if (!inimage[i] || !outimage[i]) {
            printf("\n\n ERROR IN MATRIX ALLOCATION \n");
            exit(1);
        }
    }

    printf("\n\n Please enter the input image file name : > ");
    scanf("%s", infile);
    inptr = fopen(infile, "rb");
    if (inptr == NULL) {
        printf("Error opening %s as input file\n", infile);
        exit(1);
    }

    tempptr = fopen("faulty.raw", "wb");
    if (inptr == NULL) {
        printf("Error opening faulty.raw as input file\n");
        exit(1);
    }

    /** printf("Enter output image file name : > ");
    scanf("%s",outfile);**/
    outptr = fopen("testsob.raw", "wb");
if (outptr == NULL) {
    printf("Error opening testsob.raw as output file\n");
    exit(1);
}
for (i = 0; i < 256; i++) {
    fread(inimage[i], sizeof(unsigned char), 256, inptr);
    fwrite(inimage[i], sizeof(unsigned char), 256, tempptr);
}
for (i = 0; i < 256; i++) {
    for (j = 0; j < 256; j++) {
        *(outimage[i] + j) = 0;
    }
}
/** Perform the edge detection on the input image **/
printf("\n\n****** PERFORMING EDGE DETECTION ON THE INPUT IMAGE ****** \n");
sobel(inimage, outimage);
printf("\n\n****** EDGE DETECTION DONE. DISPLAYING THE IMAGES ****** \n");
/** Write out the result in the output file **/
for (i = 0; i < 256; i++) {
    fwrite(outimage[i], sizeof(unsigned char), 256, outptr);
}
fclose(inptr);
fclose(outptr);
fclose(tempptr);
/** System commands used to display the images on the screen **/
system("rawtopgm 256 256 faulty.raw > faulty.pgm");
system("xv -wait 10 -expand 2 faulty.pgm");
system("rm faulty.pgm");
system("rawtopgm 256 256 testsob.raw > testsob.pgm");
system("xv -root -rmode 5 -rgb white -quit -expand 2 testsob.pgm");
system("rm testsob.pgm");
}
/** END OF PROGRAM **/
A2 : C Program For Alignment Of Two Images

/**********************************************************************************/
//**  PROGRAM FOR ALIGNMENT OF TWO IMAGES  **/
//**********************************************************************************/

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

/***  Global variables  ***/
float theta;
int ric, rjc; /**< They are the rotation center coordinates of the input image **/
int rid, rjd; /**< They are the center coordinates for the output rotated image **/
int imin, jmin; /**< They are the starting coordinates of the image **/
int imax, jmax; /**< They are the end coordinates of the image **/

/***  Function for rotating an image  ***/
int rotateimg(a,b)

unsigned char *a[256]; /**< Two dimensional array for input image  **/
unsigned char *b[256]; /**< Two dimensional array for output image  **/
{
    int i,j,I,J,ID,JD,ER;
    double theta_in_rad,cos_theta,sin_theta;
    float irec, jrec;
    double PI = 4.0*atan((double)1.0);

    theta_in_rad=(double)theta*PI/180.0;
    cos_theta=cos(theta_in_rad);
    sin_theta=sin(theta_in_rad);
irec = (float)(imax - imin)/2.0;
jrec = (float)(jmax - jmin)/2.0;
ER = (int)sqrt((double)(irec*irec + jrec*jrec)) + 1;

for(i = -ER; i < ER; i++)
{
    for(j = -ER; j < ER; j++)
    {
        I = (int)((float)(j)*sin_theta + (float)(i)*cos_theta) + ric;
        J = (int)((float)(j)*cos_theta - (float)(i)*sin_theta) + rjc;
        ID = i + rid;
        JD = j + rjd;
        if (I >= imin && I < imax && J >= jmin && J < jmax && ID >= 0 && ID < 256 && JD >= 0 && JD < 256)
        {
            *b[ID] + JD = *(a[I] + J);
        }
    }
}

return (0);

/** MAIN **/

main()
{
    unsigned char *inimage[256], *outimage[256], *tempimg[256],
                  *goodimg[256];
    int i, j, X_min, X_max, Y_min, Y_max, tita_min,
        tita_max, x_shift, y_shift, a, b, c, d, m, n, f, g;
    char infile[30], outfile[30], goodfile[30];
    float tita_shift;
    long prod1, prod2;
    FILE *inptr, *outptr, *goodptr;

    for(i=0; i<256; i++)
    {
        outimage[i] = (unsigned char*)calloc(256, sizeof(unsigned char));
        inimage[i] = (unsigned char*) calloc(256, sizeof(unsigned char));
    }
tempimg[i] = (unsigned char*)calloc(256, sizeof(unsigned char));
goodimg[i] = (unsigned char*)calloc(256, sizeof(unsigned char));
if(!inimage[i] || !outimage[i] || !goodimg[i] || !tempimg[i])
{
    printf("\nERROR IN MATRIX ALLOCATION \n");
    exit(1);
}
/** Input Edge Detected Test Image File **/
inptr = fopen("testsob.raw", "rb");
if (inptr == NULL)
{
    printf("Error opening testsob.raw as input file\n");
    exit(1);
}
/** Input Edge Detected Good Image File **/
goodptr = fopen("goodsob.raw", "rb");
if (goodptr == NULL)
{
    printf("Error opening goodsob.raw as good file\n");
    exit(1);
}
/** Output Aligned Test Image File **/
outptr = fopen("testcomp.raw", "wb");
if (outptr == NULL)
{
    printf("Error opening testcomp.raw as output file\n");
    exit(1);
}
for(i=0; i<256; i++)
{
    fread(inimage[i], sizeof(unsigned char), 256, inptr);
    fread(goodimg[i], sizeof(unsigned char), 256, goodptr);
}
for(j=0; j<256; j++)
{
    *(outimage[i] + j) = 0;
    *(tempimg[i] + j) = 0;
}
}

ric = 128;
rjc = 128;
rid = 128;
rjd = 128;
imin = 0;
jmin = 0;
imax = 256;
jmax = 256;
x_shift = 0;
y_shift = 0;
tita_shift = 0.0;
X_min = -5;
X_max = 10;
Y_min = -5;
Y_max = 10;
tita_min = -1.0;
tita_max = 8.0;
prod1 = 0;

for(theta = tita_min; theta < tita_max; theta += 1.0)
{
    rotateimg(inimage,tempimg);
}

for(a=X_min; a<X_max; a++)
{
    for(b=Y_min; b<Y_max; b++)
    {
        prod2 = 0;

        for(c=0; c<256; c++)
        {

            for(d=0; d<256; d++)
            {

        }
if(*(goodimg[c]+d)>150)
{
    i=c+a;
    if(i<0) i= i + 256;
    if(i>255) i= i - 256;
    j=b+d;
    if(j<0) j= j+256;
    if(j>256) j= j-256;

    f= *(tempimg[i] + j);
    g= *(goodimg[c] + d);
    prod2= (long)prod2 + (long)(f*g);
}
}
}

if(prod2 > prod1)
{
    prod1 = prod2;
    x_shift = a;
    y_shift = b;
    tita_shift = theta;
}
}
}

for(a=0; a<256; a++)
{
    for(b=0; b<256; b++)
    {
        i = a + x_shift;
        j = b + y_shift;
        if(i<0) i= i+256;
        if(i>255) i= i-256;
        if(j<0) j= j+256;
        if(j>256) j= j-256;
        *(tempimg[a] + b) = *(inimage[i] + j);
    }
}


printf("\nthe shift in x axis is %d & y axis is %d and rotation angle is %f\n\n",x_shift,y_shift,tita_shift);

theta = tita_shift;
rotateimg(tempimg,outimage);

for(i=0; i<256; i++)
{
    fwrite(outimage[i], sizeof(unsigned char), 256, outptr);
}
fclose(inptr);
fclose(outptr);
fclose(goodptr);

/** System Calls To Display Images **/
system("rawtopgm 256 256 testcomp.raw > testcomp.pgm");
system("xv -root -rmode 5 -rbg white -quit -expand 2 testcomp.pgm");
system("rm testcomp.pgm");
}

/** END OF PROGRAM **/
A3 : C Program For Finding And Isolating The Location Of Fault

/***************************************************************************/
/** PROGRAM FOR FINDING AND ISOLATING THE LOCATION OF FAULT **/
/***************************************************************************/

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

/** Algorithm Used For THINNING **/

int thinning(a, stx, sty, ex, ey)

unsigned char *a[256];  /** Input Image Array Which Needs To Be Thinned **/
int stx, sty, ex, ey;  /** Starting And Ending Coordinates For Image **/
{
    int step, z, e, f, i, j, k, l, q, x, r, p[12], t[12], xc[3000], y[3000];

    for (z = 0; z < 10; z++) {
        q = 0;
        for (step = 0; step < 2; step++) {
            q = 0;
            f = 0;
            for (i = (stx + 1); i < ex; i++) {
                for (j = (sty + 1); j < ey; j++) {
                    if (*(a[i] + j) >= 150) {
                        x = 0;
                        for (k = -1; k < 2; k++) {
                            for (l = -1; l < 2; l++) {
                                
                            }
                        }
                    }
                }
            }
        }
    }
}
if (*a[i + k] + (j + l))
    t[x] = 1;
else
    t[x] = 0;
x++;
} }
e = 0;
if (t[9] >= 2 && t[9] <= 6) {
p[0] = t[0];
p[1] = t[1];
p[2] = t[2];
p[3] = t[5];
p[4] = t[8];
p[5] = t[7];
p[6] = t[6];
p[7] = t[3];
p[8] = t[0];
for (k = 0; k < 8; k++) {
    if (p[k] == 0 && p[k + 1] == 1)
        e++;
}
if (e == 1 && step == 0) {
    f = p[3] * p[5];
    if ((p[1] * f) == 0 &&
        (p[7] * f) == 0) {
        xc[q] = i;
        y[q] = j;
        q++;
    }
} else if (e == 1 && step == 1)
    f = p[1] * p[7];
    if ((p[3] * f) == 0 &&
        (p[5] * f) == 0) {
        xc[q] = i;
        y[q] = j;
        q++;
    }
for (r = 0; r < q; r++) {
    i = xc[r];
    j = y[r];
    *(a[i] + j) = 0;
}

return (0);

/** End Of Function **/

/** MAIN **/

main()
{
    unsigned char *image[256], *goodimage[256], *targetimg[256],
    *badimg[256], *outimage[256];

    int i, j, xm, ym, xmin, ymin, xmax, ymax, a, b, c, d;

    for (i = 0; i < 256; i++) {
        outimage[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        image[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        goodimage[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        targetimg[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        badimg[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
    }
if (!image[i] || !outimage[i] || !goodimage[i] || !badimg[i] || !targetimg[i]) {
    printf("\n ERROR IN MATRIX ALLOCATION \n");
    exit(1);
}

/** Opening Input Edge Detected Test MCM Image **/  
inptr = fopen("testcomp.raw", "rb");  
if (inptr == NULL) {
    printf("Error opening testcomp.raw as input file\n");
    exit(1);
}

/** Opening Input Edge Detected Good MCM Image /**  
targetptr = fopen("goodsob.raw", "rb");  
if (targetptr == NULL) {
    printf("Error opening goodsob.raw as target image file\n");
    exit(1);
}

/** Opening Output Fault Isolated Good Image **/

goodptr = fopen("faultgood.raw", "wb");
if (goodptr == NULL) {
    printf("Error opening faultgood.raw as output file\n");
    exit(1);
}

/** Opening Output Fault Detected Test Image **/

outptr = fopen("faulttest.raw", "wb");
if (outptr == NULL) {
    printf("Error opening faulttest.raw as output bad image file\n");
    exit(1);
}

/** Opening Data File Which Contains The Image Size Of Fault Isolated Images **/

tempptr = fopen("temp1.dat", "w");
if (tempptr == NULL) {
    printf("Error opening temp1.dat as temporary data file\n");
    exit(1);
}

for (i = 0; i < 256; i++) {
    fread(image[i], sizeof(unsigned char), 256, inptr);
    fread(targetimg[i], sizeof(unsigned char), 256, targetptr);
    }
for (i = 0; i < 256; i++) {
    for (j = 0; j < 256; j++) {
        *(outimage[i] + j) = 0;
        *(goodimage[i] + j) = 0;
        *(badimg[i] + j) = 0;
    }
}

xmin = 0;
ymin = 0;
xmax = 0;
ymax = 0;

printf("\n\n****** FINDING THE LOCATION OF THE FAULT  
******\n");
a = 0;
for (i = 0; i < 256; i++) {
    for (j = 0; j < 256; j++) {
        if (*image[i] + j) != *(targetimg[i] + j) {
            a++;
            if (a == 1) {
                xmin = i;
                xmax = i;
                ymin = j;
                ymax = j;
            }
        }
    }
}
if (a > 1) {
    if ((xmin - i) < 5 || (ymin - j) < 5 || (xmax - i) < 5 || (ymax - j) < 5) {
        if (i <= xmin)
            xmin = i;
        if (i >= xmax)
            xmax = i;
        if (j <= ymin)
            ymin = j;
        if (j >= ymax)
            ymax = j;
    }
}
}
/** If No Fault Is Found **/
if (xmax == 0 && xmin == 0 && ymax == 0 && ymin == 0) {
    printf("\n\n****** THERE IS NO FAULT IN THE TEST MCM. TEST MCM IS GOOD ******\n\n");
    exit(1);
}
/** If Fault Is Found **/
else {
    printf("\n\n****** LOCATION OF THE FAULT IS DETECTED ******\n");
    xmin -= 1;
ymin -= 1;
xmax += 1;
ymax += 1;
    fprintf(tempptr, "%d\n", xmin);
    fprintf(tempptr, "%d\n", ymin);
    /* a And b Are The End Coordinates Of Fault Isolated Images */
    a = xmax - xmin;
b = ymax - ymin;

    for (i = 0; i < a; i++) {
        for (j = 0; j < b; j++) {
            c = i + xmin;
d = j + ymin;
            *(goodimage[i] + j) = *(targetimg[c] + d);
            *(badimg[i] + j) = *(image[c] + d);
        }
    }

    printf("\n\n****** PERFORMING THINNING OPERATION ON THE FAULTY AREA ******\n");
xm = 0;
ym = 0;
/** Perform THINNING Operations On Both Fault Isolated Images **/
    thinning(badimg, xm, ym, a, b);
thinning(goodimage, xm, ym, a, b);

for (i = 0; i < a; i++) {
    for (j = 0; j < b; j++) {
        c = (int) *(goodimage[i] + j) - *(badimg[i] + j);
        if (c == 0) {
            *(goodimage[i] + j) = 0;
            *(badimg[i] + j) = 0;
        }
    }
}

for (i = 0; i < 256; i++) {
    fwrite(badimg[i], sizeof(unsigned char), 256, outptr);
    fwrite(goodimage[i], sizeof(unsigned char), 256, goodptr);
}

fclose(inptr);
fclose(outptr);
fclose(goodptr);
fclose(targetptr);
fclose(tempptr);
exit(0);

/** END OF PROGRAM **/
A4: C Program For Error Detection

/**
 * PROGRAM FOR ERROR DETECTION
 */

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

/** Function For Detecting And Reporting The Error **/

int error_rep(xcor, ycor, xcor2, ycor2, xshift, yshift, e, rho, theeta, rho2, theeta2, 
               rho3, theeta3)

  unsigned char *xcor[256], *ycor[256], *xcor2[256], *ycor2[256];
  int e;
  int *xshift[256], *yshift[256];
  float *rho[256], *theeta[256], *rho2[256], *theeta2[256];

  float *theeta3[256], *rho3[256];

{

  float temp1;
  float temp2, temp3, temp4;
  int i, j;

  for (i = 1; i <= e; i += 2) {
    temp1 = fabs(*theeta[(i + 1) / 2] - *theeta2[1]);
    temp2 = fabs(*rho[(i + 1) / 2] - *rho2[1]);
    for (j = 1; j <= e; j += 2) {
      temp3 = fabs(*rho[(i + 1) / 2] - *rho2[(j + 1) / 2]);
      temp4 = fabs(*theeta[(i + 1) / 2] - *theeta2[(j + 1) / 2]);
      if (temp3 < 15.0 && temp4 < 15.0) {
        if (temp2 >= temp3 || temp1 >= temp4) {
          temp1 = temp4;
        }
      }
    }
  }

}}
temp2 = temp3;
*xshift[i] = (int) (*xcor[i] - *xcor2[j]);
*yshift[i] = (int) (*ycor[i] - *ycor2[j]);
*xshift[i + 1] = (int) (*xcor[i + 1] - *xcor2[j + 1]);
*yshift[i + 1] = (int) (*ycor[i + 1] - *ycor2[j + 1]);
*rho2[(j + 1) / 2];
*thetaa2[(j + 1) / 2];

return (0);

/** Function For HOUGH TRANSFORM **/

int
hough(xin, p, n, m, n1, n2, co, si)
{
    int n1, n2, m, n;
    unsigned char *xin[256], *p[256];
    float co[256], si[256];
    for (kk = 0; kk < 256; kk++) {
        for (ll = 0; ll < 256; ll++) {
            *(p[kk] + ll) = 0;
        }
    }
    for (k = 0; k < n1; k++) {
        for (l = 0; l < n2; l++) {
            if (*(xin[k] + l) >= 150) {
                for (i = 0; i < n; i++) {
                    r = k * co[i] + l * si[i];
                }
            }
        }
    }
}
b = SQRTD;
r = r + b;
r = r / (b * 2.0);
r = r * (m - 1);
r = r + 0.5;
j = floor(r);
*(p[i] + j) = (int) (*(p[i] + j)) + 1;
}
}
}
return (0);

/** Function For INVERSE HOUGH TRANSFORM **/

int ihough(xin, xout, p, n1, n2, n, m, co, si, xcor, ycor, rho, thetaa)
    unsigned char *xin[256], *xout[256], *p[256], *xcor[256], *ycor[256];
    float *rho[256], *thetaa[256];
    int n1, n2, n, m;
    float co[256], si[256];
{
    int k, l, i, j, e, a, q, u, v, o, lin, temp, temp2,
        temp3, temp4;
    float r, y, s, t;
    float temp11, temp22, temp33, PI;
    float SQRTD = sqrt((float) n1 * (float) n1 + (float) n2 * (float)
    n2);

    for (i = 0; i < n1; i++) {
        *rho[i] = 0;
        *thetaa[i] = 0;
        *xcor[i] = 0;
        *ycor[i] = 0;

        for (j = 0; j < n2; j++) {
            *(xout[i] + j) = 0;
        }
    }
    a = 0;
e = 0;
lin = 0;
for (k = 0; k < n; k++) {
    for (l = 0; l < m; l++) {
        y = (float) 0.0;

        if (*(p[k] + l) > 150) {
            /* Sub-Routine To Detect Vertical Lines */
            if (co[k] <= (float) 0.016) {
                for (j = 0; j < n2; j++) {
                    for (i = 0; i < nl; i++) {
                        if (*(xin[i] + j) >= 150) {
                            /* Note Down The Start Coordinates Of The Line */
                            a++;
                            if (a == 1) {
                                e++;
                                *(xout[i] + j) =
                                *(xcor[e] + 0) = i;
                                *(ycor[e] + 0) = j;
                            }
                            if (a >= 1) {
                                o = i + 1;
                                q = i + 2;
                                u = j;
                                v = j;
                                if ((*(xin[o] + u) < 150) && (*(xin[q] + v) < 150)) {
                                    /* Length Of The Line Is Less Than 2 Then Delete That Line */
                                    if (a == 1
                                        || (abs(*(xcor[e]) - i) < 2 && abs(*(ycor[e]) - j) < 2)) {
                                        a =
                                        e =
                                        e - 1;
                                    } else {
                                    }
                                } else {
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
e++;  /* Note Down The End Coordinates Of The Line */

*(xcor[e] + 0) = i;
*(ycor[e] + 0) = j;

0;

temp = *(xcor[e - 1] + 0);
temp2 = *(xcor[e] + 0);
temp = temp - temp2;
temp3 = *(ycor[e - 1] + 0);
temp4 = *(ycor[e] + 0);
temp3 = temp3 - temp4;
temp2 = abs(temp);
temp4 = abs(temp3);

*(xout[i] + j) = 255;

if (temp2 > 7) {
    lin = lin + 1;
    *rho[e / 2] = (float) (i * co[k]) + (unsigned char) (j * (fabs(si[k])));
    *thetaa[e / 2] = (float) k;
}
else if (e > 1)
    e = e - 2;
else
    *(xout[i] + j) = 255;
}
}
else
for (i = 0; i < n1; i++) {
    *(xcor[e] + 0) = i;
    *(ycor[e] + 0) = j;
    r = ((float) l * 2.0 * SQRTD / (m - 1))
    / ** Sub-Routine To Detect Horizontal Lines **/
    -0.01) {
        for (j = 0; j < n2; j++) {
            if (*(xin[i] + j) >= 150) {
                if (i > 110 && j >
                    81)
                    i = i;
                    a++;
                    if (a == 1) {
                        e++;
                        *(xout[i] + j) = 255;
                    }
                    if (a >= 1) {
                        o = i;
                        q = i;
                        u = j + 1;
                        v = j + 2;
                    }
if *(xin[o] + u) < 150 \&\& *(xin[q] + v) < 150) {
    if (a == 1 || (abs(*xcor[e]) - i) < 5 \&\& abs(*ycor[e]) - j < 5) {
        a = 0;
        e = e - 1;
    } else {
        e++;
        *(xcor[e] + 0) = i;
        *(ycor[e] + 0) = j;
        a = 0;
        temp = *(xcor[e - 1] + 0);
        temp2 = *(xcor[e] + 0);
        temp = temp - temp2;
        temp3 = *(ycor[e - 1] + 0);
        temp4 = *(ycor[e] + 0);
        temp3 = temp3 - temp4;
        temp2 = abs(temp);
        temp4 = abs(temp3);
        *(xout[i] + j) = 255;
        if (temp4 > 2) {
            lin = lin + 1;
            *rho[e / 2] = (float) (i * (fabs(co[k]))) + (unsigned char) (j * (fabs(si[k])));
        }
    }
}
*thetaa[e / 2] = (float) k;

} else if (e > 1)
    e = e - 2;

} else

*(xout[i] + j) = 255;

} else {

/** Sub-Routine To Detect Lines Other Then Horizontal Or Vertical **/

y = (r - (float) i * co[k]) / si[k];
y += 0.5;
j = floor(y);
if (j >= 0 && j < n2)
    if (*((xin[i] + j) > 150) {

        a++;
        if (a == 1) {
            e++;
            *(xcor[e])
            = i;
            *(ycor[e])
            = j;
            *(xout[i] + j) = 255;

        }

    }

if (a >= 1) {
    o = i + 1;
    q = i + 2;
    s = (r -
    (float) o * co[k]) / si[k];

122
\[ t = (r - (\text{float}) q \cdot \text{co[k]}) / \text{si[k]}; \]

\[ s += 0.5; \]

\[ u = \text{floor}(s); \]

\[ t += 0.5; \]

\[ v = \text{floor}(t); \]

\[ \text{if (u} \geq 0 \&\& \text{u} < n2 \&\& \text{v} \geq 0 \&\& \text{v} < n2) \{ \]

\[ \text{if } (*(\text{xin[o] + u}) < 150) \{ \]

\[ \text{if a} = 1 \| \| (\text{abs}(*(\text{xcor[e]}) - i) < 5 \&\& \text{abs}(*(\text{ycor[e]}) - j) < 5)) \{ \]

\[ a = 0; \]

\[ e = e - 1; \]

\} \text{ else } \{ \]

\[ e++; \]

\[ a = 0; \]

\[ *(\text{xcor[e] + 0}) = i; \]

\[ *(\text{ycor[e] + 0}) = j; \]

\[ \text{temp} = *(\text{xcor[e - 1] + 0}); \]

\[ \text{temp2} = *(\text{xcor[e] + 0}); \]

\[ \text{temp} = \text{temp} - \text{temp2}; \]

\[ \text{temp3} = *(\text{ycor[e - 1] + 0}); \]

\[ \text{temp4} = *(\text{ycor[e] + 0}); \]

\[ \text{temp3} = \text{temp3} - \text{temp4}; \]

\[ \text{temp2} = \text{abs}(	ext{temp}); \]
temp4 = abs(temp3);

if (temp2 > 2 || temp4 > 2) {
    lin = lin + 1;
    *(xout[i] + j) = 255;
    *rho[e / 2] = (float) i * (fabs(co[k])) + j * (fabs(si[k]));
    *thetaa[e / 2] = (float) k;
} else if (e > 1)
    e = e - 2;
}
else {
    *(xout[i] + j) = 255;
    }
} else {
(u >= 0 && u < n2) {
    if (*xin[o] + u) > 150)
        *(xout[i] + j) = 255;
    else {
        if (a == 1) {
            a = 0;
            e = e - 1;
        } else {

e++; 

a = 0;
*(xcor[e] + 0) = i;
*(ycor[e] + 0) = j;
temp = *(xcor[e - 1] + 0);
temp2 = *(xcor[e] + 0);
temp = temp - temp2;
temp3 = *(ycor[e - 1] + 0);
temp4 = *(ycor[e] + 0);
temp3 = temp3 - temp4;
temp2 = abs(temp);
temp4 = abs(temp3);
if (temp2 > 2 || temp4 > 2) {
    lin = lin + 1;
    *(xout[i] + j) = 255;
    *rho[e / 2] = i * (fabs(co[k])) + j * (fabs(si[k]));
    *thetaa[e / 2] = k;
} else if (e > 1)
    e = e - 2;
}
else if (u < 0 || u >= n2) {

    if (a == 1) {
        a = 0;
        e = e - 1;
    } else {
        e++;
    }

    a = 0;
    *(xcor[e] + 0) = i;
    *(ycor[e] + 0) = j;
    temp = *(xcor[e - 1] + 0);
    temp2 = *(xcor[e] + 0);
    temp = temp - temp2;
    temp3 = *(ycor[e - 1] + 0);
    temp4 = *(ycor[e] + 0);
    temp3 = temp3 - temp4;
    temp2 = abs(temp);
    temp4 = abs(temp3);
    if (temp2 > 3 || temp4 > 3) {
        lin = lin + 1;
        *(xout[i] + j) = 255;
    }
}
*\rho[e / 2] = i * (fabs(co[k])) + j * (fabs(si[k]));

*\theta[a[e / 2]] = k;

} else if (e > 1)
  e = e - 2;

/** Delete All The Duplicate Lines **/
for (i = 1; i < e; i += 2) {
  for (j = i + 2; j <= e; j += 2) {
      for (k = j; k < e; k += 2) {
        *xcor[k] = *xcor[k + 2];
        *xcor[k + 1] = *xcor[k + 3];
        *ycor[k] = *ycor[k + 2];
        *ycor[k + 1] = *ycor[k + 3];
        *rho[(k + 1) / 2] = *rho[(k + 3) / 2];
        *thetaa[(k + 1) / 2] = *thetaa[(k + 3) / 2];
      }
      e = e - 2;
      lin = lin - 1;
      j = j - 2;
    }
  }
}

/** Replace RHO And THETA Values By Actual Values Calculated From The Start And End Coordinates Of The Line **/
for (i = 1; i < e; i += 2) {
  if (((*ycor[i + 1]) - (*ycor[i])) == 0) {
    // Code to replace rho and theta values
  }
}
temp11 = (float) (*xcor[i + 1]) - (*xcor[i]);
if (temp11 > 0) {
    temp22 = -(float) (*ycor[i]);
    temp33 = -90.0;
} else {
    temp22 = (float) (*ycor[i]);
    temp33 = 90.0;
}
} else {
    temp11 = (float) (*xcor[i]);
    temp11 -= (float) (*xcor[i + 1]);
    temp33 = (float) ((float) (*ycor[i + 1]) - (float) (*ycor[i]));
    temp11 = temp11 / temp33;
    temp33 = atan(temp11);
    temp22 = (float) ((*xcor[i]) * cos(temp33));
    temp22 += (float) ((*ycor[i]) * sin(temp33));
    PI = 4.0 * atan((double) 1.0);
    temp33 = (float) ((temp33 * 180.0) / PI);
}
*rho[(i + 1) / 2] = temp22;
*thetaa[(i + 1) / 2] = temp33;

/** Joining Of Lines Having Same RHO And THETA And Coordinates Within Same Range */
for (i = 1; i < e; i += 2) {
    v = 0;
    for (j = i + 2; j <= e; j += 2) {
        if (*xcor[j] == 46 && *(xcor[j + 1]) == 51)
            i = i;
        temp = ((fabs) (*rho[(i + 1) / 2]) - (fabs) (*rho[(j + 1) / 2]));
        temp2 = ((fabs) (*thetaa[(i + 1) / 2]) - (fabs) (*thetaa[(j + 1) / 2]));
        temp3 = abs(temp);
        temp4 = abs(temp2);
        if (temp3 < 12 && temp4 < 12) {
            v++;
            if (*thetaa[(i + 1) / 2] <= 0.0) {
                if (*xcor[i] > *xcor[jj]
*xc[i] = *xc[j];
if (*yc[i] > *yc[j])
    *yc[i] = *yc[j];
if (*xc[i + 1] < *xc[j + 1])
    *xc[i + 1] = *xc[j + 1];
if (*yc[i + 1] < *yc[j + 1])
    *yc[i + 1] = *yc[j + 1];

} else {
    if (*xc[i] > *xc[j])
        *xc[i] = *xc[j];
    if (*yc[i] < *yc[j])
        *yc[i] = *yc[j];
    if (*xc[i + 1] < *xc[j + 1])
        *xc[i + 1] = *xc[j + 1];
    if (*yc[i + 1] > *yc[j + 1])
        *yc[i + 1] = *yc[j + 1];

    for (k = j; k < e; k += 2) {
        *xc[k] = *xc[k + 2];
        *xc[k + 1] = *xc[k + 3];
        *yc[k] = *yc[k + 2];
        *yc[k + 1] = *yc[k + 3];
        *rho[(k + 1) / 2] = *rho[(k + 3) / 2];
        *thetaa[(k + 1) / 2] = *thetaa[(k + 3) / 2];
    }
    e = e - 2;
    lin = lin - 1;
    j = j - 2;
}

if (v == 0) {
    if (abs(*xc[i] - *xc[i + 1]) < 7 && abs(*yc[i] - *yc[i + 1]) < 7) {
        for (k = i; k < e; k += 2) {
            *xc[k] = *xc[k + 2];
            *xc[k + 1] = *xc[k + 3];
            *yc[k] = *yc[k + 2];
            *yc[k + 1] = *yc[k + 3];
            *rho[(k + 1) / 2] = *rho[(k + 3) / 2];
            *thetaa[(k + 1) / 2] = *thetaa[(k + 3) / 2];
        }
    }
}
e = e - 2;
lin = lin - 1;
i = i - 2;
}
}
}
j = e + 2;
for (i = 1; i < e; i += 2) {
if (*thetaa[(i + 1) / 2] > 0.0) {
    *xcor[j] = *xcor[i];
    *xcor[i] = *xcor[i + 1];
    *xcor[i + 1] = *xcor[j];
    *ycor[j] = *ycor[i];
    *ycor[i] = *ycor[i + 1];
    *ycor[i + 1] = *ycor[j];
}
}
return (e);

/** Function For Thresholding The Parameter Matrix **/

int thres_par(p, n, m, t)
    unsigned char *p[256];
    int n, m, t;
{
    int a, b;
    for (a = 0; a < n; a++) {
        for (b = 0; b < m; b++) {
            if (*p[a] + b > t)
                *(p[a] + b) = 255;
            else
                *(p[a] + b) = 0;
        }
    }
    return (0);
}

/** Function For Creating The Look-Up Tables For Sine-Cosine Functions
Required By Hough And Inverse Hough Transforms **/
int
look_up_table(co, si, m)
{
    int m;
    float co[256], si[256];

    int i;
    float th;
    float R_TO_D = 0.017453;
    for (i = 0; i < m; i++) {
        th = (float) i * 180.0 / (m - 1) - 90.0;
        th = th * R_TO_D;
        co[i] = (double) cos((double) th);
        si[i] = (double) sin((double) th);
    }
    return (0);
}

/** MAIN **/  

main()
{
    unsigned char *image[256], *image2[256], *image3[256],
                   *outimage[256],
                   *houghmat1[256], *houghmat2[256], *outimage2[256],
                   *xpoint[256], *ypoint[256], *xpoint2[256], *ypoint2[256];
    float         temp1, temp2, temp3, PI, *rhe[256], *thet[256],
                   *rhe2[256], *thet2[256];
    int           minx, miny, maxx, maxi, tempx, tempy, tempx2, tempy2;
    int           a, i, j, rx, sx, sy, ex, ey, testing, line, *shiftx[256],
                   *shifty[256], xmin, ymin;
    float         costita[256], sintita[256], *thet3[256], *rhe3[256];
                   *tempptr;

    for (i = 0; i < 256; i++) {
        outimage[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        outimage2[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
        image[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
    }
image2[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
image3[i] = (unsigned char *) calloc(256, sizeof(unsigned char));
houghmat1[i] = (unsigned char *) calloc(650, sizeof(unsigned char));
houghmat2[i] = (unsigned char *) calloc(650, sizeof(unsigned char));

xpoint[i] = (unsigned char *) calloc(1, sizeof(unsigned char));
ypoint[i] = (unsigned char *) calloc(1, sizeof(unsigned char));
xpoint2[i] = (unsigned char *) calloc(1, sizeof(unsigned char));
shiftx[i] = (int *) calloc(1, sizeof(int));
shifty[i] = (int *) calloc(1, sizeof(int));
ypoint2[i] = (unsigned char *) calloc(1, sizeof(unsigned char));
rhe[i] = (float *) calloc(1, sizeof(unsigned char));
rhe2[i] = (float *) calloc(1, sizeof(unsigned char));
theta[i] = (float *) calloc(1, sizeof(unsigned char));
theta2[i] = (float *) calloc(1, sizeof(unsigned char));
rhe3[i] = (float *) calloc(1, sizeof(float));
theta3[i] = (float *) calloc(1, sizeof(float));

    printf("\n ERROR IN MATRIX ALLOCATION \n");
    exit(1);
}

/** Opening File Having Original Aligned TEST Image **/
inptr3 = fopen("faulty2.raw", "rb");
if (inptr3 == NULL) {
    printf("Error opening faulty.raw as input file\n");
    exit(1);
}

/** Opening Output File Displaying TEST Image With Faulty Area Highlighted **/
outptr3 = fopen("faultdisp.raw", "wb");
if (outptr3 == NULL) {
    printf("Error opening faultdisp.raw as input file\n");
    exit(1);
}
/** Opening Input File Containing Isolated Faulty Area For TEST Image **/
inptr = fopen("faulttest.raw", "rb");
if (inptr == NULL) {
    printf("Error opening faulttest.raw as input file\n");
    exit(1);
}
/** Opening Input File Containing Isolated Faulty Area For GOOD Image **/
inptr2 = fopen("faultgood.raw", "rb");
if (inptr2 == NULL) {
    printf("Error opening faultgood.raw as input file\n");
    exit(1);
}
/** Opening Output File For Displaying The Image After Performing Hough And Inverse Hough Transforms on TEST Image **/
outptr = fopen("houghout1.raw", "wb");
if (outptr == NULL) {
    printf("Error opening houghout1.raw as output file\n");
    exit(1);
}
/** Opening Output File For Displaying The Image After Performing Hough And Inverse Hough Transforms on GOOD Image **/
outptr2 = fopen("houghout2.raw", "wb");
if (outptr2 == NULL) {
    printf("Error opening houghout2.raw as output file\n");
    exit(1);
}
/** Opening Data File Containing The End Coordinates Of The Isolated Images **/
tempptr = fopen("templ.dat", "r");
if (tempptr == NULL) {
    printf("Error opening templ.dat as temporary data file\n");
    exit(1);
}
rx = 25;
for (i = 0; i < 256; i++) {
    fread(image[i], sizeof(unsigned char), 256, inptr);
    fread(image2[i], sizeof(unsigned char), 256, inptr2);
    fread(image3[i], sizeof(unsigned char), 256, inptr3);
}
for (i = 0; i < 256; i++) {
    *rhe[i] = 0;
for (j = 0; j < 256; j++) {
    *(houghmat1[i] + j) = 0;
    *(houghmat2[i] + j) = 0;
    *(outimage[i] + j) = 0;
    *(outimage2[i] + j) = 0;
}

minx = 255;
miny = 255;
maxx = 0;
maxy = 0;
ex = 256;
ey = 256;
sx = 230;
sy = 230;
line = 0;
testing = 0;
look_up_table(costita, sintita, ex);
hough(image, houghmat1, ex, ey, sx, sy, costita, sintita);
thes_par(houghmat1, ex, ey, rx);

/** Reading The End Coordinates Of The Isolated Images **/
fscanf(tempptr, "%d", &xmin);
fscanf(tempptr, "%d", &ymin);

line = ihough(image, outimage, houghmat1, sx, sy, ex, ey, costita, sintita, xpoint, ypoint, rhe, thet);
/** Recalculating The RHO And Theta Values Of The Lines After Joining Different Segments Of The Same Line To Get Its Real Values **/ for (i = 1; i <= line; i += 2) {

    if (((*ypoint[i + 1] + ymin) - (*ypoint[i] + ymin)) == 0) {
        temp1 = (float) ((*xpoint[i + 1] + xmin) - (*xpoint[i] + xmin));

        if (temp1 > 0.0) {
            temp2 = -(float) (*ypoint[i] + ymin);
            temp3 = -90.0;
        } else {
            temp2 = (float) (*ypoint[i] + ymin);
            temp3 = 90.0;
        }
    } else {
        temp1 = (float) ((*xpoint[i] + xmin));
        temp1 -= (float) ((*xpoint[i + 1] + xmin));
        temp1 /= (float) ((*ypoint[i + 1] + ymin) - (*ypoint[i] + ymin));

        temp3 = atan(temp1);
        temp2 = (float) ((*xpoint[i] + xmin) * cos(temp3));
        temp2 += (float) ((*ypoint[i] + ymin) * sin(temp3));

    }** Converting Theta From Radians To Degrees **/
    PI = 4.0 * atan((double) 1.0);
    temp3 = (temp3 * 180.0) / PI;

    *rhe[(i + 1) / 2] = temp2;
    *thet[(i + 1) / 2] = temp3;

/** Calculating Area Of The Actual Fault Location **/ 
 tempx = (*xpoint[i] + xmin);
 tempy = (*ypoint[i] + ymin);
 tempx2 = (*xpoint[i + 1] + xmin);
 tempy2 = (*ypoint[i + 1] + ymin);
 if (minx > tempx)
     if (tempx2 >= tempx)
         minx = tempx;
     if (minx > tempx2)
         if (tempx2 <= tempx)
             minx = tempx2;
 if (miny > tempy)
     if (tempx2 >= tempy)
         miny = tempy;
miny = tempy;
if (miny > tempy2)
  if (tempy2 <= tempy)
    miny = tempy2;
if (maxx < tempx)
  if (tempx2 <= tempx)
    maxx = tempx;
if (maxx < tempx2)
  if (tempx2 >= tempx)
    maxx = tempx2;
if (maxy < tempy)
  if (tempy2 <= tempy)
    maxy = tempy;
if (maxy < tempy2)
  if (tempy2 >= tempy)
    maxy = tempy2;

printf("\n THE COR. OF LINE %d ARE X%d=%d AND Y%d=%d", ((i + 1) / 2), i, (*xpoint[i] + xmin), (*ypoint[i] + ymin));
printf("\n THE END COR. OF LINE %d ARE X%d=%d AND Y%d=%d and the RHO=%.4f & THETA=%.4f ", ((i + 1) / 2), (i + 1),
  (*xpoint[i + 1] + xmin), (i + 1), (*ypoint[i + 1] + ymin), *rhe[(i + 1) / 2], *thet[(i + 1) / 2]);

}
printf("\n\n\n")

hough(image2, houghmatl, ex, ey, sx, sy, costita, sintita);
thres_par(houghmatl, ex, ey, rx);

testing = ihough(image2, outimage2, houghmatl, sx, sy, ex, ey, costita, sintita, xpoint2, ypoint2, rhe2, theta2);
for (i = 0; i < 256; i++) {
  fwrite(outimage[i], sizeof(unsigned char), 256, outptr);
  fwrite(outimage2[i], sizeof(unsigned char), 256, outptr2);
}

/** Recalculating The RHO And Theta Values Of The Lines After Joining Different Segments Of The Same Line To Get Its Real Values **/
for (i = 1; i <= line; i += 2) {
  if (((ypoint2[i + 1] + ymin) - (ypoint2[i] + ymin)) == 0) {
    temp1 = (float) ((xpoint2[i + 1] + xmin) - (xpoint2[i] + xmin));
    if (temp1 > 0.0) {
temp2 = -(float) (*ypoint2[i] + ymin);
temp3 = -90.0;
} else {
temp2 = (float) (*ypoint2[i] + ymin);
temp3 = 90.0;
}
} else {
temp1 = (float) ((*xpoint2[i] + xmin));
temp1 -= (float) (*xpoint2[i + 1] + xmin);
temp1 /= (float) ((*ypoint2[i + 1] + ymin) - (*ypoint2[i] + ymin));
temp3 = atan(temp1);
temp2 = (float) ((*xpoint2[i] + xmin) * cos(temp3));
temp2 += (float) ((*ypoint2[i] + ymin) * sin(temp3));
/** Converting Theta From Radians To Degrees **/
PI = 4.0 * atan((double) 1.0);
temp3 = (temp3 * 180.0) / PI;
}
*rhe2[(i + 1) / 2] = temp2;
*thet2[(i + 1) / 2] = temp3;

printf("\n THE COR. OF LINE %d ARE X%d=%d AND Y%d=%d", ((i + 1) / 2), i, (*xpoint2[i] + xmin), i, (*ypoint2[i] + ymin));
printf("\n THE END COR. OF LINE %d ARE X%d=%d AND Y%d=%d and the RHO=%.4f & THETA=%.4f ", ((i + 1) / 2), (i + 1), (*xpoint2[i + 1] + xmin), (i + 1), (*ypoint2[i + 1] + ymin), *rhe2[(i + 1) / 2], *thet2[(i + 1) / 2]);
}

testing = error_rep(xpoint, ypoint, xpoint2, ypoint2, shiftx, shifty, line, rhe, thet, rhe2, thet2, rhe3, thet3);

printf("\n\n\n");

for (i = 1; i <= (line); i += 2) {
printf("\nThe shift in xcor and ycor of point %d is %d AND %d \n", i, *shiftx[i], *shifty[i]);
printf("\nThe shift in xcor and ycor of point %d is %d AND %d \n", (i + 1), *shiftx[(i + 1)], *shifty[(i + 1)]);
printf("\nThe shift in theta of line%d is %.4f \n", ((i + 1) / 2), *thet3[(i + 1) / 2]);
}
/** Highlighting The Faulty Area In The Output Image Array (Adding White Birder Line On The Faulty Area **/ 

for (a = miny - 2; a < maxy + 3; a++)
    *(image3[minx - 1] + a) = 0;
for (a = miny - 2; a < maxy + 3; a++)
    *(image3[maxx + 1] + a) = 0;
for (a = minx - 2; a < maxx + 3; a++)
    *(image3[a] + (miny - 1)) = 0;
for (a = minx - 2; a < maxx + 3; a++)
    *(image3[a] + (maxy + 1)) = 0;

for (i = 0; i < 256; i++) {
    fwrite(image3[i], sizeof(unsigned char), 256, outptr3);
}

fclose(inptr);
fclose(inptr3);
fclose(outptr);
fclose(outptr2);
fclose(tempptr);
printf("\n\n***** DISPLAYING THE IMAGE WITH FAULTY
AREA HIGHLIGHTED  *****\n");

/** System Call To Remove The Data File **/
system("rawtopgm 256 256 faultdisp.raw > faultdisp.pgm");
system("xv -wait 10 -expand 2 faultdisp.pgm");
system("rm faultdisp.pgm");
system("rm temp1.dat");

} 

/** END OF PROGRAM  **/