OBJECT RECOGNITION USING
BOUNDARY CURVE TRACKING AND FOURIER DESCRIPTORS

by

DAVID GEORGE LUTES

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

THE UNIVERSITY OF MANITOBA

DEPARTMENT OF INDUSTRIAL ENGINEERING

WINNIPEG, MANITOBA

OCTOBER, 1991

ISBN 0-315-77984-5

OBJECT RECOGNITION USING
BOUNDARY CURVE TRACKING AND FOURIER DESCRIPTORS

BY

DAVID GEORGE LUTES

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

ABSTRACT

A computer algorithm for identifying two-dimensional binary images is presented. Objects are first chain-coded using an efficient boundary tracking method. The boundary curve is then parameterized by computing the radial distances to the centroid as a function of curve length. This method of parameterization has several advantages over other existing methods when dealing with objects that are convex and/or have thin radial extensions. With this parameterization, Fourier Descriptors (FD's) are calculated using the Discrete Fourier Transform. The resulting FD's are used to form the coordinates of a multi-dimensional array which will identify the object from a data-base of previously self taught parts. The recognition routine computes only as many FD's that are required to uniquely identify the part, avoiding the unnecessary processing time typically associated with search and match techniques.

ACKNOWLEDGEMENTS

I would like to thank David Young of the Faculty of Engineering Computer Services for his helpful advice and assistance in linking the Turbo C program to the DT-IRIS frame grabber. His patience, clarity and computing experience were greatly appreciated. Craig Muller also provided many helpful hints when I ran into trouble debugging the code. Lastly, I would like to thank Dr. John Jenness not only for his advice, but also for having the enthusiasm and approachable manner that provided an excellent environment to exchange ideas.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Problem Definition

Object recognition systems typically rely on a set of descriptors to identify a part. Although some descriptors are relatively simple (for example, area of an object), others are extracted from more lengthy routines (such as convolution), requiring large amounts of processing time even with specialized hardware. With these processing considerations in mind, a few questions are presented: How do we minimize the number of descriptors needed to identify a given object? How accurate are the descriptors?

Once the descriptors have been calculated, an additional set of problems arise when trying to implement recognition routines: What kind of search method should be used? How many parts can be successfully identified with a given number of descriptors? How reliable is the recognition algorithm?

This report discusses a computer algorithm which addresses each of these questions. The program is split into three routines. The first deals with the problem of generating accurate descriptors that are rotation and size invariant. (This is called the Boundary Tracking/FD Routine). The second provides

a method of teaching object shapes to the system by generating codes from these descriptors. (This is the Teach Mode Routine). The third then identifies the previously taught objects stored in the data-base. (This is the Identification Mode Routine).

1.2 Research Goals

In general, the goal of this project was to develop a reliable and efficient recognition algorithm for binary images. This involves generating accurate descriptors to code the objects, then extracting the correct part identification from a data-base of stored codes.

Fourier Descriptors (FD's) have been selected as suitable descriptors for object coding. Since their introduction in the late 1960's, FD's have been recognized as a very useful tool for object recognition. Given a properly parameterized representation of the object, the Discrete Fourier Transform (or Fast Fourier Transform) will generate a unique set of coefficients which are invariant to size and orientation. However, the resulting FD's are only as accurate as the parameterization. A research goal, therefore, was to develop a parameterizing method which addresses some of the weaknesses of existing techniques.

As previously mentioned, a goal of the recognition algorithm was to "extract" the correct part identification, rather than search for it. Meaning that once a code for the object is generated, its identity is known immediately - no searching and matching is to be performed.

In addition, the recognition routine was to be designed in such a way that it would call descriptor subroutines (such as the Boundary Tracking/FD routine) only until the computer could uniquely distinguish the part from the other parts in the data-base. Therefore only the minimum number of descriptors would be computed, rather than calculating a complete set each time a part is placed under the camera.

Finally, given a clear binary image, it was intended to develop a recognition algorithm that could be 100% reliable, making it attractive for industrial applications. The computer would reserve a set of memory locations for each part based on the maximum error of the descriptors. Any subsequent (new) part taught to the system that enters a previously reserved memory space would be rejected. Therefore, any possibility for error would be detected when teaching the parts to the system. Further details of this idea are presented in Chapter 4 of the report.

2.0 LITERATURE SURVEY

In many machine vision applications, an object is simply represented by its 2-D binary image. From the image, the boundary curve is often used for identification. Examples of this include machine parts recognition [10] and identification of aircrafts [4]. One of the approaches to 2-D boundary curve analysis is to use Fourier Descriptors, a popular method due to the rotation invariant property of the Fourier Transform.

Zahn and Roskies [14] have developed a parameterization method for generating Fourier Descriptors using angular bend as a function of boundary curve length. Persoon and Fu [10] use this method in a recognition algorithm which identifies characters from a data-base of pre-taught characters. Identification is accomplished by selecting the part whose set of FD's have the minimum squared distance from the object under the camera. Jiang and Merickel [6] propose a different parameterization of the boundary curve. This involves finding the centroid of the 2-D object and calculating the radial distance to the boundary edge at each angular increment around 360°.

These boundary parameterizations are often quite good for simple geometric shapes, however their inherent weaknesses become exposed when dealing with shapes that are convex or shapes that have many thin radial projections. Section 3.2 discusses these

limitations in more detail.

There is an abundance of literature dealing with the problem of recognition. A rule-based system for aerial images of airports by McKeown [9] is used to identify domain restricted objects. The restricted domain allows the rule-base to call the descriptor routines in an optimal manner, reducing the processing necessary for object identification. Similar algorithms have been developed by Smyrniotis and Dutta [12]. Draper [2] observes that a more generalized rule-base object recognition system can be developed by invoking 'expert' knowledge bases to extract specific image information. Thus a link is established between high and lower level decision making.

Rule-based systems provide an attractive option when dealing with a restricted domain of objects where the rules can be well defined. However, Niblack and Damian [9] observe that reliable rules cannot be formulated for measures such as texture, shape or region segmentation. For this reason they conclude that for low and mid-level vision, procedural programming is almost always more robust than rule-base programming.

Bolles and Cain [1] propose a recognition algorithm based on finding two or three key features of an object to narrow down the search space. If a few features can distinguish between possible interpretations of a object, a matching approach can determine

the best match between image data and object models. A similar method by Goad [3] uses preliminary feature matching to enhance runtime performance. Goad also develops a method for determining part orientation. A hypothesis about the position and orientation of the object relative to the camera predicts the location of object edges. If these edges are found in the image, a better estimate of camera location can be determined.

These are common approaches to object recognition - generate a descriptor code for the object then search for it in a data-base of previously coded objects. Although there are a number of methods which can be employed to increase the speed and efficiency of the search, all are limited by the simple fact that a search is required. These methods are further limited by their reliability of object identification. Although some are very accurate, there is no guarantee that identification will always be correct.

The recognition algorithm presented in this report uses the descriptors (in this example, FD's) as the coordinates of a large multi-dimensional array that holds part identities in its memory locations. Therefore, once the descriptors have been calculated, the identity of the part is immediately known - processing time is not required for searching, as the identity is simply extracted from the multi-dimensional array. To successfully implement such a system, the maximum possible error range of the

descriptors must be known. A correct calculation of this error will allow the system to maintain 100% reliability. Additional advantages and requirements of this method are discussed in Chapter 4. To begin, however, descriptors must first be generated. The following section provides a brief discussion on the Discrete Fourier Transform and how it can be used to provide the descriptors for our recognition algorithm.

3.0 GENERATING DESCRIPTORS

3.1 Fourier Descriptors

The Discrete Fourier Transform (DFT) is defined as follows [14]:

$$FD(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n)\ e^{(-j\frac{2\pi kn}{N})} \quad \ldots\ldots\ldots (i)$$

where,
- n = index of parameterization
- f(n) = object parameterization function
- k = frequency level, k = 1,2,3...N
- FD(k) = The Fourier Coefficient calculated at frequency level k
- N = The number of terms used

Using Euler's Identity,

$$e^{-j\theta} = \cos\theta - j \sin\theta$$

we can write equation (i) as:

$$FD(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n) \cos(\frac{2\pi kn}{N}) - j\ f(n) \sin(\frac{2\pi kn}{N}) \quad \ldots\ldots\ldots (ii)$$

From equation (ii), we can see that each FD is calculated by summing up all the values of the parameterization function f(n) while holding the frequency level k at a constant value (eg. k = 1). The next FD is calculated by incrementing k and summing all values of f(n) at the new frequency level k = 2. Changing the k value serves to change the weighting value of the cosine and sine functions. Therefore, each FD is calculated by taking the set of all values in f(n) and applying a unique weighting pattern to it according to the k value. This is the essence of the rotation invariant property of the Fourier Transform. The values of f(n) are not analyzed individually, but rather as an entire set of values having a characteristic response to a frequency level.

The number of FD's one wishes to generate (and therefore the number of k values used), is indicated by the number of terms (N) used. The selection of the N value is determined mainly from a trade-off between processing time and accuracy of results. Obviously, the more FD's we calculate, the more information we will obtain about the object. However, the processing time increases exponentially with the number of terms, since N^2 calculations are required. The Fast Fourier Transform (FFT) can reduce the processing load to $N\log_2 N$ calculations and is therefore a common substitute for the DFT (see Appendix B). For the purpose of this paper, we will simply use the DFT while keeping in mind the availability of the FFT to increase the speed. We will however use N values which are powers of 2 (ie. N = 16,32,64,128...) since this

is a requirement when implementing the FFT.

Each of the resulting FD's yield both a real and imaginary component. If we compute the magnitude of these two components, we find the result to be constant for the same object, regardless of size, orientation and starting point. Starting point refers to which value in the set f(n) is used as f(0). This value could be any element of f(n) depending on the orientation. For additional reading on FD's and rotation invariance, see Appendix D.

From the real and imaginary components, the corresponding phase angles can also be calculated. The phase angles, however, must be normalized since they shift when the orientation changes. This normalization process requires additional process time and even then does not provide constant results when using symmetrical objects [4]. Furthermore, most of the information about the shape of the object is contained in the magnitudes which confirms our decision to ignore the phase angle calculations.

Therefore, the end result is a set of magnitudes, one for each frequency level, that are characteristic of the shape of the object. For example, objects having rapidly changing values in f(n) will yield larger magnitudes at the higher frequency levels, while magnitudes at the lower frequency levels provide more information about general shape. Once the FD's have been generated, the question of what to do with them is an entirely new

problem. Before attempting to address this problem, the object parameterization function f(n) must be first determined.

3.2 Object Parameterization

The accuracy of the FD's are only as good as the accuracy of the parameterizing function. It would be nice to parameterize the object with as many variables as we wish, however each additional variable increases processing time in the DFT by an order of N. For this reason, single variable functions are used most often. Therefore we will find a way to accurately represent the object using only one variable.

Method (i):

One method is to locate the centroid of the object and calculate the radial length to the boundary as the angular displacement is incremented, thus yielding radius as a function of angle. This method is sufficient for simple shapes, however it becomes useless when dealing with objects having thin radial projections extending from the centroid at angles that are multiples of the resolution of the parameterization. The maximum possible error for this method can be determined by examining the following worst case situation.

Consider a thin rod one pixel thick and 500 pixels long. For this example, we shall use N = 256 terms, giving us angular increments of 1.40625°. In this first orientation the radial distances from the centroid as we approach the end of the rod are:

$$f(n) = \text{radial dist} = \{...0,\ 0,\ 0,\ 250,\ 0,\ 0,\ 0...\}$$

Now, if the rod is rotated 0.703125° (one-half of the angular resolution), the values of f(n) at the end of the rod become:

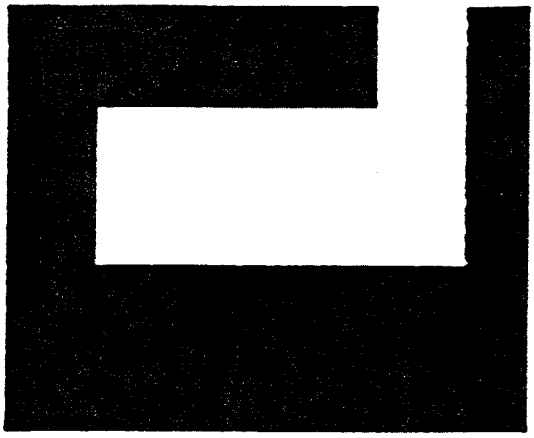
$$f(n) = \text{radial dist} = \{...0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0...\}$$

All information about the object is lost when the rod is rotated

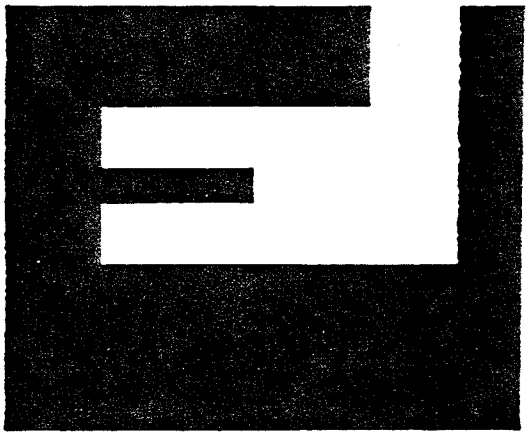
at n(r) + .5(r) degrees, where n = 0,1,2...N-1, and r = 360°/N.

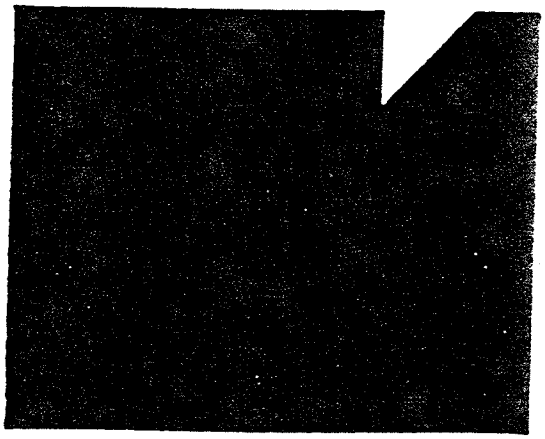
Another limitation to this technique occurs when viewing convex objects that have features hidden from the line of sight between the centroid and boundary. Figure 1a is a typical example that exposes this weakness. Since there is no knowledge of the boundary curve coordinates, the computer must search outward from the centroid along each radial length to find the boundary edge. As only one edge point can be recorded per angular increment, it becomes apparent that this method has limitations when more than one boundary edge is detected along a radial length. If, for example, only the outermost edges are recorded, the convex part of the object in figure 1a is ignored. This method, therefore, could not distinguish between other similar objects that have differences in this convex region. Figures 1b and 1c are examples of images that would produce the same parameterization function f(n). Although this problem does not reduce the repeatability of the method, it severely limits the ability to handle a wider range of object types.

Figure 1a - Convex 2D Object



Figures 1b, 1c - Objects Yielding Equivalent Values for f(n) When Using Method (i)





Method (ii):

Another method is to calculate angle change values along the boundary to yield a parameterization of bend as a function of curve length. However it is difficult to obtain accurate and repeatable angle values from boundary curves having many sharp bends.

If the curve is to be sampled using N terms, N evenly spaced points along the curve must be selected for angle calculations. Due to the spacing between the points, there is a potential for the entire set of sample points to shift when an object is rotated (i.e. a different starting point is selected).

Consider the object shown in figure 2a. Each straight line segment of the object is 50 pixels long, yielding a boundary curve length of L = 1600 pixels. If N = 32 terms are used, then the boundary curve will be sampled at every 1600/32 = 50 pixels (sample points are indicated by dots along the object boundary in the figure). For each value of f(n), the angles at two adjacent sections of the curve must be subtracted from one another to yield an angle change value. In the first orientation, the angle change values (starting at sample point 1 shown in figure 2a) are:

$$f(n) = \text{angle change}_{\Delta} = \{180,-90,-90,180,-90,-90,180....N\}$$

Figure 2a

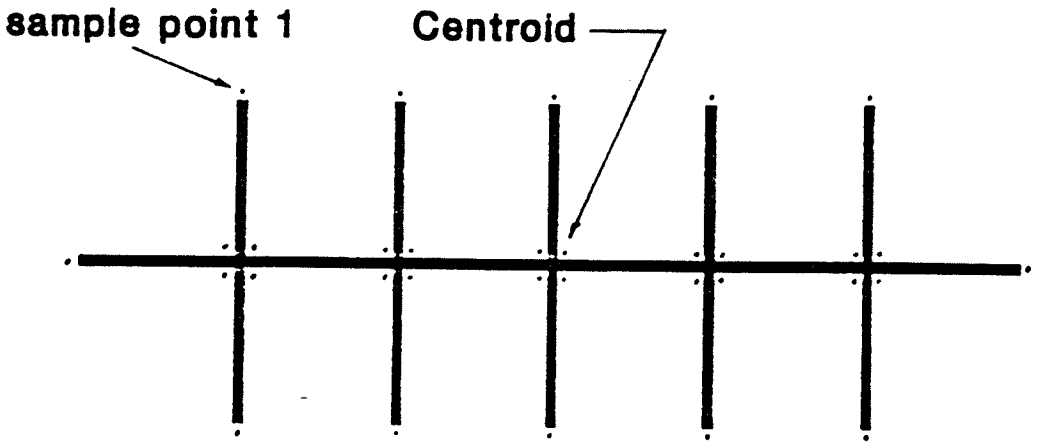
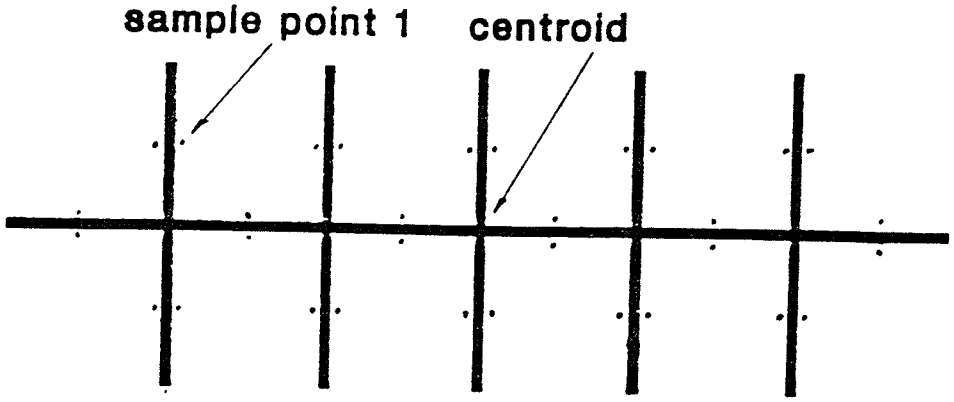


Figure 2b



If the sample points now shift due to an orientation change of the object, there is a potential for a large error in the angle change values. In a new orientation (figure 2b) it is possible that the starting point could shift by .5(L/N) = .5(1600/32) = 25 pixels (one-half a sample spacing). The angle change values are now:

$$f(n) = \text{angle change}_b = \{45,-90,45,45,-90,45,45,-90.....N\}$$

The average error per term between the two orientations is equal to 101.25 degrees. When considering the 360° range of angle change values, this translates to a %error of:

$$\%\text{error } f(n) = \frac{\text{error } f(n)}{\text{range } f(n)} = \frac{101.25^\circ}{360^\circ} \times 100 = 28.1\%$$

For objects with simple geometry, this method is quite adequate. However, a high error is realized on objects having many sharp bends, reducing the chances of generating reliable descriptors.

Method(iii):

A third method, and the one presented here in this report, is to calculate the radial distance from the centroid to selected sample points on the boundary curve, yielding a parameterizing function f(n) that describes radial distance as a function of curve length. This technique has a maximum possible error that is much less than the two methods previously described. Since the coordinates of the boundary curve are known, there is no loss of information when dealing with thin rods or convex objects such as in method (i). Furthermore, since radial distances are used, method (iii) is not as sensitive to the shifting of the sample points along the boundary as is found in the angle calculations of method (ii).

If the object in figure 2a is again considered in the first orientation (using N = 32), the radial distances as we start from sample point 1 are:

f(n) = radial dist to boundary
= {111.8, 100, 50, 70.7, 50, 0, 50, 0, 50, 70.7, 50,100...N}

If the object is now rotated such that the sample points along the curve shift by .5(L/N) = 25 pixels (figure 2b), the radial distances are now:

f(n) = radial dist to boundary
= {103.1, 75, 55.9, 55.9, 25, 25, 25, 25, 55.9, 55.9,75...N}

The average error per term between the two orientations is equal to 16.1 pixels. Considering the range of f(n) values, the percentage error for this object is:

$$\%\text{error } f(n) = \frac{\text{error } f(n)}{\text{range } f(n)} = \frac{16.1 \text{ pixels}}{150 \text{ pixels}} \times 100 = 10.7\ \%$$

The %error is much less than in method (ii). Recognizing the advantages of this technique, we shall parameterize our objects by computing radial distance as a function of boundary curve length.

In the next section an algorithm will be developed to perform curve tracking, allowing us to obtain the coordinates of the boundary points. This will provide the preliminary information required to compute the radial distances from the centroid.

3.3 Boundary Tracking

A computer program has been written to perform curve tracking of the object boundary. The flowchart shown in figure 3 will be used as a guide to explain the programming functions. The algorithm uses a blank 3x3 window that scans the binary image from the top to bottom and left to right until it encounters a "1" in the lower-right corner. The center pixel coordinates are then designated as the starting point. The 3x3 window will track the curve in a clockwise direction, generating a pixel-by-pixel 4-link chain code until the center pixel encounters the original starting point.

The moves of the tracking window are coded as 1,2,3 and 4 corresponding to forward, up, back and down respectively. Before each move, the algorithm examines the contents of the window to determine the required directional step. There are 8 pixels surrounding the center of the 3x3 window, allowing a total of 2^8 = 256 possible combinations of "1's" and "0's". Obviously, it is too time consuming to check every single combination for every pixel step around the curve, therefore a shortcut would be nice. The analysis of the window can be simplified significantly if the current window orientation is known. (This is not to be confused with the orientation of the object). Figure 4 shows the four possible window orientations which can be assigned to the current

Figure 3

Flowchart for Boundary Tracking and FD Generation

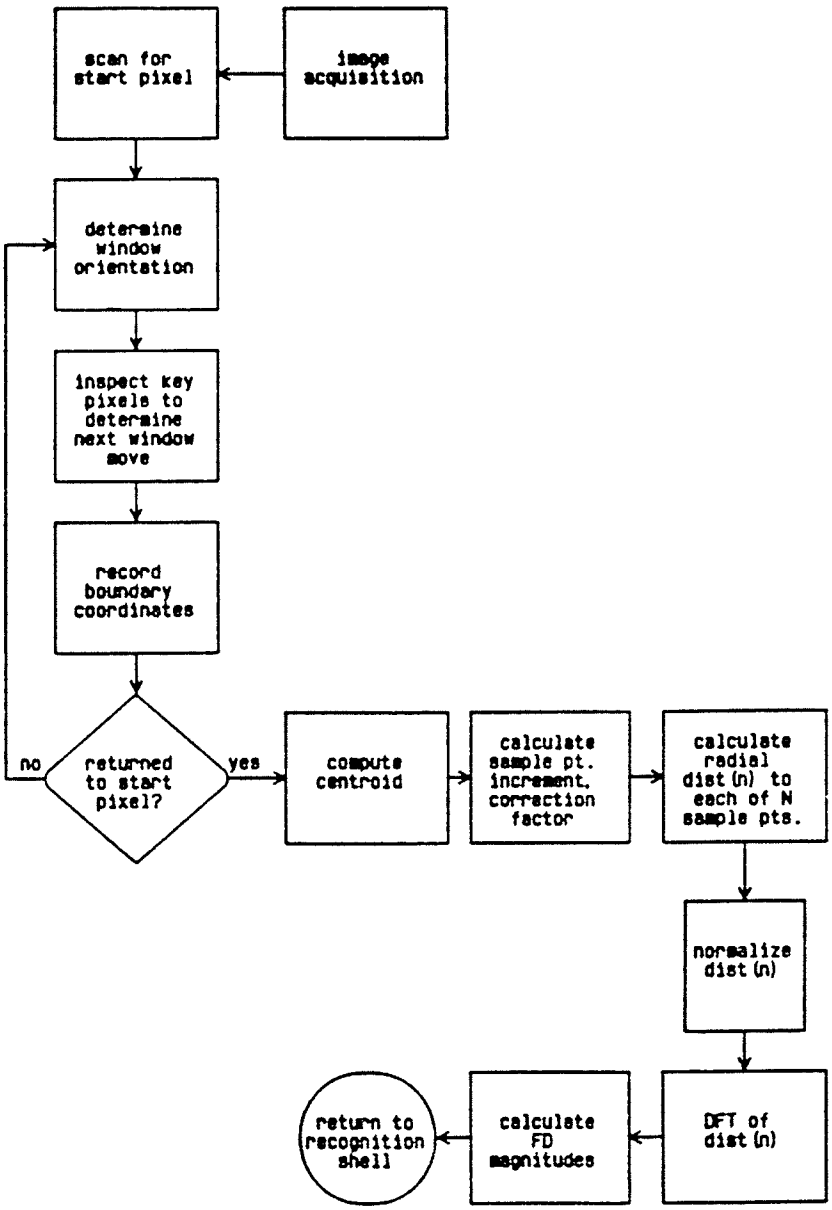
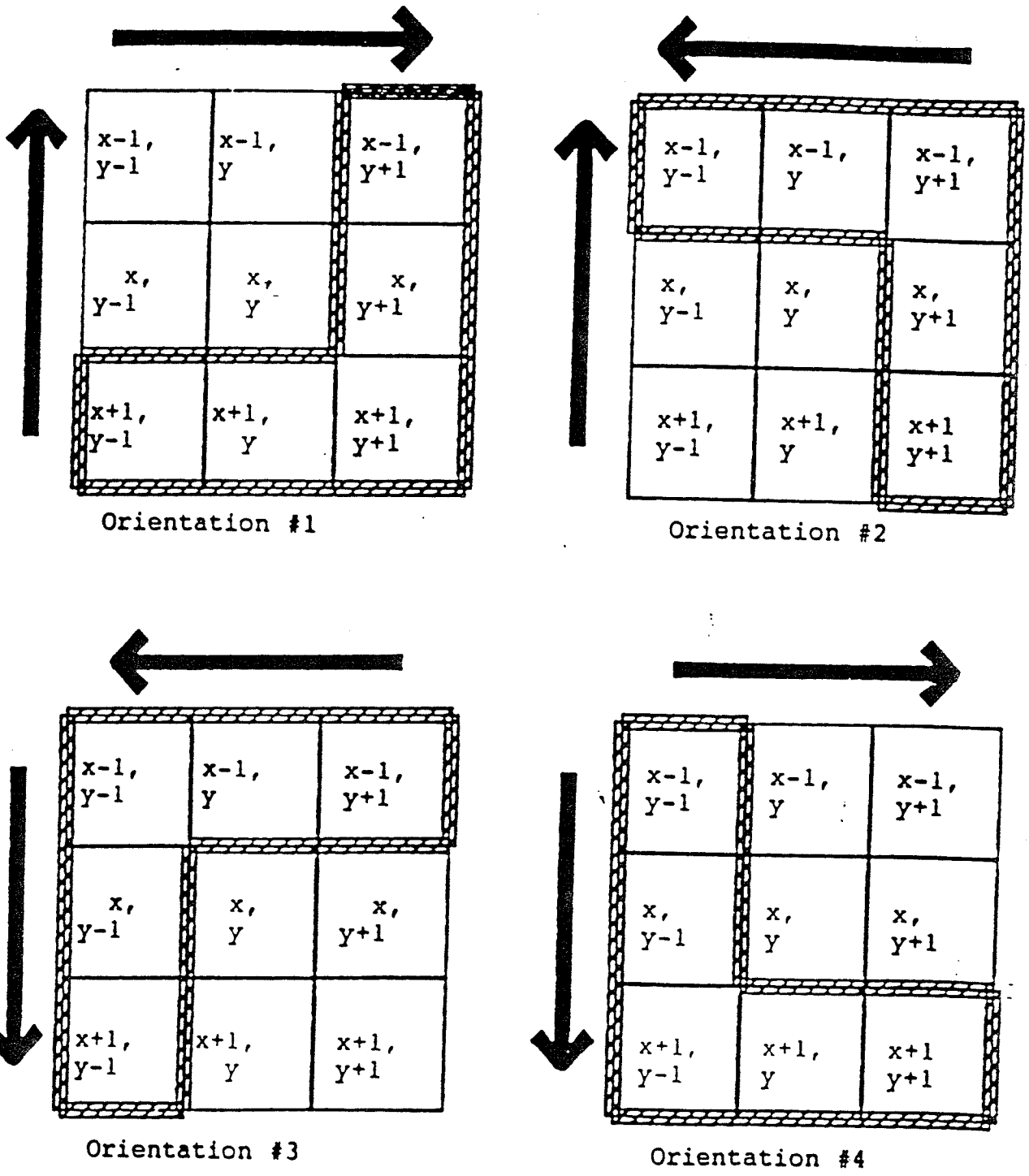


Figure 4 Window Orientations



contents of the window. Each orientation provides description of the situation the tracking window is currently in. For example, from figure 4, we can see that if we are in orientation 1, the window is considered to be tracking in either the forward or upward directions. Once this is determined, the correct move is established by inspecting the right-middle pixel for the presence of a "0". Here, a "0" would indicate there is an open path for a forward move (chain code 1), otherwise the window must move up (chain code 2).

Like all 4 orientations, orientation 1 can be determined by examining the corner pixels of the window. The total number of "1's" in the corners provide a good starting context. Once the corner number is known (ie. corners = 1 or 2 or 3), it is merely a matter of inspecting key pixels that will confirm the orientation. The key pixels for each orientation are the five contained within the hatched lines in figure 4. If the orientation and the contents of the key pixels are known, the next window move can be easily determined.

The generation of the 4-link chain code is contained within the while loop starting on page 66 in Appendix A. Since the image is scanned from top to bottom and left to right, the starting orientation is pre-set to 1. Once the chain code move is determined and the x and y coordinates are recorded, the new configuration of the window is examined and the appropriate

orientation is assigned. The while loop continues until the starting point is encountered at the center pixel.

It should be mentioned that this edge tracking algorithm is not restricted to the boundary curve. The 3x3 tracking window will follow any line or edge it comes into contact with and simply stop when it returns to the starting point. Furthermore, the curve need not be closed since, given any line, the window merely traces around each side of the line until it returns once again to the initial point. Thus, every line can be considered a closed curve, allowing the algorithm to be applied to edge images also. For the purposes of this report, however, we will only use the tracking algorithm on boundary curves.

The next step is to calculate the centroid of the object. The centroid can be easily found by:

$$centroid_x = \frac{moment_x}{mass} = \frac{\sum_{x=\min_x}^{\max_x} \sum_{y=\min_y}^{\max_y} x\, M(x,y)}{\sum_{x=\min_x}^{\max_x} \sum_{y=\min_y}^{\max_y} M(x,y)} \quad .....(iii)$$

where,

$$object\ mass = M(x,y) \in (0,1)$$

and similarly for $centroid_y$

The values of min_x, min_y, max_x, max_y, are found during the boundary tracking while loop (Appendix A, page 68).

Radial distances to selected sample points on the boundary must be calculated now. The locations of these sample points will be spaced such that different image sizes will still generate the same number of terms. For example, if we wish to obtain 256 terms (ie. 256 FD's) for the curve, we must calculate 256 radial distances around the curve regardless of the current size of the object. To accomplish this, the total curve length must be known - this was easily obtained during the chain code generation (number of chain codes = curve length). The total length is divided by the desired number of terms, yielding the increment value that is used to advance to the next sample point where a new radial distance will be calculated. The fractional part of this division is used to establish a correction factor that will provide slight adjustments to the increment value as we cycle through the chain code. The "C" language code for these calculations appears on pages 73 and 74 of Appendix A. This block of code sets up the increment values and correction factors used for the radial distance f(n) calculations.

After the desired number of radial distances have been calculated, they must be normalized for variations in object size. This is accomplished simply by dividing each element in

f(n) by the average radial distance. The DFT or FFT can now be applied to the normalized f(n) values to produce a set of FDs. The accuracy of these FDs are presented in the next section.

3.4 Descriptor Error

The error produced at a given frequency level is dependent on the object shape. For example, if we consider a perfect circle, there is no error at any of the frequency levels, since shifting of the sample points does not affect the radial distances from the centroid. However, a straight line , for example, will produce a high error at frequency level k = 2. To understand the reason for this, a closer look at the behavior of the sine and cosine waves in the Fourier Transform is required.

Figure 5 shows the sine and cosine weighting patterns (at k=2) as they apply to each term in f(n). In this example we will use N = 8 terms. At frequency level k=2, the Fourier Transform will evaluate and sum the sine and cosine components of f(n) as it cycles through two periods. A maximum error will be achieved when the sign of the error value for each element in f(n) follows the same sign change pattern as the sine or cosine waves.

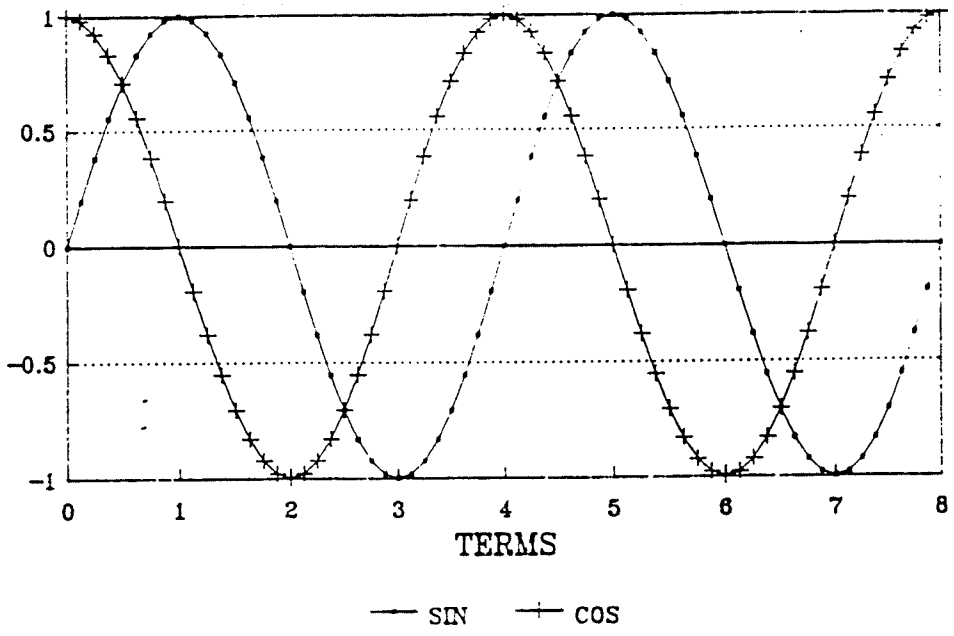


Figure 5 Weighting Patterns at k=2, N=8

For example, if the error of each term in f(n) is 5.0, the Fourier Transform will produce a maximum error at k=2 when the sign change pattern is:

$$error(n) = \{5.0,5.0,-5.0,-5.0,5.0,5.0,-5.0,-5.0\}.$$

In other words, the total error will be maximized at level k when the sign (or direction) of each elemental error in f(n) changes at every (L/N) / 2*k sample points along the boundary curve. (Note that only the pattern of positive and negative values is significant. Any element of error(n) could be selected as the starting point error(0).)

Referring back to the straight line example, it becomes apparent why a maximum error occurs at k=2. Consider the

straight line 500 pixels in length with its centroid located at midpoint. After the boundary tracking, a curve length of L=1000 pixels is obtained. Sampling the curve using N=8 terms will yield a maximum shifting error of .5(L/N) = .5(1000/8) = 62.5 pixels. Therefore, in one orientation the distances to the boundary in f(n) could be:

$f(n)_1 = \{0, 125, 250, 125, 0, 125, 250, 125\}$

Normalizing f(n) by dividing each element by the average radial distance of 125 yields:

$f(n)_1 = \{0,1,2,1,0,1,2,1\}$

In a second orientation, it is possible that the sample points could shift by as much as 62.5 pixels, yielding:

$f(n)_2 = \{62.5, 187.5, 187.5, 62.5, 62.5, 187.5, 187.5, 62.5\}$

Normalizing $f(n)_2$ yields:

$f(n)_2 = \{.5, 1.5, 1.5, .5, .5, 1.5, 1.5, .5\}$

The error between the two orientations is:

$error(n) = \{.5, .5, -.5, -.5, .5, .5, -.5, -.5\}$

For a straight line, the error changes sign at every (L/N) / 2*2 sample points, thus producing a maximum error at k=2.

To maximize the error at the other frequency levels, it is

simply a matter of determining which shapes will yield maximum shifting errors of .5(L/N) pixels that change sign every (L/N) /2*k sample points. To obtain a maximum shifting error, objects having only thin line radial projections from the centroid must be used. This will ensure that sample points will always shift directly towards or away from the centroid. The straight line has two radial projections from its centroid. Three radial projections from the centroid will generate errors that change sign at every (L/N) / 2*3 sample points and therefore will produce a maximum error at level k=3. Figure 5 shows the shapes which will produce maximum errors at each frequency level indicated. Note that for k=1, the shape required is a thin line with its centroid at one end.

A small computer routine has been written to calculate the maximum normalized error at a frequency level indicated by the user, and to propagate this error through the Fourier Transform (see Appendix A). The program was run for the first ten frequency levels using 256 terms. A summary of the results is shown in Table I.

Note that the error increases linearly as the frequency levels increase. This is due to an increase in the normalized shifting error found on the objects that maximize the higher frequencies (such as the shapes shown for k=5, k=6).

Figure 5 - Shapes Producing Maximum Error Magnitudes at Frequency = k

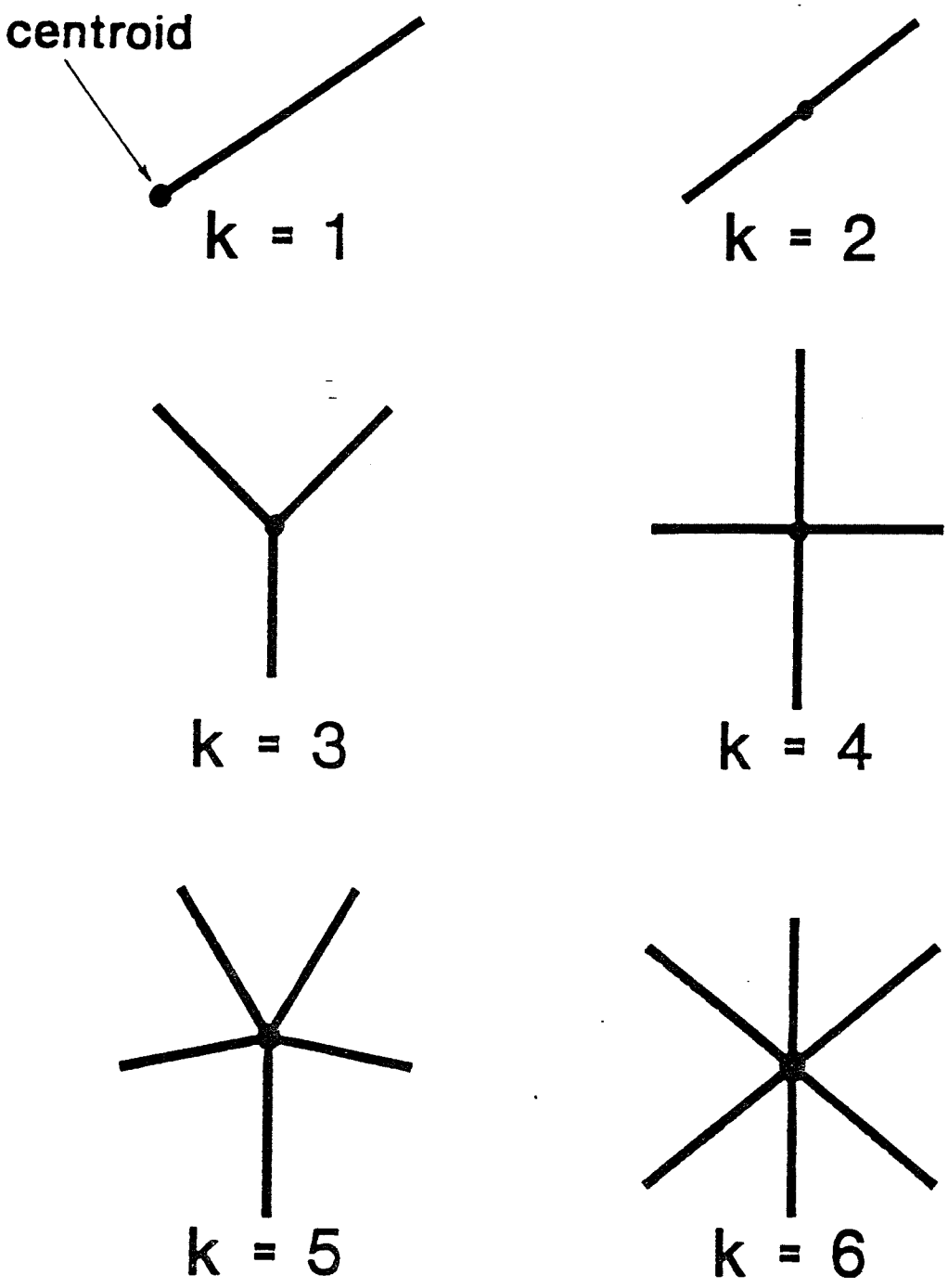


TABLE I

Frequency Level k	Normalized Error, err(k)
1	0.255
2	0.764
3	1.255
4	1.764
5	2.273
6	2.782
7	3.291
8	3.800
9	4.309
10	4.818
⋮	⋮
256	0.0000

The last frequency level k=N, represents the average radial distance. As indicated in Table I, there is no error associated with the last frequency level. The average radial distance will not be affected by the shifting of sample points along the boundary curve, provided that the sampling frequency is at least twice the highest frequency in the image. For example, for a straight line, the highest frequency is only 2. Therefore, at least 2*2=4 terms must be used to ensure that the average radial distance remains constant between rotations. Although it is highly unlikely that we will encounter images having frequencies greater than 256/2 = 128, it should be noted that some error will occur in the normalizing factor if, for example, the object is a gear with more than 128 teeth, or a star with more than 128 points. For the purpose of this report, the maximum frequency in the images will be restricted to less than 128. If greater image frequencies are to be encountered, the number of terms should be

increased to maintained a constant average radial length between rotations.

With these error calculations, we are now ready to set up a recognition algorithm that will store part identities in a multi-dimensional array. Details of how this is accomplished, and the resulting advantages and limitations, are discussed in the following chapter.

CHAPTER 4 OBJECT RECOGNITION

4.1 Multi-Dimensional Array Concept and Redundant Coding

As mentioned earlier in Chapter 2, the recognition algorithm presented in this report uses the FDs as the coordinates of a large multi-dimensional array that holds part identities in its memory locations. Once the descriptors have been calculated, the identity of the part is immediately known - processing time is not required for searching. To successfully implement this idea, the maximum possible error range at each frequency level must be known. A correct calculation of this error will allow the system to maintain 100% reliability.

Thus far we have calculated the descriptor error for each frequency level. The foundation of the recognition algorithm is based on how the real numbered FDs are classified into integer coordinates of the multi-dimensional array. This classification uses the error calculations to determine which memory locations a given object will occupy.

To classify the FD's into array coordinates, intervals of real numbers must be defined such that each interval corresponds to an integer value. Let the range R of an FD be subdivided by B boundaries $b_1, b_2 \ldots b_B$, each spaced at equal distances d. If $c_1, c_2 \ldots c_D$ are integer coordinate values, where D = the number

descriptors being used, each of the real numbered descriptors $FD_1, FD_2 \ldots FD_D$ can be classified as follows:

```
FOR k = 1  TO D
  FOR j = 1  TO B
     IF      b_j <= FD_k < b_j+1
     THEN    c_k = j
  END LOOP
END LOOP
```

Consider the following example where we wish to teach a part to the system (we shall call this the teach mode). Table II is a sample set of 6 FD's (D = 6) generated using method (iii) described in section 3.2.

TABLE II

freq. level k	FD_k
1	7.5
2	16.7
3	1.4
4	34.0
5	19.5
6	2.2

Let the real number range R of the FD_k values be 0.0 to 50.0. If we subdivide the positive real number range with 5 boundaries (B = 5) and set $b_1 = 0.0$, $b_2 = 10.0$, $b_3 = 20.0$, $b_4 = 30.0$, $b_5 = 40$, the resulting integer code is $c_1 = 1$, $c_2 = 2$, $c_3 = 1$, $c_4 = 4$, $c_5 = 2$, $c_6 = 1$. This object would then be stored in memory location array[1][2][1][4][2][1]. Clearly, this approach is insufficient since the error at each frequency level is greater than zero, allowing an object to be coded differently when

differently when different orientations are encountered. What is required is to incorporate an error space that extends a distance of +/- err(k) from the boundaries b2,b3...b_B:

TEACH MODE CLASSIFICATION SCHEME

```
FOR k = 1  TO D
  FOR j = 1  TO B
    IF    b_{j-1} + err(k) <= FD_k < b_j - err(k)
    THEN  c_k = j
    IF    b_j - err(k) <= FD_k < b_j + err(k)
    THEN  c_k = j or j+1
  END LOOP
END LOOP
```

Therefore, if an FD_k value is detected within an error space (ie. between b_j - err(k) and b_j + err(k)), the object can be identified using either one of the two array coordinates c_k = j, or c_k = j+1. Using the error values calculated in section 3.4, err(k) = {.255, .764, 1.255, 1.764, 2.273, 2.782}, the object would now be stored in 2 different memory locations:

part[1][2][1][4][2][1] part[1][2][1][4][3][1]

The FD_5 = 19.5 value has fallen within the error space defined by:

b_3 - err(5) = (20 - 2.273) = 17.727 and

b_3 + err(5) = (20 + 2.273) = 22.273

The object can now be identified using c_5 = 2, or c_5 = 3. These

redundant codes account for the maximum error that could occur when the object is rotated. An error spacing of b_j +/- err(k) must be used in the teach mode to ensure that when this same object is presented to the camera in the identification mode, it is guaranteed that the code generated will be one of the two shown above, regardless of the part's orientation.

Now in the identification mode, the classification scheme is simply:

IDENTIFICATION MODE CLASSIFICATION SCHEME

```
FOR k = 1  TO D
  FOR j = 1  TO B
     IF      bj <= FDk < bj+1
     THEN    ck = j
  END LOOP
END LOOP
```

Error spaces are not required in the identification mode since the teach mode has eliminated any possibility of misclassification. This will allow the identification mode to immediately classify the FD_k values to integer coordinates.

Of course, there is a chance that a new part taught to the system will collide with any existing memory location containing a part identity. If a collision is detected, the computer will inform the user that there are not enough descriptors to code the part. Further details about dealing with collisions are discussed in section 4.4, Data-base Capacity.

4.2 Computing a Minimum Number of Descriptors

As mentioned earlier, there are advantages to using descriptors as coordinates of an identity array. The most obvious is that search time is not required since the array coordinates can immediately extract the part identity from the data-base. In addition to this, the algorithm can also exploit the fact that sometimes only a few descriptors are required to identify a part.

In the teach mode, every descriptor is calculated and classified as a coordinate(s). However, in the identification mode it not always necessary to compute every descriptor. Often, there is sufficient information to distinguish a part from the others in the data-base after computing only two or three descriptors. To take advantage of this, we must develop a scheme to store part identities after each descriptor is calculated.

For example, if we wish to teach parts to the system using 6 FD's, with each FD subdivided by 5 intervals and 4 error spaces, the identity array is:

$$part[c_1][c_2][c_3][c_4][c_5][c_6]$$

where each c_k = 1,2,3,4 or 5

If we initially set $c_1, c_2 \ldots c_6 = 0$, then compute FD_1 and classify it as the coordinate c_1, we can store the part's identity in memory location part[c_1][0][0][0][0][0]. After computing the next descriptor FD_2, the part's identity can be stored in another memory location part[c_1][c_2][0][0][0][0]. (Note that any redundant codes generated during this process would also be stored.) The process can be repeated until all of the descriptors are calculated, resulting in an identification code for each descriptor level 1,2...6. If this was the only part taught to system, then clearly the part could be later identified (in the identification mode) after the first descriptor without requiring any further FD calculations. When more than one part is taught to the system, the memory locations generated at each descriptor level must be inspected and altered in the teach mode as follows:

TEACH MODE MEMORY LOCATION INSPECTION

IF part[c_1][c_2][c_3][c_4][c_5][c_6] = unoccupied
THEN part[c_1][c_2][c_3][c_4][c_5][c_6] = part identity

IF part[c_1][c_2][c_3][c_4][c_5][c_6] = occupied
THEN part[c_1][c_2][c_3][c_4][c_5][c_6] = -1

If a memory location already contains a part identity, its contents are changed to -1, indicating the occupying part can no longer be uniquely distinguished at the current descriptor level. Thus when a new part is taught to the system, the teach routine will store the part's identity at the first unoccupied memory

location defined at descriptor level k, and at locations defined by each subsequent level until the last level k = 6 is reached.

Once the parts have been taught to the system, the identification mode is invoked and the following questions are asked at each descriptor level:

IDENTIFICATION MODE MEMORY LOCATION INSPECTION

IF part[c_1][c_2][c_3][c_4][c_5][c_6] = a part identity
THEN part is found

IF part[c_1][c_2][c_3][c_4][c_5][c_6] = -1
THEN compute next descriptor

After each descriptor is computed in the identification mode, the corresponding memory location is checked for a part identity. If a part identity is present, the part has been found, otherwise another descriptor must be computed. Thus in the identification mode, only the minimum number descriptors are calculated, saving processing time whenever possible. Of course, as more and more parts are taught to the system, there is a decreasing chance of identifying parts at the lower descriptor levels. The following section discusses the capacity of the data-base and what options are available when it becomes saturated.

4.3 Data-base Capacity

If k descriptors are used, each uniformly distributed over a range R and subdivided into n intervals, the number of memory locations available is n^k. Using the example of 6 descriptors subdivided into n = 5 intervals yields 5^6 = 15625 memory locations. The actual capacity, however is much less than this due to the generation of redundant codes. If, say, each part on the average has six possible codes, this number would be reduced by 5/6, yielding 2604 possible parts. The capacity is simply:

$$\text{capacity} = \frac{n^k}{\text{code}_{ave}}$$

where, code_{ave} = average number of possible codes per part

Capacity is dependent on the average number of possible codes per part, which in turn depends on the size of the error spaces 2*err(k) at each frequency level. The more accurate the descriptors are, the greater the capacity. The average number of possible codes per part can be statistically calculated using the following equation:

$$code_{ave} = \sum_{h=0}^{D} \frac{D!}{h!\,(D-h)!}\,(1-p)^{(D-h)}\,p^h\,2^h \ldots\ldots (iv)$$

where, D = total number of descriptor levels used

h = number of descriptor levels having an FD detected within an error space

p = average probability of an FD landing within an error space at frequency levels k = 1,2,3...D

$$p = \frac{1}{D} \sum_{k=1}^{D} \frac{2*err(k)*(n-1)}{R} \quad \ldots\ldots\ldots\ldots (v)$$

R = range of descriptor

n-1 = number of error spaces

Equation (iv) provides a weighting factor for each possible number of redundant codes that could be generated (i.e. 1, 2, 32, 64, 128...2^h). Consider an example, using k = 6 FD's and the err(k) values calculated in section 3.4. If each descriptor range R = 50 and is subdivided into n = 5 intervals, we can calculate p from equation (v):

$$p = \frac{1}{6} \sum_{k=1}^{6} \frac{2*err(k)*(5-1)}{50} = .252$$

Now, using equation (iv), we find $code_{ave}$ = 3.85 codes. The capacity of our data-base is then:

$$\text{capacity} = \frac{n^k}{code_{ave}} = \frac{15625 \text{ memory locations}}{3.85 \text{ memory locations/object}} = 4058 \text{ objects}$$

There is enough room in the data-base to theoretically fit 4058 objects into the data-base. The problem of trying to 'fit' object codes into a data-base can be best explained by considering a 2-dimensional coordinate system of area = n^k representing the data-base, and smaller areas = $code_{ave}$ representing each object. When an object is taught to the system, its area is placed on the 2-D plane. As more parts are taught to the system, more object areas cover the plane. If, at any time, one area overlaps with another (equivalent to a collision a the last descriptor level), the current object being taught to the system must be removed. Thus, every part has a reserved memory area that cannot be accessed by any other part.

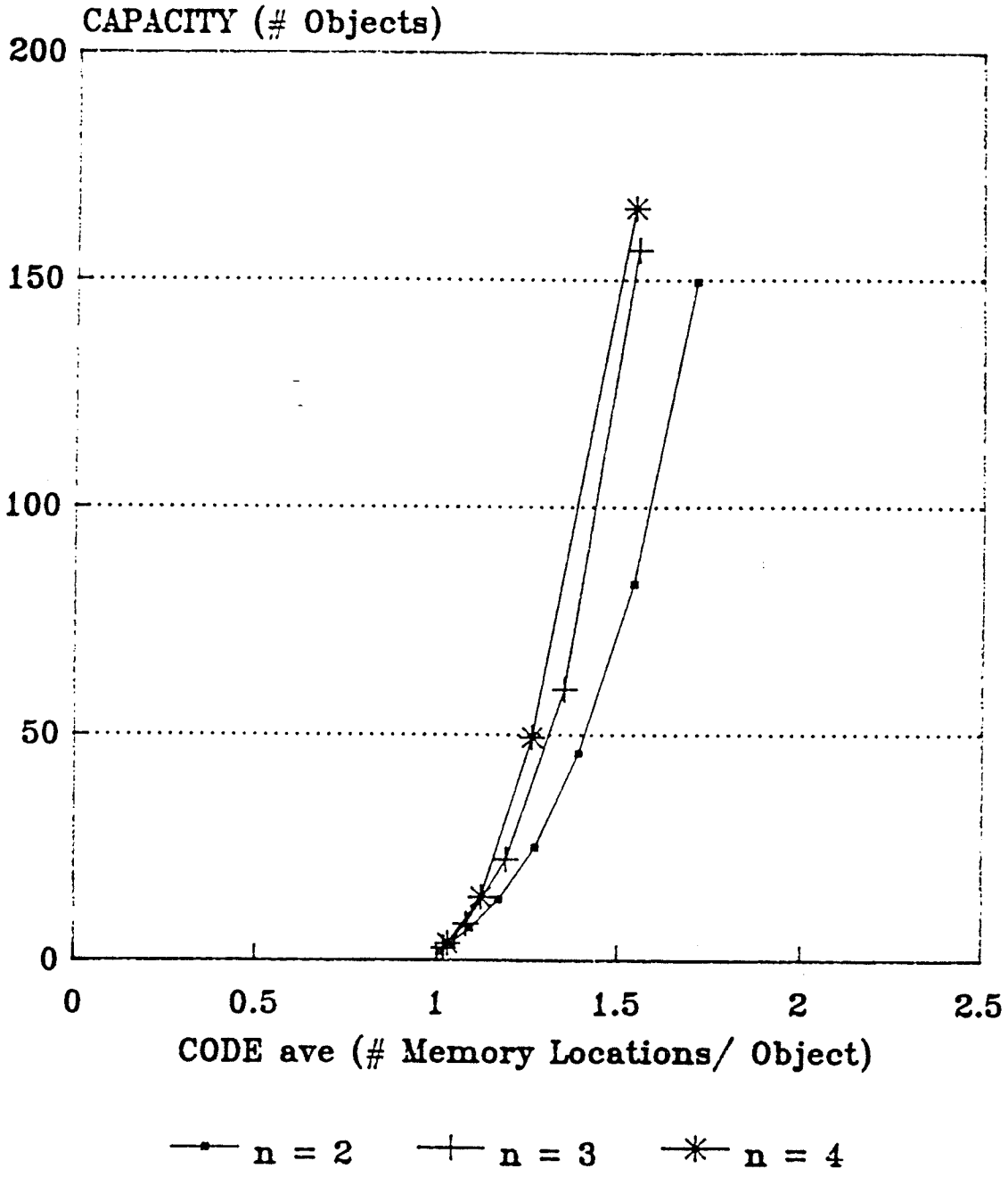
Clearly, if in our example, all 4058 objects were successfully coded, the areas for each object would have to fit together with no wasted memory space. This situation will only occur if the distribution of coordinate values for the object set is perfectly uniform at each descriptor level. Otherwise a capacity less than the theoretical value of 4058 should be expected. The expected capacity, therefore, depends on the descriptor distribution of the objects being coded. To maximize capacity, a statistical study should be performed on the object set to determine the descriptor distribution. The spacing of the boundaries can then be adjusted to allow an equal probability of

classifying descriptors in each of the n subdivisions.

At this point, one may be asking themselves if memory can be used more efficiently by varying n (the number of subdivisions used in each descriptor range). For example, say our goal is to have a theoretical capacity of 4000 objects. We know this can be achieved using 6 FDs, each subdivided into n = 5 intervals (since n^k / $code_{ave}$ = 4058). However, this can also be achieved using more FDs and less intervals. The value of $code_{ave}$ in each case must be compared to determine which method uses less memory locations per object.

Figure 6 is a plot of capacity against $code_{ave}$ obtained using equations (vi) and (v). From the graph, we can see that as n increases, less memory locations per object are required to achieve a given capacity. Therefore, our goal should be to maximize n (i.e. use as many subdivisions as possible at each descriptor level by making the subdivision width d as small as possible). It must be observed, however, that each subdivision of width d cannot be less than the size of err(k). Otherwise it would be possible for descriptors to fall into both the +/- err(k) error space surrounding boundary b_j, and the +/- err(k) space surrounding b_{j+1}. The teach mode would then not be able to generate all possible codes that the identification mode could find. Therefore, we conclude that maximum memory usage is achieved at the lower width limit of d, when d = err(k). When

Figure 6 Capacity vs Code$_{ave}$



subdividing a descriptor distribution which is not uniform, the smallest subdivision should be set equal to the size of err(k).

Perhaps a more important question is, what happens when a part is taught to the system and a collision occurs at the last descriptor level? Even if a statistical analysis is performed to determine the descriptor distribution, and the interval sizes are adjusted, someone will always try to code more objects. The computer can easily inform the user that there are not enough descriptors available to store the part's identity in unoccupied memory locations. However, what should be done with this part that cannot be 'fit' into the data-base of existing part identities? There are basically two options available. The first is to simply delete the part from the data-base, acknowledging that it cannot be coded with the existing number of descriptors being used. The second option is to add the part to an array holding the identities of parts that have previously collided at this last descriptor level memory location. When trying to identify this part later, the system will list the contents of the array, indicating that one of the listed parts is correct.

Although the second option provides some compensation to the problem, more descriptors are still needed to uniquely identify the part. The first option is simple and is consistent with our goal of maintaining a 100% guarantee that all parts

taught to the system will be correctly identified in the identification mode. Parts that cannot be uniquely coded will simply be weeded out during the teach mode. If the system is to handle more parts, it must be expanded so that it can access more descriptor routines and maintain a larger identity array.

It should be noted that expanding the system is not a problem. The identity array can be extended and new descriptor routines can be added within the main while loop of the program. Here there is an advantage over the classic rule-based approach where at least one new rule must be added to the system every time a new part is added. By lengthening the array and increasing the descriptor levels, our recognition system is equipped with the capacity to code several more objects before another expansion is required. Furthermore, as a rule-base is developed, the rules quickly become specific to types of shapes that are being dealt with. Once a rule system is developed, applying the same set of rules to an entirely new and different type of part group can be very inefficient. To fully classify the new parts, it is likely that a major portion of the rules, if not all of them, have to be rewritten. The algorithm presented in this report, although much like a rule system, is portable between object groups. It can be either expanded or the data-base can be simply reset if it is desired to totally transfer the system to the new object group. Since there are no rules to rewrite, the transfer can be immediately implemented. If it is

thought to increase the efficiency for classifying the new objects, the order in which the descriptor routines are called can be changed.

An advanced system using this recognition algorithm would have access to many descriptor routines, greatly reducing the chance of a collision at the last descriptor level. When teaching a part to the system, every descriptor routine would be executed, storing the part's identity at all the locations previously discussed. In the identification mode, the computer simply has to find the first memory location containing a part identity. The bulk of the processing time, therefore, is taken up in the teach mode, leaving a minimal amount of work to be done in the real time conditions of the identification mode.

Although the system can maintain 100% reliability and will compute only the minimum number of descriptors required to identify a part, the memory requirements are quite large. As a defense, it is pointed out that memory boards are becoming more affordable as they are now mass produced. Furthermore, it is accuracy and processing time that pose the greatest challenge for object recognition.

CHAPTER 5 EXPERIMENTAL RESULTS

5.1 Test Equipment

The program was written in the TURBO C language on a 286 microcomputer. Image acquisition was accomplished using a Sony CCD camera and a DT-IRIS frame grabber board. DT-IRIS software routines were linked to the C code to control the image acquisition from within the program. Binary images with a resolution of 512 x 480 pixels were displayed on an RGB Sony monitor.

5.2 Experimental Method

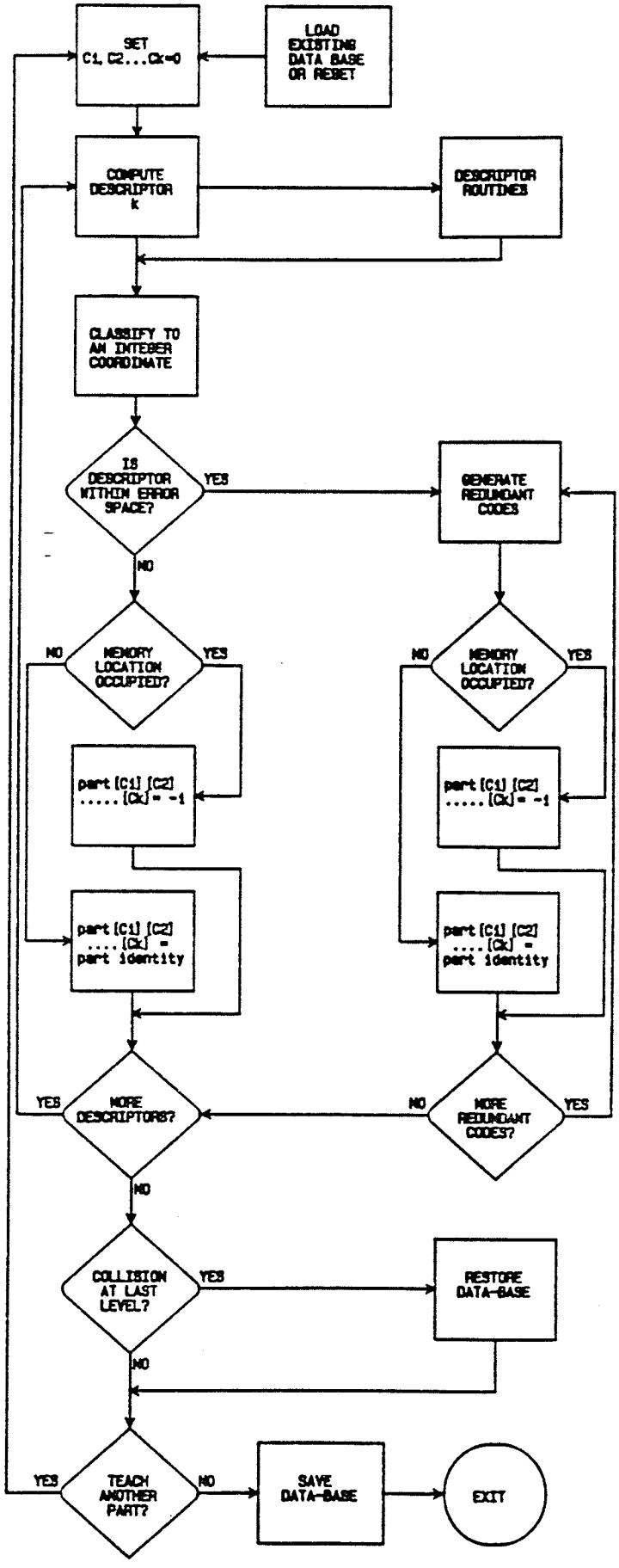
The Boundary Tracking/FD routine, the Teach Mode routine and the Identification Mode routine are listed in Appendix A. The flowcharts shown in figures 7 and 8 provide an overview of the programming functions in the teach mode and identification mode.

In the teach mode, the user can load in the existing data-base if they wish to add more parts to a set of previously taught parts. If the existing data-base is no longer needed, it can be reset, allowing a new data-base to be built.

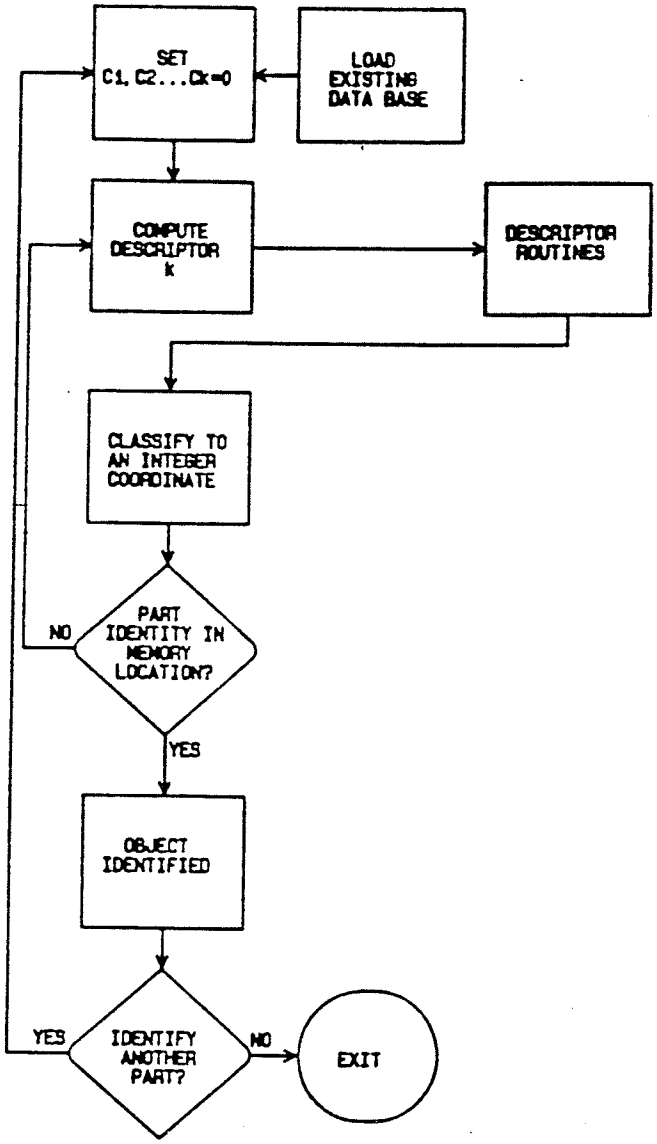
Before a new part is taught, the integer coordinates $c_1, c_2 \ldots c_k$ are set to zero. This allows the computer to store part identities at each descriptor level as described in section 4.2. The first descriptor routine is then called which returns the first descriptor value. After this real number is classified to its integer coordinate, the computer checks if the descriptor fell within an error space. If so, the appropriate number of duplicate codes will be generated. The memory location for each code is then inspected. If a part identity already exists at one of the memory locations, it is set to -1, indicating that it can no longer be uniquely identified at the current descriptor level. If unoccupied, the identity of the part currently being taught is written to the memory location. Once all descriptors have been processed in this manner, the computer checks if there was a collision at the last descriptor level. If so, the current part being taught is rejected and the memory locations of the data-base are restored to their previous values. The user then has the option to teach another part or exit and save the data-base.

The identification mode routine is similar to the teach mode except that the classification scheme does not contain error spaces and redundant codes are not generated. Descriptors are simply processed one at a time until a memory location is found that contains a part identity.

Figure 7 Flowchart for Teach Mode Routine



Flowchart for Identification Mode Routine



Test objects were obtained by drawing random shapes on white paper using a black felt marker to produce a clear binary image. A total of 23 object shapes have been taught to the system using the first 4 Fourier Descriptors, each subdivided by 4 error spaces of distance 2*err(k). During testing, it was found that objects would yield FD's having an error in excess of the calculated err(k) values. This was due to the shifting of the centroid location between rotations and size variations. As this error is dependent on the digitizing resolution rather than the parameterizing method, it was not included in the error analysis in section 3.4. A theoretical calculation of the digitizing error is beyond the scope of this report, as it depends not only on camera resolution but also on blurring and lighting conditions. Future improvements in digitizing resolution will reduce the effect of this limitation, however, for the current 512x480 array, it was found that the size of the err(k) values at the first 4 frequency levels had to be widened to at least 2.0 to account for the additional error incurred with digitization.

A total of 2 object shapes have been rejected from the system during the teach mode due to collisions occurring at the last descriptor level. These parts were later successfully coded by adding two new descriptors, shortest and longest radial length. All objects have been tested in the identification mode, each in at least 3 different orientations and at 2 different sizes. All of the test objects were correctly identified each time they were presented to the camera.

Figure 9 is a test image showing the boundary trace and 256 radial lines extending from the centroid. Figure 10 lists the output for the identification mode, indicating the real number descriptor calculations and the integer classification. Note that the object has been identified at descriptor level 4. Further descriptor calculations were unnecessary. Additional test images and program results are shown in Appendix C.

Figure 9 Boundary Trace and Radial Lines

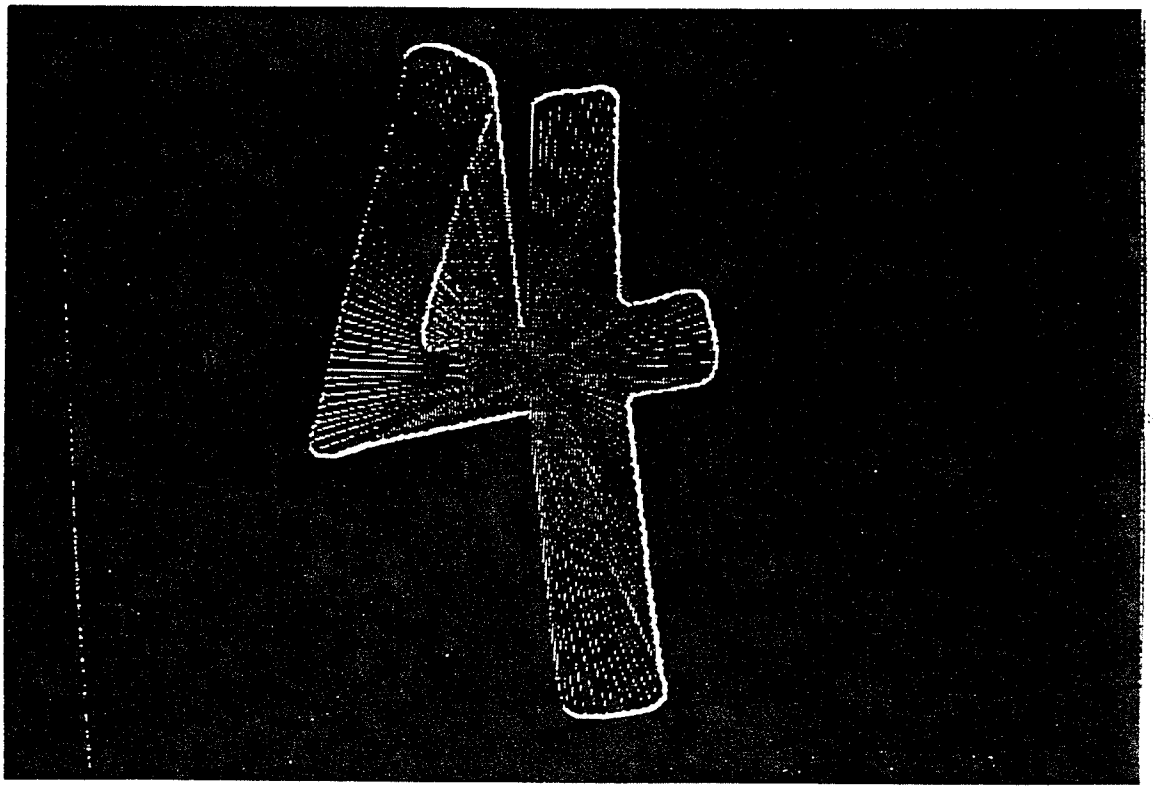


Figure 10 Identification Mode Output

```
Do you wish to identify another part?y
Position next part under camera and press any key


Normalized min_dist = 17.839012
Normalized max_dist = 186.254639


FREQ        REAL        IMAGINARY        MAGNITUDE

  1         0.40.....     5.68             5.5419
  2        13.62.....    -2.27            13.4466
  3        14.58.....    14.24            19.8561
  4        -2.69..... -12.09              12.0661

Part number is : 1
Identified at descriptor level 4


 Identity code for part no. 1 is: 2 3 2 3 0 0


 Do you wish to identify another part?
```

CHAPTER 6 CONCLUSIONS AND RECOMMENDATIONS

An algorithm for object identification of two-dimensional binary images has been presented. Although the discussion has been limited to using Fourier Descriptors, it should be noted that virtually any type of descriptor can be used provided that the maximum possible error is correctly calculated. Efficiency and reliability make this method well suited for industrial applications. The multi-dimensional array concept allows only the minimum number of descriptors to be calculated and does not require any additional processing time for searching and matching. Redundant coding accounts for the maximum possible error of each descriptor, enabling the system to maintain 100% reliability.

The largest drawback to the system is that, for a randomly selected group of parts, there is no guarantee that all parts in the group will be successfully coded with the existing number of descriptors. If a collision occurs at the last descriptor level, the identity array must be expanded to include coordinate values calculated from additional descriptor routines. Although expanding the system is not a problem, the memory requirements increase exponentially with the number of descriptors used. Increasing the efficiency of memory usage can be accomplished by subdividing the descriptor range as much as possible while

ensuring that the frequency distribution is uniform between each subdivision.

For this algorithm, the bulk of the work remains in setting up the system, choosing accurate descriptors, calculating their maximum error, and expanding the system when it is necessary to code a greater number of parts in the teach mode. Once this preparatory work is completed, the identification mode is equipped with the ability to identify the coded objects with 100% reliability.

BIBLIOGRAPHY

[1] Bolles and Cain, "Recognizing and Locating Partially Visible Objects: The local Feature Focus Methods", Int. Journal of Robotics Research, vol.1, n.3, pp.57-82, 1982.

[2] Draper et al, "Image Interpretation by Distributed Cooperative Processes", IEEE Trans Computer Vision and Pattern Recognition, pp. 129-135, 1988.

[3] Goad, "Fast 3D Model Based Vision", From Pixels to Predicates, Albex, Norwood, New Jersey, 1986

[4] Hall, Computer Image Processing and Recognition, Academic Press, pp. 416-419, 1979

[5] B.K. Horn, Robot Vision, MIT Press, pp.152-153, 1986

[6] Jiang and Merickel, "Boundary Estimation in Complex Imagery Using Fourier Descriptors", IEEE Trans Pattern Analysis and Machine Intelligence, pp. 187-190, 1988.

[7] Krzyzak, Leung and Suen, "Reconstruction of Two Dimensional Patterns by Fourier Descriptors", IEEE Trans on Systems, Man and Cybernetics, pp. 555-558, 1988.

[8] McKeown, Jr., Harvey, Jr., and McDermott, "Rule-Based Interpretation of aerial imagery", IEEE Trans Pattern Analysis and Machine Intelligence, v.7, n.5, pp.570-585, 1985

[9] Niblack and Damian, "Experiments and Evaluations of Rule-Based Methods in Image Analysis", IEEE Trans Computer Vision and Pattern Recognition, pp. 123-128, 1988.

[10] Persoon and Fu, "Shape Discrimination Using Fourier Descriptors", IEEE Trans on Systems Man and Cybernetics, v.7, pp. 170-179, 1977.

[11] Press, Flannery, S.A. Teukolsky and W.T. Vetterling, Numerical Recipes in C - The Art of Scientific Computing, Cambridge University Press, pp. 407-412, 1989

[12] Smyrniotis and Dutta, "A Knowledge-Based System for Recognizing Man-Made Objects in Aerial Images", IEEE Trans Computer Vision and Pattern Recognition, pp. 111-117, 1988

[13] Wallace and Wintz, "An Efficient Three Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors", Computer Graphics and Image Processing, v.13, pp. 99-126, 1980

[14] Zahn and Roskies, "Fourier Descriptors for Plane Closed Curves", IEEE Trans on Computers, v. c-21, n.3, pp. 269, 1972.

APPENDIX A

ERROR GENERATION
FOURIER DESCRIPTOR AND
OBJECT RECOGNITION ROUTINES

```
/************************  FD ERROR ROUTINE  ****************************

  This routine generates maximum error magnitudes at the selected
  frequency indicated by the variable FREQ.  The number of terms
  can also be changed.  The length of the radial projections from
  the centroid is indicated by max_length.  Using FREQ, max_length
  and terms, the error shifting along the boundary curve is matched
  to the sign change pattern of a periodic function having
  frequency = FREQ.

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



  #include <stdio.h>
  #include <math.h>
  #define PI 3.1415926
  #define terms 256
  #define FREQ 2
  #define max_length 250

  main()
  {
   int  s,n,c,k;
   float temp, dfcr, dfci, err[256];
   float error[256], change_sign, shift, ave_length;


 /****  Determine the normalized shifting error and the number of sample
             points required between sign changes of the error  ***********/

   ave_length = max_length / 2;
   shift = (((FREQ*max_length*2) / terms) * .5) / ave_length;
   change_sign = (terms/FREQ) * .5;



  /*********  Match the sign change pattern to that of a sine or
            cosine function of frequency = FREQ  *****************/

  n=0;
  for (c=0; c <= FREQ; c++)
  {
   for(s=1;s<= change_sign ;s++) { error[n] =  shift;  n++;}
   for(s=1;s<= change_sign ;s++) { error[n] = -shift;  n++;}
   }
```

```
/*********  Propagate the error function error[n] through the
                Fourier Transform to determine the resulting error
              magnitude     ************************************/


 printf("\n\n\n\n\n\n");
 printf("                ERROR MAGNITUDES FOR METHOD (iii) \n");
 printf("                MAXIMIZING FOR FREQ. K = %1d \n\n\n",FREQ);
 printf("                    Freq. k           Error Magnitude\n\n");


 for(k=1;k<=16;k++)
 {
  dfcr=0;
  dfci=0;
  for(n=0; n<= (terms-1); n++)
  {
   temp = (2 * PI * k * n) /  terms;
   dfcr = dfcr + (error[n] * cos(temp)) / terms;
   dfci = dfci - (error[n] * sin(temp)) / terms;
   }
   err[k] = hypot(dfcr,dfci) * 100;
   printf("                     %3d                %7.4f\n",k,err[k] );
  }

}
```

ERROR MAGNITUDES FOR METHOD (iii)
MAXIMIZING FOR FREQ. K = 2

Freq. k	Error Magnitude
1	0.0000
2	0.7640
3	0.0000
4	0.0000
5	0.0000
6	0.2549
7	0.0000
8	0.0000
9	0.0000
10	0.1532
11	0.0000
12	0.0000
13	0.0000
14	0.1097
15	0.0000
16	0.0000

ERROR MAGNITUDES FOR METHOD (iii)
MAXIMIZING FOR FREQ. K = 1

Freq. k	Error Magnitude
1	0.2547
2	0.0000
3	0.0849
4	0.0000
5	0.0510
6	0.0000
7	0.0364
8	0.0000
9	0.0284
10	0.0000
11	0.0232
12	0.0000
13	0.0197
14	0.0000
15	0.0171
16	0.0000

```
/*********** BOUNDARY TRACKING AND FD ROUTINE ************************

   The following program calculates a pixel-by-pixel 4-link chain code
   from 512x480 input images.  Radial distances from the centroid
   to points on the boundary curve are calculated to generate an
   object parameterization function (dist[n]) which describes radial
   distance as a function of curve length.  Finally, the Discrete
   Fourier Transform is applied to f(n) yielding a set of
   characteristic coefficients which are invariant under rotations,
   size and starting point.

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


#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <alloc.h>
#include <math.h>
#include "isdecs.h"
#include "iserrs.h"

#define terms 256
#define PI 3.141592654

int pixel[40][512];


fourier()
{
    extern int start;
    extern char opps;
    extern int *x_coord,*y_coord,*word;
    extern int pixel[40][512], *array;
    extern float descriptor[6];
    float  x_momentf=0,y_momentf=0,massf=0;
    float  max_dist=0,min_dist=5000;
    float dist[600], x_dist,y_dist;
    float dfcr[7], dfci[7], temp, mag[7], point2;
    float step, low, high, dec,x_centroif,y_centroif,normal,f;
    int x_centroid,y_centroid, x,y,sample_pt,L,k;
    int y_max=0, y_min=600, x_max=0, x_min=600;
    int i,j, orientation = 1, s, end,point, r, status2;
    int start_pti, start_ptj, indx, n, direction, corners, q, status;
    int split,found=0,corn1,corn2,corn3,corn4,bit;
    char FILENAME[32];
    long offset,m;
    int bitarray[] = {0x0001,0x0002,0x0004,0x0008,
                      0x0010,0x0020,0x0040,0x0080,
                      0x0100,0x0200,0x0400,0x0800,
                      0x1000,0x2000,0x4000,0x8000};
    int one=0,zero=255,color1=128;
```

```
/****** Initialize DT-IRIS frame grabber *******/

    if(start == 1)
     { is_initialize();
       start=0;
       is_select_ilut(6);
       is_select_olut(6);
       is_set_sync_source(1);
       is_select_input_frame(0);
       is_select_output_frame(1);
       is_display(1);
       }
    is_acquire(0,1);
    is_set_background(150);




/***** Acquire a binary 512x480 image and store in bit-packed format
   in *(word + offset)                              *****/

    offset = -1;
    for(i=0;i < 12;i++)
    {
      x=y=0;
      k=40*i;
      is_set_active_region(k,0,40,512);
      is_get_region(0,&pixel);
      is_put_region(1,&pixel);

      for(j=0; j < 1280; j++)
      {
       offset++;
       for (bit = 15; bit >= 0 ; bit--)
       {
         if(x == 512) { x=0;y++; }
         if(pixel[y][x] == one)
           *(word + offset) |=  bitarray[bit];
         else
            *(word + offset) &= ~bitarray[bit];

         x++;
         }
       }
      }
```

```
/***** Scan the image from top to bottom until the 3x3 window
       encounters a "1" *****/

    i = j = k = m = n = 1;
    x = y = 0;
    for (offset = 0; found == 0; offset++)
    {
      if(x == 512) {x=0;y++;}
      for (bit = 15; found == 0  &&  bit >= 0 ; bit--)
      {
        if(*(word + offset) & bitarray[bit])
            found=1 ;
         x++;
        }
      }

    x--;y--;
    x--;
    offset--;bit++;
    *(x_coord+k) = x;
    *(y_coord+k) = y;
    k++;
    start_pti = offset;
    start_ptj = bit;

    is_set_active_region(0,0,512,512);




/* Track the boundary of the object until the 3x3 window returns to
   the initial starting point.  As the window follows the curve, it
   will generate a 4-link chain code corresponding to the directional
   moves.                                                          */

      if(bit == 15) split=1;
      if(bit == 14) split=2;
      if((bit != 15) && (bit != 14)) split=3;



    while(1)
    {
    if ( ((m == start_pti) && (n == start_ptj)) ||
         ((m == start_pti-1) && (n ==  0))      ||
         ((m == start_pti+1) && (n == 15))      &&
         ( k > 50 ) ) break;
```

```
switch(split)   {
case 1 :


   switch (orientation)
   {  case 1 :  if ((*(word + offset-32) & bitarray[bit]) == 0)
                  { bit--;x++;  }
                else
                  { offset = offset - 32;y--;  } break;

      case 2 :  if ((*(word + offset-65) & bitarray[0]) == 0)
                  { offset = offset - 32;y--;  }
                else
                  { bit++;x--;  } break;

      case 3 :  if ((*(word + offset-33) & bitarray[1]) == 0)
                  { bit++;x--;  }
                else
                  { offset = offset + 32;y++;  } break;

      case 4 :  if ((*(word + offset-1) & bitarray[0]) == 0)
                  { offset = offset + 32;y++;  }
                else
                  { bit--;x++;  } break;
   }   break;


case 2 :

   switch (orientation)
   {  case 1 :  if ((*(word + offset-32) & bitarray[bit]) == 0)
                  { bit--;x++;  }
                else
                  { offset = offset - 32;y--;  } break;

      case 2 :  if ((*(word + offset-64) & bitarray[15]) == 0)
                  { offset = offset - 32;y--;  }
                else
                  { bit++;x--;  } break;

      case 3 :  if ((*(word + offset-33) & bitarray[0]) == 0)
                  { bit++;x--;  }
                else
                  { offset = offset + 32;y++;  } break;

      case 4 :  if ((*(word + offset) & bitarray[15]) == 0)
                  { offset = offset + 32;y++;  }
                else
                  { bit--;x++;  } break;
   }  break;
```

```
      case 3 :

         switch (orientation)
         {  case 1 :  if ((*(word + offset-32) & bitarray[bit]) == 0)
                        { bit--;x++;  }
                      else
                        { offset = offset - 32;y--;  } break;

            case 2 :  if ((*(word + offset-64) & bitarray[bit+1]) == 0)
                        { offset = offset - 32;y--;  }
                      else
                        { bit++;x--;  } break;

            case 3 :  if ((*(word + offset-32) & bitarray[bit+2]) == 0)
                        { bit++;x--;  }
                      else
                        { offset = offset + 32;y++;  } break;

            case 4 :  if ((*(word + offset) & bitarray[bit+1]) == 0)
                        { offset = offset + 32;y++;  }
                      else
                        { bit--;x++;  } break;
         }  break;
       }


    if(bit == 16) {offset--; bit=0; }
    if(bit == -1) {offset++; bit=15;}
    if(bit == 15) split=1;
    if(bit == 14) split=2;
    if((bit != 14) && (bit != 15)) split=3;

/* Store maximum and minimum x and y coordinates of image  */

    if(y > y_max) y_max=y;
    if(y < y_min) y_min=y;
    if(x > x_max) x_max=x;
    if(x < x_min) x_min=x;


/* Store the x and y coordinates of the current window position  */

    *(x_coord+k) = x ;
    *(y_coord+k) = y ;


/* Return to calling routine if 3x3 window becomes disoriented  */

    if((y > 511) || (y < 0) || (x > 511) || (x < 0) || (k > 10000))
       return(opps='1');



    is_put_pixel(1,y,x,1,&color1);
    k++;m = offset; n = bit;
```

```
/* Inspect key pixels to determine next orientation        */

    switch(split)    {

    case 1:

    corn1=corn2=corn3=corn4=0;
    if(*(word+offset-65) & bitarray[1]) corn1=1;
    if(*(word+offset-64) & bitarray[15]) corn2=1;
    if(*(word+offset   ) & bitarray[15]) corn3=1;
    if(*(word+offset-1 ) & bitarray[1])  corn4=1;

    corners = corn1 + corn2 + corn3 + corn4;


    if (corners <= 1)
    {
      if ( (corn1 + corn2 + corn4)  == 0)
         orientation = 1;
      if ( (corn1 + corn3 + corn4)  == 0)
         orientation = 2;
      if ( (corn2 + corn3 + corn4)  == 0)
         orientation = 3;
      if ( (corn1 + corn2 + corn3)  == 0)
         orientation = 4;
      if ( (corners == 0 ) && (*(word + offset-65) & bitarray[0]) )
         orientation = 2 ;
      if ( (corners == 0) && (*(word + offset-32) & bitarray[15]) )
         orientation = 1 ;
      }

    if (corners == 2)
    {
     if ( (corn3 + corn4) == 2 )
     { if (*(word + offset-33) & bitarray[1]) orientation = 4;
       if (*(word + offset-32) & bitarray[15]) orientation = 1 ;
       if ((*(word + offset-33) & bitarray[1]) &&
           (*(word + offset-32) & bitarray[15])) orientation = 1;  }

     if ( (corn2 + corn3) == 2 )
     { if (*(word + offset-1 ) & bitarray[0]) orientation = 1;
       if (*(word + offset-65) & bitarray[0]) orientation = 2;
       if ((*(word + offset-1) & bitarray[0]) &&
           (*(word + offset-65) & bitarray[0])) orientation = 2;   }

     if ( (corn1 + corn2 ) == 2 )
     { if (*(word + offset-33) & bitarray[1]) orientation = 3;
       if (*(word + offset-32) & bitarray[15]) orientation = 2;
       if ((*(word + offset-33) & bitarray[1]) &&
           (*(word + offset-32) & bitarray[15])) orientation = 3;  }
```

```
  if ( (corn1 + corn4) == 2 )
  {  if (*(word + offset-1) & bitarray[0]) orientation = 4;
     if (*(word + offset-65) & bitarray[0]) orientation = 3;
     if ((*(word + offset-1) & bitarray[0]) &&
         (*(word + offset-65) & bitarray[0])) orientation = 4;  }
  }

if (corners == 3)
{
  if (corn3 == 0) orientation = 3;
  if (corn2 == 0) orientation = 4;
  if (corn1 == 0) orientation = 1;
  if (corn4 == 0) orientation = 2;
 }
 break;


 case 2:

 corn1=corn2=corn3=corn4=0;
 if(*(word+offset-65) & bitarray[0]) corn1=1;
 if(*(word+offset-64) & bitarray[14]) corn2=1;
 if(*(word+offset   ) & bitarray[14]) corn3=1;
 if(*(word+offset-1 ) & bitarray[0])  corn4=1;

 corners = corn1 + corn2 + corn3 + corn4;

 if (corners <= 1)
 {
   if ( (corn1 + corn2 + corn4)  == 0)
      orientation = 1;
   if ( (corn1 + corn3 + corn4)  == 0)
      orientation = 2;
   if ( (corn2 + corn3 + corn4)  == 0)
      orientation = 3;
   if ( (corn1 + corn2 + corn3)  == 0)
      orientation = 4;
   if ( (corners == 0 ) && (*(word + offset-64) & bitarray[15]) )
      orientation = 2 ;
   if ( (corners == 0) && (*(word + offset-32) & bitarray[14]) )
      orientation = 1 ;
   }

 if (corners == 2)
 {
  if ( (corn4 + corn3) == 2 )
  { if (*(word + offset-33) & bitarray[0]) orientation = 4;
    if (*(word + offset-32) & bitarray[14]) orientation = 1 ;
    if ((*(word + offset-33) & bitarray[0]) &&
        (*(word + offset-32) & bitarray[14])) orientation = 1;  }
```

```
    if ( (corn2 + corn3) == 2 )
    { if (*(word + offset   ) & bitarray[15]) orientation = 1;
      if (*(word + offset-64) & bitarray[15]) orientation = 2;
      if ((*(word + offset-1) & bitarray[15]) &&
          (*(word + offset-65) & bitarray[15])) orientation = 2;    }

    if ( (corn1 + corn2) == 2 )
    { if (*(word + offset-33) & bitarray[0]) orientation = 3;
      if (*(word + offset-32) & bitarray[14]) orientation = 2;
      if ((*(word + offset-33) & bitarray[0]) &&
          (*(word + offset-32) & bitarray[14])) orientation = 3;  }

    if ( (corn1 + corn4) == 2 )
    {  if (*(word + offset  ) & bitarray[15]) orientation = 4;
       if (*(word + offset-64) & bitarray[15]) orientation = 3;
       if ((*(word + offset  ) & bitarray[15]) &&
           (*(word + offset-64) & bitarray[15])) orientation = 4;  }
    }

  if (corners == 3)
  {
    if (corn3 == 0) orientation = 3;
    if (corn2 == 0) orientation = 4;
    if (corn1 == 0) orientation = 1;
    if (corn4 == 0) orientation = 2;
   }
   break;



   case 3:


   corn1=corn2=corn3=corn4=0;
   if(*(word+offset-64) & bitarray[bit+2]) corn1=1;
   if(*(word+offset-64) & bitarray[bit]) corn2=1;
   if(*(word+offset   ) & bitarray[bit]) corn3=1;
   if(*(word+offset   ) & bitarray[bit+2])  corn4=1;

   corners = corn1 + corn2 + corn3 + corn4;

   if (corners <= 1)
   {
     if ( (corn1 + corn2 + corn4)  == 0)
        orientation = 1;
     if ( (corn1 + corn3 + corn4)  == 0)
        orientation = 2;
     if ( (corn4 + corn3 + corn2)  == 0)
        orientation = 3;
     if ( (corn1 + corn2 + corn3)  == 0)
        orientation = 4;
```

```
      if ( (corners == 0 ) && (*(word + offset-64) & bitarray[bit+1]) )
         orientation = 2 ;
      if ( (corners == 0) && (*(word + offset-32) & bitarray[bit]) )
         orientation = 1 ;
      }

    if (corners == 2)
    {
     if ( (corn4 + corn3 == 2 ) )
     { if (*(word + offset-32) & bitarray[bit+2]) orientation = 4;
       if (*(word + offset-32) & bitarray[bit]) orientation = 1 ;
       if ((*(word + offset-32) & bitarray[bit+2]) &&
           (*(word + offset-32) & bitarray[bit])) orientation = 1;  }

     if ( (corn3 + corn2) == 2 )
     { if (*(word + offset   ) & bitarray[bit+1]) orientation = 1;
       if (*(word + offset-64) & bitarray[bit+1]) orientation = 2;
       if ((*(word + offset  ) & bitarray[bit+1]) &&
           (*(word + offset-64) & bitarray[bit+1])) orientation = 2;

     if ( (corn2 + corn1) == 2 )
     { if (*(word + offset-32) & bitarray[bit+2]) orientation = 3;
       if (*(word + offset-32) & bitarray[bit]) orientation = 2;
       if ((*(word + offset-32) & bitarray[bit+2]) &&
           (*(word + offset-32) & bitarray[bit])) orientation = 3;  }

     if ( (corn1 + corn4) == 2 )
     {  if (*(word + offset  ) & bitarray[bit+1]) orientation = 4;
        if (*(word + offset-64) & bitarray[bit+1]) orientation = 3;
        if ((*(word + offset  ) & bitarray[bit+1]) &&
            (*(word + offset-64) & bitarray[bit+1])) orientation = 4;  }
     }

   if (corners == 3)
   {
     if (corn3 == 0) orientation = 3;
     if (corn2 == 0) orientation = 4;
     if (corn1 == 0) orientation = 1;
     if (corn4 == 0) orientation = 2;
    }
    break;

      }
}       /* End of boundary tracking     */


/* Return to calling routine if a noise point has been tracked
   instead of the object                    */

    k--;
    if(k < 250) return(opps='1');
    f=k;
```

```
/* Calculate object mass and x and y moments    */

    x=1;y=1;
    for (offset = 0; offset < 15360; offset++)
    {
      for (bit = 15; bit >= 0 ; bit--)
       {
         if(x == 513) { x=1;y++; }
         if((*(word + offset) & bitarray[bit]) && (x <= x_max) && (x >= x_mi
           && (y <= y_max) && (y >= y_min))
         {
             massf++;
             x_momentf += x;
             y_momentf += y;
          }
         x++;
       }
     }


    x_centroif = x_momentf/massf;
    y_centroif = y_momentf/massf;

    if((x_centroif - floor(x_centroif)) < 0.5)
      x_centroid = floor(x_centroif);
    else
      x_centroid = ceil(x_centroif);

    if((y_centroif - floor(y_centroif)) < 0.5)
      y_centroid = floor(y_centroif);
    else
      y_centroid = ceil(y_centroif);


      x_centroif -= 0    ; y_centroif += 0;



/* Normalize the number of slope calculations so that different image sizes
   will still generate the same number of terms.                          */

   step = f / terms ;
   low = step -  floor(step) ;
   high = ceil(step) - step  ;
   if ((high != 0) || (low != 0))
   {  if ( high <= low )
      {  step = ceil(step);
         dec  = 1 / high;
         status = 1;  }
```

```
   else
   {  step = floor(step);
      dec  = 1 / low;
      status = 2; }  }
 else
 { dec = 0 ;  }


low = dec - floor(dec);
high = ceil(dec) - dec;
if ((high != 0) || (low != 0))
{  if (high <= low )
   {  point = ceil(dec);
      point2 = 1 / high;
      status2 = 1;  }
   else
   {  point = floor(dec);
      point2 = 1 / low ;
      status2 = 2;   }
}
else
{  point = dec;
   point2 = 0 ; }


low = point2 - floor(point2);
high = ceil(point2) -point2;
if (high <= low)
  point2 = ceil(point2);
else
  point2 = floor(point2);


q = point;
r = point2;
i = j = 1;
L = f;
printf("\n\n");
s=0;
```

```
/* Calculate the distance to the centroid at each normalized sample_pt  */

    n=0;
    is_set_graphic_position(y_centroid,x_centroid);
    is_set_foreground(70);


    for  (sample_pt = 1; sample_pt <= L; sample_pt += step)
    {
      x_dist = *(x_coord+sample_pt) - x_centroif;
      y_dist = *(y_coord+sample_pt) - y_centroif;
      dist[s] = hypot(x_dist,y_dist);
      if(dist[s] < min_dist) min_dist=dist[s];
      if(dist[s] > max_dist) max_dist=dist[s];
      *(array+n) = *(y_coord+sample_pt);
      n++;
      *(array+n) = *(x_coord+sample_pt);
      n++;
      *(array+n) = y_centroid;
      n++;
      *(array+n) = x_centroid;
      s++;n++;


      if ( i == q )
      {
         if (status == 2 )
            sample_pt++ ;
         else
            sample_pt-- ;
         q = q + point;
         j++ ;
       }


       if (j == r )
       { if (status2 == 2)
            q++ ;
         else
            q-- ;
         r = r + point2;
         }

       i++;
   }


  n=n/2;
  is_draw_lines(1,n,array);
```

```
/* Apply the Discrete Fourier Transform to the function dist[n] and
   calculate the magnitudes for each coefficient.  These magnitudes are
   unique to the boundary curve and are rotation and size invariant.   */


    s=terms;
    dfcr[s] = 0;
    dfci[s] = 0;
    for (n = 0;  n <= terms; n++)
    {
      temp = (2 * PI * s * n) / terms;
      dfcr[s] = dfcr[s] + (dist[n] * cos(temp)) / terms;
      dfci[s] = dfci[s] - (dist[n] * sin(temp)) / terms;
     }
      normal = hypot(dfcr[s],dfci[s]);
      descriptor[1] = 100*(min_dist/normal);
      descriptor[2] = 100*(max_dist/normal);
      printf("\n\nNormalized min_dist = %f", descriptor[1]);
      printf("\nNormalized max_dist = %f\n\n",descriptor[2]);


  printf("\n\nFREQ       REAL      IMAGINARY        MAGNITUDE\n\n");
  j=3;
  for ( s = 1; s <= 4; s++)
  {
    dfcr[s] = 0;
    dfci[s] = 0;
    for (n = 0;  n <= terms; n++)
    {
      temp = (2 * PI * s * n) / terms;
      dfcr[s] = dfcr[s] + (dist[n] * cos(temp)) / terms;
      dfci[s] = dfci[s] - (dist[n] * sin(temp)) / terms;
          }
      mag[s] = 100*(hypot(dfcr[s],dfci[s])) / normal;
      printf(" %2d     %7.2f.....%7.2f       %10.4f \n",
                  s,dfcr[s],dfci[s],mag[s]);
      descriptor[j] = mag[s];
      j++;
    }


}
```

```
/******************  TEACH MODE ROUTINE  ***************************

      This routine calls a descriptor routine, fourier(), and uses
      the returned descriptors as coordinates of the muti-dimensional
      array *(part + offs).  Real numbered descriptors are
      classified to integer coordinates according to the boundary
      spacing defined in limit[7][7].  Redundant codes are generated
      when real descriptors fall with the error spaces defined in
      error[7][5][3].  A while loop is executed until all
      descriptors have been classified.  Once the part identity is
      stored in the location(s) of the array *(part + offs), another
      part may be taught to the system.

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



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

extern unsigned _stklen = 20000;
float descriptor[6];
int *x_coord, *y_coord;
int *word,start=1,*array;
char opps;
float limit[7][7] = {   {  0,   0,     0,     0,    0,    0,     0},
                        {  0,  -1,    12,    24,   36,   48,   600},
                        {  0,  -1,   120,   160,  200,  240,   600},
                        {  0,  -1,     6,    12,   18,   24,    50},
                        {  0,  -1,     6,    12,   18,   24,    50},
                        {  0,  -1,     6,    12,   18,   24,    50},
                        {  0,  -1,     6,    12,   18,   24,    50}};

float error[7][5][3] =
 {{{  0,   0},{    0,    0},{    0,    0},{    0,    0},{    0,    0}},
  {{  0,   0},{    4,   12},{   16,   24},{   28,   36},{   40,   48}},
  {{  0,   0},{  108,  120},{  148,  160},{  188,  200},{  228,  240}},
  {{  0,   0},{    2,    6},{    8,   12},{   14,   18},{   20,   24}},
  {{  0,   0},{    2,    6},{    8,   12},{   14,   18},{   20,   24}},
  {{  0,   0},{    2,    6},{    8,   12},{   14,   18},{   20,   24}},
  {{  0,   0},{    2,    6},{    8,   12},{   14,   18},{   20,   24}}};
```

```
main()
{
   char mode,reset='n',not_unique;
   FILE  *f1,*f3;
   static int *part;
   extern int *word;
   extern char opps;
   int   a,b,c,d,e,f,level,classified,bndry,part_no,total,i;
   int   az,bz,cz,dz,ez,fz,v,last,*restore_part_no,*restore_location;
   unsigned long offs=0;
   char  key;
   float c1,c2,c3,c4,c5,c6;
   extern float descriptor[6];
   extern float limit[7][7];
   extern float error[7][5][3];


   restore_part_no = (int *) malloc(1000);
   restore_location = (int *) malloc(1000);
   x_coord = (int *) malloc(30000);
   y_coord = (int *) malloc(30000);
   array = (int *) malloc(10000);
   part  = (int *) malloc(34000);
   word =  (int *) malloc(32000);
   if(part == NULL)
      {  printf("NUlls pointers!");
         exit(1);  }
   if(word == NULL)
      {  printf("word is NULL!");
         exit(1);  }



/*  Load in the existing data base of coded parts */

      printf("\n Do you wish to delete the existing Database\n "
                  " and build a new one (y/n)? : ");
      reset = getch();
```

```
if (reset == 'n')
{
    printf("\nPlease wait while existing Database is being loaded in");
    f1 = fopen("c:\\lutes\\pn.dat","r");
    f3 = fopen("c:\\lutes\\parts.dat","r");
    rewind(f1);
    rewind(f3);
    fscanf(f1,"%d",&part_no);

    for(offs=0; offs <= 17000; offs++)
            fscanf(f3,"%4d", (part + offs));
 rewind(f1);
 rewind(f3);
 close(f1);
 close(f3);

 }


if(reset == 'y')
{   printf("\nReseting database");
    part_no = 1;

    for(offs=0; offs <= 17000; offs++)
            *(part + offs) = 0;
 }

printf("\nPosition part under camera and press any key");
getch();



/**** This loop is executed until the user has finished teaching
     their parts ****/

while (1)
{
   opps='0';
   not_unique = '0';

   fourier();   /**** Call the descriptor routine ****/
   if(opps == '1')
     { printf("\n\n*** Part not within field of view "
                      "and/or noise spot detected *** \n"
                      "Reposition part and/or remove noise, "
                        "press any key when ready...\n");

       getch();
       continue;
      }
```

```
az=bz=cz=dz=ez=fz=0;
c1=c2=c3=c4=c5=c6=0;
v=0;


/**** This loop is executed until all descriptors are classified ****/

for (level=1; level <= 6; level++)
{
 bndry=1; classified=0;

  while (classified != 1)
 {

  if((descriptor[level] >  limit[level][bndry]) &&
     (descriptor[level] <= limit[level][bndry+1]))

  {
   switch (level) {

    case 1 : c1=bndry;
             if((descriptor[1] >= error[1][bndry][0]) &&
                (descriptor[1] <= error[1][bndry][1]))
                   az=1;  break;

    case 2 : c2=bndry;
             if((descriptor[2] >= error[2][bndry][0]) &&
                (descriptor[2] <= error[2][bndry][1]))
                   bz=1;  break;

    case 3 : c3=bndry;
             if((descriptor[3] >= error[3][bndry][0]) &&
                (descriptor[3] <= error[3][bndry][1]))
                   cz=1; break;

    case 4 : c4=bndry;
             if((descriptor[4] >= error[4][bndry][0]) &&
                (descriptor[4] <= error[4][bndry][1]))
                   dz=1; break;

    case 5 : c5=bndry;
             if((descriptor[5] >= error[5][bndry][0]) &&
                (descriptor[5] <= error[5][bndry][1]))
                   ez=1; break;

    case 6 : c6=bndry;
             if((descriptor[6] >= error[6][bndry][0]) &&
                (descriptor[6] <= error[6][bndry][1]))
                   fz=1; break;
                     }
```

```
          classified=1;
         }
        bndry++;
       }

    total=az+bz+cz+dz+ez+fz;
                    /**** calculate offset for identity array  ****/
    offs = (c1*3125)+(c2*625)+(c3*125)+(c4*25)+(c5*5)+(c6*1) - 3125;



if (total == 0)
{                   /***** no redundant codes necessary yet *****/

      if(*(part + offs) == 0)
         *(part + offs) = part_no;    /**** memory location unoccupied ***
      else
       {
         if(*(part + offs) != part_no)
         {
          if(*(part + offs) != -1)
          {
          *(restore_part_no + v) = *(part + offs);
          *(restore_location + v) = offs;
          v++;
          }
          *(part + offs) = -1;     /**** memory location occupied ***/
          if(level == 6) not_unique = '1';
          }
        }


}

else   /***  A descriptor has fallen within an error space, therefore
           generate the appropriate number of redundant codes *****/
{
    for(i=1 ; i <= 64; i++)
    { a=b=c=d=e=f=0;
```

```
switch(i) {

case 1 : a=0; break;
case 2 : f=fz; break;
case 3 : e=ez; break;
case 4 : e=ez; f=fz; break;
case 5 : d=dz; break;
case 6 : d=dz; f=fz; break;
case 7 : d=dz; e=ez; break;
case 8 : d=dz; e=ez; f=fz; break;
case 9 : c=cz; break;
case 10 : c=cz; f=fz; break;
case 11 : c=cz; e=ez; break;
case 12 : c=cz; e=ez; f=fz; break;
case 13 : c=cz; d=dz; break;
case 14 : c=cz; d=dz; f=fz; break;
case 15 : c=cz; d=dz; e=ez; break;
case 16 : c=cz; d=dz; e=ez; f=fz; break;
case 17 : b=bz; break;
case 18 : b=bz; f=fz; break;
case 19 : b=bz; e=ez; break;
case 20 : b=bz; e=ez; f=fz; break;
case 21 : b=bz; d=dz; break;
case 22 : b=bz; d=dz; f=fz; break;
case 23 : b=bz; d=dz; e=ez; break;
case 24 : b=bz; d=dz; e=ez; f=fz; break;
case 25 : b=bz; c=cz; break;
case 26 : b=bz; c=cz; f=fz; break;
case 27 : b=bz; c=cz; e=ez; break;
case 28 : b=bz; c=cz; e=ez; f=fz; break;
case 29 : b=bz; c=cz; d=dz; break;
case 30 : b=bz; c=cz; d=dz; f=fz; break;
case 31 : b=bz; c=cz; d=dz; e=ez; break;
case 32 : b=bz; c=cz; d=dz; e=ez; f=fz; break;
case 33 : a=az; break;
case 34 : a=az; f=fz; break;
case 35 : a=az; e=ez; break;
case 36 : a=az; e=ez; f=fz; break;
case 37 : a=az; d=dz; break;
case 38 : a=az; d=dz; f=fz; break;
case 39 : a=az; d=dz; e=ez; break;
case 40 : a=az; d=dz; e=ez; f=fz; break;
case 41 : a=az; c=cz; break;
case 42 : a=az; c=cz; f=fz; break;
case 43 : a=az; c=cz; e=ez; break;
case 44 : a=az; c=cz; e=ez; f=fz; break;
case 45 : a=az; c=cz; d=dz; break;
case 46 : a=az; c=cz; d=dz; f=fz; break;
case 47 : a=az; c=cz; d=dz; e=ez; break;
case 48 : a=az; c=cz; d=dz; e=ez; f=fz; break;
case 49 : a=az; b=bz; break;
```

```
        case 50 : a=az; b=bz; f=fz; break;
        case 51 : a=az; b=bz; e=ez; break;
        case 52 : a=az; b=bz; e=ez; f=fz; break;
        case 53 : a=az; b=bz; d=dz; break;
        case 54 : a=az; b=bz; d=dz; f=fz; break;
        case 55 : a=az; b=bz; d=dz; e=ez; break;
        case 56 : a=az; b=bz; d=dz; e=ez; f=fz; break;
        case 57 : a=az; b=bz; c=cz; break;
        case 58 : a=az; b=bz; c=cz; f=fz; break;
        case 59 : a=az; b=bz; c=cz; e=ez; break;
        case 60 : a=az; b=bz; c=cz; e=ez; f=fz; break;
        case 61 : a=az; b=bz; c=cz; d=dz; break;
        case 62 : a=az; b=bz; c=cz; d=dz; f=fz; break;
        case 63 : a=az; b=bz; c=cz; d=dz; e=ez; break;
        case 64 : a=az; b=bz; c=cz; d=dz; e=ez; f=fz; break;

                                }

                  /****** calculate offset ****/
      offs = ((c1+a)*3125)+((c2+b)*625)+((c3+c)*125)+((c4+d)*25)+
             ((c5+e)*5)+((c6+f)*1) - 3125;



       if(*(part + offs) == 0)
          *(part + offs) = part_no;  /*** memory location unoccupied **/
       else
       {
        if(*(part + offs) != part_no)
          {
            if(*(part + offs) != -1)
            {
            *(restore_part_no + v) = *(part + offs);
            *(restore_location + v) = offs;
            v++;
            }
            *(part + offs) = -1;      /*** memory location occupied ***/
            if(level == 6)  not_unique = '1';
          }
        }


    }           /** end of redundant coding loop ****/
  }           /*** end of else *****/



}           /*** get next descriptor ****/
```

```
      if( not_unique == '1')
      {   last=v;
          part_no--;
          printf("\n Collision at last level\n"
                 " Part will be deleted from system \n");
          for(v=0; v < last; v++)
          {   *(part + (*(restore_location + v)) ) = *(restore_part_no + v) ;
              printf("\n %d",  *(restore_part_no + v));  }
       }
       else
       {
       printf("\n\n\n Complete code for part no. %d is: %1.0f,%1.0f,%1.0f,
                %1.0f,%1.0f,%1.0f \n", part_no,c1,c2,c3,c4,c5,c6);
       printf(" Number of possible codes: %4.0f \n",pow(2,total) );
       }



       printf("\n\n Do you wish to code another part?");
       key=getch();
       putch(key);
       if (key == 'y')
       {  part_no++;
          printf("\n Position next part under camera and press any key");
          getch();
        }
       else
        { part_no++;
          break;
         }

}        /******  teach another part *******/


/****** Save the existing part codes *****/

   printf("\n Saving Part Codes...");
   free(word);

   f1 = fopen("c:\\lutes\\pn.dat","w");
   f3 = fopen("c:\\lutes\\parts.dat","w");
   rewind(f1);
   rewind(f3);
   fprintf(f1,"%d", part_no);

           for(offs=0; offs <= 17000; offs++)
               fprintf(f3,"%4d",  *(part + offs));
   rewind(f1);
   rewind(f3);
   close(f1);
   close(f3);
}
```

```
/*****************  IDENTIFICATION MODE ROUTINE  ******************

     This routine identifies 2-D binary shapes which have been
     previously coded into the multi-dimensional array
     *(part + offs).  Descriptors will be classified into integer
     array coordinates one at a time until a memory location is
     found containing a part identity.  The boundaries for integer
     classification are defined by limit[7][7].

*************************************************************************/
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <math.h>

extern unsigned _stklen = 20000;
float descriptor[6];
int *x_coord, *y_coord, *word,start=1,*array;
char opps;
float limit[7][7] =    { {   0,  0,     0,     0,     0,     0,     0},
                         {   0, -1,     8,    20,    32,    44,   600},
                         {   0, -1,   114,   154,   194,   234,   600},
                         {   0, -1,     4,    10,    16,    22,    50},
                         {   0, -1,     4,    10,    16,    22,    50},
                         {   0, -1,     4,    10,    16,    22,    50},
                         {   0, -1,     4,    10,    16,    22,    50}};

main()
{
   char fnd;
   FILE  *f1,*f3;
   static int *part;
   extern int *word;
   extern char opps;
   int   a,b,c,d,e,f,level,classified,bndry,part_no,total,i;
   unsigned long offs=0;
   char  key;
   float c1,c2,c3,c4,c5,c6;
   extern float descriptor[6];
   extern float limit[7][7];
   extern float error[7][5][3];


   x_coord = (int *) malloc(30000);
   y_coord = (int *) malloc(30000);
   array = (int *) malloc(10000);
   part  = (int *) malloc(34000);
   word =  (int *) malloc(32000);
   if(part == NULL)
      {  printf("NULL pointers!");
         exit(1);  }
   if(word == NULL)
      {  printf("word is NULL!");
         exit(1);  }
```

```
/****  Load in the existing data base of coded parts ****/

      printf("\nPlease wait while existing Database is being loaded in");
      f1 = fopen("c:\\lutes\\pn.dat","r");
      f3 = fopen("c:\\lutes\\parts.dat","r");
      rewind(f1);
      rewind(f3);
      fscanf(f1,"%d",&part_no);

      for(offs=0; offs <= 17000; offs++)
              fscanf(f3,"%4d", (part + offs));
   rewind(f1);
   rewind(f3);
   close(f1);
   close(f3);


   printf("\nPosition part under camera and press any key");
   getch();



/**** This loop is executed until the user no longer
      wishes to identify parts ***/

   while (1)
   {
      opps='0';
      fnd='0';

      fourier();  /***** call the descriptor routine ****/
      if(opps == '1')
        { printf("\n\n*** Part not within field of view "
                          "and/or noise spot detected ***\n"
                          "Reposition part and/or remove noise, "
                            "press any key when ready...\n");
         getch();
         continue;
        }

     c1=c2=c3=c4=c5=c6=0;
```

```
    if(fnd != '1')
      printf("\n This part has not been taught to the system yet\n");
    else
      printf("\n\n\n Identity code for part no. %d is: %1.0f %1.0f"
        " %1.0f %1.0f %1.0f %1.0f \n", *(part+offs),c1,c2,c3,c4,c5,c6);

    printf("\n\n Do you wish to identify another part?");
    key=getch();
    putch(key);
    if (key == 'n') exit(1);
    printf("\n Position next part under camera and press any key");
    getch();

}


}
```

THIS PAGE LEFT BLANK INTENTIONALLY

THIS PAGE LEFT BLANK INTENTIONALLY

APPENDIX B

THE FAST FOURIER TRANSFORM

In 1942, Danielson and Lanczos showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms, each of length N/2. One of the two is formed from the even-numbered points of the original N, the other from the odd-numbered points. The proof is simply:

$$FD_k = \frac{1}{N} \sum_{n=0}^{N-1} f(n)\ e^{\left(\frac{-j2\pi kn}{N}\right)}$$

$$FD_k = \frac{1}{N} \sum_{n=0}^{N/2-1} f(2n)\ e^{\left(\frac{-j2\pi k(2n)}{N}\right)} + \frac{1}{N} \sum_{n=0}^{N/2-1} f(2n+1)\ e^{\left(\frac{-j2\pi k(2n+1)}{N}\right)}$$

$$FD_k = FD_k^e + FD_k^o$$

The Danielson-Lanczos Lemma can be used recursively. Therefore, having reduced the problem of computing FD_k to that of computing FD_k^o and FD_k^e, we can do the same reduction of FD_k^o and FD_k^e to the problem of computing the transform of their N/4 even-numbered input data and N/4 odd-numbered data. Furthermore, if N is a power of 2, we can continue applying the Danielson-Lanczos Lemma until we have subdivided the data down to transforms of length 1. Consider the input function f(n) where N = 8. The problem reduction is shown on the next page.

$$
\begin{aligned}
&DFT\{f(0),f(1),f(2),f(3),f(4),f(5),f(6),f(7)\} \\
&= DFT\{f(0),f(2),f(4),f(6)\} + DFT\{f(1),f(3),f(5),f(7)\} \\
&= DFT\{f(0),f(4)\} + DFT\{f(2),f(6)\} + DFT\{f(1),f(5)\} \\
&\qquad + DFT\{f(3),f(7)\} \\
&= DFT\{f(0)\} + DFT\{f(4)\} + DFT\{f(2)\} + DFT\{f(6)\} \\
&\qquad + DFT\{f(1)\} + DFT\{f(5)\} + DFT\{f(3)\} + DFT\{f(7)\}
\end{aligned}
$$

Each successive level in the problem reduction is found by calculating the DFT on the even and odd-numbered elements of each of the Discrete Fourier transforms in the preceding level. Thus the problem has been reduced from computing a single, 8-point DFT, to that of computing eight, 1-point DFT's. The advantage to this is that the DFT of a 1-point function, for example f(4), simply equals the function itself (i.e. $DFT\{f(4)\} = f(4)$). Therefore, calculating the 8-point DFT is simply a matter of adding adjacent pairs from the 1-point level to obtain the 2-point DFT's, then adding adjacent pairs of 2-point DFT's to obtain 4-point DFT's, then combining once again to get the final 8-point DFT. The combinations at each level take N=8 operations, and there are $\log_2 N = 3$ levels, therefore the whole algorithm is of order $N\log_2 N = 24$ operations. This is significantly less than the $N^2 = 64$ operations that are required for the DFT.

Although this seems straightforward, there is one preliminary requirement to implementing the FFT. If we are to combine adjacent pairs at the 1-point level, we must first re-arrange the order of the elements of the input f(n). Observe

when we work our way down from the 8-point level to the 1-point level, f(0) is mapped into the element position 0, f(1) is mapped into position 4, f(2) into position 2, f(3) into position 6, and so on. Therefore, what we wish to do with our input :

$f(n)_1$ = { f(0),f(1),f(2),f(3),f(4),f(5),f(6),f(7) }
is to re-arrange it such that :
$f(n)_2$ = { f(0),f(4),f(2),f(6),f(1),f(5),f(3),f(7) }

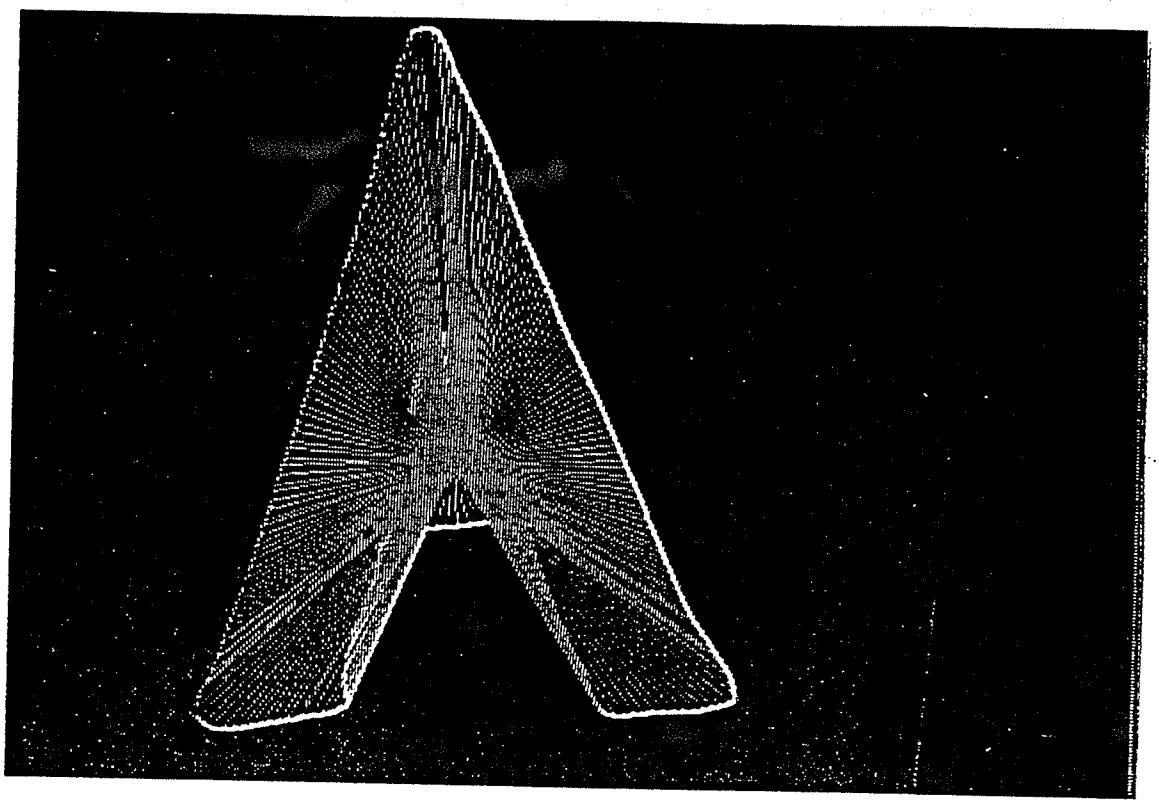
If we study this problem closer we observe that the array coordinates of $f(n)_2$ are the bit-reversed coordinates of $f(n)_1$. Denoting the array coordinates in bit notation, we have:

$f(n)_1$ = {f(000),f(001),f(010),f(011),f(100),f(101),f(110),f(111)}
$f(n)_2$ = {f(000),f(100),f(010),f(110),f(001),f(101),f(011),f(111)}

This preliminary operation is accomplished by the computer very quickly and is no greater than the order of $N\log_2 N$. Once the elements of $f(n)_1$ have been re-arranged, the FFT can be computed by combining adjacent pairs from the 1-point level until the full 8-point Fourier transform is achieved.

APPENDIX C

TEST IMAGES AND DATA RESULTS



Do you wish to identify another part?y
Position next part under camera and press any key

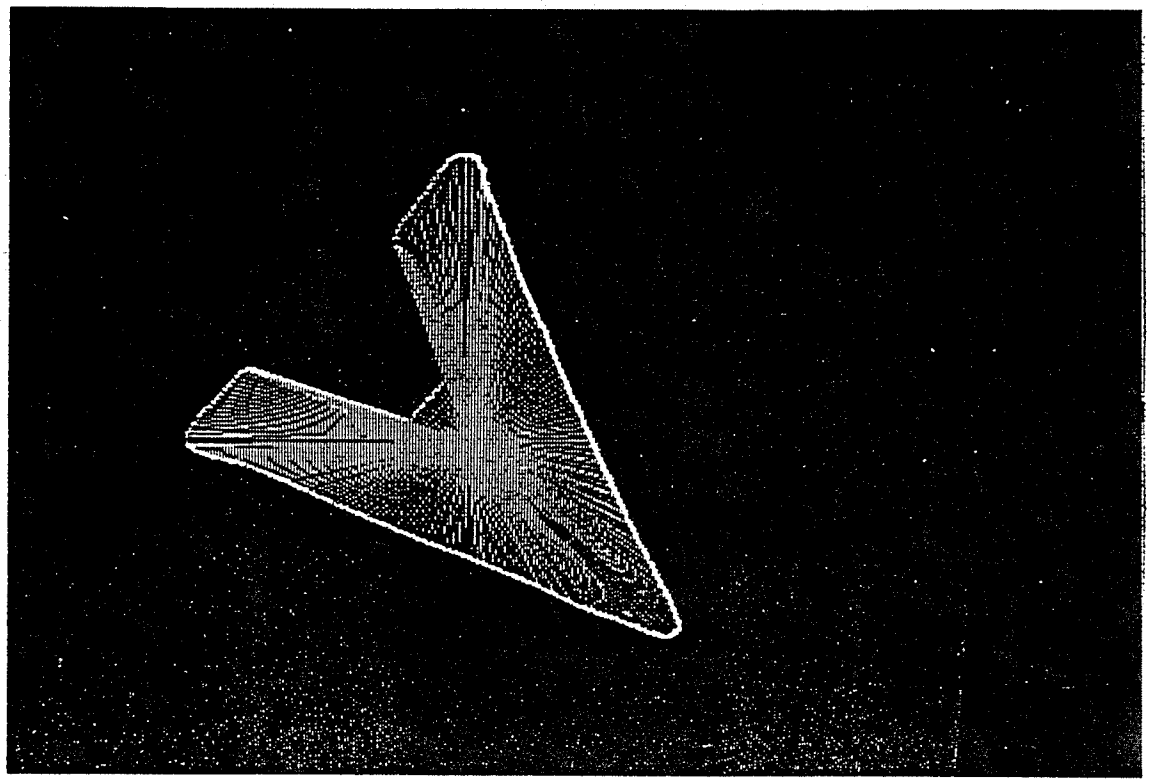
Normalized min_dist = 24.558012
Normalized max_dist = 170.274445

FREQ	REAL	IMAGINARY	MAGNITUDE
1	8.37.....	-0.33	6.4621
2	-2.07.....	3.16	2.9133
3	32.11.....	-5.93	25.1772
4	-0.64.....	1.80	1.4740

Part number is : 3
Identified at descriptor level 4

Identity code for part no. 3 is: 3 3 2 1 0 0

Do you wish to identify another part?



Do you wish to identify another part?y
Position next part under camera and press any key

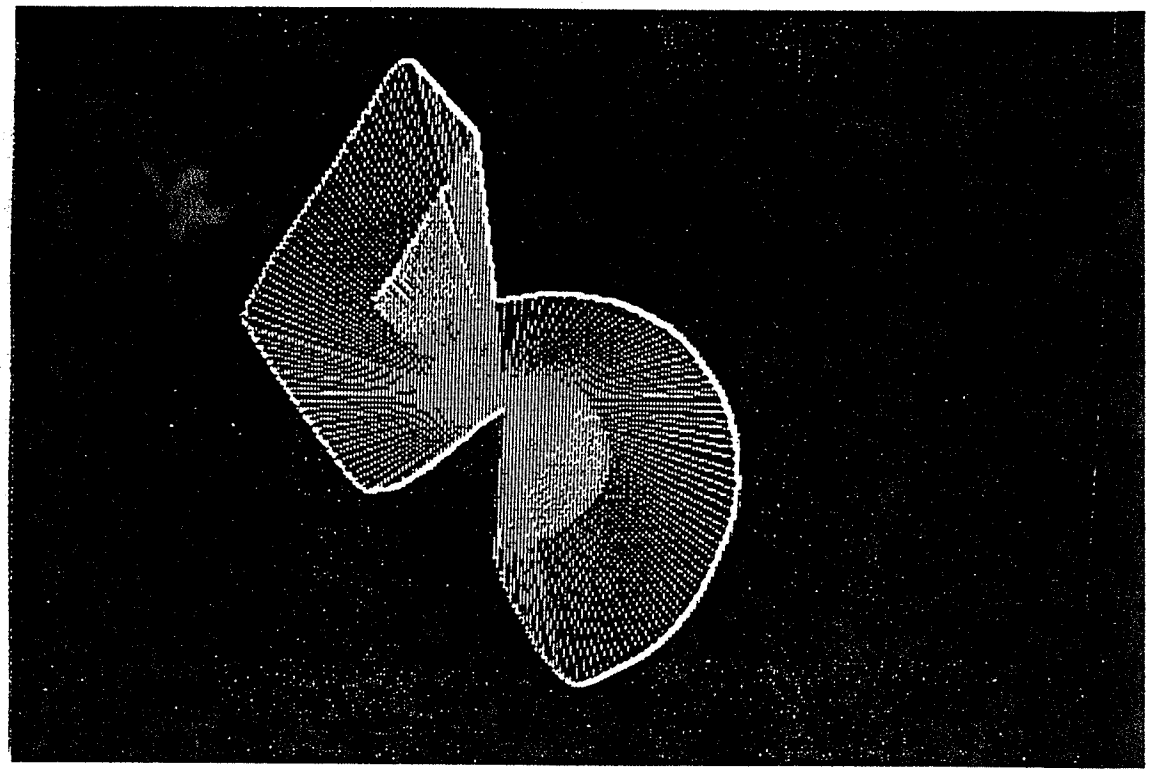
Normalized min_dist = 29.605131
Normalized max_dist = 165.968079

FREQ	REAL	IMAGINARY	MAGNITUDE
1	-1.98.....	-3.99	4.7110
2	1.84.....	-3.26	3.9604
3	23.34.....	5.75	25.4176
4	0.43.....	0.23	0.5182

Part number is : 3
Identified at descriptor level 4

Identity code for part no. 3 is: 3 3 2 1 0 0

Do you wish to identify another part?



Do you wish to identify another part?y
Position next part under camera and press any key

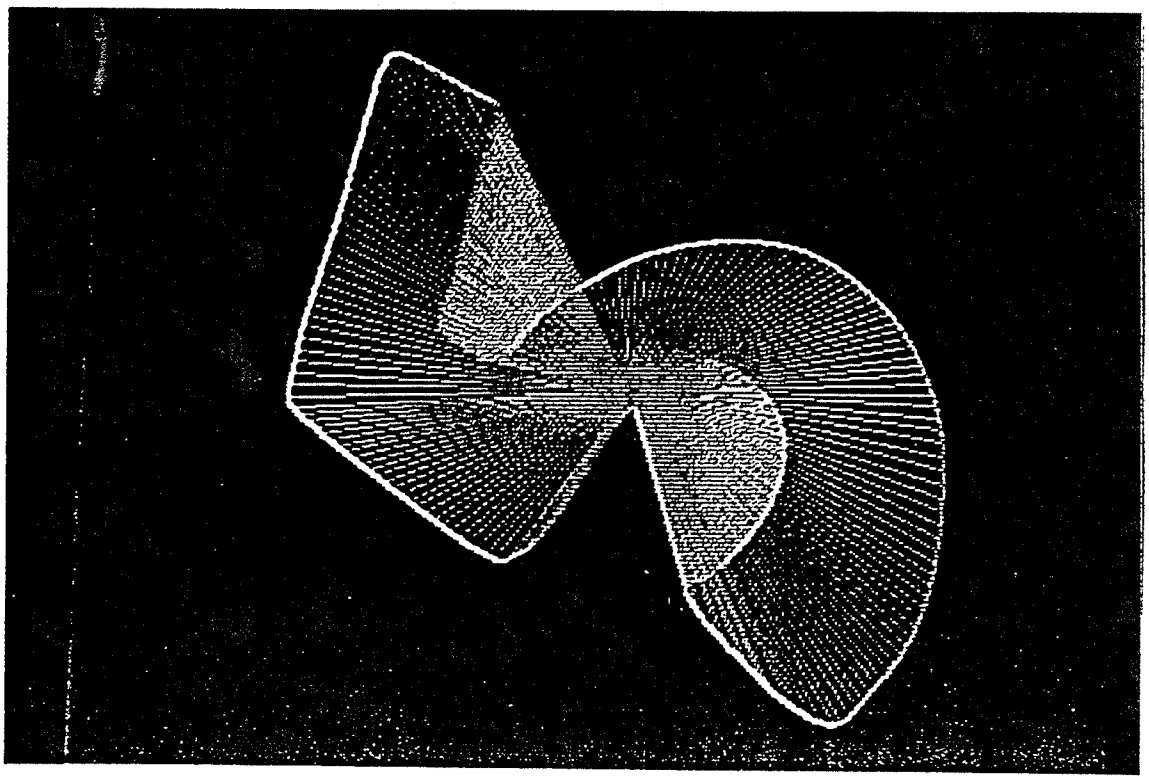
Normalized min_dist = 1.489399
Normalized max_dist = 177.332687

FREQ	REAL	IMAGINARY	MAGNITUDE
1	5.21.....	-6.64	7.1947
2	29.09.....	16.76	28.6186
3	2.08.....	0.05	1.7780
4	1.40.....	-3.77	3.4279

Part number is : 2
Identified at descriptor level 5

Identity code for part no. 2 is: 1 3 2 5 1 0

Do you wish to identify another part?



Do you wish to identify another part?y
Position next part under camera and press any key

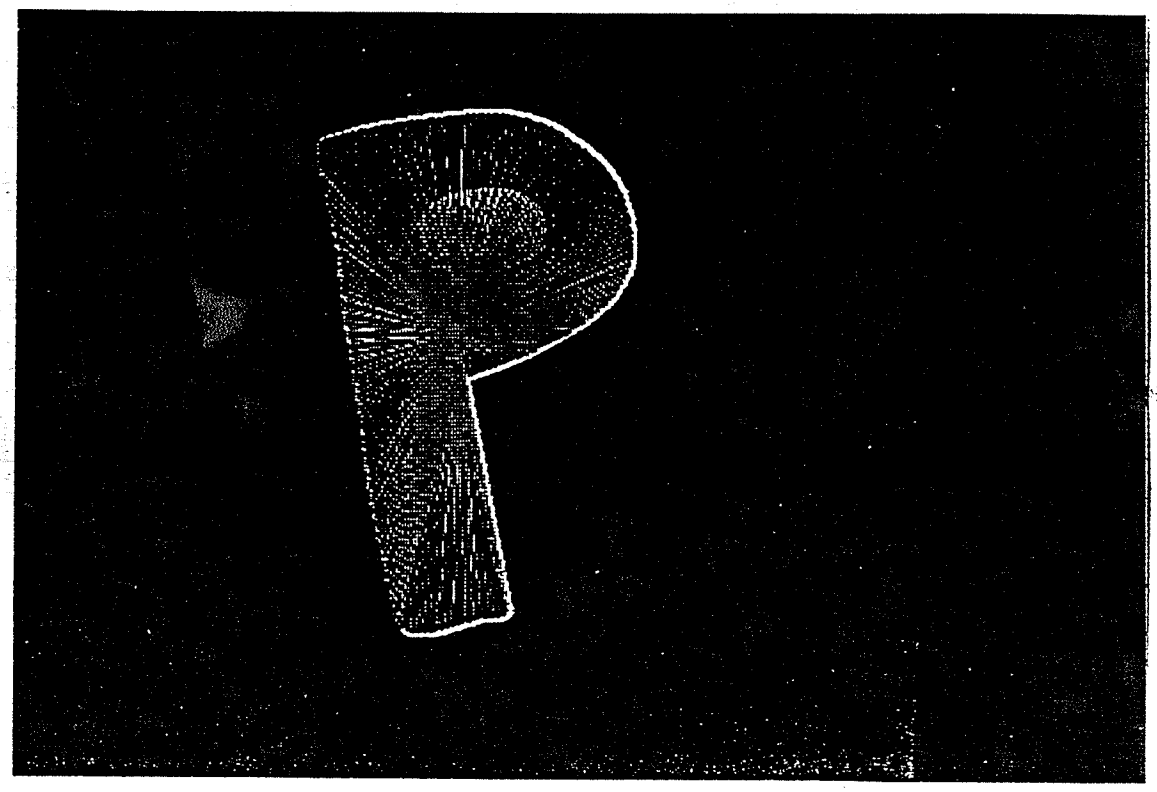
Normalized min_dist = 3.334150
Normalized max_dist = 170.777832

FREQ	REAL	IMAGINARY	MAGNITUDE
1	-3.70.....	6.52	6.2772
2	30.24.....	14.00	27.8819
3	1.20.....	2.43	2.2689
4	-0.61.....	-4.27	3.6092

Part number is : 2
Identified at descriptor level 5

Identity code for part no. 2 is: 1 3 2 5 1 0

Do you wish to identify another part?



Do you wish to identify another part?y
Position next part under camera and press any key

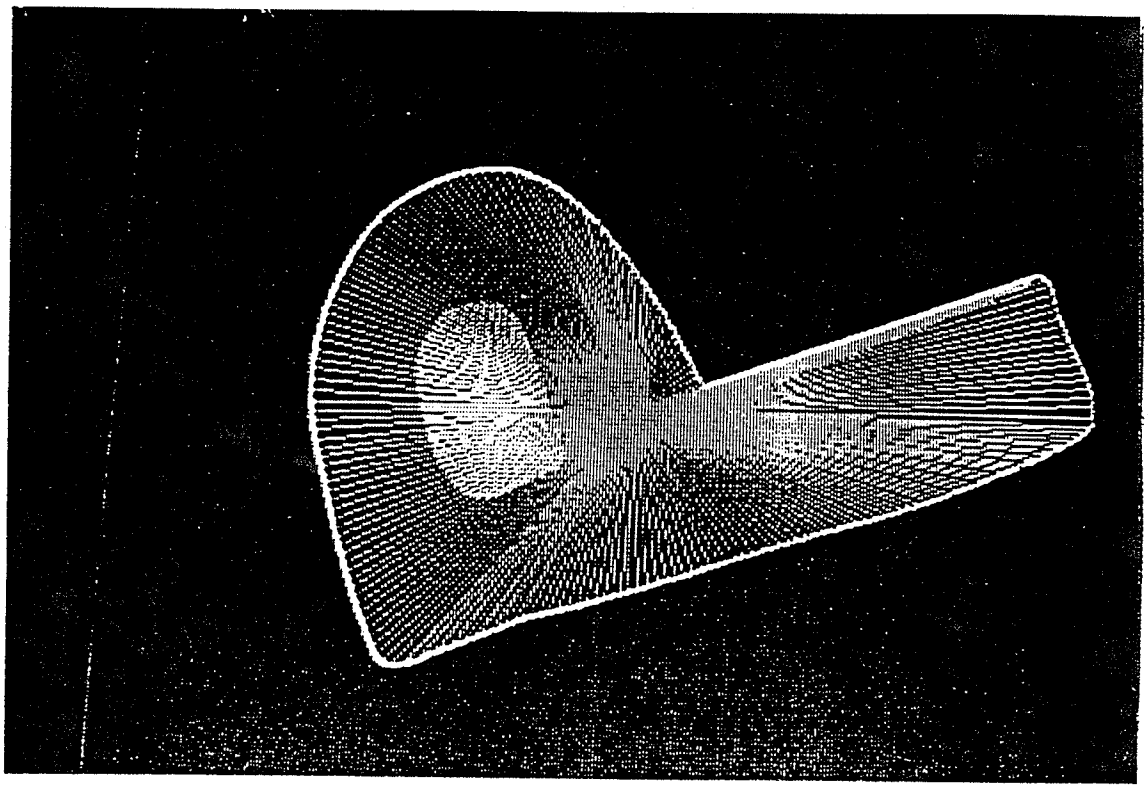
Normalized min_dist = 25.531504
Normalized max_dist = 170.197174

FREQ	REAL	IMAGINARY	MAGNITUDE
1	5.96.....	-3.46	5.5036
2	27.55.....	8.19	22.9557
3	10.17.....	7.63	10.1524
4	-2.63.....	-1.20	2.3120

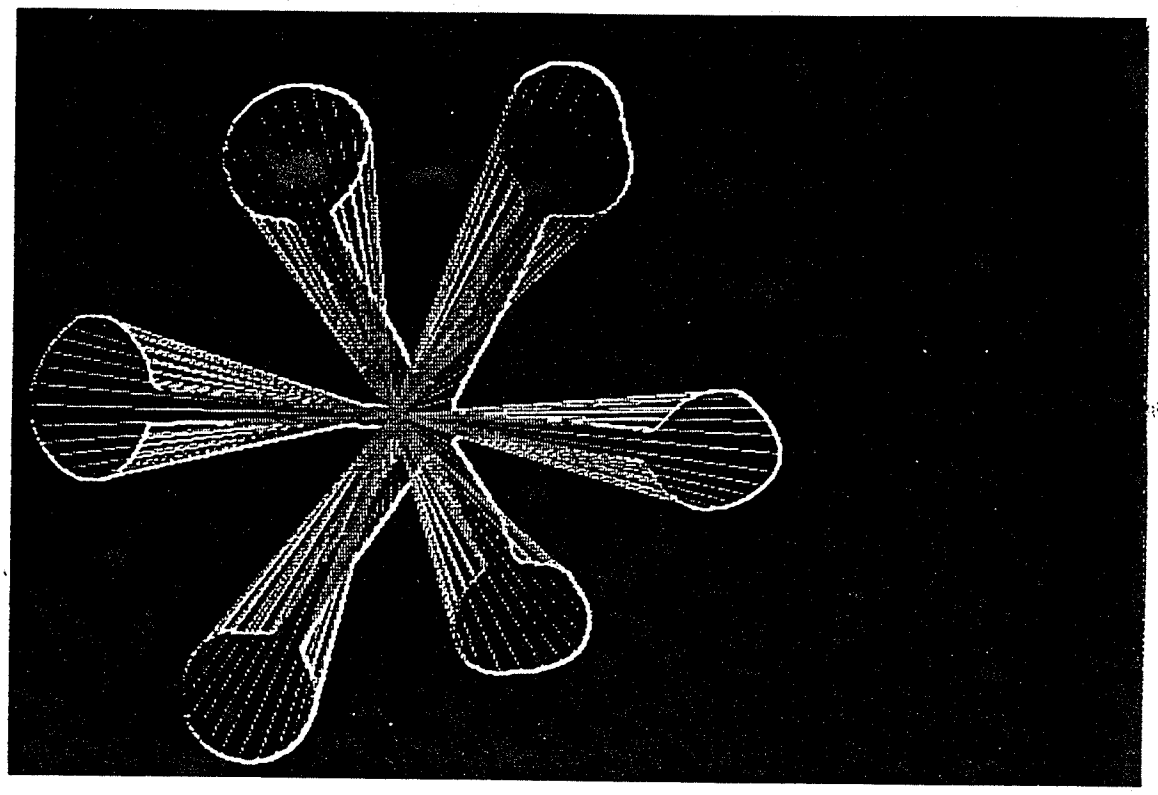
Part number is : 16
Identified at descriptor level 5

Identity code for part no. 16 is: 3 3 2 5 3 0

Do you wish to identify another part?



```
 Do you wish to identify another part?y
 Position next part under camera and press any key


Normalized min_dist = 23.669374
Normalized max_dist = 167.759201


FREQ        REAL        IMAGINARY        MAGNITUDE

  1          3.03.....     5.49             5.0096
  2         -5.13.....   -27.26            22.1492
  3         10.41.....     8.69            10.8276
  4          2.74.....    -1.25             2.4024

Part number is : 16
Identified at descriptor level 5


 Identity code for part no. 16 is: 3 3 2 5 3 0


 Do you wish to identify another part?
```



Do you wish to identify another part?,
Position next part under camera and press any key

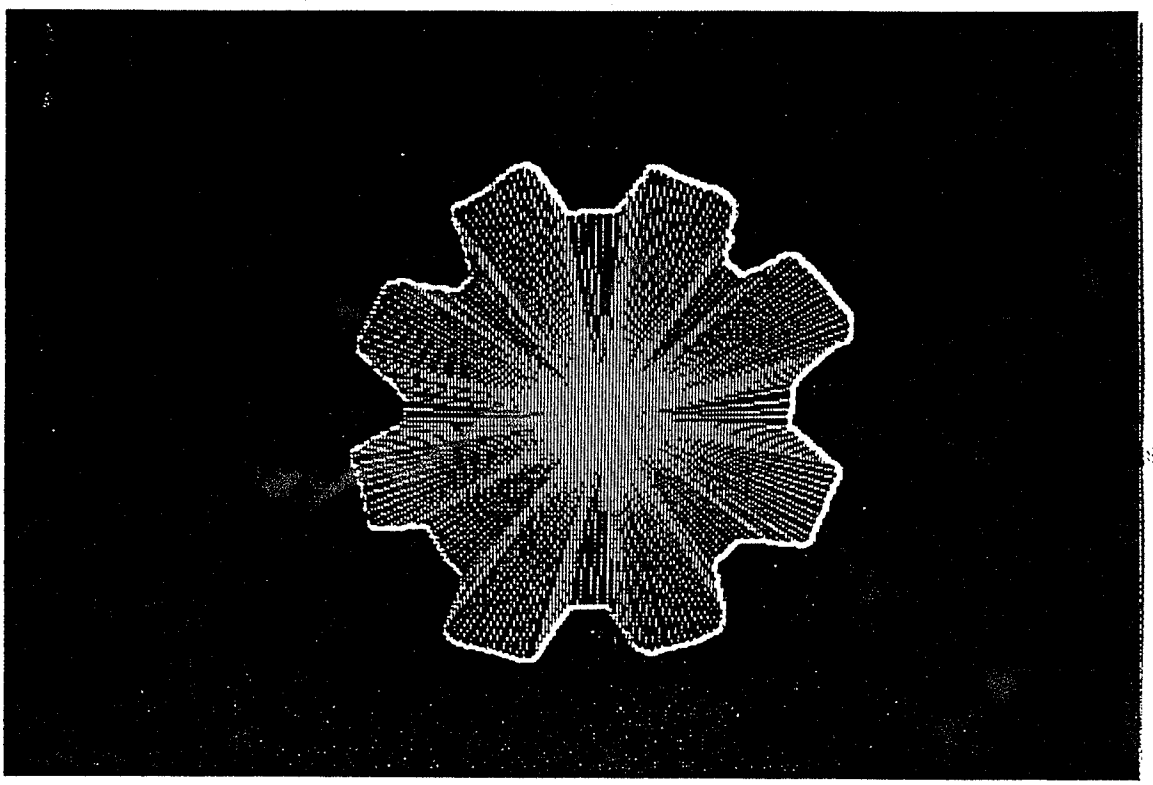
Normalized min_dist = 5.147645
Normalized max_dist = 182.147675

FREQ	REAL	IMAGINARY	MAGNITUDE
1	3.21.....	0.04	4.2734
2	1.93.....	3.30	5.0900
3	1.34.....	1.33	2.5087
4	4.98.....	-3.98	6.4246

Part number is : 12
Identified at descriptor level 6

Identity code for part no. 12 is: 1 3 2 2 1 2

Do you wish to identify another part?



```
 Do you wish to identify another part?y
 Position next part under camera and press any key


Normalized min_dist = 80.519577
Normalized max_dist = 119.370422


FREQ        REAL        IMAGINARY        MAGNITUDE

  1         1.14.....     0.57             0.8917
  2        -1.16.....     1.09             1.1165
  3         0.89.....    -1.08             0.9848
  4        -0.32.....    -0.53             0.4351

Part number is : 5
Identified at descriptor level 6


 Identity code for part no. 5 is: 5 2 1 1 1 1


 Do you wish to identify another part?
```

APPENDIX D

GENERATING FOURIER DESCRIPTORS, ROTATION INVARIANCE, AND INHERENT WEAKNESSES

The following paragraphs attempt to provide an understanding of how Fourier Descriptors (FDs) are generated, why they are rotation invariant, and to expose some of the inherent weaknesses of the method.

To examine how FDs are generated, consider the square object shown in figure 1(a). Figure 1(b) shows the radial distances, dist(n), from the centroid to each evenly spaced sample point along the boundary (only 32 sample points, N=32, have been used to simplify the analysis). Also shown is the sine weighting pattern of the Fourier Transform applied to dist(n) at frequency = 1. When calculating the FD, each element of dist(n) is multiplied with its corresponding sine weight, and the sum of each of these products is evaluated. For the square at frequency = 1, the result is zero. This makes sense since the elements of dist(n) from 0 to π are repeated in the same pattern from π to 2π. Therefore the positive sum of the products from 0 to π is equal to the negative sum of the products from π to 2π, yielding a total sum of zero. This condition holds true for frequency levels 2 and 3 also and can be described as follows:

Now if the frequency = 4 is applied (figure 1(c)), there is a dramatic change in the response. From the graph, we can see that

Figure 1(a)

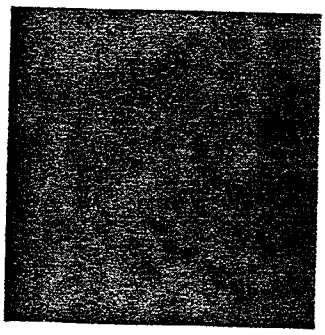


Figure 1(b)

Freq. = 1 Applied to Square

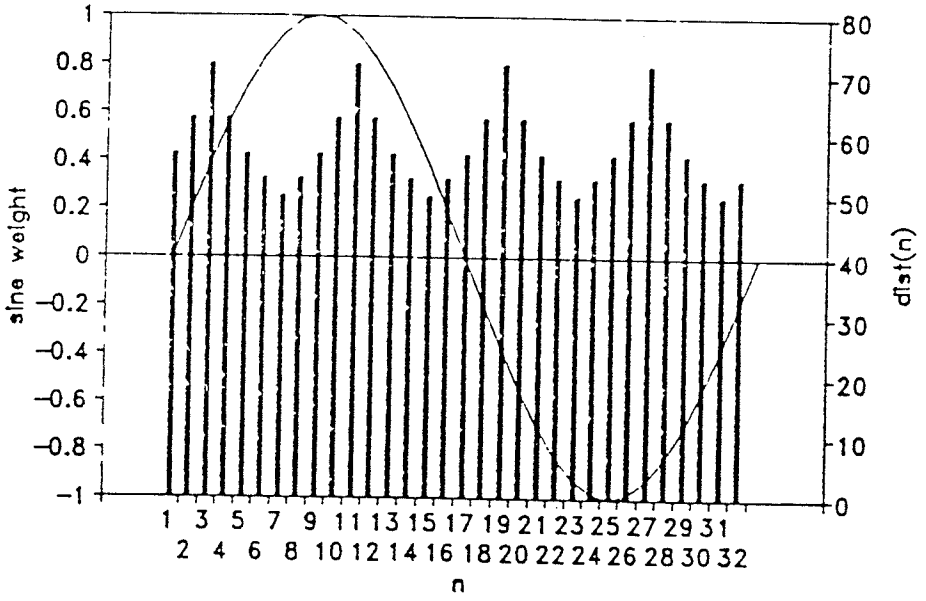


Figure 1(c)

Freq = 4 Applied to Square

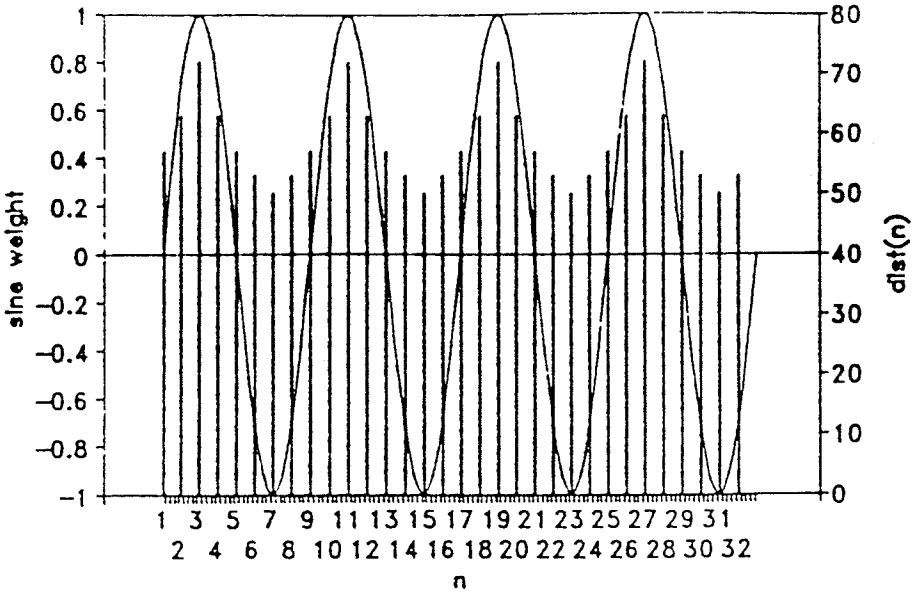
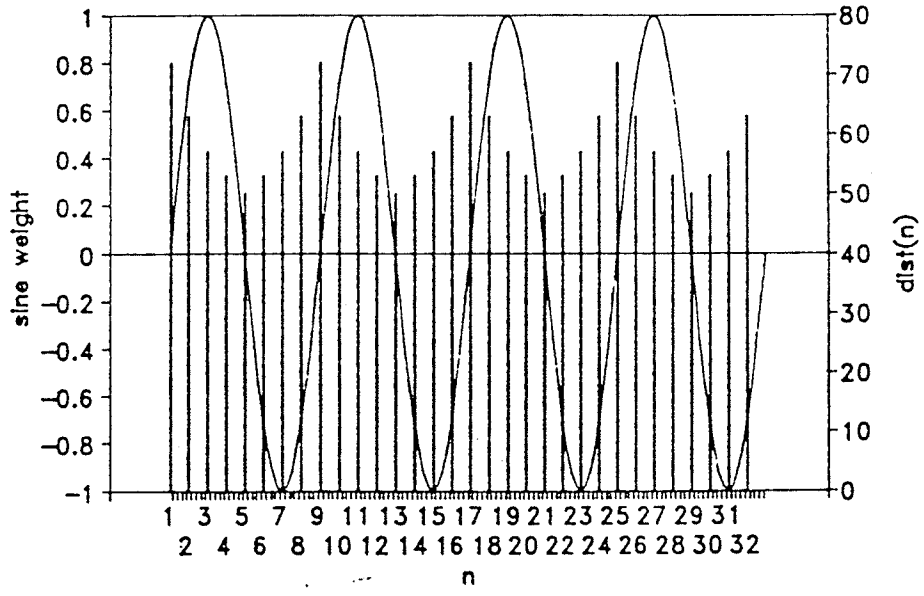


Figure 1(d)

Freq = 4 Applied to Square (shifted)



the peaks of the sine wave follow the same pattern of the peaks of the dist(n) function. The positive sine weights are always applied to the maximum dist(n) values whereas the negative sine weights are applied to the minimum dist(n) values, yielding a total sum response greater than zero. When the dist(n) function has a pattern similar to the selected frequency, a large response can always be expected.

One may be asking themselves if the same response can be obtained when selecting a different starting point in dist(n) (equivalent to a rotation of the object or a phase shift of dist(n)). If only the sine weighting pattern is used, the answer is no. Figure 1(d) shows dist(n) shifted by $\pi/2$ at frequency = 4. A quick examination of the graph will verify that condition (i) now exists and the response is zero. To capture the total response regardless of the starting point, both the sine and cosine weighting patterns must be used.

If the cosine curve is applied to the graph (figure 1(e)), we see that it has recaptured all the response that the sine curve lost from the phase shift. This represents the extreme case - one of the weighting patterns is providing all the response while the other is zero. Between these two extremes the total response is shared. Observing the well-known identity,

Figure 1(e)

Freq = 4, Cosine Curve Added

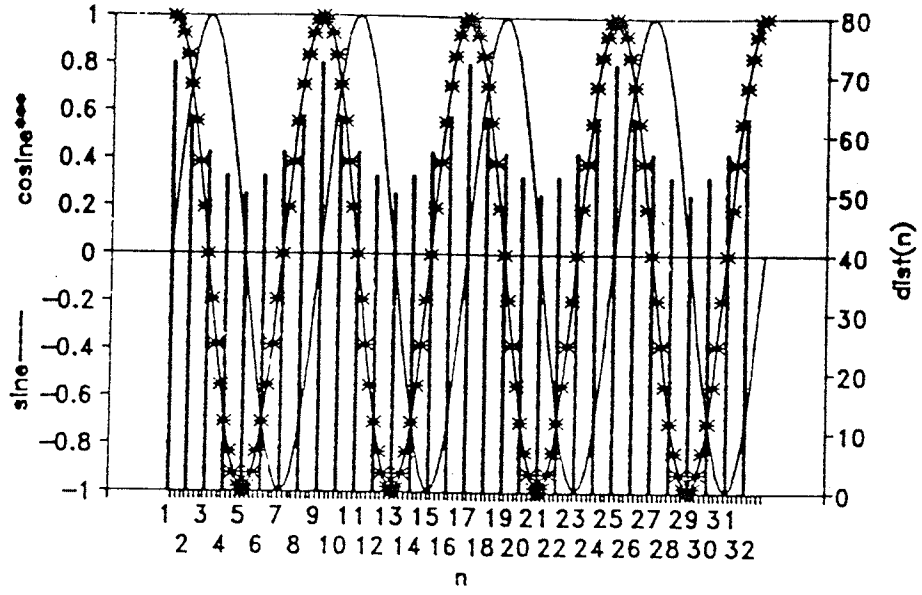
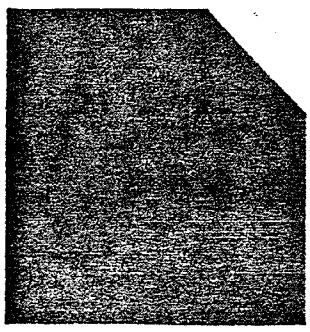


Figure 2



it is evident that if we square and add the sine and cosine responses, then take the square root to find the magnitude, this magnitude (or Fourier Descriptor) will remain constant regardless of the starting point (or rotation) of the object. This is the basic principle behind the rotation invariance of the Fourier Transform - what is lost in one weighting pattern (due to a phase shift) will be picked up by the other, and the magnitude derived from these two components will remain unchanged.

Although the Fourier Transform offers advantages such as rotation invariance, the method is not without inherent weaknesses. Distinguishing between objects that differ only slightly can sometimes be a difficult task. We will first examine a situation where the Fourier Transform can easily detect a small change in an object. A second example will apply a small change that is much more difficult to detect using FDs.

Consider the object shown in figure 2. There has been only one change made to the square - a corner has been cut off. Despite this small change, there is a significant difference in the FD output as shown in Table I. The first frequency level has jumped from zero to 1.051, the second from zero to 1.610.

TABLE I - FD Outputs for Square and Cut Square

Frequency Level	FD Magnitude for Square	FD Magnitude for Cut Square
1	0.000	1.051
2	0.000	1.610
3	0.000	2.049
4	4.518	3.323
5	0.000	0.431
6	0.000	0.501

There are a number of reasons for this change. The average radial distance (which is used to normalize the object for variations in object size) has decreased. Since normalization is accomplished by dividing each element of dist(n) by the average radial distance, there is an immediate change in the dist(n) function. The centroid has also shifted and the object is no longer symmetrical. The elements of dist(n) between 0 to π are not repeated in the same pattern from π to 2π and condition (i) does not hold true at any of the frequency levels. Therefore, all FDs for the new object are non-zero. Furthermore, the FD at level 4 has decreased since one of the four strong peaks in dist(n) has been reduced by the cut corner.

The change between the square and the cut square is detected quite easily. However, consider the two objects shown in figures 3(a) and 4(a). Although there is a notable difference between the two objects, the centroid location and average radial distance have

remained unchanged. If we look at a plot of the dist(n) function for each case (figures 3(b) and 4(b)), we see that only elements 4 through 9 are different. The remaining pattern is identical between the two objects. Detecting the difference in this small area is a difficult task for the Fourier Transform, especially at the lower frequency levels where the sine and cosine weighting patterns change only gradually between the elements. Figures 3(b) and 4(b) also show the sine and cosine curves for each object at frequency = 1. To calculate the difference between the FDs at the first frequency we need only look at the elements in dist(n) that are different (elements 4 to 9) (see Tables II and III).

Note in the tables that the average distance for elements 4 - 9 is equal for the two objects. Therefore if a uniform distribution (with constant weight was applied to dist(n), there would be no difference between $magnitude_1$ and $magnitude_2$. If frequency = 1 is applied (such as this case), a small change begins to show due to the slightly changing weights of the sine and cosine curves (as indicated in the tables this change of magnitude is .02). If frequency = 2 is applied, the difference in magnitudes increases further due to the more rapidly changing sine and cosine weights that exaggerate the differences between the elements of the objects.

Tables IV and V show the same calculations at frequency = 4. Note how the more extreme maximums and minimums of the weighting

Figure 3(a)

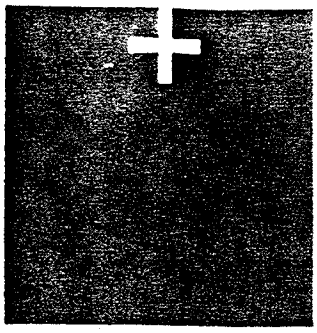


Figure 3(b)

Freq = 1 Applied to Fig. 3(a)

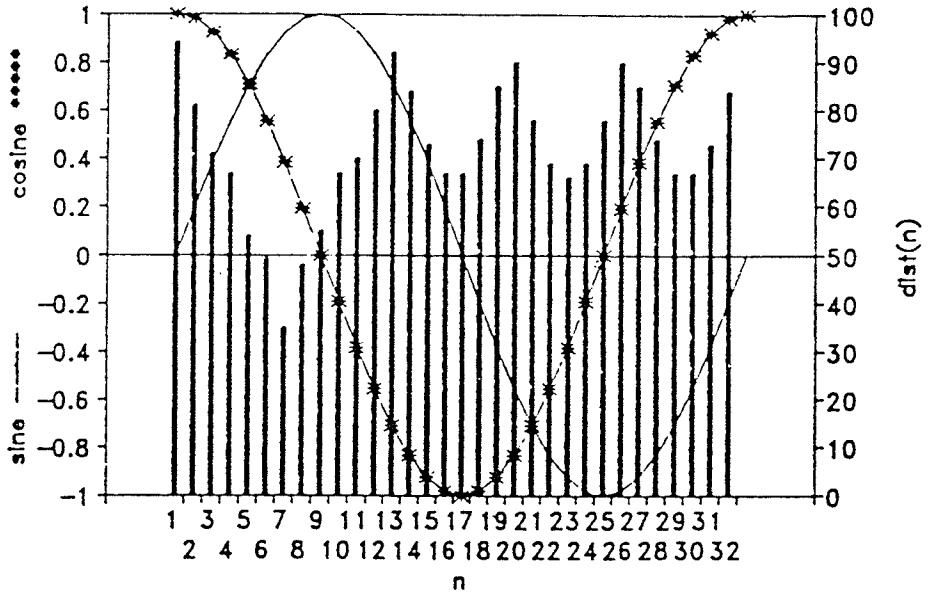


Figure 4(a)

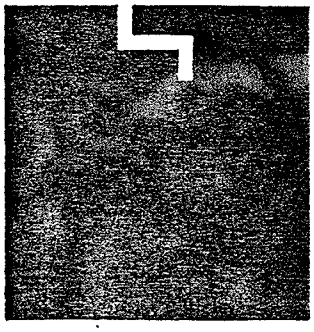
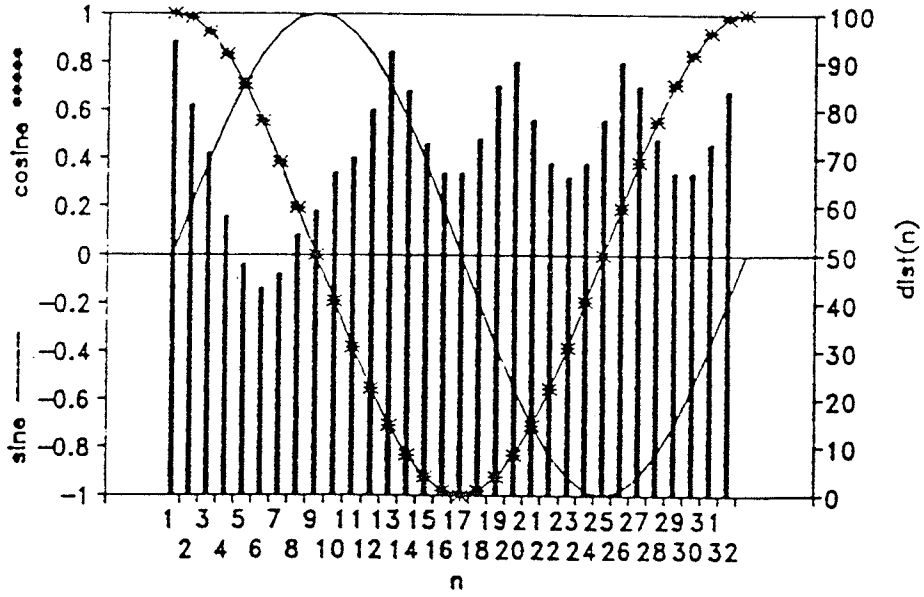


Figure 4(b)

Freq = 1 Applied to Fig. 4(a)



functions improve on emphasizing the difference in the dist(n) * sine and the dist(n) * cosine products between the two objects. The difference between the resulting FDs of the two objects is now .52. Although this is better than the magnitude difference found at frequency = 1, more FDs had to be calculated before the change in object shape could be detected. Furthermore, in this example, the change in object shape occurs over a fairly large percentage of dist(n). If the changed area was smaller with respect to the object size, the differences in the FDs would be reduced even further.

This example demonstrates how it is difficult for the Fourier Transform to distinguish between objects having dist(n) functions that differ by only a few elements. At the lower frequency levels, the change will be almost undetectable and any noticeable difference will not appear until higher frequency FDs are calculated. In spite of this weakness, FDs offer a good definition of shape if a sufficient number of FDs are calculated. The rotation and size invariant properties of the transform are particularly useful for any type of recognition system.

TABLE II - Magnitude of Elements 4-9 for Figure 3(a)
Using Frequency = 1

n	dist(n)	sine weight	cosine weight	dist(n) * sine weight	dist(n) * cosine weight
4	2.09	.555	.830	1.16	1.73
5	1.69	.707	.707	1.19	1.19
6	1.53	.830	.555	1.27	.85
7	1.09	.923	.382	1.01	.42
8	1.50	.980	.195	1.47	.29
9	1.72	1.000	0.000	1.72	0.00
	ave dist = 1.60			total = 7.82	total = 4.48
				$magnitude_1$ = 9.01	

TABLE II - Magnitude of Elements 4-9 for Figure 3(b)
Using Frequency = 1

n	dist(n)	sine weight	cosine weight	dist(n) * sine weight	dist(n) * cosine weight
4	1.81	.555	.830	1.00	1.50
5	1.50	.707	.707	1.06	1.06
6	1.34	.830	.555	1.11	.74
7	1.44	.923	.382	1.33	.55
8	1.69	.980	.195	1.66	.33
9	1.84	1.000	0.000	1.84	0.000
	ave dist = 1.60			total = 8.00	total = 4.18
				$magnitude_2$ = 9.03	

Difference in FD at Frequency 1: $= |magnitude_1 - magnitude_2|$
$= |\ 9.01 - 9.03\ |$
$= .02$

TABLE IV - Magnitude of Elements 4-9 for Figure 3(a)
Using Frequency = 4

n	dist(n)	sine weight	cosine weight	dist(n) * sine weight	dist(n) * cosine weight
4	2.09	.707	- .707	1.48	-1.48
5	1.69	0.000	-1.000	0.00	-1.69
6	1.53	- .707	- .707	-1.08	1.08
7	1.09	-1.000	0.000	-1.09	0.00
8	1.50	- .707	.707	-1.06	1.06
9	1.72	0.000	1.000	0.00	1.72
	ave dist = 1.60			total = -1.75	total = .69
				$magnitude_1$ = 1.88	

TABLE V - Magnitude of Elements 4-9 for Figure 3(b)
Using Frequency = 4

n	dist(n)	sine weight	cosine weight	dist(n) * sine weight	dist(n) * cosine weight
4	1.81	.707	- .707	1.28	-1.28
5	1.50	0.000	-1.000	0.00	-1.50
6	1.34	- .707	- .707	- .95	- .95
7	1.44	-1.000	0.000	-1.44	0.00
8	1.69	- .707	.707	-1.19	1.19
9	1.84	0.000	1.000	0.00	1.84
	ave dist = 1.60			total = -2.30	total = - .70
				$magnitude_2$ = 2.40	

Difference in FD at Frequency 1: $= |magnitude_1 - magnitude_2|$
$= |\ 1.88 - 2.40\ |$
$= .52$

This example demonstrates how it is difficult for the Fourier Transform to distinguish between objects having dist(n) functions that differ by only a few elements. At the lower frequency levels, the change will be almost undetectable and any noticeable difference will not appear until higher frequency FDs are calculated. In spite of this weakness, FDs offer a good definition of shape if a sufficient number of FDs are calculated. The rotation and size invariant properties of the transform are particularly useful for any type of recognition system.