NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Visual position recovery for an automated guided vehicle using pseudo-random encoding.

by
Khalfallah Habib.

A thesis
submitted to the school of graduate studies
in partial fulfillment of the requirements for the
Degree of Master of Applied Science
in
Electrical Engineering

at the

University of Ottawa

ISBN 0-315-75080-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

The thesis presents a visual position measurement system that allows an Automated Guided Vehicle (AGV) to recover its absolute position anywhere on a pseudo-random encoded guide-path. It is based on the "window property" of pseudo-random binary sequences. The guide-path consists of an (2^n-1) bit long encoding pseudo-random binary sequence marked on the floor as a sequence of geometric binary symbols ('E' for the binary value 1, and 'H' for the binary value 0). A long portion of the encoded guide-path has to be visible to the camera mounted on the AGV, in order to provide enough information to recover the n-bit pseudo-random window attached to any point in the field of view. The binary contents of this window uniquely identifies the absolute position of the encoded point. Image analysis and pattern recognition algorithms have been developed for the visual recovery of the pseudo-random window. The image processing essentially consists of the following steps: thresholding of the original image, boundary tracing each object in the image, polygon approximation of each boundary, extracting vertices and corners from each polygonal contour, and finally binary-symbol recognition by tree search. An original implementation of the split and merge polygon approximation results in an improvement of the run-time performance of this algorithm. An experimental vision system was implemented to test its performance for AGV absolute position recovery applications.

ACKNOWLEDGMENT

I wish to gratefully thank Emil for what I have learned from your natural enthusiasm and your work. I will always treasure your scintillating discussions and I hope that our friendship will continue for the years to come.

I wish to thank professor F. C. A Groen, from the University of Amsterdam, for his considerable help in the achievement of this work.

I would also like to acknowledge the friendship of M.A Greenspan, a master student in the research group, with who work was such a pleasure.

My parents are deserving the most gratitude for without their support I wouldn't be where I am today.

Table of contents

CHAPTER I

INTRODUCTION

Material flow management is recognized today as one of the fundamental problems of the modern automated factory. A new more flexible asynchronous material flow approach has been developed as an alternative to the traditional synchronous assembly line organization. This new approach largely makes use of the Automated Guided Vehicles (AGVs). They are driveless carriers employed on the factory floor as mobile production platforms for material transportation.

The underlying paradigm in AGV systems involves the repeated application of the *perception-planning-action* (PPA) cycle. Each element of the cycle is a specific area of research. The importance and degree of elaboration of element of the PPA cycle differs from one application to another.

Nevertheless, today, perception techniques attempt to provide intelligent (providing the maximum amount of information), robust and reliable interpretations. Beginning with the analysis of a single object in the work environment, researchers have been increasingly studying cluttered environments and partially occluded objects and upgrading robot intelligence using multiple visual and nonvisual sensors (data fusion).

The essential requirements of sensing devices consist of:

- production of intelligent environment descriptions after sensed data processing;
- easy and fast processing of the sensed data.

Low level image processing routines are usually time consuming. However, recent developments in image processing hardware, as well as software, made real time visual applications realizable. Visual applications in robotics have increased greatly during the last decade. Problems concerning visual guidance and obstacle avoidance, for instance, have been subject to intensive research.

Range data is important in AGV navigation. It provides information about the vehicle's environment and is useful for avoiding obstacles. As far as vision is concerned, only triangulation methods and structured light patterns have been used for *direct* range measurement. These methods require no information about the object shape.

However, the recent development in pattern recognition and image analysis has made range measurement using monocular vision possible. The object's range is estimated by considering the perspective deformations the object's image has been subject to. This method, called the *inverse perspective* technique, requires the a priori recognition of the object and its reference to a specified model. In general, the method is *indirect* and estimates the object range using an iterative least square algorithm.

Obviously inverse perspective methods require more processing then triangulation or structured light techniques. However, they provide more information since they estimate the pose (position orientation and scaling estimation) of the object. Unlike stereo vision and structured light techniques, monocular vision can not be used in an unknown environment and drawing *occupancy grids*, using monocular vision, is impracticable. Nevertheless, it has been successfully employed in visual guidance applications such as road and path following [Dick90]. In such cases, the object to be recognized, the path, is well defined and its model can be easily generated.

However, few results for the problem of AGV position measurement have been produced. Most systems presently rely on indirect techniques, such as "dead reckoning", in order to determine their position. Other techniques use optical docking and bar-code labeling of specially designed locations. In this case, a limited number of reference points can be used.

An original guide-path encoding technique, based on the window property of *pseudo-random binary sequences* provides an efficient alternative for AGV position measurement. Pseudo-random binary sequences have the property that each n consecutive bits form a unique pattern and may be used to fully identify the AGV position on the guide-path.

The first application of PRBS in AGV navigation was presented in [Petr89]. A sensing board containing optical reading heads for guide-path tracking and absolute position code reading attached to the single steering wheel of the robot. The translation of the scanned pseudo-random n-tuple into the natural representation (rank of the n-tuple in the sequence) was done sequentially. From the point of view of processing time, this system performs well. However, an AGV with such a minimal (1-bit) depth perspective of its environment can provide only limited navigation abilities.

Our research deals with the development of a visual system for AGV navigation in a pseudo-random encoded environment [Kha90]. The system proposed is similar to the one presented in [Petr89]. Two geometric symbols ('E' and 'H') are used to represent the two binary numbers '0' and '1'. These symbols are put on the floor in such a way they constitute a guide-path for the AGV. A camera is mounted on the vehicle and is constantly oriented to the guide-path. Every time there is a need for absolute position measurement, a snapshot of the path is taken. The image obtained is processed in order to recognize and locate each code appearing in the image. n successive binary codes are grouped together in a *pseudo-random binary window* . The

resultant n-tuple uniquely identifies the AGV position on the guide path.

From the viewpoint of navigation, the use of a camera to read the pseudo-random codes has the advantage of offering a wide view of the navigation path. This permits reconstruction of the 3D profile of the path in front of the vehicle. By solving the inverse perspective for each binary code in the image, a 3D profile of the path can be estimated. Slopes and turns along the track can be estimated well in advance to enable speed adjustment.

In chapter II general considerations in visual navigation as well as absolute positioning are made. Also the proposed solution for visual absolute positioning is described and finally a quick review of the pseudo-random sequence theory is made.

Two computer vision difficulties were encountered when designing the system. The first is related to the AGV mobility and its effect on the lighting conditions. The second is the result of a dilemma between the guide-path encoding resolution and the code recognition reliability. That is, the possession of a high encoding resolution implies that the number of codes covered by the camera are also high. In such situations, the binary codes, appearing in the image, are small and therefore their recognition becomes more difficult. This dilemma was solved by turning to robust shape analysis algorithms, although they may be more time consuming than other, more simple, techniques. The binary symbol shape analysis programs we have developed are described in chapter III.

Once the codes have been recognized, they are grouped together in a pseudo-random binary window which allows for absolute position recovery. In order to facilitate the binary window positioning as well as the estimation of the 3D profile of the path in front of the vehicle, the 3D position of each binary code in the image is determined. Straightforward as well as indirect methods for 3D binary code positioning are presented in chapter IV. Straightforward

methods are used when *a priori* information about the plane the code belongs to is available whereas indirect techniques use no such an *a priori* information. The type of estimation employed by indirect techniques is an iterative least square solution to the inverse perspective problem. At each iteration the 3D position of the code is refined using a singular value decomposition.

Chapter V describes the software implementation and presents the experimental results. A number of interesting data structures as well as recursive functions are employed in the feature generation, the recognition and the 3D positioning modules of the program are described. The search strategy which permits tracing the 3D profile of the visible path portion as well as locating the binary window along the path, employing 3D position of each visible code, is also described. Finally the experimental results illustrating the functionality of the designed system are presented.

CHAPTER II

THE USE OF PSEUDO-RANDOM ENCODING FOR VISUAL AGV NAVIGATION

II.1) INTRODUCTION

In this chapter a description of the proposed system for AGV navigation is made. In section II.2 a review of the existing visual navigation systems is made. The importance of range measurement for AGV navigation is discussed and a review of the existing visual range measurement techniques is made; a particular emphasis is made on the use of monocular vision. In section II.3 we discuss the problem of absolute positioning using pseudo-random encoding and discusses the system presented in [Petr89]. In section II.4 the principles of pseudo-random encoding as well as the code conversion algorithm is discussed. Finally in section II.5 we describe the proposed solution and its basic principles.

II.2) STATE OF THE ART OF AUTOMATED GUIDED VEHICLE VISUAL NAVIGATION

Autonomous vehicles have been the focus of increasing research and development effort. Applications such as remotely operated vehicles, autonomous land, underwater, and space vehicles, multiple interacting robots, autonomous systems for nuclear power plant maintenance, diagnosis and repair, hazardous waste handling, domestic robots and automated manufacturing have provided additional impetus to the AGV systems research. This effort has been

aided by significant developments in technologies such as VLSI, microcontrollers, reliable sensing and microsensing, actuators, computer hardware and software development environments.

Environment perception is an important component in AGV navigation. The perception system should permit:

- fast sensory data processing (to satisfy real-time functioning requirements*);
- intelligent sensory data interpretation.

Low level image processing routines, are usually considerably time consuming; this is due to the fact that, during processing, every pixel in the image has to be manipulated at least once. However, there has been a considerable development in image processing hardware; fast parallel processors performing low level image processing routines are commercially available [Schm88]. Also, from the software point of view, recently proposed serial implementations of low level image processing routines have interesting run-time performances [Vliet88]. Moreover, as far as machine intelligence is concerned, a considerable development in visual pattern recognition and image analysis has been made during the last decade. Monocular and binocular vision, nowadays, have reached a high level of technology.

For these reasons, visual applications to robotics and particularly AGV navigation have considerably increased during the last decade. Problems concerning the use of visual systems for range measurement, obstacle avoidance and path following have been subject to intensive research.

---

* The real-time functioning requirements are determined by the frequency at which measurements have to be made so that an adequate functionning of the system is assured.

Range measurement is important in AGV navigation. Range information can be exploited in a number of different applications such as vehicle mobility, path following and obstacle avoidance. two types of ***direct*** visual range measurement techniques can be distinguished: stereoscopic [Nitz88] and structured light patterns [Wax88] ranging techniques.

The **stereoscopic** ranging technique, also called the passive triangulation technique, has been the focus of increasing research during the last decade. This common ranging technique with ancient Greek and Egyptian origins has historically been used in ship navigation, surveying and civil engineering applications. Passive triangulation presents a simple trignometric method for calculating the distances and angles needed to determine object location. By comparing features in two images of the same scene taken by two cameras separated by a known baseline, one may determine the ranges to these features from their measured disparities in a simple fashion. Stereo visual information has been widely used in 3D object recognition and positioning [Kak88]. As far as AGV navigation is concerned, stereo vision has been used for path following and obstacle avoidance [Thor88].

However, the major difficulty encountered in stereo vision, is the so called *correspondence problem*: identifying features in the right image which correspond to those in the left image. Sorting this out computationally can be time consuming. One can alleviate this problem by going to an active, bi-static triangulation system based on the concept of "**structured light**". One of the stereo cameras is replaced by a light source which projects a known pattern of light on the scene. The camera then images the illuminated scene from a different vantage point. The range information manifests itself in the apparent distortions of the projected pattern. The spatial position of a point is uniquely determined as the intersection of a straight line with a plane, three planes, or as the intersection of two straight lines within a plane.

Estimating the range of each projected pattern, provides information about the 3D surface of the object, hence the object shape. Therefore object recognition can be performed using structured light patterns. Again, one is faced with a correspondence problem to solve, essentially to label the grid points in the imaged pattern according to their coordinates in the projected pattern. Several methods have been proposed to encode the grid in order to allow for correspondence between grid points and their projected patterns. Among these methods is the binary encoding technique [Vuy90].

Structured light patterns permit construction of a topographic map of the robot's immediate environment [Wax88]; this map, also called **occupancy grid**, is used for planning a path over the terrain while avoiding obstacles.

While stereo vision and structured light patterns are the only available *straightforward* visual range sensing devices, there are certain **monocular methods** which permit indirect range estimation. In such techniques, range measurement is considered as a *parameter estimation problem*. They estimate the *six* geometric parameters that determine the position of an object in the 3D space (**pose**: Position, Orientation and scaling estimation). However to be capable of estimating the *pose*, it is necessary to recognize the object beforehand and to refer it to a specific model. Monocular vision, in general, can be divided in two groups: the **dynamic vision and** the **static vision**. In dynamic vision, which uses the so called *spatio-temporal* approach, a sequence of images is used to analyze the same scene, while in the static approach, a single image is considered to describe a scene.

In dynamic vision, also called **image motion**, the 3D position of an object is estimated by considering a sequence of discrete images taken successively, at a constant period of time. The **optical flow**, or **instantaneous velocity field**, assigns to every point on the visual field a 2D **retinal velocity** at which it is moving across

the visual field. If the object has been recognized and referred to a model, using classical parameter estimation techniques (such as Kalman filtering [Dick90]), the 3D position and velocity of this object can be estimated. A clever application of image motion to automatic visual guidance is the one proposed by Dickmanns *et al* [Dick90]. The Kalman filter approach to recursive state estimation is extended to image sequence processing. The *road profile* and the eventual existence of *obstacles* in front of the vehicle are *estimated recursively* from the available sequence of images.

In the case of static vision, the 3D position of the object is computed by solving the **inverse perspective** problem. The method is based upon, having recognized the object of interest, evaluating the deformations, due to the perspective transformation, that affected the object image. These deformations being dependent on the relative position of the object and the camera, the 3D object position can be estimated. Straightforward methods, such as the *flat-earth* technique used in road following problems [Mor90], use a priori information about the object position to find the pose. However, in the general case where no access to such information is possible, only model based techniques which require the recognition of the object beforehand can be used. Morgenthaler *et al* [Morg90] use static images for automatic visual guidance. The 3D profile of the road in front of the vehicle is estimated by solving the inverse perspective.

Monocular visual raging techniques nowadays are subject to increasing research interest. Certainly, dynamic vision provides more information and may be more robust than static vision. However, it requires more time processing. We have chosen to use static monocular vision for the system to be designed.

II.3) USING PSEUDO-RANDOM ENCODING FOR AUTONOMOUS VEHICLE NAVIGATION

As mentioned above, the AGV visual guidance system to be designed has to meet two requirements:

1) automatic visual guidance using a navigation line, and
2) ability to measure at any time and location the absolute position.

The first problem is a classical path following problem. In [Ishi88] and [Vey86] two different visual navigation techniques using white line recognition were presented. The problem of AGV path following is much similar to road following ([Dick90] and [Mor90]). Among the requirements the path following system has to satisfy is the possibility to *reconstruct the profile of the 3D path in front of the vehicle*. Indeed, *constructing the path over as large a distance as possible is profitable* since the *reconstructions from several vehicle positions overlap*, and can thus be *combined for added reliability*.

As regards the position measurement problem, little research dealing with this problem has been made. Presently, most AGV control systems rely on indirect measuring methods, such as "dead reckoning", in order to find their position. Since the errors encountered in these methods are cumulative, the AGV may eventually lose its position. Some attempts have been made to compensate for this drawback by using either optical calibration methods (optical docking and bar-code labeling [Hon87], [Kab87]) of specially designed locations [Del81]. For economic reasons, these solutions are restricted to a limited number of a priori defined reference points.

It is a common approach to recover the absolute position p of a vehicle following a guide-path by marking the full length of the path

and reading the particular mark corresponding to the current position of the vehicle. This marking is usually done by writing a distinct absolute code-word on each step of the path. Of course, the resulting number of code tracks increases proportionally with the desired marking resolution as well as the desired guide-path length, making practically impossible to use this absolute encoding method for real-life applications.

An original guide-path encoding technique, based on the properties of ***pseudo-random binary sequences*** (**PRBS**), [Petr89], [Petr90] and [Petr], provides a more efficient alternative for AGV absolute position measurement. This encoding technique solves the position accuracy problem by allowing the AGV to recover its absolute position at any point on the vehicle path. The method has the advantage of requiring only ***one bit of code per quantization interval*** (see Sect. II.4), making AGV absolute position measurement a practical feature even for very high resolution applications.

Pseudo-random binary sequences are generated by direct modulo-two feedback n-bit shift registers [Petr89]. Such a sequence has the property that each n consecutive bits form a unique pattern and may be used to fully identify the AGV position on the guide-path (n is called PRBS order).

The first application of PRBS to AGVs was presented in [II.16]. Reflective-type optical reading heads for guide-path tracking were attached to the steering wheel of the AGV. The AGV followed a guide-path painted on the floor under the control of a path tracking algorithm implemented by software in the main robot controller. The absolute-type encoding method employed required two supplementary tracks along the guide-path. These two tracks were 1-bit wide each, one being used for the absolute position encoding and the other for the synchronization of the position-code readings. The translation of the scanned pseudo-random n-tuple into the natural representation (rank of the n-tuple in the sequence) was

done sequentially. The code track has had to be scanned bit by bit using a single reading head and the n-tuple was assembled sequentially in an n-bit shift register. The sequential code conversion algorithm was based on the idea that it is possible to find the natural value of the AGV position by simply counting the steps which are required for the scanned pseudo-random n-tuple in order to arrive by successive "back-shifts" in the "zero-state" (see section IV).

The system proposed in [Petr89] is fast and robust. However, an AGV with such a minimal (1-bit) depth perspective of its environment could provide only a rather limited navigation ability. For path planning purposes, we need to reconstruct the profile of the largest visible portion of the path in front of the vehicle.

In the work presented here, we propose the use of a visual system for an automated guided vehicle navigation in an pseudo-random encoded environment. The system will use a camera to visually inspect the binary encoded guide-path (see Schm88).

Two geometric symbols ('E' and 'H') are used to represent the two binary numbers "0" and "1". These geometric symbols are called ***binary symbols***. These symbols are put on the floor in such a way that they constitute a guide-path for the AGV. The camera is mounted on the vehicle and is constantly oriented to the guide-path. Every time there is a need for absolute position measurement, a snapshot of the path is taken. The distance of the camera to the floor and the sizes of the binary symbols are determined by the fact that at least $2n$ binary codes must be in range, where n is the PRBS order defined above. The image taken by the camera is processed and the codes appearing in the image are recognized and located. n successive binary codes are isolated, thus constituting ***a pseudo-random binary window*** (**PRBW**) that uniquely identifies the AGV's position (see Fig II.3). The purpose of having at least $2n$ codes in range is first to have a redundant number of codes visible to the camera, second to allow for the reconstruction of a large portion of the guide-path appearing in front of the vehicle.

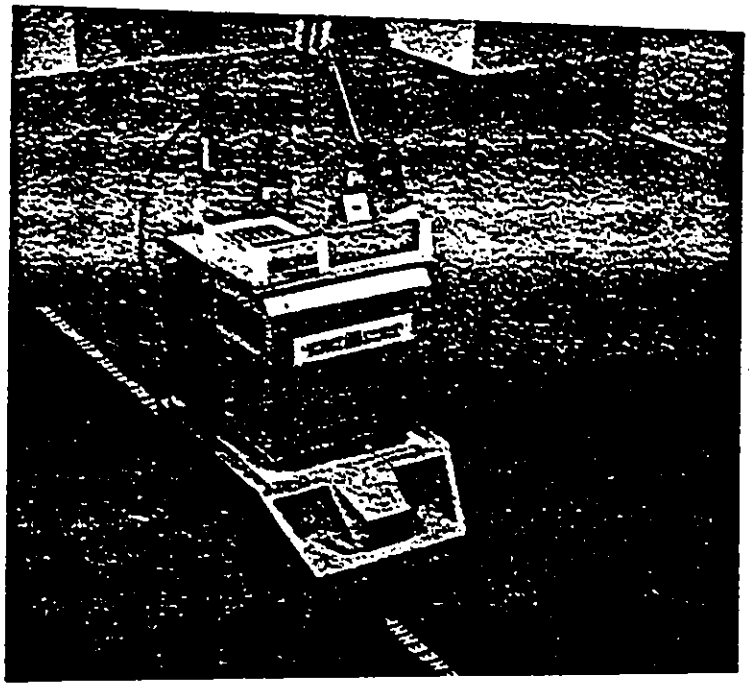


Fig II.1 AGV with guide-path on the floor.

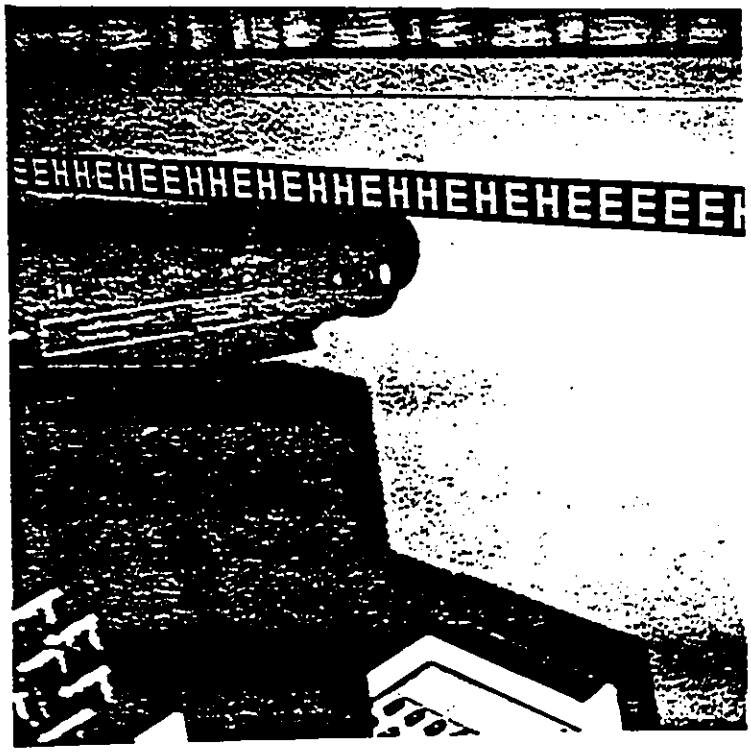


Fig II.2 AGV with guide-path on the wall.

This system has several advantages over the one presented in [Petr89]:

1) Because the camera covers a large portion of the path, one snapshot is sufficient to recover the absolute position of the vehicle, while, in [Petr89], the code-track is scanned bit-by-bit.

2) no synchronization problems exist and not more than one track is necessary, whereas in [Petr89] three tracks were needed.

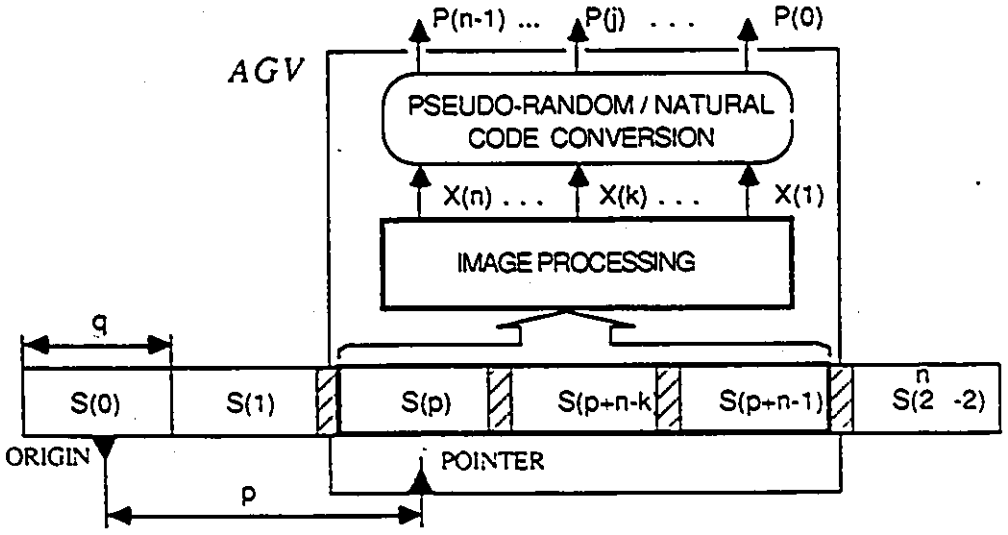


Fig II.3 Pseudo-random/natural code conversion.

3) unlike in [Petr89], the orientation of the path is easily determined (the orientation of the letter 'E' indicates the orientation of the track), and no difficulties are encountered when the moving direction is changed.

4) a larger view of the track is available thus conveniently offering information about the 3D profile of the path.

5) because most of the space in front of the vehicle is visible to the camera, obstacle detection techniques, such as the one presented in [Dick90], can be employed.

6) unlike in [Petr89], it is not compulsory for the encoded track to be on the floor; it can be put anywhere, as long as it is visible by the camera. In figure II.3 we can see that the track is painted on the wall.

7) an extended 2D pseudo-random encoding can be employed for position measurement [Will76].

From the geometrical point of view in the 1D pseudo-random encoding case, the ***3D profile of the track*** in front of the vehicle can be estimated by *determining the 3D position of each binary code* appearing in the image. Slopes and turns along the track can be estimated well in advance to enable speed adjustment.

In the following section we present the most important, as far as our application is concerned, mathematical properties of PRBS as well as the code conversion algorithm proposed in [Petr89].

II.4) MATHEMATICS OF PSEUDO-RANDOM ENCODING AND CODE CONVERSION

II.4.A) MATHEMATICS OF PSEUDO-RANDOM SEQUENCES

Pseudo-random binary sequences, which are also called pseudo-noise sequences, are deterministic strings of binary digits

that have statistical properties similar to those of random sequences. A shift register consisting of n stages represents the binary memory elements. At each time unit, the contents of these binary memories are shifted one place to the right and the present value of the feedback, R(n+1), is placed into the leftmost memory element (see Fig II4). The bits which are shifted out of the rightmost box represent the values of an infinite binary sequence.

The feedback is performed using modulo two addition or an exclusive or gate. Having:

$$R(n+1)= c(n)\cdot R(n) \oplus c(n-1)\cdot R(n-1) \oplus \cdots\cdots c(2)\cdot R(2) \oplus c(1)\cdot R(1)$$

the determination of the array c specifies the particular feedback connections to be made with the shift register. A polynomial of degree n can be used to define the feedback elements required for an n-bit shift register:

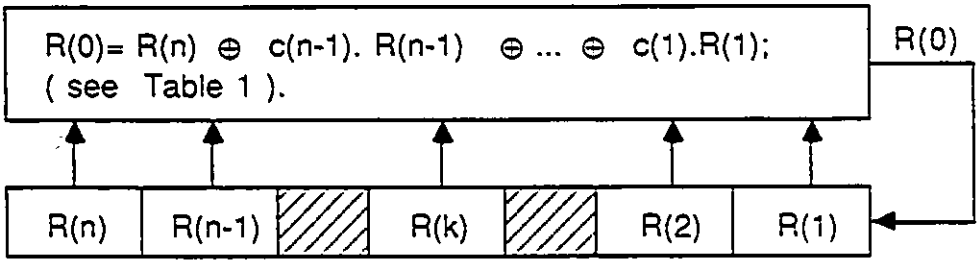


Fig II.4 PRBS generated by a shift register.

$$f(x)= x^n + c_n\cdot x^{n-1} + \cdots + c_2\cdot x^1 + c_1\cdot x^0 \qquad (II.1).$$

In order to to construct a PRBS sequence of length 2^n-1, one needs a primitive polynomial of degree n. For instance the primitive polynomial in the case of n=4 is:

$$p(x)= x^4 + x + 1 \qquad (II.2).$$

Assuming the register contains a_{i+3}, a_{i+2}, a_{i+1}, a_i, at time i, then at time i+1 it contains a_{i+4}, a_{i+3}, a_{i+2}, a_{i+1} with:

$$a_{i+4} = a_{i+1} \oplus a_i \qquad (II.3).$$

In order to produce the infinite sequence, it is necessary to specify the initial values for $a_0, a_1, \ldots, a_{n-1}$. Because each of the n-bits of the shift register can only be 0 or 1, there are 2^n possible states which means that the infinite sequence $a_0, a_1, a_2 \ldots$ must be periodic. Moreover, with modulo two or exclusive or feedback, the zero state 0, 0, 0,... cannot occur unless the infinite sequence is the trivial case of all zeros. The exclusion of the zero state results in the maximum possible period of a useful PRBS to be 2^n-1.

Let $p(x)$ = fixed primitive polynomial of degree n, and

∂_n = the set consisting of the PRBSs obtained from p(x) plus the sequence of 2^n-1 zeros.

If $b=b_0 b_1 \ldots b_{2^n-2}$ is any PRBS in ∂_n then any cyclic shift of b, say

$$b_j b_{j+1} \ldots b_{2^n-2} b_0 \ldots b_{j-1} \qquad (II.4)$$

is also in ∂_n. This constitutes the so called shift property.

Moreover, if

$$p(x) = \sum_{i=0}^{n} h_i \, x^i \qquad (II.5).$$

with $h_i=0$ or 1 for $0<i<n$. Now any PRBS b in ∂_n satisfies the recurrence:

$$b_{i+n} = h_{n-1} b_{i+n-1} \oplus h_{n-2} b_{i+n-2} \oplus \cdots \oplus h_1 b_{i+1} + b_i \qquad (II.6)$$

for i=0,1... . This is just a generalization of the feedback equation which is obtained from the primitive polynomial and used with the feedback shift register.

If a *window of width n* is slid along a PRBS in ∂_n, each of the 2^n-1 nonzero binary *n-tuple is seen exactly once*. This so called the ***window property*** follows from the fact that p(x) is a primitive polynomial which is irreducible.

Of the PRBS characteristics mentioned above, the most important to the position encoding technique for AGVs is the window property. With such a sequence, each n successive bits form a unique pattern; hence, *may be used to fully identify the AGV's absolute position on the guide-path.*

Shift register length n	Feedback for direct p.r.b.s. $R(0) = R(n) \oplus c(n-1) \cdot R(n-1) \oplus \ldots \oplus c(1) \cdot R(1)$	Feedback for reverse p.r.b.s. $R(n+1) = R(1) \oplus b(2) \cdot R(2) \oplus \ldots \oplus b(n) \cdot R(n)$
4	$R(0) = R(4) \oplus R(1)$	$R(5) = R(1) \ominus R(2)$
5	$R(0) = R(5) \oplus R(2)$	$R(6) = R(1) \oplus R(3)$
6	$R(0) = R(6) \oplus R(1)$	$R(7) = R(1) \ominus R(2)$
7	$R(0) = R(7) \oplus R(3)$	$R(8) = R(1) \ominus R(4)$
8	$R(0) = R(8) \oplus R(4) \oplus R(3) \oplus R(2)$	$R(9) = R(1) \oplus R(3) \oplus R(4) \oplus R(5)$
9	$R(0) = R(9) \ominus R(4)$	$R(10) = R(1) \oplus R(5)$
10	$R(0) = R(10) \oplus R(3)$	$R(11) = R(1) \oplus R(4)$

Table 1.

Another important feature of PRBS is the fact that there do exist complementary primitive polynomials. The generation of the PRBSs has been such that the resulting sequence shifts out the right

end of the illustrated diagrams for a given primitive polynomial. It is also possible for a primitive polynomial to produce the same PRBS, but this time have the feedback entering at the right end while the output is being produced in the reverse order at the left end. Table 1 summarizes the required modulo two forward and reverse feedback connections for shift registers with bit lengths from 4 to 10.

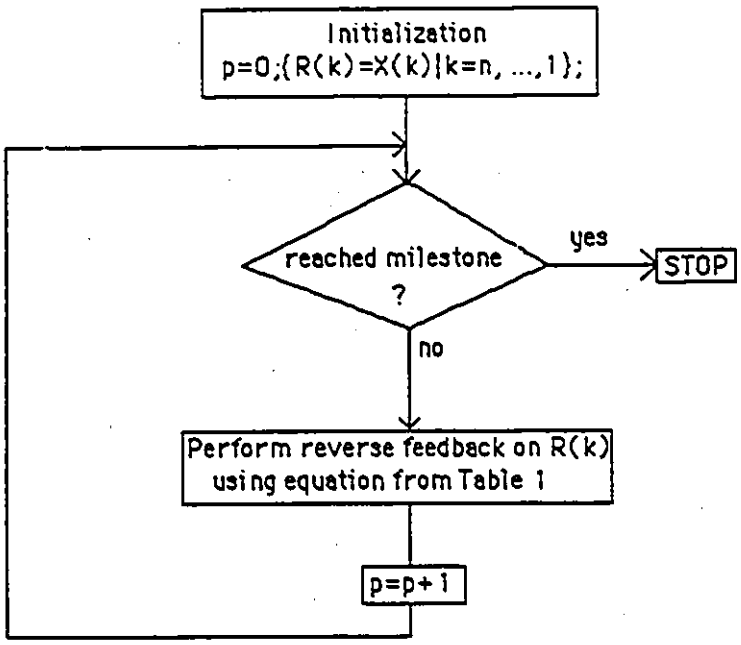
II.4.B) CODE CONVERSION

A translation from the pseudo-random binary code into the natural binary representation is always necessary for practical AGV applications. The code conversion methods that can be employed vary in the extent that translation is a serial and/or parallel process. A strictly parallel translation from the pseudo-random binary code into the natural position can be done using a code conversion table stored in ROM. Obviously this solution is equipment expensive for long PRBSs.

A strictly serial translation exploits the reversibility of the PRBS generating algorithm. This method is based on the idea that it is possible to find the value of the AGV's position by simply counting the number of reverse feedback shifts that it takes for the given pseudo-random code to arrive back into the initial code for the "zero position". In this case the solution is less equipment expensive but more time expensive for long PRBSs.

A better solution to code conversion is a combination of the serial and parallel methods. Consider a pseudo-random encoded track where certain positions (uniformly distributed with a period of t) are employed as milestones. The code conversion for a general position, $p = m \cdot t + r$, where $m \cdot t$ is the position of the nearest down the track milestone Q(m). All intermediate states of this serial shift-back operation are checked in parallel against all possible milestone pseudo-random patterns. Thus with this method the code conversion of the relative position r distance is found serially while the

milestone code conversion is done in parallel. This code conversion algorithm is given in Flowchart II.1.



Flowchart II.1

II.5) GENERAL DESCRIPTION OF THE PROPOSED SYSTEM FOR AGV VISUAL NAVIGATION

As indicated in Fig II.5, the hardware setup of the system consists of a TV camera connected to a microcomputer which has an image acquisition board as well as a code conversion board plugged in. The video signal coming from the camera is loaded to the frame board. Every time a snapshot is taken the image is stored in the frame buffer of the frame board. The computer, which has access to this image, performs a number of filtering as well as image analysis operations as described below. The result is a list of *binary symbols*

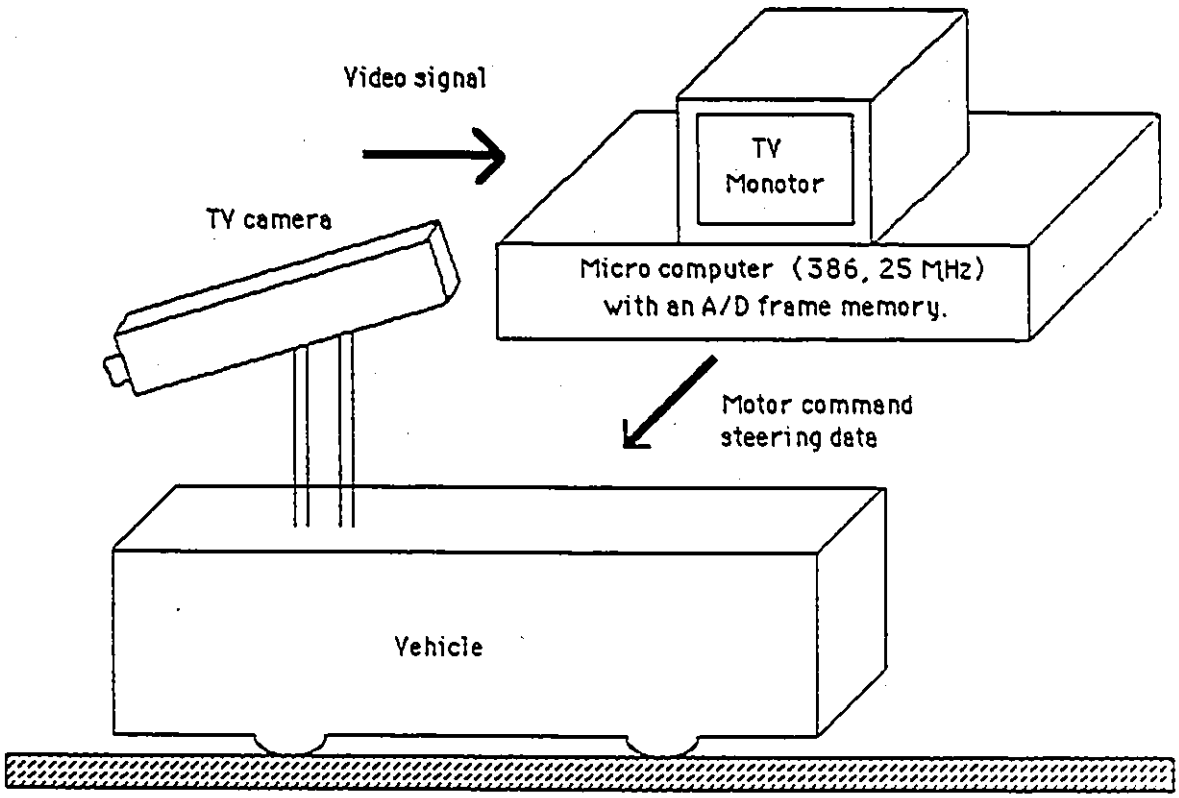


Fig II.5 Setup for the proposed system for AGV visual navigation.

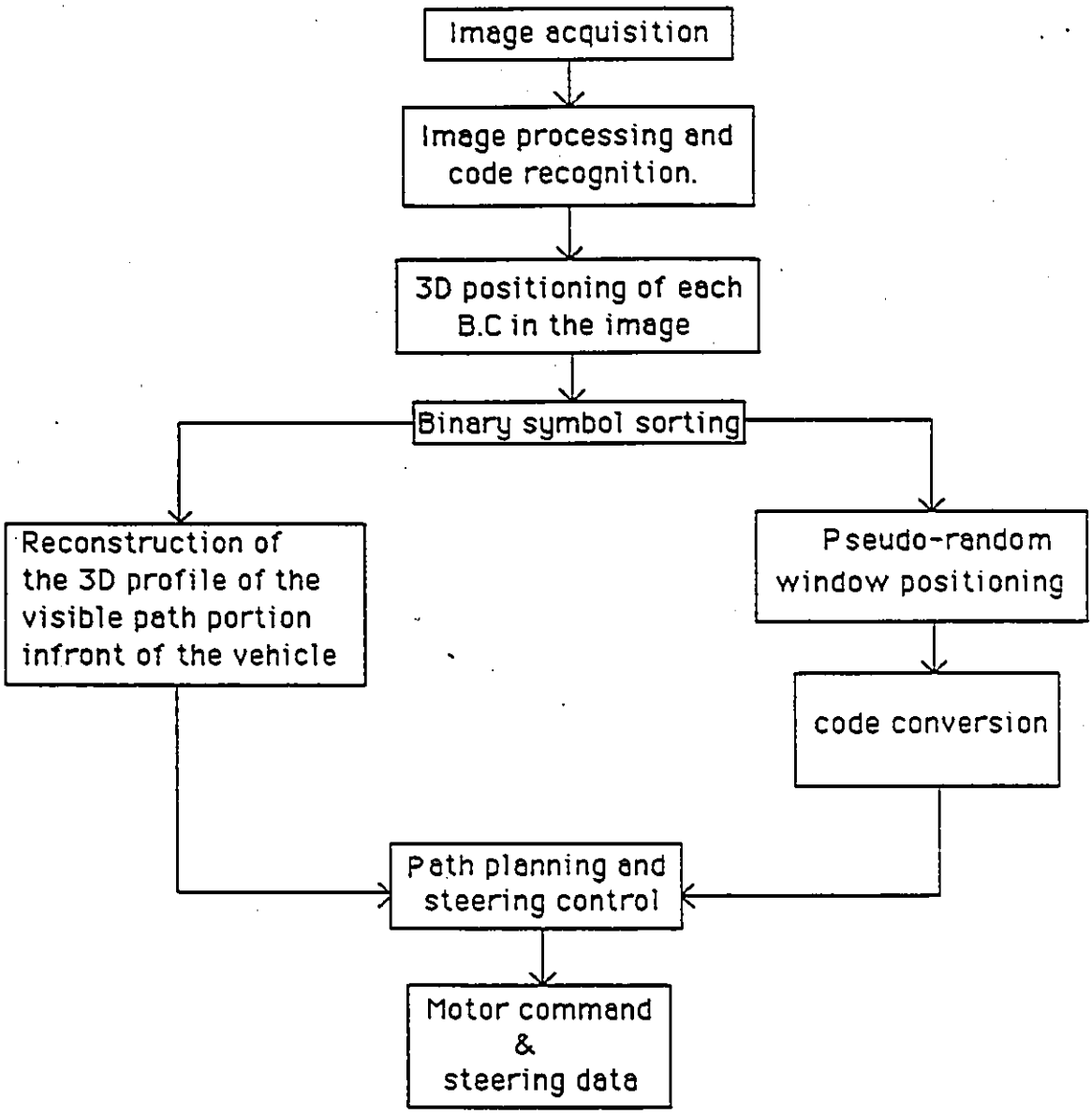


Fig II.6. General organization of the system software.

sorted in the order they follow each other on the guide-path. A sequence of n concatenated binary symbols (n order of the PRBS) on the visible portion of the guide-path is extracted and the corresponding sequence of n binary numbers (pseudo-random window) is supplied to the code conversion procedure. This procedure returns the absolute position of the AGV along the guide-path. The latter information together with the reconstructed 3D profile of the guide-path in front of the vehicle are provided to the steering controller which is responsible for AGV navigation.

The microcomputer used was 386 personal computer which operates at a speed of 25 MHz (megahertz) and has a 4 MB (megabytes) memory. The frame board is a "Matrox" video frame grabber-digitizer that has a resolution of 512x512 pixels with eight bits per pixel. Pixels can be individually addressed by the computer using address registers or they can be addressed sequentially using an auto-increment register. The address as well as the auto-increment registers are accessed using the C language interfacing commands. The code conversion board has been described in section II.4.2.

As regards the software part (which was of our concern), it involves three steps (see Fig II.6): binary symbol recognition, 3D positioning of the recognized symbols and finally sorting of the resulting list of binary symbols in the order they follow each other on the guide-path.

Binary symbol recognition consists of contour tracing each binary symbol in the image (see III.3.2) and approximating the contour by polygon (see section III.3.3) in order to locate corners in the symbol. The symbol shape is recognized by comparing the lengths of the resulting polygon segments and checking certain hypotheses (see section III.4).

3D binary symbol positioning is necessary for binary window positioning and for tracing the 3D profile of the track in front of the vehicle.

After the codes have been recognized and their 3D position determined it is necessary to sort them in the order they follow each other on the guide path. The method is based on finding the neighboring binary symbols in the image. Once these are determined it becomes simple to sort the list of the binary codes in the order they follow each other on the track.

II.6 CONCLUSION

General considerations about AGV visual absolute position recovery have been undertaken in this chapter. The use of monocular vision for automatic guidance has been discussed and the different techniques employed have been reviewed. The problem of absolute position measurement has been discussed and the proposed visual system using pseudo-random encoding for this purpose has been introduced. The mathematical properties of PRBSs which are of particular interest for the case of our application have been discussed. In the following chapters a detailed description of the image processing software is carried out. Chapter III will deal with binary symbol recognition and chapter IV will be concerned with 3D positioning of the binary symbols.

Chapter III

BINARY SYMBOL VISUAL RECOGNITION

III.1. INTRODUCTION

This chapter deals with the image processing and pattern analysis techniques employed to recognize each of the two symbols that represent the binary values "1" and "0".

Using a simple shape analysis method appears to be sufficient to solve this problem. However, two major difficulties have to be considered when designing the code recognition system. The first is associated with the AGV environmental lighting conditions. The second problem is related to guide-path encoding and code recognition for AGV position recovery.

The first problem is encountered in most AGV visual navigation applications. Because the vehicle is in permanent motion, there may be substantial variations in its environmental illumination. Such situations result in low-contrast images, specular reflections, shadows and extraneous details.

The second problem arises from the need for a high resolution guide-path encoding as required by industrial applications. Having a high encoding resolution leads to a rather long pseudo-random sequence and, therefore, a large pseudo-random window to be read from each image. The need for redundancy imposes even longer pseudo-random code-track portions to be actually covered by the camera. This results with each image containing a large number of binary symbols which are affected by considerable geometric perspective deformations. This might cause the symbols which are further away from the camera to have a smaller size. The symbol recognition procedure must deal with such variations in the size of

the symbol. Note that as the symbols get smaller, the quantification noise increases and symbol recognition becomes more difficult.

For the above reasons it is necessary to use robust techniques for symbol recognition. The symbol recognition procedure employed consists of:

1) the generation, for each object in the image, of a set of primitive features, providing a significant and reliable object description; and,

2) actual binary symbol recognition by a tree search.

The remainder of this chapter is structured as follows: section II states the different steps of processing involved in shape analysis programs and describes the order in which they are executed. Section III includes the data processing necessary for primitive generation; and, section IV deals with the recognition problem using tree search.

III.2 OBJECT SHAPE ANALYSIS - GENERAL PROBLEMS

Object shape analysis involves two major feature based steps: shape description and shape recognition.

From the grey level picture we need to generate a convenient description of the object shape that permits simple and robust shape recognition. This operation is referred as *primitive feature generation.*

Three different approaches are known [Pav80a] for shape description:

i) *scalar transform* techniques;

ii) *global* or *internal* techniques; and

iii) *local* or *external* techniques.

Scalar transform techniques are based on the evaluation of scalar measurements (moments) from the brightness function of the grey level image. However, as Pavlidis T. mentions, "it is difficult to relate the higher order moments to shape and it is not a popular technique any more" ([Pav80b] pp 308.).

Global, or *internal*, techniques examine both the interior and the boundary of the object. The skeletal method (also called the thinning method) is among the most popular of the global techniques. It is based on the thinning of the object using a repetitive erosion until the skeleton is obtained. The computational effort, however, is fairly complex [Pav80a] and small amounts of noise can severely alter the form of the skeleton [Pav82a]. Moreover, the execution time is proportional to the object area and in some cases even to its square [Pav82b].

The *local*, or *external*, techniques deal with the object boundary represented either in the frequency or in the space domain.

The frequency domain boundary representation consists of computing the Fourier transformation of the boundary. The boundary can be expressed as the complex function x(t) + j y(t) where x(t) and y(t) are the coordinates of the boundary point and t is the length parameter. The Fourier transform (FT) of the contour will be the complex function X(s) + j Y(s) where X and Y are respectively the Fourier transforms of x and y. Strokes on the contour, if repeated, produce a high component in the frequency description of which the magnitude is proportional to the number of strokes. Therefore, the coefficients of the FT can be used for shape description. Nevertheless, as mentioned by Pavlidis, "their major disadvantage is that of all

transform techniques, the difficulty is to describe local information." ([Pav80b] pp 302).

When using the space domain representation two situations may occur where:

- the boundary is a collection of straight segments (polygon type shape);

- the boundary has a curved shape.

In the first case, the location of corners is sufficient to characterize the shape, while in the second case supplementary information about each boundary segment curvature is needed. Obviously, the first type of shapes are easier to deal with. This is the reason we chose polygonal type symbols, 'E' and 'H', to encode the binary elements, '0' and '1', of the pseudo-random sequence.

Several corner extraction methods are known. The most popular among them is polygon approximation. Polygons offer simple and compact boundary representations (segments and vertices), they are easy to implement and have good noise immunity. For this reason we have decided to use polygon approximation to describe symbol boundary.

The recognition of the binary symbols 'E' and 'H' will consist of a simple tree search checking certain features. Section III.4 details this step.

It is important to specify the order in which objects in the image are treated. The image is scanned left to right and top to bottom. Every time a new object (a binary symbol or any other object) is encountered the shape description routine is executed. A tree search is made following this to recognize the object shape and finally the 3D position and orientation are recovered only if the object in question is recognized to be a binary symbol (see Chap IV).

The program continues scanning the image and the same operations are repeated, in each case when new objects are located, until the bottom right pixel is reached.

III.3. BINARY SYMBOL SHAPE DESCRIPTION

The feature based shape description of the two binary symbols ('H' and 'E') consists of the following steps:

a) thresholding the original image by using the grey level histogram technique, [Wes78];
b) boundary detection by contour tracing, [Pav82a];
c) two-phase polygonal boundary approximation using a fast scan-along method [Wal84], followed by the split and merge algorithm, [Pav82b],[Pav74],[Pav73]; and,
d) corner-vertex extraction using syntactic stroke analysis, [Pav82b],[Fu87].

We will now proceed to a description of each of these steps.

III.3.1 IMAGE SEGMENTATION BY THRESHOLDING

Thresholding is the main industrial image processing technique used to separate the objects from their background.

As mentioned above, one of the practical problems encountered is related to the changes in lighting conditions. It is generally possible to compensate for effects such as shadows and "hot-spot" reflectances using one of the following methods:

- image enhancement [Gon77] (pp 115-181) followed by a global thresholding, [Wes78]; or,

- local thresholding (adaptive thresholding), [Fu87] (pp 374-384).

The major disadvantage of both methods is their long processing time that prohibits their use when only modest hardware (PC type computer) with a simple image acquisition board are used. For these reasons we were forced to use global thresholding [Wes78] for image segmentation. Such segmentation is ostensibly inaccurate. However, the subsequent processing algorithms are robust enough to compensate for this inaccuracy.

Let us assume the grey level image that will process is defined by the function f(x,y) (where x and y are the pixel coordinates). Thus we create a thresholded image g(x,y) by defining:

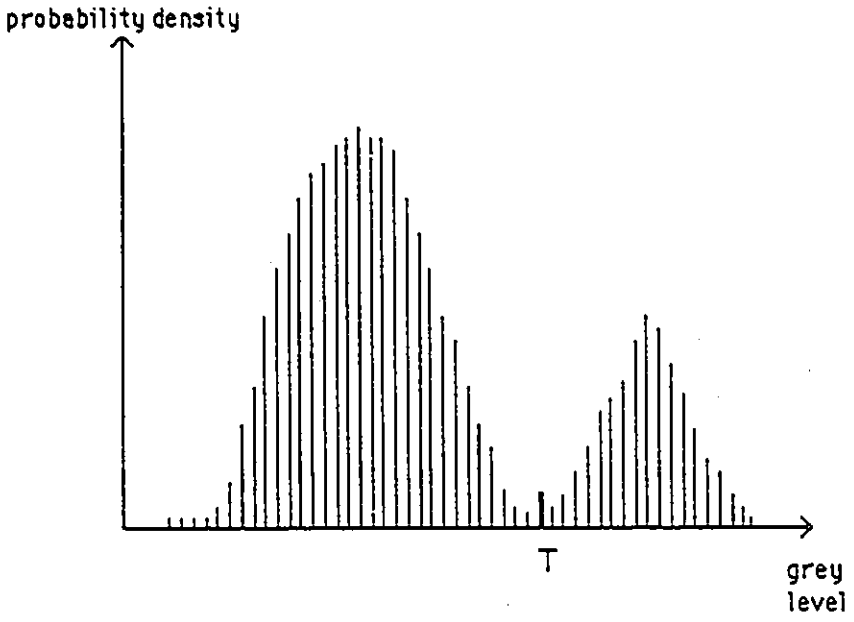


Fig III.1 Grey level hystogram of guide-path image.

$$g(x,y)=\begin{cases} 1 & \text{if } f(x,y) > T \\ 0 & \text{if } f(x,y) < T \end{cases} \qquad \text{(III.1)}.$$

with T the threshold value.

The images we are concerned with are composed of light objects on a dark background. The grey level histogram of such an image has two dominant modes as shown in Fig III.1.

One obvious way to extract objects from their background is to select a threshold, T, which separates the two intensity modes. Therefore, T will be the grey-level value corresponding to the local minimum that separates the two intensity modes on the grey level image histogram.

Obviously, when the lighting is not constant, if the thresholding method metioned above is used, some objects in the image might be deformated. Thus a more robust method is required. However, for the type of computing equipement we have, it is difficult to use such robust algorithms for a real time application.

After thresholding we obtain a binary image consisting of a set of black and white pixels. Each concentration of white pixels, assuming that the background of the image is darker than the objects, corresponds to a region in the scene. At this stage the problem is to separate and locate each region. This problem is termed "region growing". In our case, region growing is accomplished using boundary tracing. In the following section we proceed to a description of the boundary tracing method used.

III.3.2 BOUNDARY DETECTION BY CONTOUR TRACING

The use of contour tracing assumes that every traced boundary encloses a region. Unfortunately, this is only true for external boundaries. Since internal contours in hollow objects enclose an empty area, not all traced contours represent a region. However, this

does not pose a problem in our case since the objects we want to recognize have full interiors (the symbols 'E' and 'H').

The contour tracing routine, which we will use [Pav82a], traces boundaries by ordering successive edge points. For each traversed pixel the coordinates are memorized and the pixel's grey level on the thresholded image is set to 2 (the grey levels of the black and the white pixels are, respectively, 0 and 1). This operation, called contour marking, avoids tracing the same contour twice. The procedure terminates when the current pixel reaches the initial one.

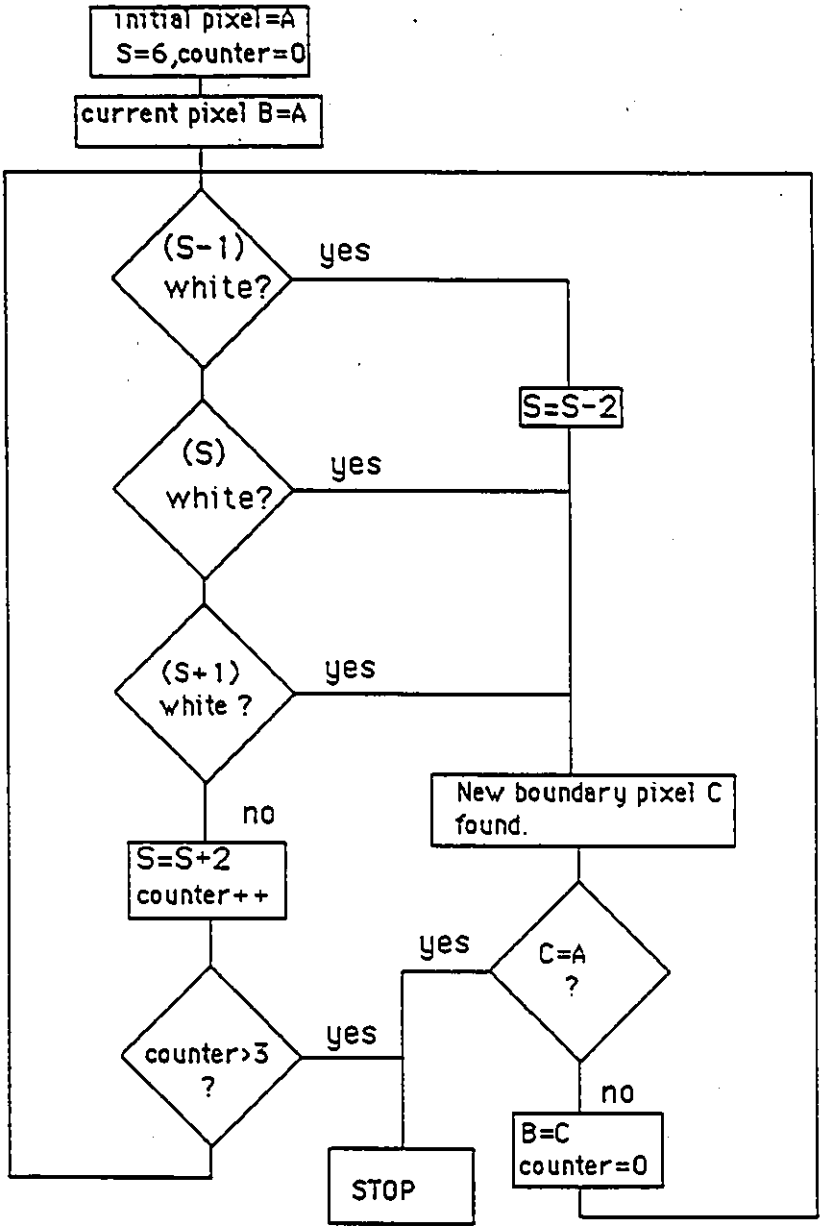
Boundary tracing involves two steps:
1) locating each region in the image; and,
2) boundary tracing each region that has been found.

Locating regions (step 1) is accomplished by scanning the thresholded image from left to right and top to bottom. Every time a new region is encountered, the tracer routine is called (step 2). Such situations, where a new region is encountered, are characterized by the pattern "black pixel followed by a white pixel". The white pixel in this case, as far as the contour tracer routine is concerned, constitutes the initial boundary pixel.

(I-1,J-1) 3	(I,J-1) 2	(I+1,J-1) 1
(I-1,J) 4	(I,J)	(I+1,J) 0
(I-1,J+1) 5	(I,J+1) 6	(i+1,J+1) 7

Fig III.2. Principles of chain-coding.

We now proceed to the description of the actual contour tracing technique utilized. From the initial boundary point, found by the above search strategy, the contour is followed in such a manner that



Flowchart III.1

the region within the boundary is always kept to the left of the path being followed [Pav82a]. The search scheme consists of finding, for the current boundary pixel, the rightmost (with respect to the current boundary orientation) white pixel. This is accomplished through the concept of (Freeman) chain coding (see Fig III.2).

Let S be the search direction in terms of the code of Fig III.2. S, then, indicates which of the neighbors of the preceding boundary cell correspond to the cell currently considered. For the starting boundary point, S is equal to 6. The neighbors of the current cell (I,J) are now checked, as indicated by flowchart III.1. Note that the neighbors are tested in an order of decreasing likelihood. The most likely next boundary neighbor is the one that has the same chain-code value as the current one. This is related to the fact that most of the contours we are dealing with are piecewise linear and, consequently, most of the chain-code values are constant. In this way, the algorithm is performed quickly. The loop is executed three times at the most in order to avoid circling around a set that only has one pixel.

This procedure produces a closed path and follows external contours in a counter-clockwise fashion. Memorization of the boundary points is done as follows: first the coordinates of the initial boundary point are memorized then, for each boundary pixel traced, the chain-code value indicating the direction to the next neighbor on the contour is memorized in a stack. It should be noted that preserving the chain-code value instead of the pixel coordinates saves a great deal of memory space. Also, by knowing the coordinates of the initial edge point and the chain-code values between the different pixels on the boundary, we can recover the coordinates of any boundary pixel whenever necessary.

The complexity analysis shows that for each pixel on the contour, seven tests at most are made. This corresponds to the worst possible case imaginable where we must test the brightness of all 8 neighbors of the current pixel. However, such a situation seldom

occurs. Nevertheless, the complexity of the algorithm is of the order(N), N being the number of points on the contour.

III.3.3 POLYGONAL BOUNDARY APPROXIMATION

III.3.3.1 Introduction

The problem of piecewise linear approximation, for 2D curves, has received considerable attention during the last two decades. Nevertheless, research is still under the way to develop faster and more precise techniques [Dunh86], [Wal84].

Due to the perspective nature of image formation, symbols placed further from the camera will look smaller in the image. In such cases, the noise affecting the symbol shape might be of the same order of magnitude as the small details in the boundary. This makes it difficult to find the corner locations. Figure III.3 illustrates the complexity of the corner positioning task when the shapes appearing in the image are small. In the ideal case of a noiseless contour, corners are located in (A) and (B). In the real case four corners appear on the contour: A1, A2, B1 and B2.

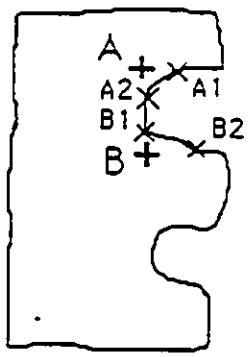


Fig III.3 Corner location problem in polygon approximation.

In view of this matter, a robust polygonal boundary approximation technique imposes itself. Obviously, the precision of the polygonal approximation and its time computation are two antagonist factors.

Surveys of the available polygonal approximation techniques are given in [Pav82a], [Dun86] and [Wal84]. The existing techniques can be classified according to the type of error norm they use, whether the knots are fixed or variable and whether or not the approximation is continuous [Dun86]. In the following we review the most important polygonal approximation schemes. Three approaches can be distinguished: the scan-along, the minmax and the split and merge approach. Most of the existing polygon approximation algorithms are more or less similar to one the approaches mentioned above.

An example of a *scan-along* method is the one proposed by Williams [Wil78] which uses cone intersection to find the longest allowable approximating segments. For each of these, the maximum Euclidian distance between the segments and the points they approximate is less then a prescribed value. Circles are drawn around each point. An acceptable approximating polygon should pass through each of the circled areas. Moving from point to point, tangents are drawn to the circles to find sectors in which an acceptable segment must lie. Cone intersection methods are relatively fast due to simplicity and sequential processing of points. However, they do not produce optimal results.

Kurozumi [Kur81] proposed a *minmax* method which finds a polygon satisfying the condition that the number of sides are minimum and the maximum distance between the sides and the data points is less than a prespecified tolerance. First the maximum polygon is constructed, that is, a convex polygon containing all points in the current subset. Then a side in the polygon is found and found

vertex giving the maximum perpendicular to it. The side must be the one for which this perpendicular is shortest. The current segment is then a line parallel to the side passing through the midpoint of the perpendicular. If the deviation is greater then the given tolerance the segment is not accepted and the previous segment is used as approximation. *Minmax* methods give optimal results but are rather complex. They process the points only partly in sequence so several points near the current segment are needed in each step.

For a given error norm, the *split and merge* algorithm ([Pav82a], [Dun86] and [Pav74]) finds the minimum number of vertices of which the total error norm is less than a prescribed value. An arbitrary set of vertices on the contour initializes the algorithm. The error norm on each resulting segment is then calculated. These segments are split in order to drive the total error norm under the given limit value. After this, each neighboring segments are merged when merging does not cause the total error norm to exceed the limit value. The final step of the algorithm is vertex adjustment; every vertex position on the contour is adjusted so that the total error norm is minimized.

The *split and merge* polygonal approximation, introduced by T. Pavlidis, has been widely used for shape description. As far as the optimality of the number of the resulting polygon vertices, after processing, and their location are concerned, the split and merge method is one of the most powerful among the available techniques. Based on these considerations, we opted to use this technique for the polygonal boundary approximation of the employed symbols 'E' and 'H'.

It is shown that, as far as the number and the position of the vertices are concerned, the split and merge algorithm, in cases of rare configurations, is suboptimal [Pav74]. However, in the case of the 'E' and 'H' symbols, the algorithm leads to very satisfactory results.

The *known* implementations of the split and merge algorithm are time consuming. This is due to the following reasons:

- if the initial partition is chosen arbitrarily (which is most of the time - considerably far from the optimal solution) the number of splits, of merges and vertex adjustments operated will be large.

- the error norms used by Pavlidis, either the maximum Euclidian distance or the integral square error, require rather extensive computation time.

In order to correct the first drawback, a rough, but very fast, *scan-along* polygonal approximation [Wal84] precedes the split and merge algorithm. Vertices resulting from this scan-along procedure serve as initial partition for the split and merge algorithm.

The second drawback is addressed by using area deviation per length unit as an error norm. This norm originally used for the scan along technique [Wal84], mentioned above, allows the split and merge algorithm to have a better execution time .

Thus the polygon approximation we will use consists of the following steps:

- fast scan-along sequential polygon approximation using area deviation [Wal84]; and,
- a split and merge algorithm using area deviation per length unit as an error norm and the output of step 1) as an initial partition.

The reminder of this section is structured as follows: section III.3.3.2 discusses the concept of area deviation; section III.3.3.3 describes the fast scan-along algorithm [Wal84]; and section III.3.3.4 deals with the split and merge algorithm.

III.3.3.2 The area deviation concept

The concept of area deviation has been introduced by Wall and Danielsson [Wal84]. The area deviation of a contour between two boundary points A and B is defined as the surface that separates the segment [A,B] from the contour portion between A and B (see fig III.4).

An interesting property of area deviation is the simplicity of the computation of its variation when one of the segment endpoints is shifted to its boundary neighbor.

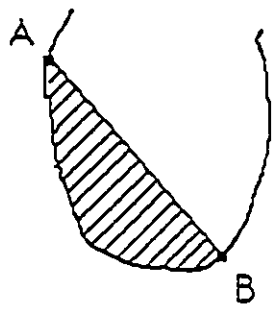


Fig III.4. Area deviation concept.

Let A_i (respectively A_{i+1}) be the area deviation between the points P_0 and P_i (respectively P_0 and P_{i+1}). P_i and P_{i+1} being two boundary neighbors as mentioned on Fig III.5. We want to calculate A_{i+1} as a function of A_i. Let us assume the following additive model:

$$A_{i+1} = A_i + dA_i \quad (III.2)$$

It is easy to see in figure III.5 that the area dA_i corresponds to that of the triangle of vertices P_0, P_i and P_{i+1}. From Fig III.6 we have:

$$dA_i = \frac{1}{2} (P_0 P_{i+1}). (P_i H)$$

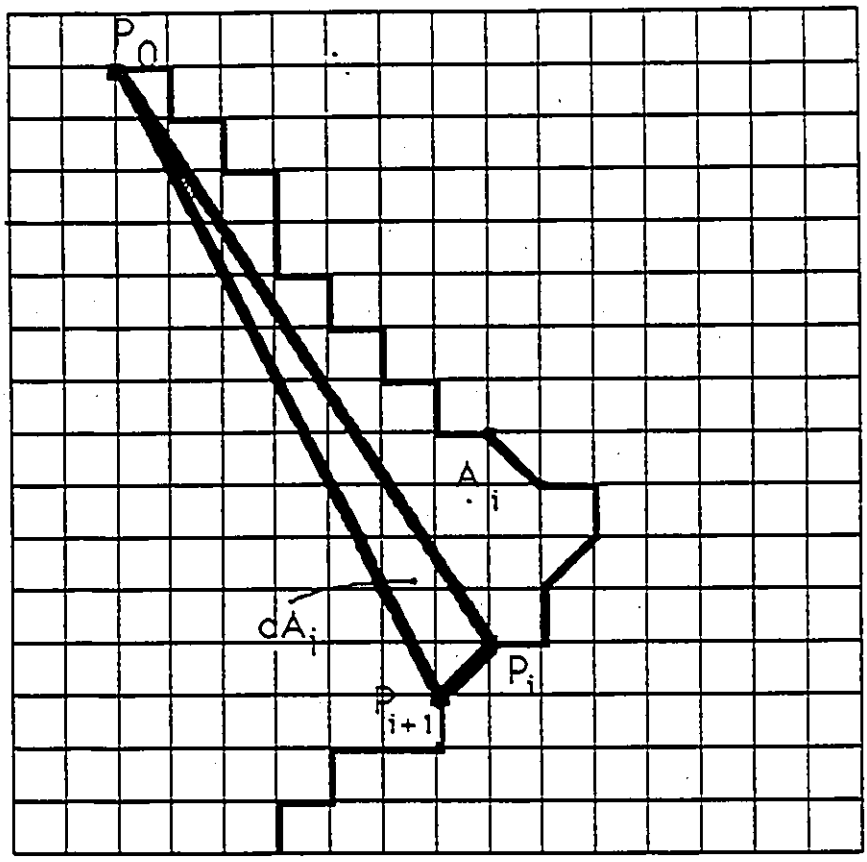


Fig III.5 Area deviation variation between two concatenated boundary pixels.

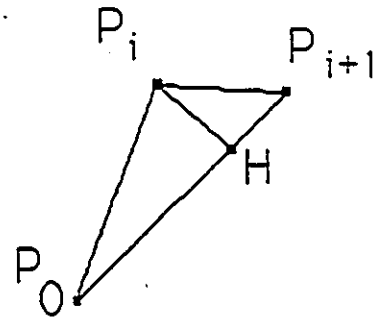


Fig III.6 Computation of area deviation vazriation.

$$= \frac{1}{2} \| \overrightarrow{P_0 P_{i+1}} \times \overrightarrow{P_i P_{i+1}} \|$$

$$= \frac{1}{2} \left((y_{i+1}-y_i) . x_{i+1} - (x_{i+1} - x_i) . y_{i+1} \right)$$

$$= \frac{1}{2} \left((y_{i+1}-y_i) . x_i - (x_{i+1} - x_i) . y_i \right)$$

where (X) is the vector product operator and (x_i,y_i) and (x_{i+1},y_{i+1}) are respectively the coordinates of $\overrightarrow{P_0P_i}$ and $\overrightarrow{P_0P_{i+1}}$. This leads to equation III.3:

$$dA_i = \frac{1}{2} . (dy_i.x_i - dx_i .y_i) \qquad \text{(III.3).}$$

where

$$dx_i = x_{i+1}-x_i \qquad \text{(III.4.a).}$$
$$dy_i = y_{i+1}-y_i \qquad \text{(III.4.b).}$$

S_i	dx_i	dy_i	dA_i
0	1	0	$-y_i$
1	1	-1	$-x_i -y_i$
2	0	-1	$-x_i$
3	-1	-1	$-x_i + y_i$
4	-1	0	y_i
5	-1	1	$x_i + y_i$
6	0	1	x_i
7	1	1	$x_i - y_i$

table III.1

Since P_i and P_{i+1} are concatenated pixels on the boundary, dx_i and dy_i can only take the values 0,-1 and 1. For this reason, as indicated in table III.1, the area deviation variation dA_i can be quickly computed using only additions and subtraction of integer numbers (x_i and y_i). In table III.1 the area deviation dA_i is mapped directly as a function of all chain code values. Note that the chain code values are automatically generated by the contour tracing module.

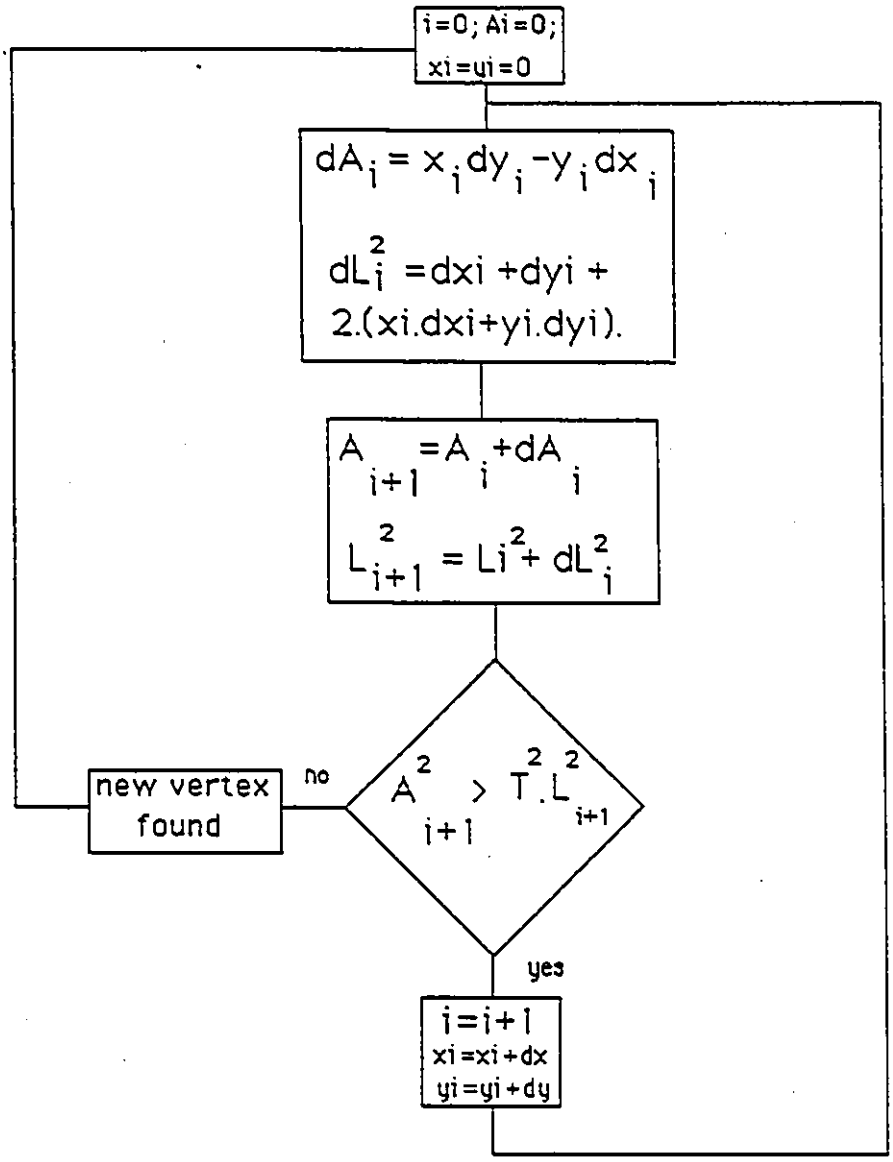
The area deviation between two boundary points, P_m and P_n, with m and n the indices of each pixel along the connected contour, can be easily computed using the following additive model:

$$A_{m,n} = \sum_{i=m}^{n} dA_i \qquad \text{(III.5).}$$

It results that the computation of $A_{m,n}$ comes down to the addition and subtraction of integers. This explains the rapidity of the sequential algorithm for polygonal approximation presented in III.3.3.3 and the vertex adjustment algorithm presented in III.3.3.4.c.

III.3.3.3 Scan-along polygon approximation using area deviation

This technique is very fast but, being sequential, it does not produce optimal approximations. It uses a scan-along technique where the approximation depends on the area deviation for each line segment. The algorithm (Flowchart III.2) finds the longest allowable approximating segment by stepping from boundary point to another until the area deviation per length unit exceeds a prespecified value. The algorithm stops when the last pixel on the boundary is reached.



Flowchart III.2.

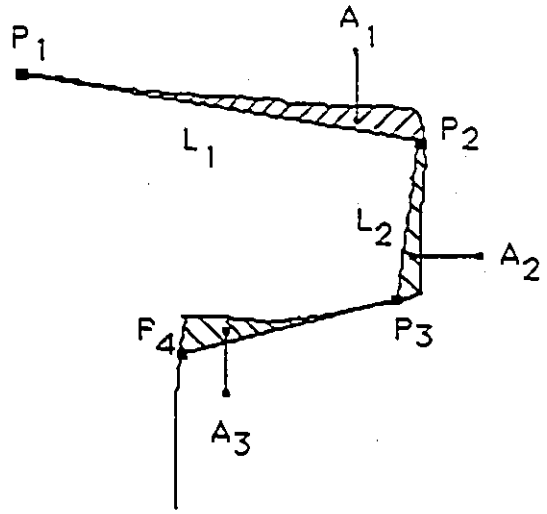


Fig III.7. Problem of corner location by scan along polygon approximation with small error norm.

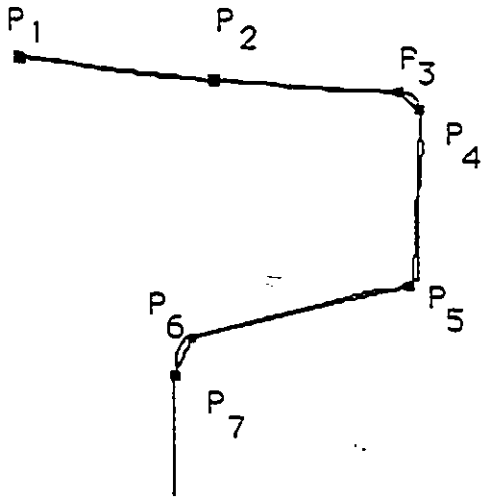


Fig III.8. Problem of corner location by scan along polygon approximation with large error norm.

Like area deviation, it is shown in Appendix I that segment length L_i can be calculated recursively; in the same way we have proceeded with area deviation, a look up table for the segment length variation - when one endpoint of the approximating segment is shifted to its neighbor - as a function of the chain-code values can be drawn.

The sequential polygon approximation conformable to the flowchart in Flowchart III.2 is very fast (an average of 30 milliseconds of execution time per contour). However, as far as the number of vertices and their location are concerned, the approximations of the method are not optimal. Figure III.7 shows a situation where the vertices are not located exactly at the corners. This problem is related to the fact that the algorithm is sequential (scan-along). The program does not label the current pixel a vertex as long as the ratio between the area deviation and the segment length has not exceed the prescribed value. This criterion is generally not satisfied at the corner but a few pixels away from it. The larger the threshold value the further the generated vertices are from the real corners.

A solution to this problem would be to choose a small threshold value. However, because of noise, fictive corners may be detected by the program in that case. Figure III.8 shows that too many vertices may be generated.

Therefore, increasing the threshold value minimizes the number of vertices but worsens their positions. Conversely, decreasing the threshold value improves the vertex position but increases the number of vertices. Thus the algorithm can optimize either the number of vertices or their positions but not both at the same time. For this reason, we can't content our selves with only this scan-along polygon approximation. The split and merge algorithm which we use to refine the partition generated by the scan-along method is presented in the following section.

III.3.3.4 The split and merge polygon approximation

This section deals with different problems related to the split and merge algorithm. In section III.3.3.4.a we describe the split and merge mechanism and discuss the optimality of its approximations. Section III.3.3.4.b studies the problem of the validity of area deviation per length unit as an approximation error measure. In section III.3.3.4.c the vertex adjustment algorithm is presented. A topological analysis made in this section permits to understand the functioning of the vertex adjustment process when area deviation per length unit is used as an error norm.

III.3.3.4.a) The split and merge paradigm

This section will describe the split and merge algorithm independently of the type of error norm used. We will discuss how, by split and merge, we can optimize the number as well as the position of the generated vertices.

Unlike the scan-along methods, which are sequential, the split and merge polygon approximation is parallel: at any moment during processing, the whole boundary is considered as a single entity. A set of break points divide the contour into intervals. Each interval S_i is approximated by the linear segment joining its endpoints. To each interval S_i corresponds an error norm, called segment error norm, which measures the error made when approximating S_i by a linear segment. The split and merge algorithm minimizes the error norm over all segments and makes sure it is less than an upper boundary. The total error norm is defined as follows:

$$E = \sum_{i=1}^{n} E_i \quad \text{(III.6)}$$

where Ei denotes the segment error norm along the i-th polygon segment (the different computational schemes for Ei are detailed in section III.3.3.4.b). Starting from an initial partition, the algorithm finds the vertex configuration that has the minimum number of vertices such that the total error norm E is less than an a priori upper limit value E_{max}. As mentioned above, the initial partition, in our case, consists of the vertices generated by the scan-along algorithm described in section III.3.3.3. Generally the initial partition has an E greater than E_{max}. The execution of the split and merge algorithm has two stages:

i) when the total error norm E is greater than Emax; and,
ii) when the total error norm E is smaller than Emax.

In the first stage only the split routine is active (see flowchart III.3); no merge is made as long as the error norm is bigger than Emax. During the split operation, the segment with the maximum error norm, S_i, is split into two equal segments, S_{i1} and S_{i2} (see Fig III.9). The error norms on each one of the new segments, E_{i1} and E_{i2}, are computed. Obviously, after each split, the sum of the error norms on the resulting segments, $E_{i1}+E_{i2}$, is smaller than the error norm on the original segment, E_i:

$$E_i > E_{i1} + E_{i2} \qquad \text{(III.7)}.$$

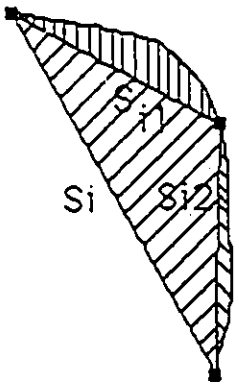
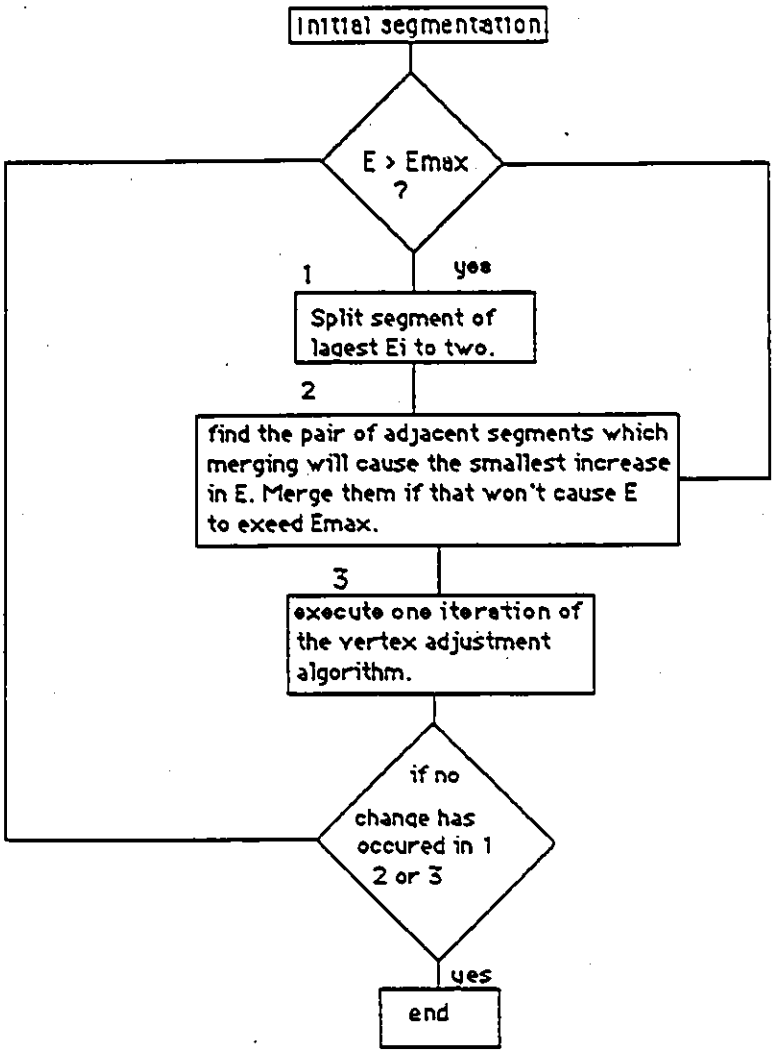


Fig III.9 Merge operation increases segment error norm.



Flowchart III.3

Therefore, considering equations III.6 and III.7, it easy to see that the split operation decreases the total error norm E.

As indicated on flowchart III.3, as soon as E becomes smaller than Emax the merge operation takes presence over the split operation. Algorithm III.1 describes the merge routine. For each polygon segment S_i, i=1,....,*n* (*n* is the total number of segments on the contour), the error norm F_i which would have occurred if S_i and S_{i+1} were merged is evaluated and:

$$G_i=E+F_i-(E_i+E_{i+1}) \quad \text{(III.8)}$$

is computed. Gi represents the value the total error norm would have taken if S_i and S_{i+1} were merged. After this, the segment list $\{S_i\}_1^n$ is sorted in the decreasing order of Gi. The segment S_i having *the smallest Gi* is merged with its neighbor S_{i+1} *provided that* G_i *is smaller than* E_{max}. Every time a new merge is made the Gi's of all segments are updated (see algorithm III.1). After the merge operation one iteration of the vertex adjustment algorithm is executed.

Since at most all segments can be merged into one, there comes a time when the merge part of the algorithm becomes inactive. From then on, only the vertex adjustment part is active.

ALGORITHM III.1

Let the polygon segments be numerated from 1 to n.

1) for i=1 to i=n-1
{
- evaluate F_i, the error norm which would have occurred if segments i and i+1 were merged.
-compute : $G_i=E-(E_i+E_{i+1})+F_i$.
}

2) Sort the segments' list in the decreasing order of Gi.

3) while(G_j < Emax)
{
-merge segments j and j+1.
-update the elements in the rest of the segment list as follows:
$G_i = G_i + F_j - (E_j + E_{j+1})$.
-consider the next segment in the list.
}

4) end

Note that by the very nature of the merge routine, any decrease in the number of segments will cause the total error norm to exceed E_{max}. Hence the final segmentation has a (locally) minimum number.

III.3.3.4.b The use of the area deviation per length as an error norm

So far, we have studied area deviation only from the computational point of view. It was shown that it can be computed recursively using a simple arithmetic operation. The problem which will be discussed in this section is related to the use of area deviation per length unit (ADPLU) as an error norm for the split and merge algorithm. It is necessary, at this point, to show that ADPLU constitutes a valid error measure for piece-wise polygonal approximation. In other words, we want to estimate, for a given value of the ratio area deviation per segment length, how far the boundary points can be from the approximating segment.

In the following we start with a review of the different error norms used in the split and merge algorithm and then proceed to a

geometrical analysis that proves the validity of ADPLU as an error norm.

In [Pav74], the pointwise error between a boundary point and the approximating segment is defined as:

$$e_i = \sin(\beta).x_i + \cos(\beta)\ y_i - d \qquad \text{(III.9)}$$

Where :

$$\sin(\beta).x + \cos(\beta)\ y - d = 0 \qquad \text{(III.10)}$$

is the equation of the segment approximating the contour portion (S_k) and x_i and y_i are the coordinates of the considered boundary point.

The two error norms proposed in [Pav74] for the split and merge algorithm are:

i) the integral square error E_2 defined as:

$$E_2 = \sum e_i^2 \qquad \text{(III.11)}$$

for (x_i,y_i) element of a point set S_k.

ii) the maximum pointwise error (or maxim Euclidian distance) defined as:

$$E_\infty = \max (e_i) \qquad \text{(III.12)}$$

for (x_i,y_i) element of a point set S_k.

The maximum Euclidian distance error norm (E_∞) results in approximations which present a closer fit as judged visually than the integral square error norm (E_2). This raises the question of how to

derive an estimation of the split and merge maximum total error norm E_{max}, when the E_2 error norm is used, from an estimation of the maximum pointwise error norm (e_{max}).

Fortunately, the E_2 measure is bounded by the E_∞ measure:

$$E_2=\sum e_i^2 <= n_k \cdot \max (e_i^2) \qquad \text{(III.13)}$$

where n_k is the number of points in the interval S_k of the boundary. It is known from the theory of approximation that the sign of the error (e_i) *changes* at least *as many times* as *the number of degrees of freedom* plus one and that values *close to the maximum* are achieved at least as many times, [Pav82b]. In practice, it is assumed that the number of degrees of freedom is equal to one and therefore, assuming that (L) is the interval length, E_{max} is evaluated as follows:

$$E_{max} = \frac{e_{max}^2 \cdot L}{3} \qquad \text{(III.14)}$$

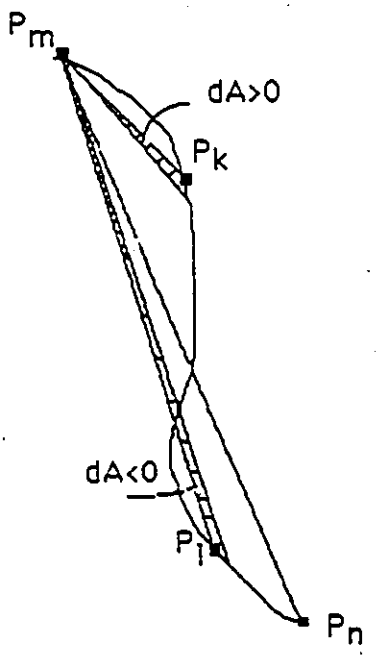


Fig III.10 The sign of A.D variation depends on the contour concavity.

In the case area deviation per length unit is used, the segment error norm along one point set S_k is:

$$E_a = \frac{|A_k|}{L_k} \qquad \text{(III.15)}.$$

where A_k is the area deviation along S_k and L_k is the length of the linear segment approximating S_k.

Figure III.10 shows one specific problem to area deviation: the sign of the area deviation variation dA depends on the boundary concavity at the point where this variation is being measured. For instance, dA at P_k, in Fig III.10, is positive while it is negative at P_l. Therefore since the total area deviation is the sum of all dA's along the boundary portion to be approximated by a segment (see equation III.5), the result may be small in absolute value and so may be the ADPLU.

This raises the following question: *in general, how large can the pointwise error be for a given value of the ratio area deviation-segment length?* In the sequel an analysis, similar to the one made for the E2 error norm, is carried out in order to relate the ADPLU to the pointwise error norm .

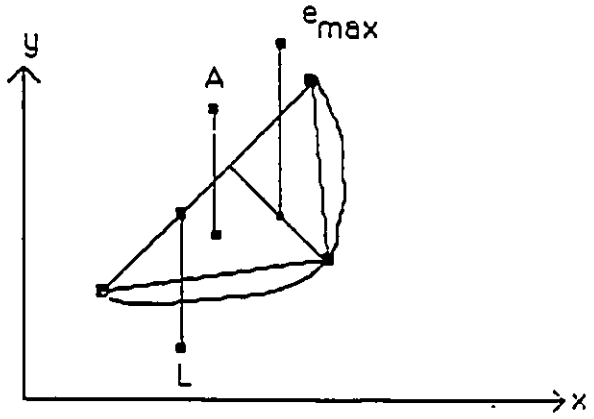


Fig III.11 Area deviation in the case of a convex curve.

In the case area deviation per length unit is used, the segment error norm along one point set S_k is:

$$E_a = \frac{|A_k|}{L_k} \qquad \text{(III.15)}.$$

where A_k is the area deviation along S_k and L_k is the length of the linear segment approximating S_k.

Figure III.10 shows one specific problem to area deviation: the sign of the area deviation variation dA depends on the boundary concavity at the point where this variation is being measured. For instance, dA at P_k, in Fig III.10, is positive while it is negative at P_l. Therefore since the total area deviation is the sum of all dA's along the boundary portion to be approximated by a segment (see equation III.5), the result may be small in absolute value and so may be the ADPLU.

This raises the following question: *in general, how large can the pointwise error be for a given value of the ratio area deviation-segment length?* In the sequel an analysis, similar to the one made for the E2 error norm, is carried out in order to relate the ADPLU to the pointwise error norm .

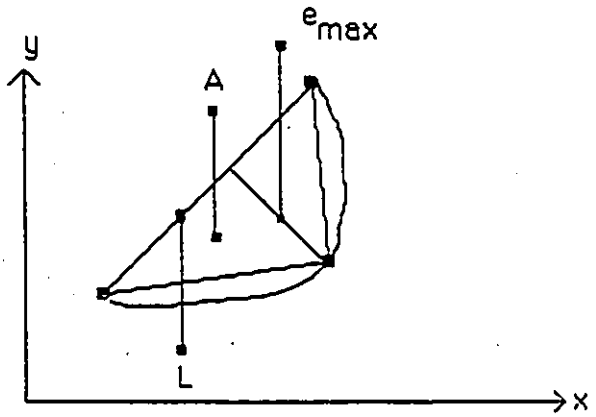


Fig III.11 Area deviation in the case of a convex curve.

By summing both sides of equations III.19 and III.20 we have:

$$S2+S3 < E_{max}\,(LB+LC)$$

thus

$$\frac{LC \cdot e_{max}}{2} < E_{max} \cdot (LB+LC)$$

$$e_{max} < 2\,E_{max}\,(1 + LB/LC) < 4.E_{max} \qquad (III.21).$$

This is valid if LB is smaller than LC which is only the case of smooth curves. In the case of rough curves, this assumption is no longer valid and e_{max} can take bigger values. Furthermore, in the case where the number of boundary concavity changes is more than one, we expect e_{max} to take larger values. Nevertheless, for the type of shapes we are dealing with ('E' and 'H'), we can make the assumption that the contour is smooth and that the pointwise error does not change very often.

Hence, from the analysis made above, we can conclude that *when the ratio area deviation per segment length along a segment* Sk *of the contour is bounded the maximum pointwise error along this segment is also bounded and conversely* (the converse proposition is evident).

This proves the validity of area deviation per length unit as an error norm. Moreover, from equations III.19 and III.22 we can derive an estimation of E_{max} from an estimation of the pointwise maximum error e_{max} :

$$E_{max} = \frac{e_{max}}{4} \qquad (III.22)$$

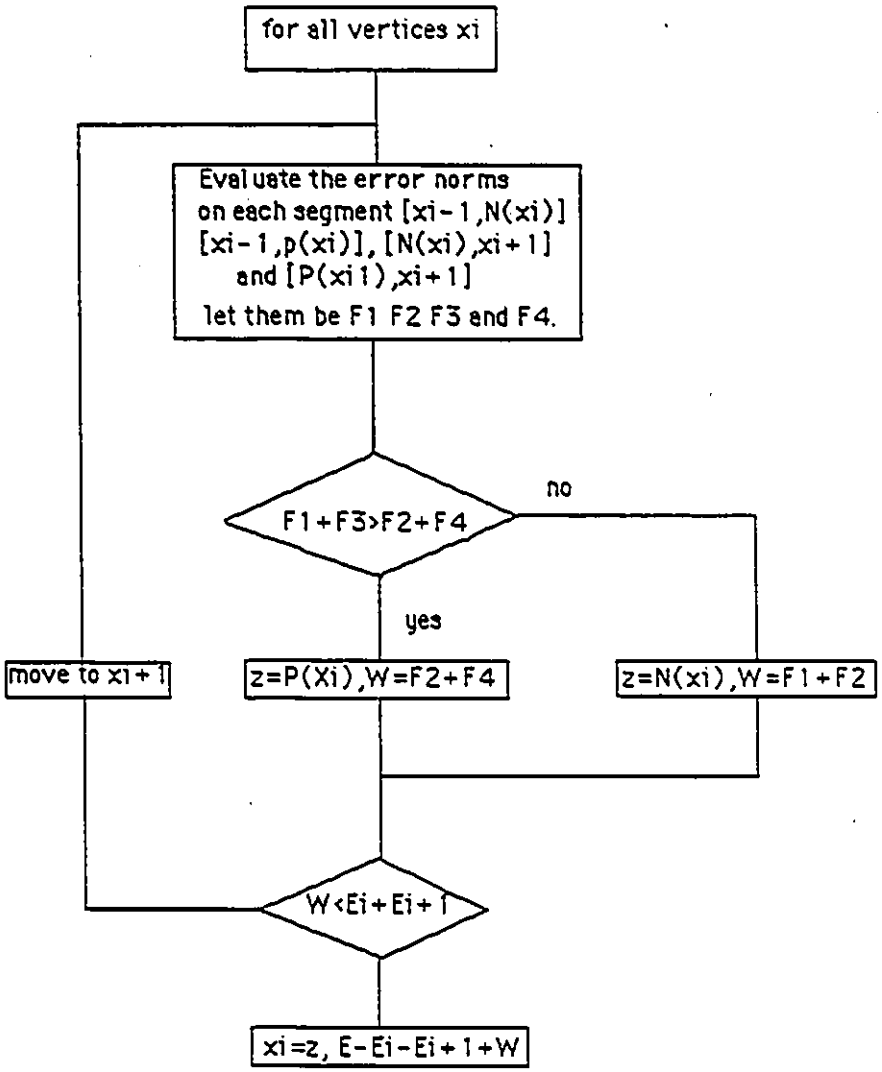
III.3.3.4.c Vertex adjustment

Vertex adjustment, also called *piecewise approximation with variable joints*, has received considerable attention in the literature ([Pav82b] pp 31-45). The vertex adjustment routine constitutes the third step of the split and merge algorithm. The two first steps, the split and the merge routines guarantee a minimum number of vertices to the polygon approximation. The role of vertex adjustment is to optimize the vertex positions so that they are as close as possible to the real corners.

The problem of vertex adjustment can be formulated as follows: *given a contour (C) and an integer number n, find the optimal position of each one of n points* $\{X_i\}_1^n$ *along (C) such that the n segment polygon joining these points approximates the contour (C) with the minimum error.*

Depending on the type of error norm employed, different techniques can be used for vertex adjustment. When the maximum pointwise error norm (E_∞) is employed, vertex adjustment comes down to a mathematical programming problem involving the minimization of a linear function which is subject to both linear and quadratic constraints. Dynamic programming methods have been proposed to solve this case [Dun86]. On the other hand vertex adjustment, in the case of the integral square error (E_2), is an analytical minimization problem ([Pav82b] pp 35-39). Pavlidis in [Pav77] uses the Newton-Raphson to solve this problem [Pav77].

Both the mathematical programming and the analytical minimization techniques are optimal. However, they are time consuming [Dun86]. To overcome this, Pavlidis proposed a fast vertex adjustment method, the *descent* algorithm [Pav73], that works for both, the E_∞ and the E_2 error norms. The descent algorithm



Flowchart III.4

becomes even faster when the ADPLU error norm is employed (see section III.3.3.2).

The descent algorithm (flowchart III.4) for vertex adjustment finds the optimal position of the n vertices by treating each vertex separately. The algorithm keeps turning around the boundary by stepping from vertex to another. The position of each traversed vertex is adjusted in order to minimize the total error norm. The algorithm keeps cycling around the boundary until no more vertex adjustment can be made.

During the adjustment procedure of a vertex X_i, only the position of X_i can change while all the other (n-1) vertices remain fixed. Therefore only the two segments $[X_{i-1},X_i]$ and $[X_i, X_{i+1}]$ are affected by the vertex adjustment of X_i and hence the only segment error norms which may be modified are E_i and E_{i+1}. Since the total error norm is equal to the sum of the segment error norms:

$$E=\sum E_i = E_1+E_2+.....+(E_i + E_{i+1})+...+E_n .$$

finding the position of X_i which minimizes E is equivalent to finding the position of X_i which minimizes $(E_i + E_{i+1})$. For each vertex X_i, the changes $G_{i1}=F_1+F_3$ and $G_{i2}=F_2 + F_4$ in (E_i+E_{i+1}) that would have occurred if X_i has been moved respectively to its preceding neighboring pixel on the boundary, $P(X_i)$, and its following neighboring pixel, $N(X_i)$, are computed. Obviously, the vertex position giving a negative G_i is opted for.

One iteration of the descent algorithm corresponds to one pass through the whole contour. As mentioned in section III.3.3.4.a, one iteration of the vertex adjustment algorithm is executed at the end of each split and merge cycle. Once the minimum number of vertices has been reached only the vertex adjustment routine remains active. The latter, as mentioned above, stops when no more vertex adjustment can be made.

Note that when the algorithm converges $\frac{\partial(E)}{\partial(X_i)}$ is equal to zero for all Xi's. Therefore the gradient of E is null and Γ reaches its minimum. However, this minimum can be local and the vertex adjustment algorithm may generate suboptimal approximations.

Using the ADPLU error norm speeds up the vertex adjustment algorithm. This is due to the simple and fast computation required by this error norm.

III.D CORNER-VERTEX EXTRACTION

Despite the robustness of the polygon approximation employed it may happen that vertices not corresponding to corners on the real object be generated. This is due to gross segmentation errors that may occur. In order to extract the real corners, more processing of the polygon approximating the boundary is necessary.

Unlike polygon approximation which operates on boundary points, corner-vertex extraction uses the polygon segments approximating the boundary as a description of the object. There are basically two techniques to solve the vertex extraction problem. In the first technique proposed by Davis [Dav77] curvature is computed to decide if a vertex corresponds to a real corner or not. The second method, proposed by Pavlidis [Pav79], is based on a *syntactic* analysis of the polygon segments which permits the extraction of corners. Both techniques have comparable performances [Pav82a]. We will use the syntactic technique.

An object boundary can always be described as a sequence of basic features (*primitives*), such as corners, arcs, protrusions, intrusions, lines etc., which characterizes (as a *signature*) that object shape. Encodings of curves as a sequence of such *primitives* are particularly suited for *syntactic* descriptions. If each entity, also called *primitive* is labeled from a *finite alphabet*, then the curve is

mapped into a string over such an alphabet and the classical theory of *formal languages* [Fu74] is directly applicable.

The number of *primitives* in the alphabet depends on the variety and the complexity of the shapes considered. We will assume in our case that the boundaries of the 'E' and 'H' symbols can be described in terms of the following primitives:

- corners with the formal name COR;
- long linear segments with the formal name LINE; and,
- short segments having no regular shape with the formal name BREAK.

The polygon approximations of the contour results in a sequence of vectors. A LINE will be either a single vector, a set of mutually almost collinear vectors or a set of mutually almost collinear segments separated by BREAKs. A BREAK can be approximated by one or two vectors of very short lengths. A CORNER can be considered as consisting of two lines either in direct sequence or separated by a BREAK. Following these definitions we have the following production rules.

<LINE> := <LINE1> + <LINE2> (angle LINE1,LINE2 < ß)
<LINE> := <LINE> + <BREAK>
<LINE> := <LINE1> + <BREAK> + <LINE2> (angle LINE1,LINE2 < ß)
<CORNER>:= <LINE1> + <LINE2> (angle LINE1,LINE2 > ß)
<CORNER>:= <LINE1> + <BREAK> + <LINE2> (angle LINE1,LINE2 > ß)

Based on these production rules, we can build the *contour parser*, which is also termed *trusion parser*. The parser is a *finite automaton* with a buffer of size four. The automaton always looks at the buffer contents. A decision is made whether a segment is short (S) or long (L) depending not only on the current line but also on the contents of the buffer. Fig III.13 describes the automaton. The operation "Shift k" means obtain the next k lines as the return of a line detector operating on the contour. "Reduce k" means emptying

the first k elements of the buffer. The "relabel" operation consists of relabeling the remaining contents of the buffer by L. The symbol # stands for end of input while ? denotes an empty portion of the buffer.

buffer contents	Action
LL??	reduce 2 & shift 1
LSL?	reduce 3& shift 2
LSSL	reduce 4 and shift 3
LSSS	reduce 2 relable & shift 1
LSS#	reduce 3 and stop
LS##	reduce 2 & stop
L###	stop
####	stop

Fig III.13 Implementation of the trusion parser.

III.4 SHAPE RECOGNITION USING POLYGONS

IV.4.1) INTRODUCTION

As explained in section III.1. the shapes to be recognized are two distinct symbols which represent the two binary numbers '0' and '1' (binary symbols). Other shapes appearing in the picture will have to be recognized as not being binary symbols and will eventually be considered obstacles to be avoided during AGV navigation. Consequently, our data base will contain only two templates each one representing the two binary symbols.

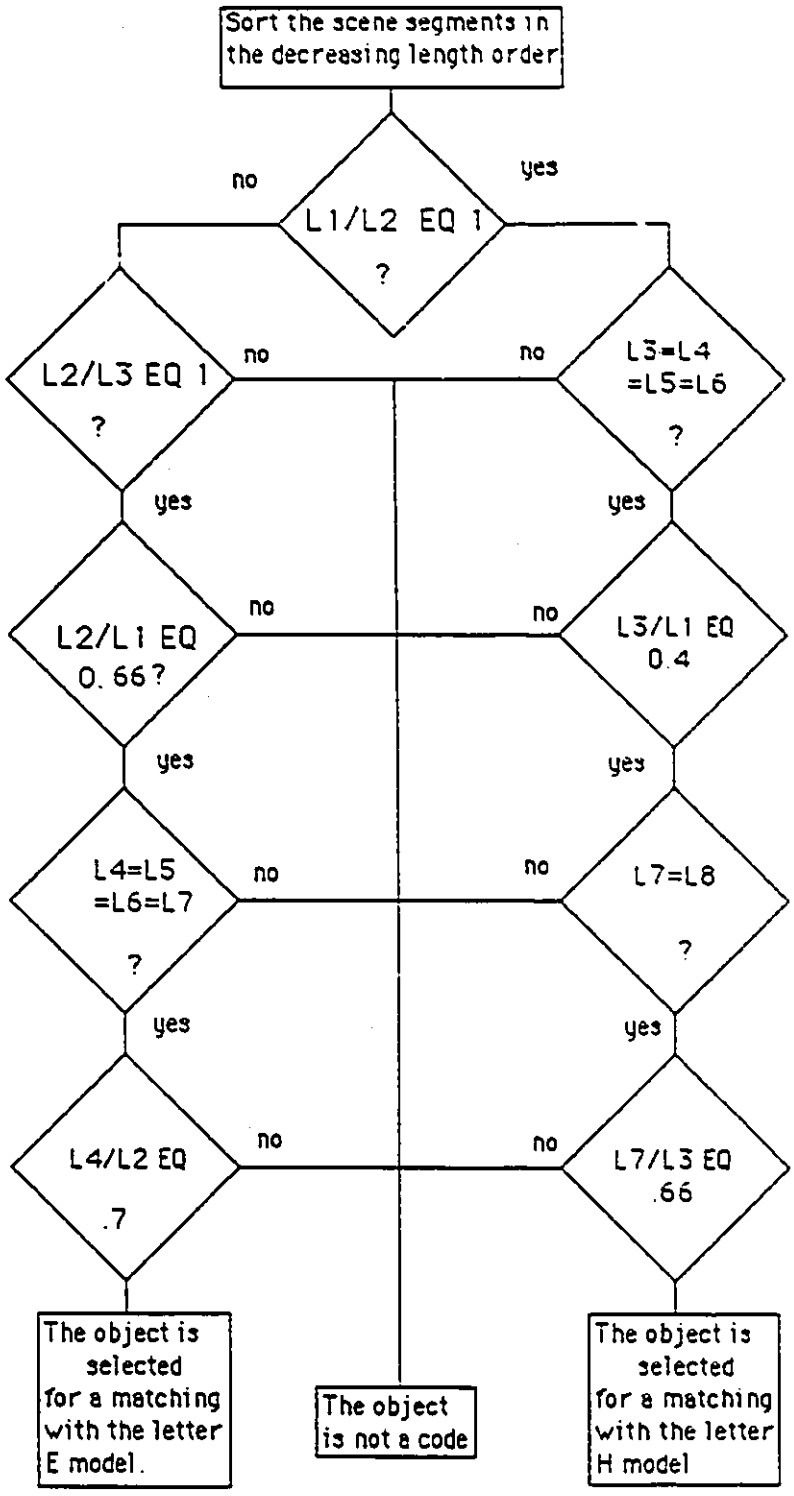
Our problem is not limited merely to the recognition of the binary symbols but also deals with their positioning in the 3-D environmental space (as further discussed in chapter IV). Recognizing shapes like printed characters, especially when the feature generation scheme is as robust as the one we have employed, is a rather simple problem to solve. A simple tree search, that checks a certain number of features characterizing the object, is quiet sufficient. However, to be able to determine the 3-D position and orientation of each symbol, the information provided by the recognition procedure is not sufficient. To determine the "POSE" (Position Orientation and Scaling) of the recognized code, we need a certain number or vertex correspondences between the scene description and a model description. This can be performed, as it will be explained in chapter IV by using a model based shape recognition technique.

The next section will discuss a fast technique for object recognition used to identify the binary symbols. In this way the 3D object positioning algorithm will be performed only on objects which we are interested in.

III.4.2) SYMBOL CODE RECOGNITION BY TREE SEARCH

After processing, the search algorithm outputs one of the following results:

- the object is not binary code;
- the object is the symbol 'E'; or
- the object is the symbol 'H'.



Flowchart III.5

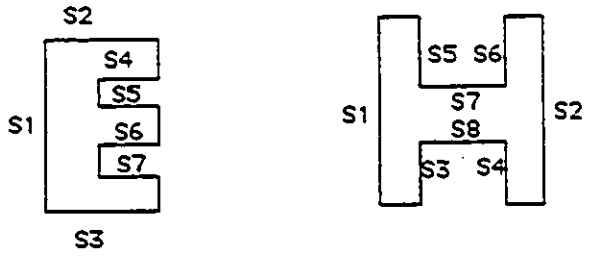


Fig III.14 Segment indexing for the characters 'E' and 'H'.

The search procedure consists of checking intrinsic properties (relations) of the polygon description of each binary symbol. Flowchart III.5 describes this procedure.

The polygon segments approximating the contour are first sorted in a decreasing order of their lengths, S_1 , S_2, S_3... (see Fig III.14). The tree search is based on comparing segment lengths. An easy to check property which differentiates the symbol 'E' from 'H' is the fact that $L_1=L_2$ is only true for the symbol 'H'. Therefore, we start by checking this feature. Following this some other segment lengths are checked. For instance, in the case of the letter 'E' we have $L_2=L_3$ $L_4=L_5$, $L_2=L_3$ etc.... Every time a test fails, the program execution stops and the object is recognized as not being a binary symbol.

III.5 CONCLUSION

In this chapter we have investigated the problem of binary symbol recognition. This task has been divided in two phases: feature generation and symbol recognition. Feature generation is accomplished through a sequence of steps: boundary tracing of each object appearing in the image, polygon approximation of each boundary, and finally corner-vertex extraction from each approximated contour. The recognition phase consists of a simple tree search that checks intrinsic relational properties..

As regards polygon approximation, a new method to implement the split and merge algorithm was proposed in this chapter; we have shown that using area deviation per segment length as an error norm offers polygon approximation better run-time performances.

The robustness of the polygon approximation method employed offers the recognition process considerable accuracy and precision. However, the exact localization of the pseudo-random window that permits the recovery of the AGV absolute position is a problem that needs further considerations. Also, for path planning purposes, it is important to collect information about the 3D profile of the path seen in front of the vehicle. These geometrical considerations will make the subject of the next chapter.

CHAPTER IV

GEOMETRIC CONSIDERATIONS IN VISUAL POSITION MEASUREMENT

IV.1) *INTRODUCTION*

As mentioned in chapter I, the visual navigation system to be designed should have position measurement as well as automatic guidance abilities. It was shown in Chap II that the AGV position is recovered by reading the n-bit pseudo-random code that identifies the absolute position. On the other hand, automatic visual guidance requires the 3D profile reconstruction of the guide-path in front of the vehicle [Mor90].

Obviously both problems would be easy to solve if the binary symbol recognition process was error free. In practice, because of inadequate lighting conditions and the eventual presence of objects hiding a portion of the path to the camera, several symbols may not be recognized. Despite the redundant number of codes in range, it is not an easy task to find n binary symbols in the image that follow each other on the guide-path. In fact, because of perspective, the binary symbols further away from the camera appear very close to each other in the image. In order to solve this problem it is necessary to determine the 3D position of each binary symbol. Once this is done, since the actual distance between two successive codes on the guide-path is known, it is possible to determine when a symbol has not been recognized. Moreover, this facilitates the estimation of the 3D profile of the visible guide-path portion. The estimated profile will consist of the 3D piecewise linear curve which joins the centroids of all the neighboring binary symbols.

This chapter discusses problems related to 3D binary symbol positioning. In section IV.2 there is a description of the geometric aspect of image formation. Section IV.3 presents the flat-earth geometric model technique for 3D. binary symbol positioning and section IV.4 discusses the iterative least-squares solution to the problem.

IV.2) *THE GEOMETRIC ASPECT OF IMAGE FORMATION*

The study of the *perspective* problem consists of investigating the *geometric aspect of the imaging process* [sha89]. The problem consists of finding the geometrical transformation between object and the corresponding image points. Fig IV.1 shows an arbitrary point P(xo,yo,zo) in the 3D space. The reference frame of the 3D space is defined by the camera position. Its origin is the focal point of the camera. The image, I(xi,yi,-f), of P(xo,yo,zo) is obtained by projecting the line issued from P and passing by O, the focal point, on the image plain. The points I, O and P are therefore collinear and we can write:

$$(xo,yo,zo)= k \cdot (xi,yi,-f) \qquad (VI.1)$$

where k is a real constant such that:

$$k= - zo/f \qquad (IV.2)$$

Thus we can write:

$$xi= - (f/zo) \cdot xo \qquad (IV.3.a)$$
$$yi= - (f/zo) \cdot yo \qquad (IV.3.b)$$

The geometric transformation defined by the IV.3 equations is called the ***perspective-projective*** transformation. This model is

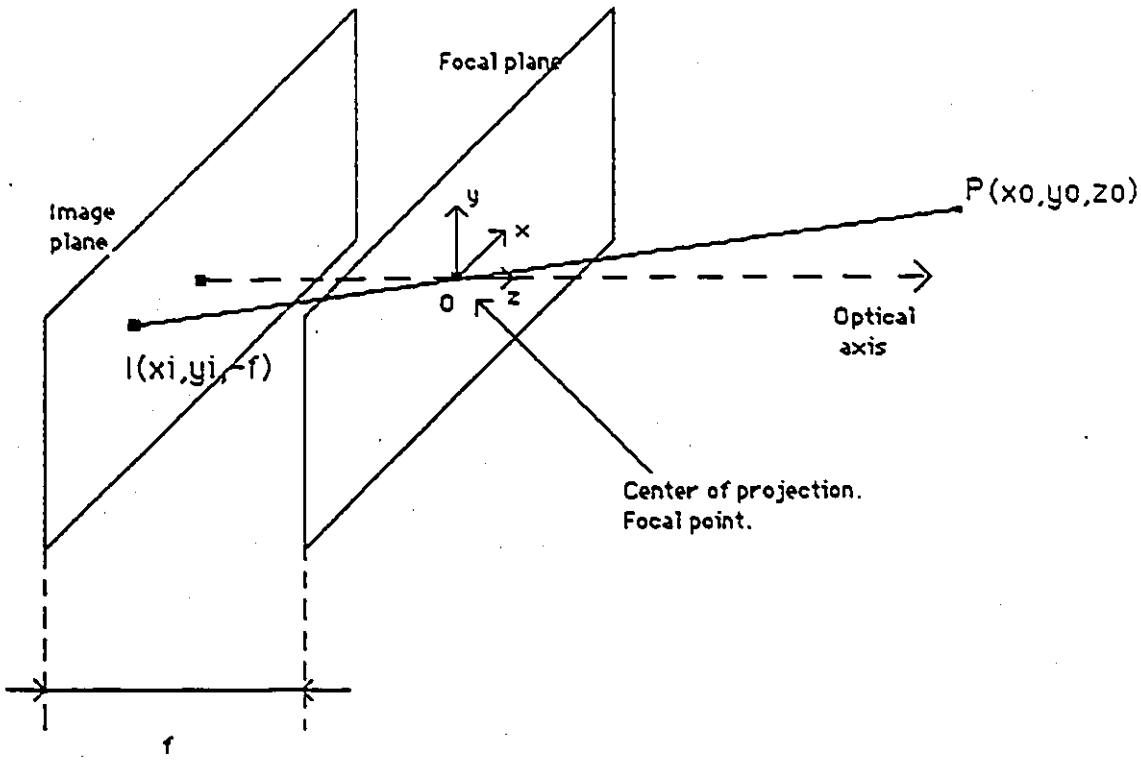


Fig IV.1 Geometry of the perspective transformation.

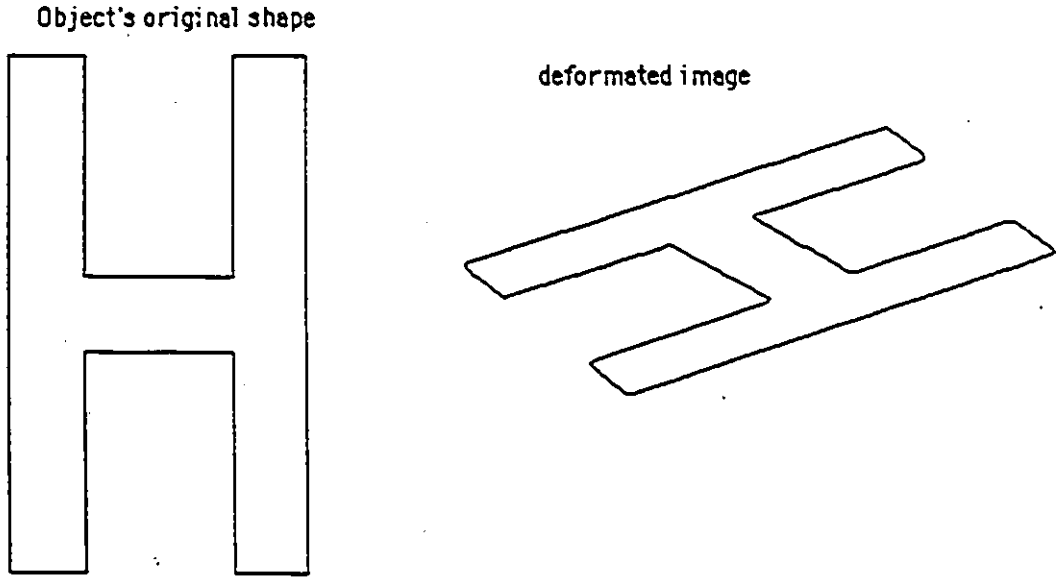


Fig IV.2. Shape deformation by perspective transformation.

visibly nonlinear. Note that when the object is planar and the camera optical axis is perpendicular to the object plane, zo is constant. The geometric transformation in this situation is reduced to a scaling and no shape deformation occurs.

On the other hand, when the object is still planar but the optical axis is not perpendicular to the object plane, zo is no longer constant and the shape is subject to deformations. Fig IV.2 shows how the image of the letter H is deformed for a 30° angle between the optical axis and the normal to the object plane.

When the distance of the object to the camera is large relative to the object dimensions, it can be assumed that zo is constant for all object points. Based upon this assumption, the transformation is reduced to the composition of a projection and a scaling.

As far as object recognition and range measurement using monocular vision are concerned, the inversibility of the perspective-projective transformation is an important problem.

Let I(xi,yi,-f) be any point in the image plane. Every point P(xo,yo,zo) which has I(xi,yi,-f) as image point verifies equation IV.3. This implies that:

$$xo = (-xi/f) \cdot zo \qquad (IV.4.a).$$
$$yo = (-yi/f) \cdot zo \qquad (IV.4.b).$$

IV.4 shows that the point set having I(xi,yi,-f) as image point is the line containing both the focal and the image point. Therefore, the perspective-projective transformation is not invertible.

This raises the question of how to determine the 3D position of an object *visible in a single image*. Solving this so called ***inverse perspective problem***, because the perspective-projective transformation is noninversible, requires, on top of the image, supplementary information about the object. Two situations may

occur: information about either the *object position* or the *object shape* are available.

The first case is much easier to deal with and has a straightforward solution. Generally the a priori information about the object position can be formulated into either a linear or nonlinear constraint which makes up a third equation of the system IV.3. A unique solution in that case exists. An example of such a situation is the flat-earth geometric model technique used in visual automatic guidance to estimate the 3D profile of the road in front of the vehicle [Mor90]. We use this technique, in our case, to determine the 3D position of each binary symbol appearing in the image (see section IV.3).

In the second case, the case of the *model based 3D position estimation*, the object recognition and its reference to a specified model are required. Considering the *model* and the *scene* of the object the 3D position is determined by computing the *perspective deformations* the scene has been subject to. Unlike the first method, the 3D position of the object is determined *indirectly* using a least-squares estimation. This case is thoroughly discussed in section IV.4.

IV.3: THE USE OF THE FLAT-EARTH GEOMETRY MODEL FOR 3D CODE POSITION DETERMINATION

In the *flat-earth* geometry model [Mor90], we assume that the path is planar, and that the plane containing the visible portion of the path is the same plane which is giving support to the vehicle.

In this case, the 3D reconstruction, P(xo,yo,zo), of an image point I(xi,yi,-f) is made by finding the point of intersection of the line from the focal point of the camera through the image point I(xi,yi,-f) with the ground plane. Fig IV.3 describes this principle.

Using equations IV.3 we have:

$$xi = -(f/zo) \cdot xo \qquad \text{(IV.5.a)}$$

$$yi = -(f/zo) \cdot yo \qquad \text{(IV.5.b)}$$

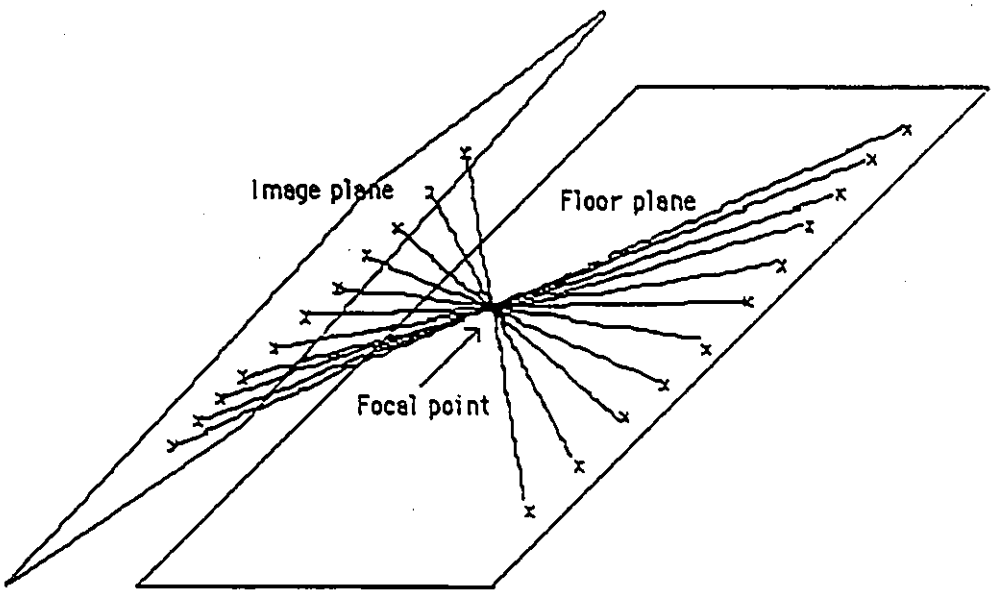


Fig IV.3 The flat earth model.

Since Po belongs to the floor plane it verifies the equation of the latter:

$$a\,xo + b\,yo + c\,zo = d \qquad \text{(IV.6).}$$

Where a,b,c and d are determined using the distance along the vertical axis between the camera and the floor and the tilt angle formed by the camera to floor plane. By substituting IV.5 in IV.6 we have:

$$xo = -\,(d \cdot xi)/\,[-a \cdot xi - b \cdot yi + f \cdot c] \qquad \text{(IV.7.a).}$$

$$yo = -\,(d \cdot yi)/\,[-a \cdot xi - b \cdot yi + f \cdot c] \qquad \text{(IV.7.b).}$$

$$zo = (f \cdot d)\,/\,[-a \cdot xi - b \cdot yi + f \cdot c] \qquad \text{(IV.7.c).}$$

Equation IV.7 shows how to determine the 3D position of one point on the ground given its image.

In our case, each binary symbol is represented by its centroid. Using equation IV.7, with I(xi,yi,-f) being the centroid of the binary symbol image we can determine P(xo,yo,zo) the 3D position of the binary symbol centroid on the floor.

The flat-earth geometry technique has the advantage of requiring the simple calculation of (IV.7) for each symbol on the track. For this reason the method is fast. However, there two major problems which limit the applicability of this method:

- since no estimation is made, the error in the output 3D locations is only a function of the extent to which the flat-earth assumption is violated; and

- the sensitivity to inaccuracies in the assumed tilt angle formed by the camera to the floor plane.

In industrial environments, the flat-earth assumption can *easily be violated*. Moreover, the camera is in a fixed position relative to the body of the AGV, but the body of the vehicle is able to rock forward and backward on the undercarriage. Hence, the accuracy of the tilt angle of the camera to the floor is not reliable.

Taking into account these considerations, the decision to employ or not the flat-earth technique is dependent on the environment in which the AGV will be used. In the next section, another more robust technique that makes no assumptions of evenness of the floor surface is discussed. This technique determines, iteratively, the 3D position of each code appearing in the image. Unlike the flat-earth geometric model technique the calculations are not straightforward; a least-squares estimation of the 3D position is made and therefore the technique is more time consuming.

IV.4: MODEL BASED 3D POSITION ESTIMATION

The case of 3D object position determination using a model of the object is discussed in this section. As explained above this method requires the recognition of the object beforehand. Considering the model and the scene of the object, the 3D position is determined by computing the perspective deformations the scene has been subject to. This method is indirect and uses least-squares estimation to find the object 3D position.

This so called ***pose*** (Position Orientation and Scaling Estimation) problem or ***LDP*** (Location Determination Problem) has been the interest of considerable research. In section IV.4.1 we define the pose problem and review the existing literature related to the subject. Section IV.4.2 presents the geometric hashing technique for model and scene point matching. Finally section IV.4.3 deals with the numerical solution to the pose problem.

IV.4.1 DEFINITION OF THE POSE PROBLEM AND LITERATURE REVIEW

Model based 3D position estimation is a problem of *fitting a model to experimental data* (the object image in this case). In fact, as mentioned by Fishler and Bolles (in [Fish81] pp 381):

> to a large extent, scene analysis is concerned with the interpretation of sensed data in terms of a set of predefined models.

Fishler and Bolles go on to describe the data interpretation process and divide it into two tasks:

first, there is the problem of finding the best match between the data and one of the available models (the classification problem); second, there is the problem of computing the best values for the free parameters of the selected model (the parameter estimation problem).

The *parameter estimation* problem mentioned above, in our case, corresponds to pose estimation. The parameters to be estimated are the *six degrees of freedom of the object*. Generally, these consist of *three translational and three rotational parameters* (position and orientation).

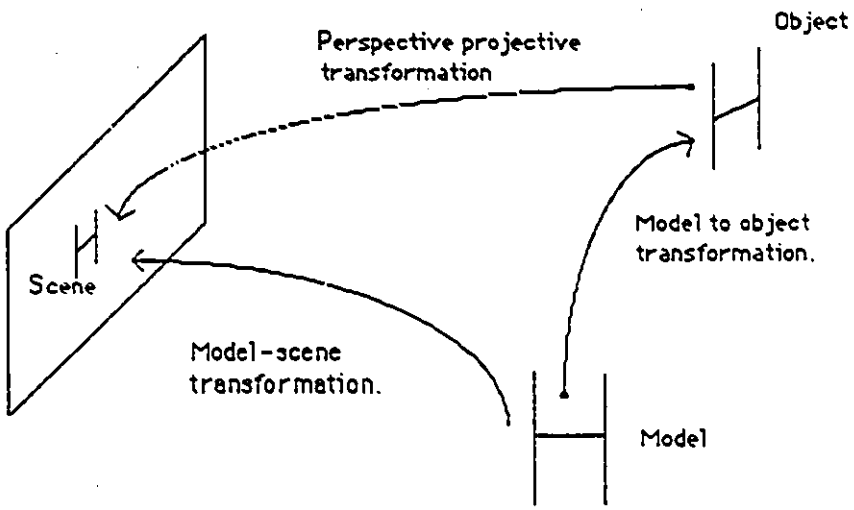


Fig IV.4 General scheme for 3D object positioning.

Fig IV.4 describes the general scheme of 3D object positioning. After the scene has been recognized, an arbitrary 3D position for the model is chosen. The 3D object position is defined by the geometrical transformation, called model-object transformation, between the model and the object. The model-object transformation is the composition of a rotation and a translation in the 3D space. If Pm is a point in the model and Po is another point in the object, we have:

$$\vec{Po} = R \cdot Pm + T \qquad (IV.8).$$

R is the composition of three rotations: one along the X axis, one along the Y axis and one along the Z axis. Consequently R can be written as:

$$R = \begin{pmatrix} C_x.C_y.C_z+S_y.S_z & -C_x.C_y.S_z-S_x.C_z & C_x.S_y \\ S_x.C_y.C_z+C_x.S_z & -S_x.C_y\,S_z+C_x.C_z & S_x.S_y \\ -S_y.S_z & -S_y.C_z & C_y \end{pmatrix} \qquad (IV.9)$$

where C and S respectively represent "cos" and "sin" operators. The indices X,Y and Z stand for the angles of the rotation respectively along the axes X,Y and Z. Note that R can be defined as the composition of rotations along three different axes.

The translation T is defined by:

$$T = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \end{pmatrix} \qquad (IV.10)$$

As indicated in Fig IV.4, the model scene transformation is the composition of the model-object transformation and the perspective transformation.

R and T are computed as follows: first a number of scene points and their corresponding model points are matched. Then R and T are determined by minimizing a quadratic function which calculates the sum of the distances between the matched scene points and the image by the model-scene transformation of their corresponding points in the model. Note that segments instead of points can be matched.

Therefore model based position estimation involves two steps:

1) model or scene point (or segment) matching; and,

2) estimation of the model-object transformation by least-squares minimization.

These two steps can be performed either successively or simultaneously. In the first case, the point (or segment) matching and the least-squares minimization steps are performed independently ([Aru80], [Joo88], [Dar88], [Faug90], [Fab88] and [Dho89]). These methods are based on the classical techniques for parameter estimation. Therefore they have no internal mechanism for detecting and rejecting gross errors. However, when a robust matching technique is employed, gross errors are rejected automatically [Gar90].

In the second case the least-squares minimization is performed as the matching is made. After each matching, the new point (respectively segment) correspondence is taken into account to refine the 3D object position. To initialize the algorithm, a minimum number of point (or segment) matchings are necessary to permit a first estimation of the pose (in [Fish81] and [Faug86] a study of the minimum number of point correspondences for pose computation is carried out). Performing matching and least-squares minimization simultaneously was first introduced by Fishler and Bolles [Fish81]. The advantage of using this so called RANSAC method (Random Sample Consensus) is that it permits the filtering of data which has a significant percentage of gross errors. The method presented by Faugeras *et al* in [Faug86] (named HYPER: Hypothesis Prediction and Estimation Recursively) to solve the 2D pose problem is also based upon the same idea.

A review of the model-scene matching techniques and the least-squares methods for 3D object positioning is further given.

The model-scene matching problem has been thoroughly studied during the last two decades. Matching model and scene points (or segments) is used considerably in model based object recognition. Because of the large amount of publications available,

it is difficult to survey all the existing techniques. In the following we review only some of these.

Graph matching techniques [Dav79] consist of dividing the scene and the model contour into subparts. Using a local evaluation function, for each element in the template, a subset of object elements that the template element may be associated with is determined. Each association of a template element with a single matching object element will correspond to a node in the association graph. Graph matching techniques have widely been used in model based pattern recognition, because of their suitability for programming and their **structural** nature that gives them position and orientation independence. Some graph matching techniques are probabilistic [Groe85].

The **distance measure technique** is a worth mentioning segment matching method. It calculates an error norm (or distance measure) between each scene segment and the corresponding segment in the model [Faug86].

Another recent point matching method is the **geometric hashing technique** [Gav90]. This technique is based on the principle that, assuming the **linearity** of the **perspective-projective** transformation, the coordinates of any model point, calculated in a frame related to the model, are equal to the coordinates of their corresponding scene points calculated in the corresponding scene frame. This principle permits the determination of model and scene point correspondences. The method is fast and easy to implement. We have decided to use this point matching scheme for our problem.

The least-squares minimization problem, which constitutes the second step of the 3D positioning scheme previously described, is known in the photogrametry literature as the ***exterior problem***. It consists of, given n scene points (respectively segments) and their corresponding model points (respectively segments), finding the

object 3D location for which the model points best fit the scene points. As mentioned by Haralick and Joo (in [joo88] pp 385): " The dissertation by Szczepanski (1958) surveys nearly 80 different solutions beginning with one given by Karlsruhe in the year 1879". This quotation indicates the importance and the complexity of the pose problem. We classify the solutions in two groups: the geometrical methods and the analytical methods.

The **geometrical** methods compute the pose by making considerations on the 3D relative geometric position of the model and scene points. Examples of such analyses are presented in [Dav88], [Fish81] and [Dho89]. The advantage of using these technique is the possibility they offer to physically understand the inverse-perspective problem. As for the **analytical** methods, their advantage resides in the possibility of using the well established numerical techniques that exist for parameter estimation. One of the best publications that surveys analytical techniques for pose estimation, is Haralick and Joo's paper [Joo88] in which three numerical techniques for least-squares estimation of the pose are described.

The first method described in [Joo88] (the one adopted for our work) employs *a least-squares fitting algorithm of two 3D point sets* [Aru80] to find *the best fit of a set of 3D points (model points) in a set of 2D points (scene points)*. The method is iterative: at each iteration, in a first stage, the optimal range of each point in the scene, assuming the rotation and translation which define the 3D object position are known, are calculated. In a second stage, the scene point ranges are supposed known and the optimal rotation and translation are calculated using the method presented in [Aru80].

The second technique presented in [Joo88] finds the least-squares solution by **linearization** of the least-squares function. The method is also iterative. The solution is adjusted at each iteration using a linearized model. Faugeras *et al* [Faug90] propose the same technique using segment matching instead of point matching.

The third method presented in [Joo88] employs more elaborated nonlinear regression techniques: **the M-estimators**. M-estimators minimize objective functions more general than the sum of squared residuals associated with the sample mean. This method is unfortunately time consuming.

Other methods which can not be classified as being geometrical or analytical are worth being surveyed. Faber and Stokely [Fab88] use **a tensor based moment function** to estimate the pose. Faugeras *et al* [Faug86] use a *Kalman filter* to solve the minimization problem and Davis *et al* [Dav88] **match triangle pairs** in the model and the scene to do the same.

In the following two sections we present in detail the first and the second step of the 3D object positioning scheme. In section IV.4.2 we discuss the geometric hashing method for point matching and in section IV.4.3 we describe the least-squares solution adopted to the LDP problem.

IV.4.2: MODEL AND SCENE POINT MATCHING USING GEOMETRIC HASHING

Geometric hashing is a general technique for model-based object recognition. It represents a robust method where matching is done on *local* features of an object, *independent* of the other features. A ***voting*** scheme is used to evaluate a number of correct matches. The object modeling phase of geometric hashing consists of describing the object by a set of points, called ***interest points***, which are such local features as physical points, line segments, curve segments, etc., together with their geometrical relation. The same features, interest points, are extracted from the scene together with their ***geometrical relations***.

The purpose of employing geometric hashing in our case does not consist of recognizing the binary symbols. Indeed these have been recognized using a tree search (which is a much faster algorithm). Geometric hashing permits one to match a number of vertices in the scene with a number of vertices in the model. This vertex correspondence will be useful in 3D position determination of binary symbols. Consequently, interest points, in our case, will be vertices localized on the boundary corners. In the following the problem of geometrical relation is studied.

As already mentioned the perspective projective transformation can be considered linear when the object range is considerably larger than its size. In the following analysis this hypothesis is assumed true. The perspective-projective transformation, assuming this hypothesis, is an affine transformation which consists of an orthogonal projection of the object on the viewing plane. As known, an affine transformation can be described by 2x2 non-singular matrix A and 2x1 translation vector b, mapping vector x to vector $Ax+b$. The transformation is fully determined by three point to point correspondences. Given a set of points describing the model and a set of points describing the scene, the question is: *is there an affine transformation which maps a subset of scene points onto a subset of the model points?*

Let p_0, p_1 and p_2 be three non-collinear points in 2D space and let (p_1-p_0, p_2-p_0) be a basis spanning the 2D space with p_0 as origin (see Fig IV.5). The coordinates (a,b) of an arbitrary fourth point p with respect to the chosen basis are given by:

$$p = a\,(p_1 - p_0) + b\,(p_2 - p_0) + p_0$$

If the whole system undergoes an affine transformation T then:

$$T\,p = T\,(\,a(p_1-p_0) + b\,(p_2 - p_0) + p_0\,)$$

$$= a\,(T\,p_1 - T\,p_0) + b\,(T\,p_2 - T\,p_0) + T\,p_0$$

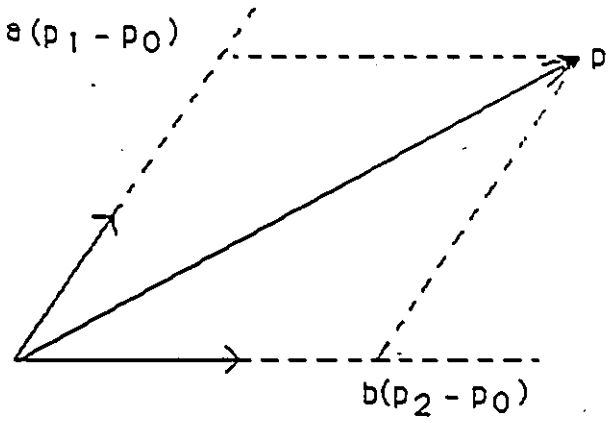
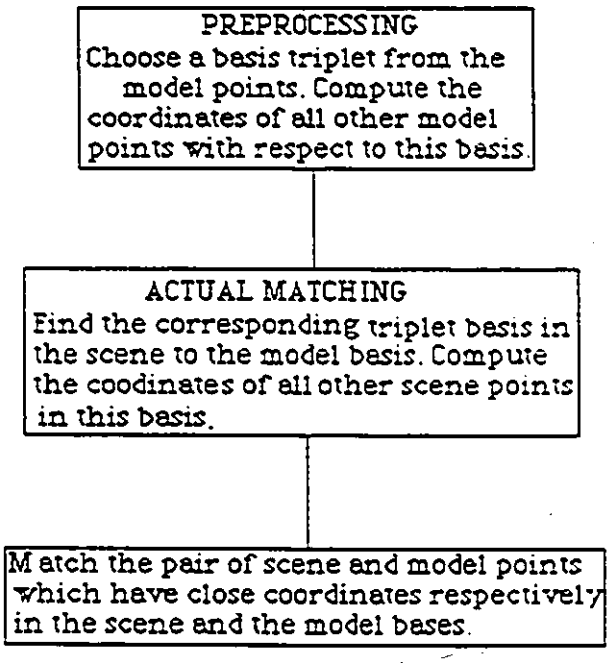


Fig IV.5. Coordinates of a point in a three point frame.



Flowchart IV.1.

where it can be seen that the coordinates of the transformed point **Tp** with respect to the transformed coordinates system are still the

same as those of p before applying T. In other words the coordinates of points with respect to a basis are transformation invariant. This is actually the key observation of geometric hashing.

The matching process is divided in two stages: **preprocessing** (object representation) and **actual matching**. In the preprocessing stage a basis triplet is chosen from the model points and the coordinates of all other model points are computed with respect to this basis. In the **actual matching** stage the three points corresponding to the model basis triplet are determined and again the coordinates of all other scene points are computed with respect to this basis. The voting scheme for correct matches is achieved by searching for the pair of scene and model points which have the closest coordinates in the bases defined above. Flowchart IV.1 illustrates this principle.

IV.4.3: LEAST-SQUARES MINIMIZATION FOR LOCATION DETERMINATION

With the software we have presented so far, we can recognize the binary symbols appearing in the image and match a number of vertices in each symbol scene with a number of vertices in the symbol template. As mentioned in section IV.3, model based 3D position estimation is based upon consideration of the geometrical deformations due to perspective. The pose estimation is made by calculating the model-object transformation (see Fig IV.4) that minimizes a quadratic error norm.

This section deals with the least-squares minimization for 3D position estimation problem. In section IV.4.3.1 the mathematical formulation for the LPD is made, section IV.4.3.2 will present the numerical solution adopted and section V.3.3 deals with the *singular value decomposition problem.*

IV.4.3.1 *Maximum likelihood estimation of 3D location*

Let $p_1, p_2, \ldots p_n$ be the observed 3D model points in the Euclidean space and $q_1, q_2, \ldots q_n$ be the corresponding 2D perspective projections of the 3D points. The point correspondence is made using the geometric hashing technique presented in section IV.4.2. By applying the principle shown on Figure IV.4 and using equation (IV.8), for each i less than n we have:

$$q_i = Ps.(R \cdot p_i + T) \qquad \text{(IV.11)}.$$

Where Ps is the perspective-projective transformation defined by equation (IV.3) and R and T are respectively the 3D rotation and 3D translation defined in (IV.9) and (IV.10).

Obviously, equation (IV.11) is only valid in the ideal case where the scene is not noisy. In reality we have:

$$q_i = P.(R \cdot p_i + T) + w_i \qquad \text{(IV.12)}.$$

where w_i represents zero-meanvalue noise components.

$$E(w_i)=0 \qquad \text{(IV.13)}$$

Equation IV.12 is a linear system with three unknowns: R, T and wi. It should be noted, however, that wi's are random variables and there is no way to compute R and T by solving (IV.12). We can only *estimate* R and T given the set of *observations* $\{q_i\}_1^m$. Therefore the problem of 3D position estimation is that of estimating the model-object transformation based upon the set of measured data $\{q_i\}_1^m$.

Estimation theory offers a large variety of estimators of which performance varies according to the problem. We will use a ***maximum likelihood*** estimator. This estimator provides estimates R* and T* of R and T that maximize the likelihood function:

$$P(\{q_i\}_1^m \mid (R^*,T^*)) \qquad (IV.14)$$

P is the probability density of $\{q_i\}_1^m$ given R and T. Note that this is not a conditional probability. It is the evaluation of the probability of having $\{q_i\}_1^m$ assuming R and T are true.

The maximum likelihood estimator is generally biased. However, it has the following interesting asymptotic properties:

- when the number (m) of samples increases, the estimator is asymtotically efficient, i.e unbiased and of minimum variance. It is proved that efficient estimators have the smallest mean square errors.
- if the observation vector $\{q_i\}_1^m$ is Gaussian, the maximum likelihood estimator comes to a least-squares estimator. Obviously, the latter is much more simple to compute.

In the following we will show that under certain assumptions we can make the second property applicable to our estimation problem.

Assuming the measurements $\{q_i\}_1^m$ are independent we can write:

$$P(\{\{q_i\}_1^m\}_{(R,T)}) = P(\bigcup_{i=1}^{m} q_i)_{(R,T)} \qquad (IV.15)$$

The noise in each measurement qi is the cumulative effect of the independent following stochastic processes:

1) the quantization noise;
2) the segmentation errors due to unconstant lighting conditions; and
3) the errors due to suboptimal polygon approximation.

Each one of the enumerated noise sources is the result of many independent stochastic processes. Under this assumption, the *central limit* theorem permits consideration of the error on qi as being Gaussian.

On the other hand from equation (IV.12) we can write:

$$E(q_i) = E(P \cdot (R \cdot p_i + T)) + E(w_i)$$

Using (IV.13) we can write:

$$E(q_i) = P \cdot (R \cdot p_i + T) \qquad (V.16)$$

Thus, qi is a normal iid with mean-value

$$p'_i = Ps \cdot (R \cdot p_i + T) \qquad (V.17)$$

and standard deviation s_i:

$$P(q_i)_{(R,T)} = \frac{1}{\sqrt{2\pi}} \cdot s_i \cdot \exp\left\{ -\frac{1}{2s_i^2} (q_i - p'_i)^T \cdot (q_i - p'_i) \right\} \qquad (IV.18)$$

We can write equation (IV.15) as:

$$P(\{q_i\}_1^m \mid_{(R,T)}) = \prod_{i=1}^{m} \{\frac{1}{\sqrt{2\pi}\cdot s_i} \cdot \exp\left(-\frac{1}{2s_i^2}(q_i-p'_i)^T \cdot (q_i-p'_i)\right)\} \qquad (IV.19).$$

Let s be:

$$s = \prod_{i=1}^{m} s_i. \qquad (IV.20)$$

We can write:

$$P(\{q_i\}_1^m \mid_{(R,T)}) = \frac{1}{(\sqrt{2\pi})^m \cdot s} \exp\{\sum_{i=1}^{m} \frac{-1}{2s_i^2}(q_i-p'_i)^T \cdot (q_i-p'_i)\} \qquad (IV.21).$$

Thus maximizing $P(\{q_i\}_1^m \mid_{(R^*,T^*)})$ comes down to minimizing:

$$R_m = \sum_{i=1}^{m} \frac{1}{2\cdot s_i^2}(q_i-p'_i)^T \cdot (q_i-p'_i) \qquad (IV.22)$$

Therefore, the maximum likelihood estimator, under the assumed Gaussian noise, comes to a least-squares estimator minimizing IV.22.

In general, all the s_i's are considered to be equal. Nevertheless, the quadratic function used is not that mentioned in (IV.22). Some modifications have to be made so that the minimization function conforms to the general form of quadratic estimators. Assume that the vector (u_i) is defined by:

$$u_i = (q_i, f)^T \quad . \qquad (IV.23)$$

f being the focal distance. We assume that d_i is the distance to the focal point of the object point (O_i) of which image is q_i. Using equation IV.3 the coordinates of (O_i) are determined by the vector:

$$\frac{d_i}{f}\ u_i. \qquad \text{(IV.24)}$$

Therefore the quadratic function to be minimized becomes as indicated in (IV.25). Note that in equation (IV.25) the unknowns are R, T and $\{d_i\}_1^m$.

$$\sum = \sum_{i=1}^{m} \| (R \cdot p_i + T) - \frac{d_i}{f}\ (q_i, f)^T \|^2 \qquad \text{(IV.25)}$$

In the following section we show the use of such a representation and the numerical solution to the problem.

IV.4.3.2 *The numerical solution to the pose problem*

If the distances $\{d_i\}_1^m$ in equation (IV.25) were known, determining R and T would become a *least-squares fitting of two 3D point sets* problem. This problem is solved using the single value decomposition technique proposed in [Aru80]. Section IV.4.3.3 discusses the mathematics of this problem and proposes simplifications for the case of planar objects.

Conversely, if R and T are known, the computation of $\{d_i\}_1^m$ is simply made by setting all the partial derivatives of $\sum$ with respect to (di) to zero.

$$\frac{\partial(\sum)}{\partial(d_i)} = 0 \qquad \text{(IV.26)}$$

By substituting IV.25 in IV.26 and deriving the equations we obtain the following result:

$$d_i = f \frac{(q_i.f) \cdot (R.p_i + T)}{(q_i.f) \cdot (q_i.f)^T} \qquad (IV.27)$$

The case where R , T and $\{d_i\}_1^m$ are all unknown is a nonlinear minimization and equation IV.25 can only be solved *iteratively*. As mentioned above we are going to use the method presented in [Joo88]. Note that extended Kalman filtering can be used to solve the problem. Algorithm IV.1 shows the numerical solution to the problem. At each iteration two activities are performed: first the $\{d_i\}_1^m$ are supposed given and R and T are computed using the singular value decomposition technique presented in [Aru80] (see section IV.4.3.3); second, given the computed values of R and T the distances di's are determined using equation (IV.27). To initialize the algorithm reasonable values of the di's are chosen. We decided to take the same constant value for each point. It represents an initial guess of how far the object is. In the situation we are dealing with, since the objects are planar, the ratio scene perimeter-object perimeter permits the calculation of this distance. Assuming the distance of each point on the contour is a constant (d) and using equation (IV.3) we have:

$$\frac{P_s}{P_m} = \frac{f}{d} \qquad (IV.28)$$

Where P_s and P_m are the perimeters of respectively the scene and the object. Hence:

$$d_i = d = \frac{P_m}{(P_s.f)} \qquad (IV.29)$$

ALGORITHM IV.1

1)Initialize the depth di of each point (see equation IV.29).

2)Iterate: Suppose the di's are given. Compute the new values for each di by:

a) Finding R and T which minimize Σ_n in equation V.25 using the singular value decomposition technique [Aru80] (see section IV.4.3.3).

b) Finding $\{di\}_1^m$ using equation IV.27.

3) Compute Σ_n in equation IV.25. If Σ_n is less than a boundary value stop, otherwise go to 2).

An interesting property of the algorithm is the fact that the residual error Σ_n is monotonously decreasing. In other words :

$$\Sigma_{n+1} > \Sigma_n \qquad \text{(IV.30).}$$

In the following we give a proof of this property. The proof we present is different and much simpler than the one presented in [Joo88]. Indeed, by substituting (IV.27) in (IV.25), we have:

$$\Sigma_n = \sum_{i=1}^{m} \| R_n \cdot p_i + T_n) - \frac{(q_i.f)\cdot(Rn.p_i + Tn).(q_i.f)^T}{(q_i.f)\cdot(q_i.f)^T} \|^2 \qquad \text{(IV.31)}$$

R_{n+1} and T_{n+1} minimize the term:

$$\Delta_{n+1} = \sum_{i=1}^{m} \| R_{n+1} \cdot p_i + T_{n+1}) - \frac{(q_i.f).(R_n.p_i + T_n).(q_i.f)^T}{(q_i.f) \cdot (q_i.f)^T} \|^2 \qquad \text{(IV.32)}$$

Therefore:

$$\Delta_{n+1} < \Sigma_n \qquad \text{(IV.33).}$$

On the other hand, the terms $\{ d_i = \frac{(q_i.f).(R_{n+1}.p_i+T_{n+1})}{(q_i.f).(q_i.f)^T} \}_1^m$ minimize the following term:

$$\Sigma_{n+1} = \sum_{i=1}^{m} \| R_{n+1}\, p_i + T_{n+1} - (\frac{d_i}{f}\ (q_i.\ f)^T \|^2 \qquad \text{(IV.34).}$$

This implies:

$$\Sigma_{n+1} < \Delta_{n+1} \qquad \text{(IV.35).}$$

Finally by combining (IV.33) and (IV.35) we show (IV.30).

Using the initialization indicated above, the algorithm converges within three to five iterations depending on how noisy the binary symbol is.

IV.4.3.3 *The least-squares fitting of two 3D point sets using sigular value decomposition*

IV.4.3.3.1 Introduction

It is worth recalling, at this stage, that the 3D object positioning problem, formulated in section IV.4.3.1, consists of minimizing a quadratic function defined by equation (IV.25). The parameters to be estimated, when solving the pose problem, consist of a 3D rotation R, a 3D translation T and the set of distances $\{d_i\}_1^m$. The minimization task is performed using the iterative algorithm presented in IV.4.3.2 (algorithm IV.1). At each iteration of the algorithm, in the first stage,

the optimal rotation R and translation T which minimize the quadratic function (IV.25), knowing $\{d_i\}_1^m$ are determined. In the second stage, R and T are assumed to be known and the optimal set $\{d_i\}_1^m$, which minimizes (V.25), is computed. In section IV.4.3.2 we presented the solution to perform the second stage of each iteration. This current section deals with the first stage performance.

Let us define a new set of 3D points $\{r_i\}_1^m$ such that:

$$ri = \frac{d_i}{f} (q_i, f)^T \qquad (IV.36)$$

Minimizing IV.25 when all di's are constant is equivalent to solving:

$$\sum = \sum_{i=1}^{m} \|(R.p_i + T) - r_i\|^2 \qquad (IV.37).$$

Hence, the minimization of solving (IV.25) in this case comes down to finding the least-squares fitting of two sets of 3D points $\{p_i\}_1^m$ and $\{r_i\}_1^m$.

This problem has been addressed by Arun, Huang and Blostein in [Aru80]. In section IV.4.3.3.2 we present the solution proposed in [Aru80]. In section IV.4.3.3.3 we define the notion of singular value decomposition and state the decomposition theorem and finally we show the simplifications made for the case of 2D objects. .

IV.4.3.3.2 The numerical solution to least-squares fitting of two 3D point sets

Obviously the following equation holds for each i less than m.

$$r_i = R \cdot p_i + T + N_i \qquad \text{(IV.38)}$$

Where Ni is the noise component.

In the same way we have operated in section IV.4.3.1, we can assume that Ni is a zero mean-value random variable. This assumption will allow to use a classical decoupling technique to separate the estimation of the rotation R from that of the translation T.

We define two variables p and r as the respective average values of $\{p_i\}_1^m$ and $\{r_i\}_1^m$:

$$p = (1/m) \sum_{i=1}^{m} p_i \qquad \text{(IV.39.a)}$$

$$r = (1/m) \sum_{i=1}^{m} r_i \qquad \text{(IV.39.b)}$$

Substituting (IV.38) in (IV.39.b) we have:

$$r = R \cdot p + T + (1/m) \sum_{i=1}^{m} N_i \qquad \text{(IV.40)}$$

Assuming Ni is a zero mean-value random process and m is large we can write:

$$r = R \cdot p + T \qquad \text{(IV.41).}$$

Let:

$$si = p_i - p \qquad \text{(IV.42.a)}$$

and

$$ti = r_i - r \qquad \text{(IV.42.b).}$$

If we substitute (IV.42.a) and (IV.42.b) in (IV.37) we have:

$$\sum = \sum_{i=1}^{m} \|(R \cdot p_i + T) - r_i\|^2$$

$$= \sum_{i=1}^{m} \|(R \cdot s_i + R.p + T) - t_i - r\|^2$$

Therefore:

$$\sum = \sum_{i=1}^{m} \| R \cdot s_i - t_i\|^2 \qquad \text{(IV.43).}$$

In this way we have reduced the estimation of R and T into two separate activities: first the estimation of R by minimizing IV.43, second the computation of T using the following equation derived from IV.41:

$$T = r - R.\ p \qquad \text{(IV.44).}$$

The minimization of IV.43 presented in [Aru80] uses the *singular value decomposition.*

Let us derive equation IV.43:

$$\sum = \sum_{i=1}^{m} \| R \cdot s_i - t_i \|^2$$

$$= \sum_{i=1}^{m} (R \cdot s_i - t_i)^T (R \cdot s_i - t_i)^T$$

$$= \sum_{i=1}^{m} s_i^T \cdot R^T \cdot R \cdot s_i - 2 \cdot s_i^T \cdot R^T \cdot t_i + t_i^T \cdot t_i$$

Since R is an orthogonal transformation we have:

$$R^T \cdot R = I$$

where I is the identity matrix. Therefore we can write:

$$\sum = \sum_{i=1}^{m} s_i^T \cdot s_i - 2 \cdot t_i^T \cdot R \cdot s_i + t_i^T \cdot t_i \cdot \qquad (IV.45)$$

Finally, finding R which minimizes $\sum$ is equivalent to maximizing:

$$\Delta = \sum_{i=1}^{m} t_i^T \cdot R \cdot s_i \qquad (IV.46).$$

It is easy to see that:

$$\Delta = \text{trace} \left(\sum_{i=1}^{m} R \cdot s_i \cdot t_i^T \right)$$

$$= \text{trace} \left(R \cdot \sum_{i=1}^{m} s_i \cdot t_i^T \right)$$

Therefore:

$$\Delta = \text{trace} (R \cdot Q) \qquad (IV.47)$$

where:

$$Q = \sum_{i=1}^{m} s_i . t_i^T \qquad \text{(IV.48).}$$

Q is called the **model-scene correlation matrix**. We will show in section IV.4.3.3.3 that there exist two orthonormal matrices U and V and a diagonal positive definite matrix Σ such that:

$$Q = U . \Sigma . V^T \qquad \text{(IV.49)}$$

The operation in (IV.49) is called *singular value decomposition* of the matrix Q (see section IV.4.3.3.3). Because Σ is diagonal, positive definite there exists a diagonal matrix B such that:

$$\Sigma = B . B^T \qquad \text{(IV.50)}$$

Therefore:

$$Q = (U.B) . (V.B)^T \qquad \text{(IV.51)}$$

Thus assuming that bi are the column vectors of B:

$$\begin{aligned} \text{trace}(R.Q) &= \text{trace}\{ ((R.U).B).(V.B)^T \} \\ &= \text{trace}\{ (V.B)^T((R.U).B) \} \\ &= \text{trace}\{ B^T.(V^T R.U).B \} \\ &= \sum_{i=1}^{m} b_i^T.(V^T R.U).b_i \qquad \text{(IV.52)} \\ &= \sum_{i=1}^{m} b_i^T.(V^T R.U.b_i) \qquad \text{(IV.53)} \end{aligned}$$

Applying Cauchy-Shwarz inequality to IV.53:

$$\text{trace}(R.Q) <= \sum_{i=1}^{m} \|b_i\|.\|V^T R.U.b_i\| \qquad (IV.54)$$

It is clear that $V^T R.U$ is an orthonormal matrix and therefore we can write:

$$\|b_i\| \, . \, \|V^T R.U.b_i\| = \|b_i\|^2 \qquad (IV.55)$$

Using (IV.53), (IV.54) and (IV.55) we finally can write:

$$\text{trace}(R.Q) = \sum_{i=1}^{m} b_i^T.(V^T R.U).b_i <= \sum b_i^T.b_i \quad (IV.56).$$

Equation (IV.56) shows that trace(R.Q) is bounded and that it reaches its maximum when:

$$V^T R.U = I$$

or:

$$R = V.U^T \qquad (IV.57).$$

Therefore the least-squares fitting of two 3D point sets is solved using algorithm IV.2.

Algorithm IV.2.

1) Compute p,r using V.3.3.2.2 and si, ti using V.3.3.2.5.
2) Compute Q using V.3.3.2.11 and perform its singular value decomposition (see section V.3.3.3).
3) Compute the rotation R using V.3.3.2.20.
4) Compute T using V.3.3.2.7.
5) End.

In the following section we will show how to compute the singular value decomposition of the Q matrix .

IV.4.3.3.3. Singular value decomposition of the model-scene correlation matrix

In this section we are first going to state and prove the singular value decomposition theorem [Grif89] and then decompose, for the particular case of planar objects, the model-scene correlation matrix introduced in section IV.4.3.3.2.

Note that the aim of carrying out the proof of the singular value decomposition theorem is that, through this proof, we can show the plan of matrix decomposition calculation and thus be capable to determine the form the decomposed matrix Q will take in the particular case of planar objects.

IV.4.3.3.3.1: The singular value decomposition theorem.

Let A be any n x n matrix. The square roots of the nonzero eigenvalues of $(A^T.A)$ are called **singular values** of A. Let v be a normalized eigenvector of $(A^T.A)$ belonging to a nonzero eigenvalue μ. We can write $\|A.v\|^2 = \mu$. Hence $(A.v/\sqrt{\mu})$is a normalized vector. Also $(A.A^T).A.v=A.(A^T.A.v)=\mu.A.v$. Consequently, $(A.v/\sqrt{\mu})$ is a **normalized eigenvector of** $(A.A^T)$.

The singular value decomposition is based on the property stated above. The result is a generalization of the $(P.D.P^T)$ factorization of a symmetric matrix.

Theorem (The singular value decomposition theorem):

Let A be an (n X n) matrix of rank r. Then there are U and V such that:

$$A = U.\Sigma.V^T \qquad \text{(IV.58)}.$$

where Σ is an (n x n) diagonal matrix with $\Sigma_{11},\ldots,\Sigma_{rr}$ the r singular values of A, and all other entries zero.

Proof.

$(A^T.A)$ is a symmetric matrix of rank r and has r orthonormal eigenvectors $v_1, v_2, \ldots, v_r$ belonging to the nonzero eigenvalues $\mu_1, \mu_2, \ldots, \mu_r$ and (n-r) eigenvectors belonging to the zero eigenvalue, giving an orthonormal set of eigenvectors $v_1, v_2, \ldots, v_n$. Let V be the (n x n) unitary matrix of columns $v_1, v_2, \ldots, v_n$.

Define $u_1, u_2, \ldots, u_r$ by:

$$u_j = A.v_j/\sqrt{\mu_j} \quad \text{for } j = 1,\ldots,r.$$

As explained in the beginning of this section the u's are normalized eigenvectors of $(A.A^T)$. Since $(A.A^T)$ is symmetric these vectors are orthogonal. $(u_1, u_2, \ldots, u_r)$ is extended to an orthonormal basis $(u_1, u_2, \ldots, u_r, \ldots, u_n)$. At this point it is easy to see that:

$$(U^T.A.V)_{ij} = \sqrt{\mu_j}.\partial_{ij}. \quad \text{for } 1 <= j <= n.$$

With ∂_{ij} the Kronecker symbol. This shows that: $A = U.\Sigma.V^T$.

IV.4.3.3.3.2: The singular value decomposition of the correlation matrix when the object is planar:

As mentioned in section IV.4.1 we have the choice to take any 3D position of the object model. Obviously, it is preferable to take the one that offers the most suitable computations. We assume the object model is in the (O,x,y) plane of the camera frame (see Fig IV.5). In this way, the model points $\{p_i\}_1^m$ have their z coordinate null. Hence, from IV.39 and IV.42, we see that all $\{s_i\}_1^m$ have their z coordinate null. Following this, the model-scene correlation matrix, using IV.48, has the following form.

$$Q = \sum_{i=1}^{m} s_i . t_i^T$$

$$= \sum_{i=1}^{m} \begin{pmatrix} Sx_i \\ Sy_i \\ 0 \end{pmatrix} (tx_i ty_i tz_i)$$

$$= \begin{pmatrix} X & X & X \\ X & X & X \\ 0 & 0 & 0 \end{pmatrix} \qquad \text{(IV.59).}$$

Equation IV.59 shows that the model-scene intercorrelation matrix has its third row null. This simplifies the singular value decomposition of Q considerably. Note that matrix Q has its third row null because the third coordinates of all si's are null; this explains the 3D model position we have chosen above.

We have to write Q in the form mentioned in (IV.49):

$$Q = U . \Sigma . V^T.$$

To be capable to calculate U and V, we have to follow the plan indicated in section IV.4.3.3.1: calculate $Q^T.Q$, compute its eigenvalues and eigenvectors and make V the unitary matrix having these eigenvectors as columns. The U matrix is also defined by its columns which are equal to the product by Q of each column of V divided by the square root of the eigenvalue of $Q^T.Q$ this column of V belongs to.

A simple matrix multiplication shows that $(Q^T.Q)$ is a full matrix. This makes the computation of its eigenvalues more complicated. However, $Q.Q^T$ has its third row and third column null. Therefore, we will proceed by carrying out the singular value decomposition of Q^T and then, by simply interchanging U and V matrices we have the singular value decomposition of Q. We give in the following the results of the decomposition, the development being fairly simple (solving a second degree equation and a second order linear system).

Let:

$$Q.Q^T = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \text{(IV.60).}$$

We have:

$$U=[u1\ u2\ u3] \qquad \text{(IV.61).}$$

with:

$$u1= \left[\frac{c}{\sqrt{c^2+e^2}}, \frac{-e}{\sqrt{c^2+e^2}}, 0 \right] \qquad \text{(IV.62.a).}$$

$$u2= \left[\frac{e}{\sqrt{c^2+e^2}}, \frac{c}{\sqrt{c^2+e^2}}, 0 \right] \qquad \text{(IV.62.b).}$$

$$u3= [0,0,1] \qquad \text{(IV.62.c)}$$

where:

$$e = \frac{d-a-\sqrt{\Delta}}{2} \qquad \text{(IV.63)}$$

and

$$\Delta = (d-a)^2 + 4\,b.c \qquad \text{(IV.64).}$$

As for V:

$$V=[v1\ \ v2\ \ v3] \qquad \text{(IV.65).}$$

We have

$$v1 = (Q^T .u1)/\sqrt{e+a} \qquad \text{(IV.66.a).}$$
$$v2 = (Q^T .u2)/\sqrt{d-e} \qquad \text{(IV.66.b).}$$
$$v3 = v1 \times v2 \qquad \text{(IV.66.c)}$$

where (x) is the vector product between two vectors.

Hence, we see that the singular value decomposition of the model-scene intercorrelation matrix, in the case of planar objects, is a simple task of which computation should not be time consuming.

V) CONCLUSION

In this chapter were discussed some geometrical problems related to locating a window of consecutive binary symbol that permits recovery of the absolute position of the AGV. It was shown that, to avoid errors in placing such a window, it is necessary to

determine the 3D position of each binary symbol appearing on the image.

Two solutions to 3D binary symbol positioning have been discussed. The first solution, the flat-earth technique, was originally used in AGV road following. This technique assumes that the guide-path permanently belongs to an even plane. The second technique proposed is more robust and makes no assumption about the evenness of the guide-path. In this technique, an iterative method is used to solve the pose problem for each code appearing in the image. The second method is more time consuming but has a better robustness.

CHAPTER V
SOFTWARE IMPLEMENTATION AND EXPERIMENTAL RESULTS

V.1 INTRODUCTION

In this chapter, problems related to the software implementation are discussed. The algorithms described in chapters III and IV can be considerably memory and time consuming if the appropriate data types and structures are not used. Such problems are important and sometimes not easy to solve. Because the programs manipulate huge amounts of data (hundreds of pixels in an object boundary for instance) problems such as memory fragmentation, memory saturation etc., can easily occur. Also, considering the fact that the pixel coordinates in the image are integer numbers, the computations should be reduced, as much as possible, to integer type operations in order to speed up the program execution.

This chapter is structured as follows: section V.2 deals with the software implementation and section V.3 presents some experimental results.

V.2 SOFTWARE IMPLEMENTATION

In this section problems related to software implementation are discussed. Each module of the software is independently discussed.

The main module

This module contains the "main" program. Contour tracing, polygon approximation, corner-vertex extraction, 3D pose estimation and binary window positioning routines are called from this module. The image is scanned left to right-top to bottom and every time an object is encountered the contour tracing routine is executed. Only objects having a number of boundary pixels ranging between 50 and 250 are eligible for further processing. This avoids processing objects which, because of their size, can not be binary symbols. Polygon approximation, corner-vertex extraction and binary symbol recognition are then performed. 3D position estimation is carried out in the case the object is recognized as being a binary symbol. Once the bottom right pixel of the image is reached the list of the recognized binary symbols is sorted in the order they follow each other on the guide-path.

Situations where a new object is encountered are characterized by the pattern "black pixel followed by a white pixel". The white pixel in this case, as far as the contour tracer routine is concerned, constitutes the initial boundary pixel.

The scanning of the image is performed by using the auto increment register of the frame board. The image is read by block transfer using C interfacing commands. In this way the transfer is done fast.

The image segmentation by thresholding module

This module provides the threshold selection and the image segmentation by thresholding routines.

The selected threshold is the grey-level value corresponding to the local minimum that separates the two intensity modes on the grey level image histogram. The image is thresholded by initializing the lookup table. The latter is a part of the frame board device that maps the incoming data (the grey level pixel values) to values set up by the user. In this way the thresholding operation is almost instantaneous.

The boundary tracing module

This algorithm has been described in detail in section III.3.2. Note that boundary points are memorized in the form of a stack of chain code values. Each boundary pixel is represented by the chain code value between this pixel and the one coming next on the boundary. This method permits saving a lot of memory space. The chain code values in the stack follow the order of their corresponding pixels on the boundary.

The polygon approximation module

This module includes the two steps involved in the polygonal approximation scheme we have adopted: scan-along and split and merge polygon approximation, both using area deviation .

The polygon segments are stored in a double linked list of type POLY. The type POLY is a structure which includes the following elements:

- the coordinates (x_1,y_1) and (x_2,y_2) of the endpoints P_1 and P_2 of the polygon segment S_i;
- index i in the boundary stack of the polygon vertex P_1;
- length L_i of the segment S_i;

- area deviation A_i along the segment Si:
- segment error E_i along S_i which is equal to the absolute value of the ratio between A_i and L_i.
- integer flag indicating the most likely search direction to take when adjusting the vertex position.
- integer indicating the index of the vertex in the list.

In the scan-along algorithm, the first pixel on the contour is generated as a vertex. Then in a "while" loop the program traverses the contour and calculates the ratio area deviation per segment length. Note that instead of testing:

$$\frac{A_i}{L_i} < T$$

where A_i is the area deviation, L_i the segment length and T is the upper limit for are deviation per segment length the following test is actually used:

$$A_i^2 < L_i^2 \cdot T^2$$

This speeds up the performance of the vertex generation test. In fact, as mentioned in section III.3.3.3 the square of the segment length is computed incrementally and therefore we have access to L_i^2 rather than L_i. On the other hand calculating a product is faster than calculating a ratio.

In the case of the split and merge algorithm three basic operations are performed: split, merge and vertex adjustment. The total error norm we take, as mentioned in section III.3.3.4.b, is equal to the fourth of the pointwise maximum error norm. In the case of our application the maximum pointwise error is equal to one. This means we don't allow the contour points to be more than one pixel away from the approximating segment..

During the split operation the segment having the maximum error norm is split into two. The new vertex generated is halfway, along the boundary, between the endpoints of the segment to split. After the split has been performed the error norms along the new segments are calculated using the incremental properties of area deviation and segment length.

During the merge operation, for each segment i, the value Gi the total error norm would take if segments i and i+1 were merged is evaluated. The segment list is sorted in the decreasing order of the Gi's using the quick sort algorithm.

As regards vertex adjustment, a flag is used for each vertex for the sake of speeding up the adjustment process. This flag indicates the direction in which the vertex has been moved during its last position adjustment. In practice, for the same vertex, this direction seldom changes. In this way, by checking the flag value, only one test, instead of two, can be performed to adjust the vertex position.

Note that error norm updating after a split, a merge or a vertex adjustment operation, when a classical error norm (the maximum pointwise or the integral square error norm) is used is time consuming. In fact error norm updating constitutes the basic source of time consumption for the split and merge algorithm. In our case, this operation is carried out fast because of the recursive computation of area deviation and segment length.

The corner-vertex extraction module

This module includes three basic routines: polygon decomposition, trusion parsing and the semantic analysis.

Polygon decomposition serves to represent the object as a union of subsets (clusters). The shape of the latter may be simpler

and therefore some of the simpler descriptors will be applicable. The method used, in our case, consists of the decomposition on the basis of convexcity: pairs of concave vertices are related. Therefore the boundary will be divided into substrings consisting of successive vertices of the same angle sign.

For each cluster the trusion parser described in III.D is applied. This parser permits to locate the real vertices on the object boundary. As mentioned in III.D the parser is a finite automaton with a buffer of size four and using the "shift reduce" parsing. The bound of the buffer size is restricted on the production of BREAK. We assume that the decision can be made whether a line is short or long depending not only on the current line but also on the contents of the buffer. The automaton is implemented by an algorithm (the function trus()) where a "shift" (see section III.D) is replaced by a *recursive call* to the algorithm it self. This calls a line detector routine and, depending on the state, examines the length of its return. The functions of "reduce" are performed by the subroutine sflush() (see appendix II) which also performs a number of semantic checks. The semantic analyzer sflush() decides if the pattern detected is a corner or a line. In this way real vertices are extracted from the polygonal representation of the boundary.

In our case, since the symbols 'E' and 'H' have right angle corners, we take the parameter COLANG (the upper limit of the angle between two collinear segments) equal to 45°. The parameter BREAK, the maximum length for a BREAK segment is taken equal to 5.5.

The symbol recognition module

In this module symbol recognition is carried out using a tree search. The segment list is sorted in the decreasing order of the segment lengths. The quick sort algorithm is used for this purpose. After this, the ratio between the length of the second long segment

and the that of the longest segment is computed. In the case this ratio is greater to 0.65 the symbol is eligible to be recognized as the symbol 'H' and other segment length features are checked. In the case this ratio is smaller than 0.65 the object is eligible to be recognized as the symbol 'E' and other segment length features are checked.

The model-scene point matching module

In this module, model and scene points are matched using the geometric hashing technique. The matched model and scene points are used for the further determination of the 3D position of the symbol.

The model and scene reference frames for the symbols 'E' and 'H' are defined as indicated on Fig V.1 and Fig V.2. The choice of these frames is based on the fact that the segments $[p_0 , p_1]$ and $[p_0 , p_2]$ are the less noise affected (as far as their length and orientation are concerned) boundary segments. In other words the model and scene frame is required to have noise robustness.

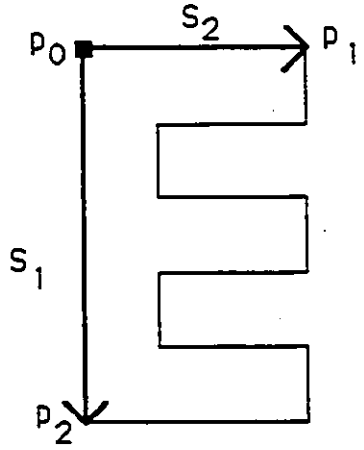


Fig V.1. Frame defintion for the character 'E'.

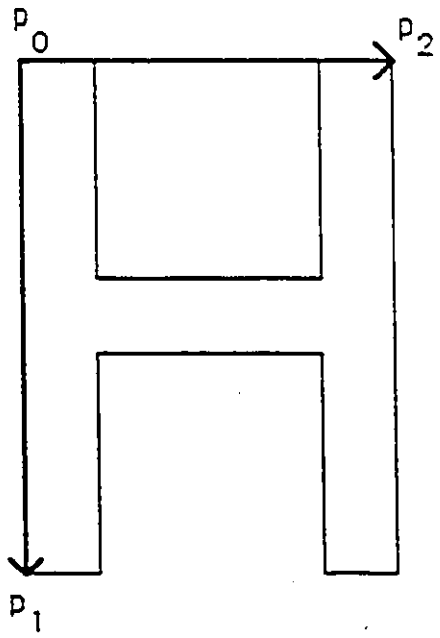


Fig V.2. Frame defintion for the character 'H'.

For instance, Fig V.3 shows the worst case where the image of the symbol 'E' can be distorted. Note that, despite the considerable amount of noise affecting the image, the orientation and length of each one of the segments $[p_0 . p_1]$ and $[p_0 . p_2]$ have remained almost unchanged.

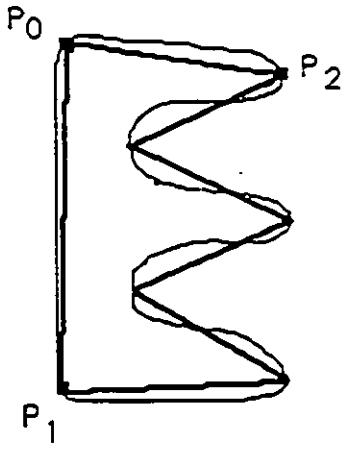


Fig V.3. Worst case of image distortion for the character 'E'.

Note that locating p_0, p_1 and p_2 in the scene is a simple task. Either in the case of the symbol 'H' or the symbol 'E', p_0, p_1 and p_2 are the endpoints of the two longest segments of the boundary polygon.

After the scene frame has been determined, the coordinates of all the scene vertices are calculated in this frame. For each scene vertex, the model vertex having the closest coordinates in the model frame is determined and the resulting pair of vertices is added to the list of matching model and scene point pairs.

The model based 3D position estimation module

This module deals with model based pose estimation as presented in section IV.4.3. Conformably to algorithm IV.1 the program mainly involves the computation of the model-scene intercorrelation matrix, its singular value decomposition and the updating of the depth of each point. The singular value decomposition is done fast by applying the simplifications introduced in section IV.4.3.3.3.2. In fact, using these simplifications, the computation of the singular value decomposition of the model-scene intercorrelation matrix becomes straightforward and no iterative algorithm is required.

Because the initial values of the point depths are fairly close to the solution, the algorithm converges within less than five iterations. At the end of the program the 3D coordinates of the symbol centroid are computed and stored in a double linked list.

3D binary symbol position determination using the flat earth geometric model

As shown in section IV.3 this method assumes the guide-path belongs to a planar surface. The algorithm is based upon equation IV.7. Note that model-scene point matching is not required. This method is straightforward and therefore faster than the model based 3D position estimation method. However, since no estimation is made, the error in the output 3D locations is only a function of the extent to which the flat-earth assumption is violated. Moreover, the camera is in a fixed position relative to the body of the AGV, but the body of the vehicle is able to rock forward and backward on the undercarriage.

Pseudo-random window positioning

Positioning the pseudo-random binary window as well as tracing the 3D profile of the guide-path in front of the vehicle requires sorting the list of the recognized codes in the order they follow each other on the guide-path.

In the sequel a search technique that solves this problem is presented. This method is capable of sorting the binary codes even when the path in front of the vehicle is not linear.

Sorting the list of binary codes is achieved by finding, for each code in the list, the neighboring codes on the guide-path. Two types of neighbors can defined: *first order* and *second order neighbors*. Two binary symbols are *first order neighbors* if the distance that separates them is equal to the constant distance $d0$ that separates the codes along the guide-path. On the other hand, two binary

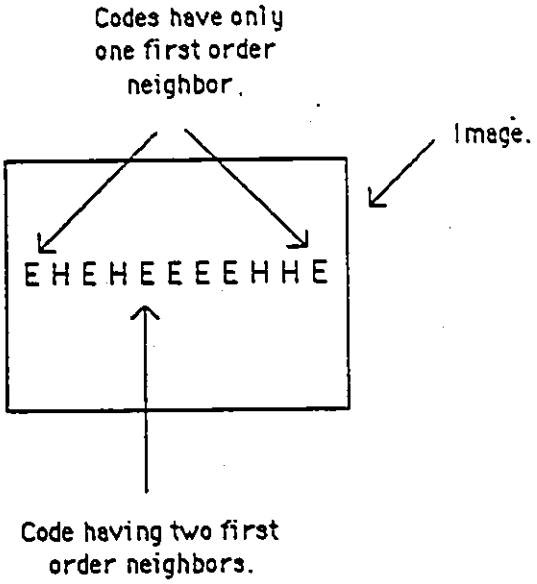


Fig V.4. First order neighbors.

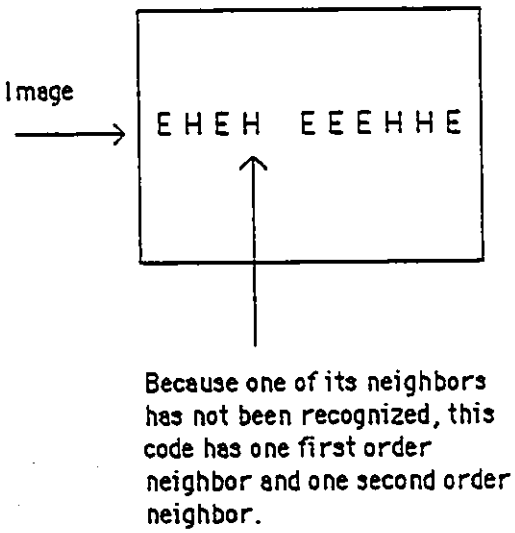


Fig V.5 Second order neighbors.

symbols are *second order neighbors* if the distance that separates them is greater than *d0* .

Note that in the ideal case where the recognition process is a hundred percent accurate, all binary symbols, except the two on the border of the image, have two first order neighbors (see Fig V.4). In the case where certain binary symbols have not been recognized, some codes, other than the ones on the border of the image, will not have two first order neighbors (see fig V.5). Second order neighbors permit to determine the pseudo-random window position.

The sorting program sorts the binary symbol list in the order they follow each other on the guide-path. In this list binary codes are represented by a structure defined as follows:

```
Structure BINARY-CODE
{
- Type of code ('E' or 'H').
- Coordinates in 3D space of the code centroid.
- Order of neighborhood ( zero one or two).
- Pointer to the next element in the list.
- Pointer to prior element in the list.
}
```

The element "Type of code" in the structure indicates if the code in question is 'E' or 'H'. The element "Order of neighborhood" is initialized to zero. When a neighbor of the code is found, depending on the order of this neighbor, the element "Order of neighborhood" is set to one or to two. The two pointers "next" and "prior" define the rank of the code in the double liked list.

Algorithm V.1 describes the sorting scheme. Since the two ends of the visible portion of the path appear in the image as the intersections of the path with the image border, the top of the double linked list will be the closest binary code to the border of the image. The algorithm steps from binary code to neighbor while marking

each traversed code by setting the element "Order of neighborhood" to either one or two. The marking operation avoids considering the same code as a neighbor more then once. The routine "Find-neighbor" finds neighbors by searching for the closest, unmarked, binary code.

ALGORITHM V.1

FUNCTION **Binary-Code-Sort** ()

{

1) Find the closest code B to the border of the image, and define B as being the top element of the double linked list. Set order of neighborhood of B equal 1.

2)while ((B1= find-neighbor(B)) not null)
{
Add B1 to the bottom of the list and set B to B1.
}

}

FUNCTION **Find-neighbor** (C)

{

1) Find Ci, for which the flag is equal to zero, that minimizes the absolute value of d-d0; d being the distance between C and Ci and d0 a constant indicating the normal distance between two neighboring codes on the track.

2) If d-d0 is close to zero set order of neighborhood of Ci to 1. Else set it to 2.

}

Once the binary codes sorted in the order they follow on the guide-path, it becomes easy to trace the 3D profile of the track in front of the vehicle and to position the pseudo-random binary window. The reconstructed visible part of the track will be the piece-wise linear 3D curve which joints the centroids of the successive binary codes.

As for pseudo-random binary window positioning, the search is based on checking the "Order of neighborhood" of each element in the list and finding n successive binary symbols all having neighbors of order 1 (algorithm V.2 explains it principle).

ALGORITHM V.2

FUNCTION **Binary-Window-Positioning** ()

{
1) Go to the top of the binary code list and set C to the corresponding code; set counter to zero.

2) While (bottom of list not reached)

{

2.1) if the order of neighborhood of the code is equal to 1 increment counter. Else set counter to zero and C to the current code.

2.2) if counter is equal *n* take binary window equal codes from C to the current code. exit loop.

2.3) go to next code in the list.

}

}

V.3 EXPERIMENTAL RESULTS

The results discussed in the sequel have been presented in [Kha90].

Fig 6 shows an image of the guide-path taken by the camera. Obviously the binary symbols further away from the camera look smaller in the image. Fig V.7 shows that the system has recognized the binary symbols along the guide-path and generated the binary numbers they correspond to ('E.' corresponds to 1 and 'H' to 0). Note that even the far away symbols from the camera have been recognized. Fig V.8 shows the reconstructed 3D profile of the guide-path. Because the symbol 'E' is asymmetric it is possible to determine the orientation of the guide-path . In Fig V.8, the arrows indicate the orientation of the guide-path. The order of the pseudo-random sequence in the case of our application is equal to eight. Note that twenty four binary symbols have been recognized. In this case we have 16 redundant codes. Because all visible binary symbols have been recognized, positioning the pseudo-random window is a simple task in this case.

Fig V.9 shows the case where an obstacle hides a portion of the guide-path. Fig V.10 shows the recognized binary symbols. Note that one visible binary symbol has not been recognized. This is due to the

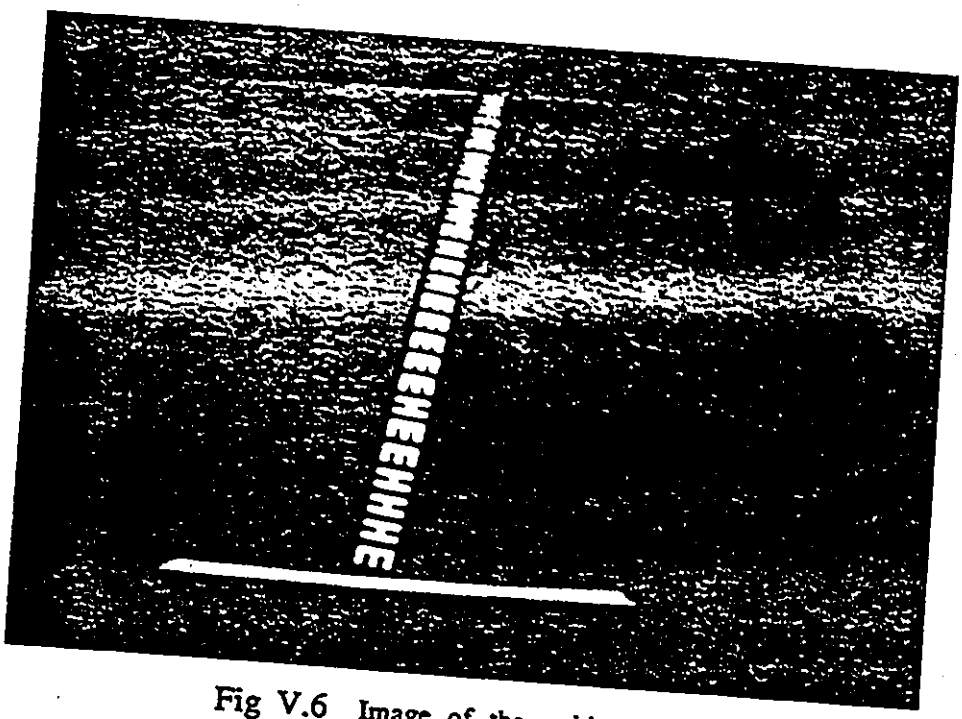


Fig V.6 Image of the guide path.

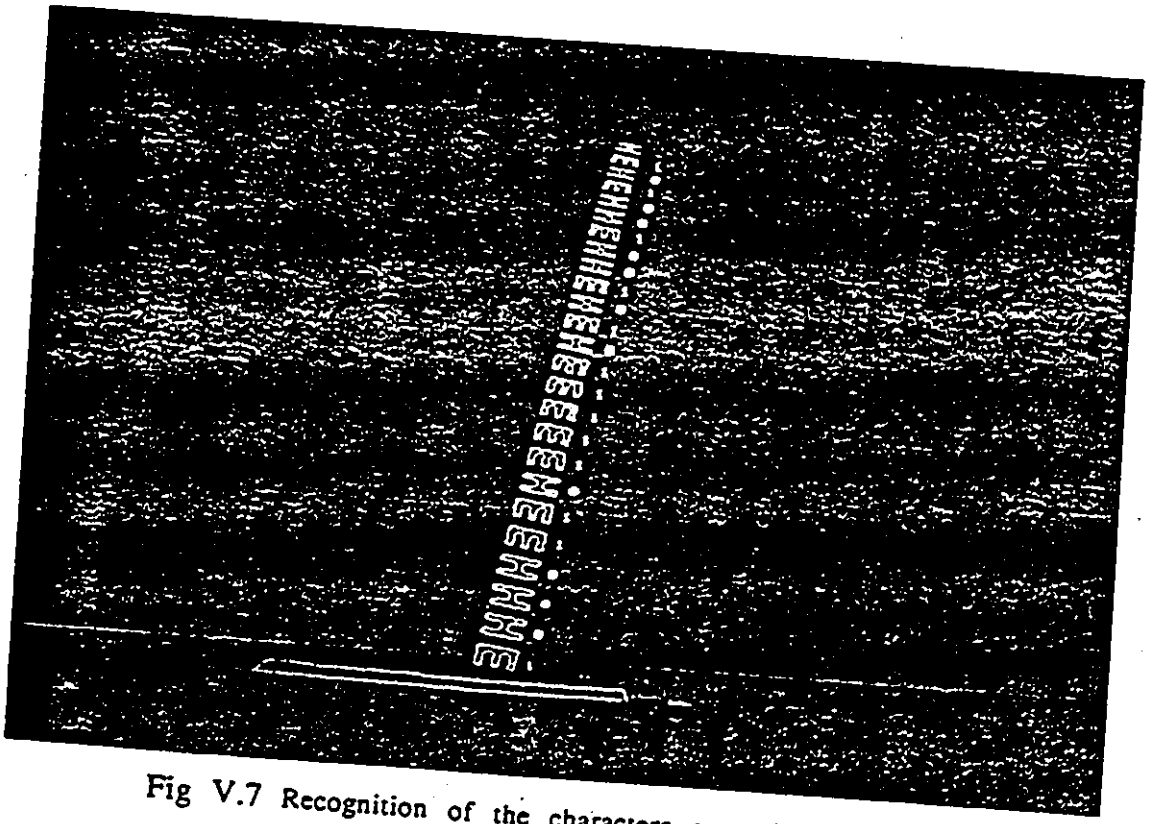


Fig V.7 Recognition of the characters appearing in Fig V.6.

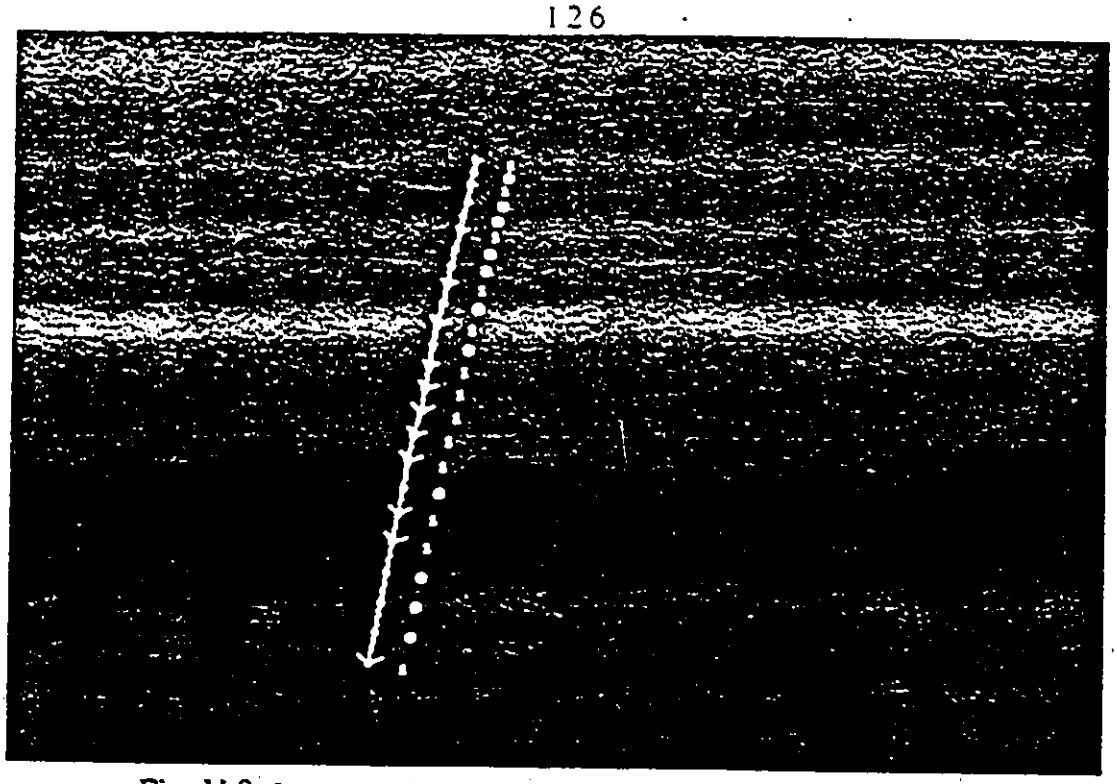


Fig V.8 Reconstruction of the guide-path conform to Fig V.6.

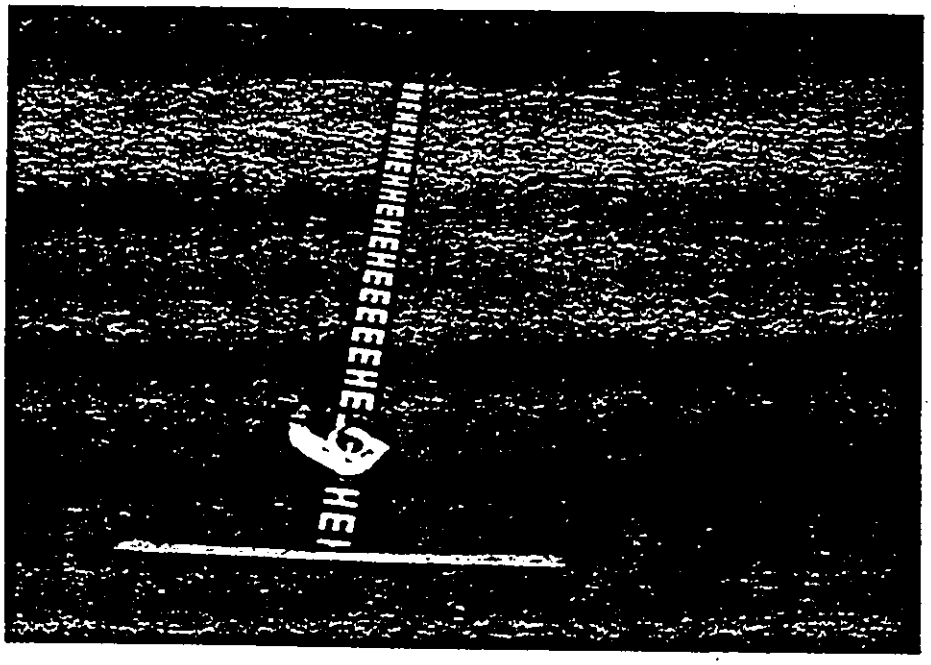


Fig V.9 Guide-path with a hidden portion.

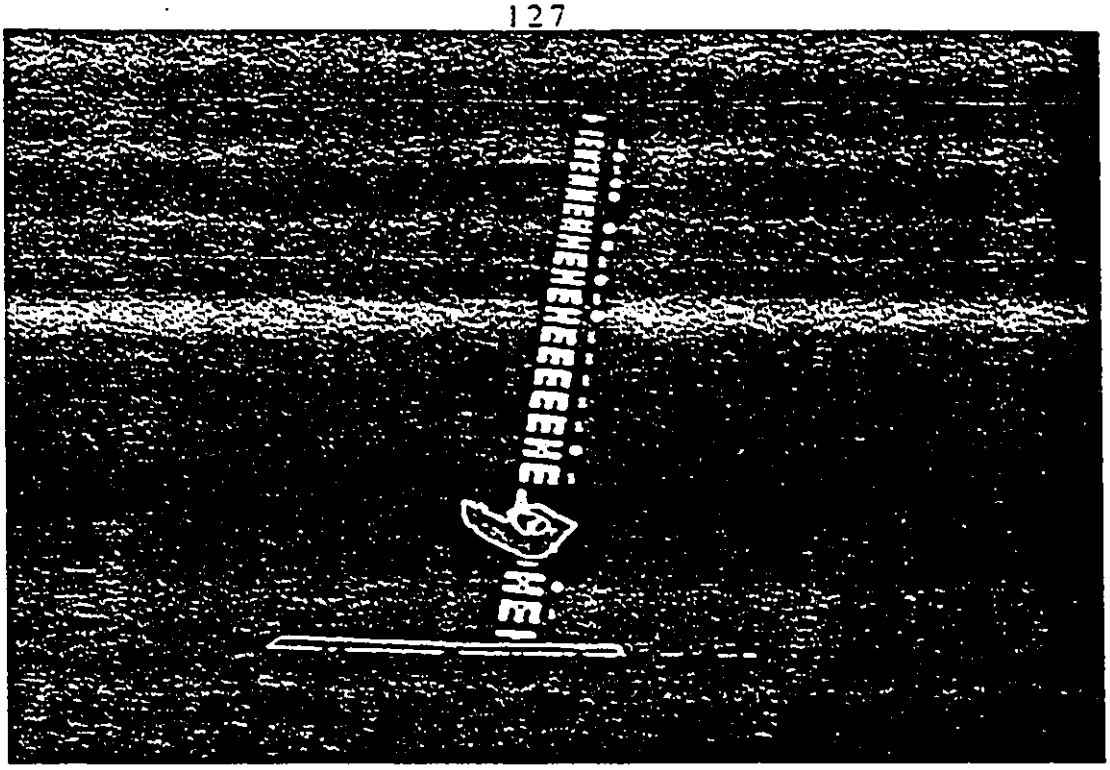


Fig V.10 Recognition of the characters appearing in V.9.

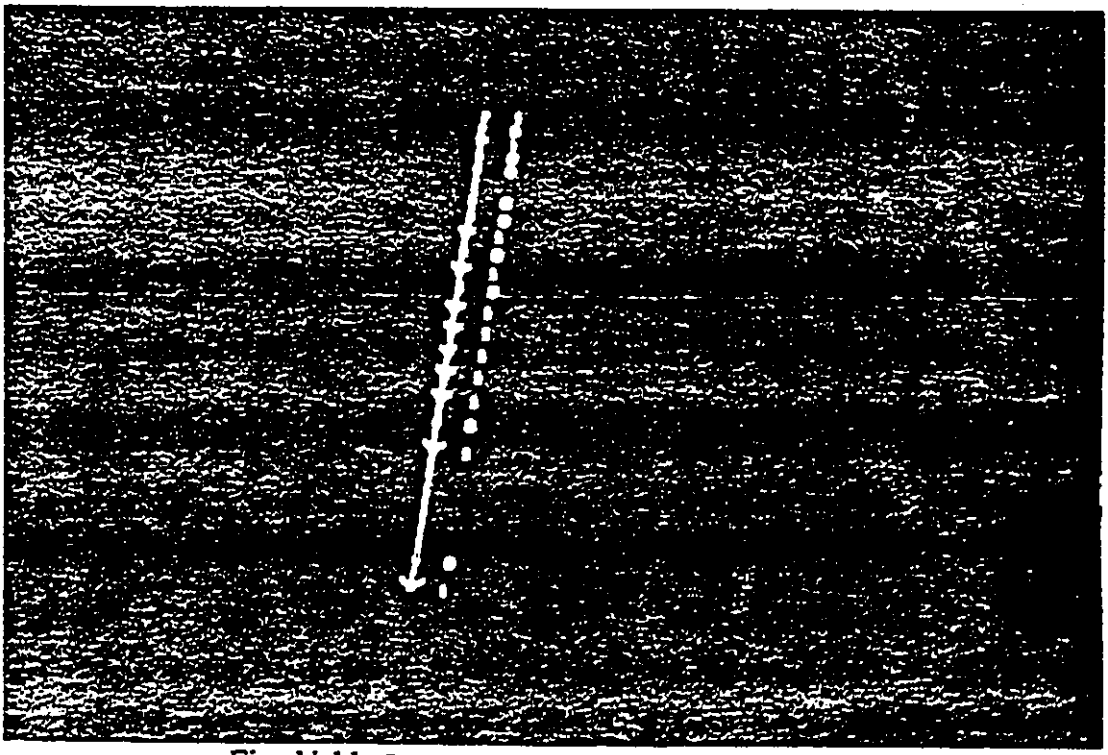


Fig V.11 Reconstruction of the guide-path in V.9.

fact that the presence of the obstacle changes the grey level hystogram distribution and, because the thresholding method used is global, affects the recognition process. Fig V.11 shows the 3D reconstruction of the guide-path. Four binary symbols, among the ones recognized, have second order neighbors. Nevertheless, we can find thirteen consecutive binary symbols that have first order neighborhood. Hence the pseudo-random window can be located.

CONCLUSION

In this chapter problems related to the software implementation have been discussed. We have shown how simplifications as well as special data types can be used to speed up the program execution as well as to reduce its memory consumption. In our belief not much improvement in time processing can be made by trying to refine this software. We think most of the time consumption originates in the data acquisition part of the processing, that is the early image processing stage. A faster image acquisition system is definitely required if better time performances are needed.

CHAPTER VI

CONCLUSION

Automatic visual guidance, nowadays, has become a classical problem in AGV navigation. Quiet performing road as well as white line following systems have been designed [Dick90]. However, the problem of automatic AGV positioning has seldom been addressed in the literature. The first implementation of an efficient position measurement system for AGVs was presented in [Petr89]. However, in this implementation the navigation problem has not been considered.

A new AGV navigation system having, both, self positioning and automatic guidance abilities was the *subject of this thesis*. The vehicle visually follows a guide-path made of a sequence of binary codes represented by two geometric shapes. The binary symbols on the guide-path follow the order of a pseudo-random binary sequence. Such a sequence has the so called *window* property which permits recovery of the vehicle absolute position along the code track. Every time there is a need for absolute position measurement, a camera mounted on the AGV takes a snapshot of the guide-path. Assuming n is the order of the PRBS, among the codes appearing in the image, n binary symbols are grouped together to make an n-tuple which uniquely identifies the AGV position along the track.

Descriptions of the image analysis and pattern recognition programs that permit binary symbol recognition have been undertaken. The code recognition task consists of the original image by thresholding, tracing the boundary of each object in the image, approximating each boundary by a polygon, extracting vertices and

corners on each contour and finally recognizing the object using a tree search.

An original implementation of the split and merge polygon approximation, which improves the run-time performance of the algorithm, was presented. It consists of using area deviation per segment length as an error norm. The mathematical proof of the validity of such an error norm was carried out.

For the sake of a high encoding resolution, the camera has to cover a large number of codes in front of the vehicle. This causes the binary codes to have smaller sizes in the image and consequently makes code recognition more difficult. For this reason we went to robust techniques, the syntactic corner-vertex extraction algorithm for instance, although they may be more time consuming than other, more simple, techniques.

To be capable of locating the pseudo-random window permitting absolute position measurement, it is necessary to compute the 3D location and orientation (*pose*) of each binary code in the image. This so called the *inverse-perspective* problem, has been discussed in detail. A review of the existing solutions as well as the different engineering applications where the problem is encountered has been made. Note that in all pose estimation problems, an a priori recognition of the object under inspection and its reference to a specific model are necessary. Two methods have been discussed to solve the inverse-perspective problem. The first, called the *flat-earth* geometric model method, is straightforward: it assumes that the guide-path lies on a planar surface. The second technique is more general; it makes no assumption about the evenness of the surface of the floor and estimates iteratively, using a least square algorithm, the 3D position and orientation of each code. This second technique requires matching a number of points in the scene with a number of corresponding points in the model. This operation is made using a recent structural matching technique called *geometric hashing*.

Once the 3D position of each binary symbols has been estimated, tracing the 3D profile of the visible portion of the path in front of the vehicle becomes an easy task. The reconstructed visible part of the track is the piece-wise linear 3D curve which joints the centroids of all the binary codes.

In the experimental part examples of the system running have been shown. Situations where the system can be induced in error have been tested. We have tested, for instance, the case where an obstacle hides a portion of the track. In that case, because the camera covers a sufficient number of redundant codes, the system has been able to recover its position.

A great deal of research is still to be made to improve the performance of the system:

- implementation of an adaptive thresholding technique. Because the camera covers a large surface of the floor, it happens that the light intensity changes dramatically within the same image and therefore a variable threshold value is necessary;

- investigation of the possibility of a parallel implementation of the system;

- study of the problem of obstacle avoidance. A few visual obstacle avoidance methods have been proposed in the literature.[Dick90], [Vey86].

- design of a steering controller that uses the information produced by the system proposed.

- use of the same constructions to obtain pseudo-random sequences and arrays with entries which, instead of being 0's and 1's, are taken from the alphabet of q symbols. In

that case more then two symbols will have to be recognized by the visual system.

BIBLIOGRAPHY

[Arun80] K. S. Arun, T. S. Huang and S. D. Blostein, " Least-squares fitting of two 3-D point sets". *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 9, No. 5, pp. 698-700, Sept.1980.

[Dav77] L. S. Davis, " Understanding shape: Angles and sides." *IEEE Trans. Comput.*, Vol 26, No 3, pp 236-242, June 1977.

[Dav79] L. S. Davis, " Shape matching using relaxation techniques." *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 1, No. 1, pp. 60-72, Jan.1979.

[Dav88] L. S. Davis, S. Linnainmaa, D Harwood, " Pose determination of a 3D object using triangle pairs," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 5, pp. 634-647, Sept.1988.

[Del81] R. Deliban, D. G. Lieby, " Vehicle guidance system employing radio blocking," US Patent 4 287 160, Aug. 18, 1981.

[Dho89] M. Dhome, M. Richetin, J Lapreste, G Rives, " Determination of the attitude of 3D objects from a single perspective view," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 11, No. 12, pp. 1265-1278, Dec.1989.

[Dick90] E.D Dickmanns, B. Mysliwetz, T. Christians, " An integrated spatio-temporal approach to automatic visual guidance of autonomous vehicles," *IEEE Trans. Syst. Man & Sybernetics* , Vol 20, No. 6, pp 1273-1284, Dec.1990.

[Dun86] J. G. Dunham, " Optimum uniform piecewise linear approximation of planar curves," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 8, No. 1, pp 67-75, jan.1986.

[Fab88] T. L. Faber, E. M. stokely, " Orientation of 3D structures in medical images," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 5, pp. 626-633, Sept.1988.

[Faug86] O. D. Faugeras, N. Ayache, " HYPER: A new approach for the recognition and positioning of two-dimensional objects," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 8, No. 1, pp. 44-54, Jan.1986.

[Faug90] O. D. Faugeras, Y. Liu, T. S. Huang, " Determination of camera location from 2D to 3D line and point correspondence," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 12, No. 1, pp. 28-37, Jan.1990.

[Fish81] M. A. Fishler, R. C. Bolles, " Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, Vol 24, NO 6, pp. 381-395, June 1981.

[Fu74] K. S. Fu, *Syntactic Methods in Pattern Recognitionl*. New York: academic, 1974.

[Fu87] K. S. Fu, R. C. Gonzalez, C. S. G. Lee, *Robotics: Control, Sensing, Vision and Intelligence*. Mc Graw-Hill, 1987.

[Gav90] F. C. A. Groen, D. M. Gavrila, " 3D object recognition from 2D images using geometric hashing," Ma.Sc. Dissertation, Dep. Comput. Scie., Vrije Universiteit, Amsterdam, The Netherlands,1990.

[Gon77] R. C. Gonzalez, P. Wintz, *Digital Image Processing*. Massachusets, 1977.

[Grif89] D. H. Griffel, *Linear Algebra and its Applications*. Vol 2, England, 1989.

[Groe85] F. C. A. Groen , A. C. Sanderson, J. F. schlag, " Symbol recognition in electrical diagrams using probabilistic graph matching" *Pattern Recognition Letters*, Vol 3, pp. 343-350, Sept 1985.

[Hon87] T. Hongo et al., " An automatic guidance system of a self-controlled vehicle," *IEEE Trans. Ind. Electron*, Vol 34, No 1, pp 5-10, Feb. 1987.

[Ishi88] S. Ishikawa, H. Kuwamato, S. Ozawa, " Visual navigation of an autonomous vehicle using white line recognition," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 5, pp 743-749, Sep. 1988.

[Joo88] R. M. Haralick and H Joo, " 2D and 3D pose estimation," CH2614-6 IEEE, pp. 385-391, 1988.

[Kab87] M. R. Kabuka, A. E. Arenas, " Position verification of a mobile robot using a standard pattern," *IEEE J. Robotics Automation*, Vol 3, No 6, pp 505-516, Dec. 1987.

[Kak88] K. L. Boyer, A.C Kak, " Structural stereopsis for 3-D vision," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 2, pp 144-166, Mars1988.

[Kha90] E. Petriu, H. Khalfallah, J. Basran, " Visual navigation for atomated guided vehicles in an encoded environment," *Proc. VISION '90 SME-Conference*, Detroit, V.S.A, Nov. 1990.

[Kur81] Y. Kurozumi and W. Davis, " Polygonal approximation by the minmax method," *Computer Graphics and Image Processing*, Vol 19, pp 248-264, 1981.

[Mor90] D. G. Morgenthaler, S. J. Hennessy, D. DeMenthon, " Range-video fusion and comparison of inverse perspective algorithms in static images," *IEEE Trans. Syst. Man & Sybernetics* , Vol 20, No. 6, pp 1301-1312, Dec.1990.

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

[Nitz88] D. Nitzan, " Three dimensional structure for robot applications." *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 3, pp 291-309, May.1988.

[Pav73] T. Pavlidis, " Waveform segmentation through functional approximation," *IEEE Trans. Comput.*, Vol 22, No 7, pp 689-697, July 1973.

[Pav74] T. Pavlidis, S. L. Horowitz, " Segmentation of plane curves," *IEEE Trans. Comput.*, Vol 23, No 8, pp 860-870, Aug.1974.

[Pav75] H. Yuan F. Feng & T. Pavlidis, " Decomposition of polygons into simpler components: Feature generation for syntactic pattern recognition," *IEEE Trans. Comput.*, Vol 24, No 6, pp 636-650, June 1975.

[Pav77] T. Pavlidis, " Polygonal approximation by Newton's method," *IEEE Trans. Comput.*, Vol 26, No 8, pp 800-807, Aug. 1977.

[Pav79] F. Ali, T. Pavlidis, " A hierarchical syntactic shape analyzer," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 1, No 1, pp 2-9, jan. 1979.

[Pav80a] T. Pavlidis, " Algorithms for shape analysis of contours and waveforms,"*IEEE Trans. Pattern Anal. Machine Intell.*, Vol 2, No. 4, pp 301-312, july1980.

[Pav80b] T. Pavlidis , S Kahan, H. S. Baird " On the recognition of printed characters of any font and size," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 9, No. 4, pp 301-312, july1980.

[Pav82a] T. Pavlidis, *Algorithms for Graphics and Image Processing*. New York: Computer Science Press, 1982.

[Pav82b] T. Pavlidis, *Structural Pattern Recognition*. New York: Springer-Verlag, 1982.

[Petr88] E. Petriu, " Absolute position recovery for automated path-guided vehicles," *Proc. of the Robots 12 and Vision '88 Conference and Exhibition*, Detroit, Michigan, June 5-9, 1988.

[Petr89] E Petriu, J. S. Basran, " On the position measurement of automated guided vehicles," *IEEE Trans. Instrum. Meas.*, Vol 38, No 3, pp 799-803, June 1989.

[Petr90] E Petriu, J. S. Basran, F.C.A Groen, " Automated guided vehicles position recovery,". *IEEE Trans. Instrum. Meas.*, Vol 39, No1, pp 254-258, Feb. 1990.

[Petr] E. Petriu, " Using pseudo-random encoding for AGV's absolute position recovery," (private communication).

[Scha89] R. J. Schalkoff, *Digital Image Processing and Computer Vision.*, New York,1989.

[Schm88] L. A. Schmitt and S. S. Wilson, " The AIS-5000 parallel processor," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 3, pp 320-341, May.1988.

[Thor88] C. Thorpe, M. H. Herbert, T. Kakade, S. A. Shafer, " Vision and navigation for the Carnagie-Mellon Navlab," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 10, No. 3, pp 362-373, May1988.

[Vey86] E. S. Mc Vey, K. C. Drake, R. M. Inigo, " Range measurement by mobile robot using a navigation line," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 8, No. 1, pp 105-109, Jan.1986.

[Vliet88] L. J. van Vliet, B. J. H. Verwer, " A contour processing method for fast binary neighborhood operations," *Pattern Recognition Letters*, Vol 7, pp 27-36, Jan. 1988.

[Vuy90] P. Vuylsteke, A. Oosterlinck, " Range image acquisition with a single binary-encoded light pattern," *IEEE Trans. Pattern Anal. Machine Intell.*, Vol 12, No. 2, pp 148-163, Feb.1990.

[Wal84] Karin Wall & Per-Erik Danielsson, " A fast method for polygonal approximation of digital curves," *Computer Graphics and Image Processing*, Vol 28, pp 220-227, 1984.

[Wax88] A. M. Waxman, J. J. Le Moigne, " Structured light patterns for robot mobility," *IEEE Jour. Robotics & Automation*, Vol. 4, No. 9, pp 541-548, Oct. 1988.

[Wes78] J. S. Weszak, " Survey: a survey of threshold selection techniques," *Computer Graphics and Image Processing*, Vol 7, pp 259-265, 1978.

[Wil78] C. Williams, " An efficient algorithm for the piece-wise linear approximation of planar curves," *Computer Graphics and Image Processing*, Vol 8, pp 286-293, 1978.

[Will76] J. McWilliams & Neil J.A Sloan," Pseudo-random sequences and arrays." *Proceedings of IEEE* 1976.

APPENDIX I

Let L_i (respectively L_{i+1}) be the length of the segment joining the points P_0 and P_i (respectively P_0 and P_{i+1}), P_i and P_{i+1} being two boundary neighbors as mentioned on Fig III.5. Therefore we have:

$$L_{i+1}^2 = \|\overrightarrow{P_0P_{i+1}}\| = x_{i+1}^2 + y_{i+1}^2 . \qquad \text{(A.I.1.a)}$$

$$L_i^2 = \|\overrightarrow{P_0P_i}\| = x_i^2 + y_i^2 . \qquad \text{(A.I.1.b)}$$

We want to calculate L_{i+1}^2 as a function of L_i^2. Let us assume the following additive model:

$$L_{i+1}^2 = L_i^2 + dL_i^2 \qquad \text{(A.I.2).}$$

Using III.4.a and III.4.b:

$$dx_i = x_{i+1} - x_i ;$$
$$dy_i = y_{i+1} - y_i ;$$

we can compute dL_i^2 as follows:

$$L_{i+1}^2 = (x_i + dx_i)^2 + (y_i + dy_i)^2 .$$

$$= x_i^2 + y_i^2 + 2.x_i . dx_i + 2.y_i . dy_i + dx_i^2 + dy_i^2 .$$

$$= L_i^2 + 2.x_i . dx_i + 2.y_i . dy_i + dx_i^2 + dy_i^2 .$$

Hence,

$$dL_i^2 = 2.x_i \,.\, dx_i + 2.y_i \,.\, dy_i + dx_i^2 + dy_i^2 \,.$$

In the same manner we have proceeded with area deviation, we can draw a table, similar to table III.1, which looks up dL_i^2 for each value af the chain-code. In this way the segment length computation, like the area deviation computation, comes down to the addition and subtraction of integers.

```
/*                    APPENDIX II

                    ************************
                          MAIN MODULE
                    ************************

   This module contains the main program. Contour
   tracing, polygon approximation, corner-vertex
   extraction, 3D pose estimation and binary symbol
   list sorting routines are colled from this module.

*/

   #define  BASE   0x0000
   #define  SNAR   BASE+0x066C
   #include <stdlib.h>
   #include <dos.h>
   #include <tracer1.h>

   void scan();
   extern tos;

main()

{
   int y,thre;
   unsigned char buf[512];

        /* FRAME BOARD INITIALIZATION */

   fg_init(0x26c);
   fg_chan(0);
   fg_sync(1);
   fg_sbuf(0);
   fg_sbuf(1);

        /* IMAGE ACQUISITION.*/

   inp(SNAR);

        /*THRESHOLD SELECTION*/

   thresh();
```

```
        /*START PROCESSING THE IMAGE*/

   scan();

}

   void scan()

{
   int re;
   register int x,y,b,addr,a1;
   XY c;

/*
   Scan the image left to right-top to bottom.
*/

   for(y=1;y<=510;y+=5)

   {

/*
   Image read by block transfer. Setup the x and y
   auto-increment registers.
*/
   addr=fg_setxy(1,y);
   for(x=1;x<=512;x++)
         {
         b=inp(addr);

/*
   Pattern "black pixel followed by a white"
   characterises a new object encountered.
*/

         if(b==100&&a1==0)
               {
               fg_setind(255);
               c=tracer(x,y);
/*
   If the number of boundary pixels is not in the
   range 50-250 discard the object.
*/
               if(tos>50 && tos< 250)
               {
               polygon(c.x,c.y,.4);
               vertex_extraction();
               recognition();
               }
               addr=fg_setxy(x+1,y);
               }
```

```
            a1=b;
            }
    }

    binary_code_sort();
}
```

```
/*
                    ****************************
                      BOUNDARY TRACING MODULE
                    ****************************


/*
   This module includes the contour tracing routine.
*/


   #include <stdio.h>
   #include <tracer1.h>


   void push(char);
   char pop();
   int modulo(int);


   int tos=0; /* tos is the index indicating
                 the top of the boundary stack.*/


   char point[MAX]; /* point is the boundary stack.*/

   int h[8][2]={      {1,0},{1,-1},{0,-1},{-1,-1},
                 {-1,0},{-1,1},{0,1},{1,1}};

/* h is the array which converts chain code values
   to x and y coordinates.
*/


   XY tracer(int xx,int yy)

{
/*
   This is the contour tracing routine.
*/
   XY c;
   int first,found,S,S1;
   int x,y,count;

/*
   INITIALIZATION.
*/
   x=xx;y=yy;
   S=6;
   first=0;
```

```
/*
   CONTOUR TRAVERSAL.
*/

   while(first==0)
   {
         found=0;
         count=0;

         while(found==0 && count<=3)
         {
               S1=modulo(S-1);
               if(fg_pixr(x+h[S1][0],y+h[S1][1])!=0)
               {

                     x+=h[S1][0];
                     y+=h[S1][1];
                     push(S1);
                     fg_pixw(x,y,200);
                     S=modulo(S-2);

                     found=1;

               }

               else if(fg_pixr(x+h[S][0],y+h[S][1])!=0)
               {

                     x+=h[S][0];
                     y+=h[S][1];
                     push(S);
                     fg_pixw(x,y,200);
                     found=1;

               }

               else if(fg_pixr(x+h[(S1=modulo(S+1))][0],y+h[S1][1])!=0)
               {

                     x+=h[S1][0];
                     y+=h[S1][1];
                     push(S1);
                     fg_pixw(x,y,200);
                     found=1;
               }

               else {
               S=modulo(S+2);count++;}

         }
```

```
   if(found==0){first=1;}
   if(x==xx && y==yy)first=1;

   }

       c.x=x;
   c.y=y;
   return c;

}

   void push(char i)

{

/*
   This function adds a new element at the top
   of the boundary stack.
*/

   if(tos>MAX)
   {
   printf("stack full");
   return;
   }

   point[tos]=i;
   tos++;

}

   char pop()

{

/*
   This function retrives an element from the top of the
   boundary stack.
*/
   tos--;
   if(tos<0)return 'n';
   return point[tos];
}


   int modulo(int S)

{

   int SR;
```

```
    if(S>=8)SR=S-8;
    else if(S<0)SR=S+8;
    else SR=S;

    return SR;

}

    float dot(float x1,float x2,float y1,float y2)

{
    return (x1*x2+y1*y2);
}

    float sign(float x)
{
    if(x>=0) return 1;
    else return -1;
}
```

```
/*

                ***********************************
                   POLYGON APPROXIMATION MODULE
                ***********************************


   This module includes the polygon approximation routines.
   It involves the scan along and the split and merge algorithms.
*/


   #include <stdlib.h>
   #include <math.h>
   #include <tracer1.h>


   extern char point[MAX];
   extern int tos;
   int counter;
   extern int h[8][2];

   POLY *p0;
   POLY *last;
   POLY *LI[30];
   float E,Emax;
   int X[MAX],Y[MAX];


void polygon(int x0,int y0,float thresh)

{
   POLY *p;

   scan_along(x0,y0,thresh);
   split_and_merge();
   p=p0;

   while(p->next!=NULL)
   {
   ang(p,p->next);
   }
}


void scan_along(int x0,int y0,float thresh)
```

```
{

/*
   Scan along polygon approximation using area deviation.
*/

   POLY *p,*p1;
   register int i;
   register char S;
   register int x1,y1;
   int x1,y1,x2,y2,x3,y3;
   register int l,ll,area2;
   float a;
   int a1;

/* Intialisation*/

   i=tos;
   last=NULL;
   thresh=4*thresh*thresh;
   fg_setind(255);
   x1=y1=0;
   area2=0;
   l=0;


/*
   The first pixel (x0,y0) of the boundary stack is
   taken as a vertex. It is therefore stored in the vertices list.
*/

   p=malloc(sizeof(POLY));
   p->x1=x0;
   p->y1=y0;
   p->flag=1;
   p->i=i-1;
   store_segment(p);

   X[0]=x0;Y[0]=y0;


/*     In the following loop the area deviation is computed using
   the incrimental property of area deviation. A new polygon
   vertex is generated when the ratio area deviation per segment
   length exeeds an upper limit value.
*/

   while((S=pop())!='n') /* Read chain code value S from stack
                            using the function pop.*/
```

```
{

i--;

area2+=darea(xi,yi,S);          /*Update area deviation*/

l+=dl(xi,yi,S);                 /*Update segment legth*/

xi+= -h[S][0];yi+= -h[S][1];    /*Update pixel coordinates*/
X[i]=xi;Y[i]=yi;


if( (a=(float)(area2*area2)) > (float) (thresh*l))/*Test on area
                                                   deviation*/
      {
      /*
         Generate a new vertex and append it to the polygon
         vertex list
      */

      p->x2=x1=xi+p->x1;p->y2=y1=yi+p->y1;
      p->area2=area2;
      p->l=l;
      p->error=area2*area2/l;
      xi=yi=0;
      area2=0;
      l=0;
      p=malloc(sizeof(POLY));
      p->x1=x1;p->y1=y1;
      p->flag=1;
      p->i=i-1;
      store_segment(p);
      }

}


/*
   The following test investigates the possibility of merging
   the last segment with the first one.
*/

   if(a< (float) (thresh*l))
   {
         x1=p->x1;y1=p->y1;
         x2=p0->x1;y2=p0->y1;
         x3=p0->x2;y3=p0->y2;
         a1=(x2-x1)*(y3-y1) - (y2-y1)*(x3-x1);
         l1=(x3-x1)*(x3-x1)+(y3-y1)*(y3-y1);

         if(l>.3 && (float) (a1*a1)< (float) (thresh*l))
```

```
                {
                p0->x1=p->x1;p0->y1=p->y1;
                p0->flag=1;
                p0->area2=a;
                p0->l=l1;
                p0->error=a*a/l1;
                p->prior->next=NULL;
                }
          else
          {
          p->x2=p0->x1;p->y2=p0->y1;
          p->area2=area2;
          p->l=l;
          p->error=area2*area2/l;
          }
    }

    else
    {
    p->prior->next=NULL;
    }
    return;

}


void split_and_merge()

{
/*
   This is the split and merge routine.
*/

    POLY *p,*pmax;
    float emax=.5; /* emax is the maximum poinwise error*/
    float Fi,Ei,Ej;
    int a,l,j,k,i;


    Emax= emax/4; /* Emax is the maximum total error norm. This
                result follows the analysis made in section
                III.3.3.4.b*/


/*      initial total error norm evaluation.*/

    p=p0;
    E=0;

    while(p!=NULL)
    {
```

```
   E+=abs(p->error);
   p=p->next;
   }

/*
   Split operation.
*/

   while(E>Emax)

   {
   pmax=max_error_segment();
   split(pmax);
   }

/* Merge operation
*/

   p=p0;
   i=0;
   while(p->next!=NULL)
   {

/*      Evaluation of the error norms Fi which would have occurred

   if the ith and the (i+1)th segments were merged.
*/

   a=0;
   l=0;
   j=p->i;
   k=p->next->next->i;

   for(i=j;i<=k;i++)
   {
   a+=darea(X[i],Y[i],point[i]);        /*Update area deviation*/
   l+=dl(X[i],Y[i],point[i]);       /*Update segment legth*/
   }

   Fi=a*a/l;

/*
   Evaluation of the total error norm Gi if the ith and
   the (i+1)th segments were merged.
*/

   p->Gi=E+Fi- (p->error+p->next->error);

   LI[i]=p;
   i++;
   p=p->next;
```

```
   }

/*
   Sort Gi's in the decreasing order.
*/

   sort_Gi(0,i);
/*
   Merge all segments which have a Gi<Emax.
*/
   for(j=i;j>=0;j--)
   {
   p=LI[i];
   if(p->Gi< Emax)
   {
   Ei=p->error;
   Ej=p->next->error;
   merge(p);
   for(k=0;k<=j-1;k++) LI[k]->Gi+=p->error-Ei-Ej;/* Update Gi's of
                                          nonmerged segments.*/

   }
   else j=-15;
   }
/*
   Vertex adjustment.
*/
   vertex_adj();

}

void split(POLY *p)
/*
   The split routine.
*/

{

   int i,j,k;
   POLY *p1;
   int a1,a2,l;
   float e;

   j=p->i;k=p->next->i;
/*
   The segment is split by inserting a new
   element in the vertex list.
*/
   e=p->error;
   p1=malloc(sizeof(POLY));
   p1->next=p->next;
```

```
   p1->next->prior=p1;
   p1->prior=p;
   p->next=p1;

/*
   The new vertex takes the place of the pixel wich is
   half way, along the boundary, between the two endpoints
   of the segment to split.
*/

   p1->i=(j+k)/2;

/*
   Updating the segment error norms,
   lengths and area deviations.
*/
   a1=0;
   l=0;
   for(i=j;i<=(j+k)/2;i++)
   {
   a1+=darea(X[i],Y[i],point[i]);        /*Update area deviation*/
   l+=dl(X[i],Y[i],point[i]);       /*Update segment legth*/
   }
   p->area2=a1;
   p->l=l;
   p->error=a1*a1/l;

   a2=0;
   l=0;
   for(i=(j+k)/2;i<=k;i++)
   {
   a2+=darea(X[i],Y[i],point[i]);
   l+=dl(X[i],Y[i],point[i]);
   }
   p1->area2=a2;
   p->l=l;
   p->error=a2*a2/l;

/*
   Updating the total error norm.
*/

   E+=p->error+p1->error-e;

}

POLY *max_error_segment()

/*
   This fuction finds the polygon segment
```

```
   with the maximum error norm.
*/


{
   POLY *p;
   float emax=0;
   POLY *pmax;

   p=p0;

   while(p!=NULL)
   {
   if(p->error>emax){emax=p->error;pmax=p;}
   p=p->next;

   }
   return pmax;

}

void sort_Gi(int first,int last)

/*
   This function sorts the vertex list in the decreasing
   orderof the Gi's. It uses the quick sort algorithm.
*/

{
   int i,j;
   float pivot;

   if(first<last)
   {
   pivot=LI[first]->Gi;
   i=first;
   j=last;

   while(i<j)
         {
         while(LI[i]->Gi>=pivot && i!=last) i++;
         while(LI[j]->Gi<=pivot && j!=first) j--;
         if(i<j) exchange_Gi(i,j);
         }

   exchange_Gi(first,j);
   sort_Gi(first,j-1);
   sort_Gi(j+1,last);

   }
   return;
```

```
}

void exchange_Gi(int i,int j)

{
   POLY *a;

   a=LI[i];
   LI[i]=LI[j];
   LI[j]=a;

}

void merge(POLY *pi)
/*
   This function merges the segment pi with next segment
   in the list.
*/

{

   int At,l;
   float a;
   int xp,yp,xi,yi,xn,yn;
   POLY *pp;

   pp=pi->next;

   xp=pp->x1;yp=pp->y1;
   xi=pi->x1;yi=pi->y1;
   xn=pi->x2;yn=pi->y2;

/*
   New area deviation and segment length computation.
*/

   At= (xi-xp)*(yn-yp) - (yi-yp)*(xn-xp)  ;
   l=(xn-xp)*(xn-xp)+(yn-yp)*(yn-yp);
   a=(float) (At+pi->area2+pp->area2);

/*
   Segment merge
*/
   pi->x1=xp;pi->y1=yp;
   pi->i=pp->i;
   pi->area2=(int) a;
   pi->l= l;
   pi->error=a*a/l;
   pi->flag=pi->flag;
   pi->prior=pp->prior;
```

```
   if(pp!=p0)
   pp->prior->next=pi;
   else p0=pi;

}

void vertex_adj()
/*
   This routine loops through the contour by stepping
   from vertex to vertex and performing a single vertex
   adjustment.
*/

{
   POLY *p,*pl;
   int i,j;
   float a;

   counter=1;/* "counter" indicates the number of adjustments
            performed.If no more vertex adjustment be made
            the program stops.*/

   while(counter!=0)
   {
   p=p0->next;
   pl=p0;
   counter=0;

   while(p!=NULL)
         {
         single_vertex_adj(p,pl);
         pl=p->next;
         if(pl!=NULL)
         p=p->next->next;
         else p=NULL;
         }
   }
}

void single_vertex_adj(POLY *pi,POLY *pp)
/*
   This routine, called from the function vertex_adjustment,
   adjusts the position of one vertex.
*/

{

   int F1,F2,dA1,dA2,dA3,dA4,dl1,dl2,dl3,dl4;
   int xp,yp,xi,yi,xn,yn,dx1,dy1,dx2,dy2;
   int S,S0;
```

```
/*
   Start by defining three successive vertices:Pp,Pi,Pn. Pi
   is the vertex to be adjusted.
*/

   xi=pi->x1;yi=pi->y1;
   xn=pi->x2;yn=pi->y2;
   xp=pp->x1;yp=pp->y1;


/*
   Computation of the area deviation variation when the
   vertex is moved to the following neighbor on the contour.
*/
   if(pi->flag==1)
   {
   S=point[pi->i];

   dA1=darea(xi-xp,yi-yp,S);
   dA2=darea(xn-xi,yn-yi,S);
/*
   Computation of the total error norm variation
*/

   F1=(pi->area2+dA2)*(pi->area2+dA2) - (pi->area2*pi->area2)
   +  (pp->area2+dA1)*(pp->area2+dA1) - (pp->area2*pp->area2);

   if(F1<0)     /* Case where moving the vertex
                has decreased the total erro norm.*/
         {
         pi->i--;
         pi->x1-=h[S][0];pp->x2=pi->x1;
         pi->y1-=h[S][1];pp->y2=pi->y1;;
         pi->area2+=dA2;pp->area2+=dA1;
         pi->l+=dl(xi-xn,yi-yn,S);
         pp->l+=dl(xi-xp,yi-yp,S);
         counter++;
         return;
         }
   }

/*
   Computation of the area deviation variation when the
   vertex is moved to the preceeding neighbor on the contour.
*/

   S=point[pi->i+1];

   dA3=darea(xi-xp,yi-yp,S0=modulo(S-4));
   dA4=darea(xn-xi,yn-yi,S0);
```

```
/*
   Computation of the total error norm variation
*/

   F2=(pi->area2+dA4)*(pi->area2+dA4) - (pi->area2*pi->area2)
   +  (pp->area2+dA3)*(pp->area2+dA3) - (pp->area2*pp->area2);

   if(F2<0)
         {

         pi->i++;
         pi->x1+=h[S][0];pp->x2=pi->x1;;
         pi->y1+=h[S][1];pp->y2=pi->y1;;
         pi->area2+=dA4;pp->area2+=dA3;
         pi->l+=dl(xi-xn,yi-yn,S0);pp->l+=dl(xi-xp,yi-yp,S0);
         pi->flag=2;
         counter++;
         return;
         }

    else if(pi->flag==2) {pi->flag=1;vertex_adj(pi,pp);}

}


void store_segment( POLY *p)

{
/*
   This funtion appends a new element to the segment
   list.
*/

   if(!last) last=p0=p;
   else last->next=p;
   p->next=NULL;
   p->prior=last;
   last=p;
}


void ang(POLY *pi,POLY *pp)
{
/*
   .This function calculates the angle between two
    succesive segments in the polygon.
*/

   int xi,yi,xp,yp,xn,yn;
   float Ct,Si;
```

```
    xp=pp->x1;yp=pp->y1;
    xi=pi->x1;yi=pi->y1;
    xn=pi->x2;yn=pi->y2;

    Ct= (float) ( (xi-xp)*(xn-xi)+(yi-yp)*(yn-yi) );
    Si= (float) ( (xi-xp)*(yn-yi)-(yi-yp)*(xn-xi) );


    if(Ct==0)
    pi->ANG=sign(Si)*3.14/2;

    else if(Ct>0)
    pi->ANG=atan(Si/Ct);

    else
    pi->ANG=atan(Si/Ct)+sign(Si)*3.14;


}

int darea(int xi,int yi,int S)

{
/*
    This function calculates the area deviation variation.
*/
    int j[8];

    j[0]=yi;j[1]=yi+xi;j[2]=xi;j[3]=xi-yi;j[4]=-yi;
    j[5]=-(yi+xi);j[6]=-xi;j[7]=yi-xi;

    return  (j[S]);

}

int dl(int xi,int yi,int S)

{
/*
    This function calculates the segment length variation.
*/

    int j[8];

    j[0]=1-2*xi;j[1]=2+2*(yi-xi);j[2]=1+2*yi;j[3]=2+2*(xi+yi);
    j[4]=1+2*xi;j[5]=2+2*(xi-yi);j[6]=1-2*yi;j[7]=2-2*(yi+xi);

    return  (j[S]);

}
```

```
/*
                ************************************
                  CORNER-VERTEX EXTRACTION MODULE.
                ************************************

/*
   This module includes three basic routines: the polygon
   decomposition, the trusion parser and the semantic analyser.
*/


   #include <tracerl.h>
   #include <math.h>
   #include <stdlib.h>
   #include <dos.h>

   extern int X[],Y[];
   extern POLY *p0;
   SEGMENTS *j0;

   SEGMENTS *lastj=NULL;

   float BREAK;
   float COLANG;
   int state;

   int IS;
   LINES L1,L2,L3,L4;
   int xm,ym;


void poly_decomp()
/*
   The polygon decomposition routine serves to partition the
   boundary into parts where the concavity is constant.
*/

{

   POLY *i,*i1,*i2;
   int sig;


   sig=sign(i0->ang);
   i=p0;
   i1=p0;

   xm=X[i->i];ym=Y[i->i];
```

```
   for(;;)
   {
   while(i->next!=NULL&&sig==sign(i->ang))i=i->next;
/*
   Search for the vertex where concavity changes.
*/
   i2=i;
   if(i2!=i1->next)
   {
   state=0;
/*
   Apply the parser for trusion to the part of the contour
   that has a constant concavity.
*/
   trus(i1->ind,i2->ind);

   }

   if(i->next==NULL)break;

   i1=i;
   sig=-sig;
   }
}


   void trus(int i1,int i2)

/*
   Parser for trusion.
*/

{
   LINES lin;
   BREAK=5.5;
   COLANG=30*3.14/180;

   lin=line(i1,i2,COLANG);

   if(lin.ret!=1){return;}

/*1*/
   IS++;

   switch(state)
   {

/*2*/   case 0:
/*
   This is the case where one line has been found.
```

```
*/
   state=1;
   L1=lin;
   trus(IS,i2);
   return;

/*3*/    case 1:

/*      This is the case where two lines have been found. If the
second
   line is smaller then the first one "trusion" is called again to
   extract the following line. If not, either a corner or a line
   has been found and the result is analysed by sflush.
*/
       L2=lin;
   if(L2.seg.L<BREAK)

/*4*/          {
          state=2;
          trus(IS,i2);
          return;
          }

/*5*/    else
          {
          sflush(state);
          state=1;
          L1=lin;
          trus(IS,i2);
          return;
          }

/*6*/    case 2:
/*
   Case where 3 lines have been found. Either the second line is
   a break and the three lines make either a corner or a line, or
   the third line is short and a fourth one is needed in order to
   recognize the pattern.
*/
   L3=lin;
   if(L3.seg.L<BREAK)

/*7*/          {
          state=3;
          trus(IS,i2);
          return;
          }

/*8*/    else
          {
          sflush(state);
```

```
        state=1;
        L1=lin;
        trus(IS,i2);
        return;
        }

/*9*/   case 3:
/*
   Four lines have been found. Two situations may occur: either
   the last line is a line (happens very seldom) or the pattern
   is a either a corner or a line having two breaks in the middle.
*/
      L4=lin;
   if(L4.seg.L>max(L2.seg.L,L3.seg.L))

/*10*/        {
        sflush(state);
        state=1;
        L1=lin;
        trus(IS,i2);
        return;
        }

/*11*/  else
        {
        state=4;
        sflush(state);
        IS++;
        state=1;
        L1=lin;
        trus(IS,i2);
        return;
        }
   }
}


void sflush(int state)

/*
   This function is a semantic analyser. It decides if the pattern
   is a line or a break.
*/

{
   static int ind=0;
   float ang,si,co,e=45*3.14/180,c,x;
   SEGMENTS *j;
   int a1,a2;
   float l;
```

```
   switch(state)
   {
   case 1:
/*
   Computation of the angle between the two lines.
*/

   x=dot(L1.seg.a1,L2.seg.a1,L1.seg.a2,L2.seg.a2)/(L1.seg.L*L2.seg.L);

   if(fabs(x)>=.99) ang=acos(sign(x)*.99);

   else ang=acos(x);

   if(ang<e) /* Case where the pattern is a line*/
   {
   ind++;
   }

   else  /* Case where the pattern is a corner.
            Generate a vertex.*/
   {

   a1=L1.xc-xm;;
   a2=L1.yc-ym;;
   j->x=(L1.xc+xm)/2;
   j->y=(L1.yc+ym)/2;
   j->L=l=sqrt(dot(a1,a1,a2,a2));
   j->ang=acos(a1/l)*sign(a2);
   xm=L1.xc;ym=L1.yc;
   dlstore1_free1(j,1);

   }

   break;

   case 2:
/*
   Computation of the angle between the first and the last lines.
*/
   x=dot(L1.seg.a1,L3.seg.a1,L1.seg.a2,L3.seg.a2)/(L1.seg.L*L3.seg.L);

   if(fabs(x)>=.99) ang=acos(sign(x)*.99);

   else ang=acos(x);

   if(ang<e) /* Case where the patter is a line.*/

   {
   ind++;
   }
```

```
   else  /*Case the pattern is a corner.
              A vertex is generated.*/
   {

   a1=L1.xc-xm;;
   a2=L1.yc-ym;;
   if((j=pop_to_seg())==NULL)
   j=calloc(sizeof(SEGMENTS),1);
   j->x=(L1.xc+xm)/2;
   j->y=(L1.yc+ym)/2;
   j->L=l=sqrt(dot(a1,a1,a2,a2));
   j->ang=acos(a1/l)*sign(a2);
   xm=L1.xc;ym=L1.yc;
   dlstorel_freel(j,1);
   ind++;

   }

   break;

   case 3:
/*
   Computation of the angle between the first and the last line.
*/

   x=dot(L1.seg.a1,L4.seg.a1,L1.seg.a2,L4.seg.a2)/(L1.seg.L*L4.seg.L);

   if(fabs(x)>=.99) ang=acos(sign(x)*.99);

   else ang=acos(x);

   if(ang<e)/* Case where the pattern is a line.*/

   {
   ind++;
   }

   else  /* Case where the pattern is a corner.
         A vertex is generated.*/

   {

   a1=L1.xc-xm;;
   a2=L1.yc-ym;;
   j=calloc(sizeof(SEGMENTS),1);
   j->x=(L1.xc+xm)/2;
   j->y=(L1.yc+ym)/2;
   j->L=l=sqrt(dot(a1,a1,a2,a2));
   j->ang=acos(a1/l)*sign(a2);
   xm=L1.xc;ym=L1.yc;
   dlstorel_freel(j,1);
```

```
    ind++;
    }
    break;

    default:break;

    }

}


    LINES line(int i1,int i2,float COLANG)
/*
    This function, by calculating angles, finds a vertex i between
    i1 and i2 such that (i1,i) is a line and (i1,i+1) is not.

{
    LINES lin;
    VERTICES *i;

    int xc1,yc1,xc2,yc2;

    if(i1>i2){lin.ret=-1;return(lin);}
    i=i0;
    while(i->ind<i1)i=i->next;

    xc1=X[i->j];
    yc1=Y[i->j];

    while(i->ind<i2 && fabs(i->ang)<COLANG) i=i->next;


    xc2=X[i->k];
    yc2=Y[i->k];

    lin.seg.a1=yc1-yc2;
    lin.seg.a2=xc2-xc1;
    lin.seg.L=sqrt(pow(lin.seg.a1,2)+pow(lin.seg.a2,2));
    lin.xc=xc2;
    lin.yc=yc2;

    lin.ret=1;

    IS=i->ind;
    if(lin.seg.L<.1){lin=line(IS,i2,COLANG);return(lin);}
    return(lin);

}

}
```

```
void dlstore1_free1(SEGMENTS *j,int f)
{

if(f==1)
{
if(lastj==NULL){j0=j;}
else lastj->next=j;
j->next=NULL;
j->prior=lastj;
lastj=j;
}
else if(f==2)
{
while(lastj!=NULL)
      {
      push_to_seg(lastj);
      lastj=lastj->prior;
      }
}

}
```

```
/*
                    ******************************
                      SYMBOL RECOGNITION MODULE.
                    ******************************

   In this module symbol recognition is carried out using
   a tree search.
*/

   #include <stdlib.h>
   #include <conio.h>
   #include <tracer1.h>
   #include <math.h>


   POLY *A[30];
   extern POLY *p0;

   extern void E_vote();
   extern void H_vote();



void recognition()
/*
   This is the symbol recognition by tree search function.
*/

{
   float a,a1,a2,a3;
   int i,xp,xp1,xp2,yp,yp1,yp2,xa,ya,j;
   POLY *p,*p1;

/*
   Initialization of the matrix A which contains the pointers to the
   boundary vertices of the symbol.
*/

   p=p0;
   i=0;
   while(p1=NULL)
   {
   A[i]=p;
   i++;
   p=p->next;
   }
/*
   The list of polygon segments is sorted in the decreasing order
   of their length. The quick sort algorithm is used for this
   purpose.
```

```
*/

   quicksort(0,i-1);


   a=(float) (A[0]->l);

   if((a1=(float)(A[1]->l)) >.65*a) /* Test if the ratio between
                           the length of the second longest
                           segment and the longest segment is
                           greater than 0.65.*/
   {
/*
   In the affirmative case the object is eligible to be the
   symbol 'H'. A few other features of the symbol 'H' will
   have to be tested then.
*/

   a2=.50*a;a3=.08*a;
   if((a1=(float)(A[2]->l))<a2 && a1>a3
   && (a1=(float)(A[3]->l))<a2 && a1>a3
   && (a1=(float)(A[4]->l))<a2 && a1>a3
   && (a1=(float)(A[5]->l))<a2 && a1>a3)
         {
/*
   The object is recognized as the symbol 'H'. Perform model-scene
       segment      matching by geometric hashing.
*/

         H_vote();

         }
   }

   else if(a1<.6*a || a1>.15*a) /* Test if the ratio between
                           the length of the second longest
                           segment and the longest segment is
                           smaller than 0.6 or greater than
                           0.15.*/
   {
/*
   In the affirmative case the object is eligible to be the
   symbol 'E'. A few other features of the symbol 'E' will
   have to be tested then.
*/

   a2=.65*a;a3=.12*a;
   if((a1=(float)(A[1]->l))<a2 && a1>a3
   && (a1=(float)(A[2]->l))<a2 && a1>a3)
         {
```

```
        a2=.4*a;a3=.07*a;
        if((a1=(float)(A[3]->l))<a2 && a1>a3
        && (a1=(float)(A[4]->l))<a2 && a1>a3
        && (a1=(float)(A[5]->l))<a2 && a1>a3
        && (a1=(float)(A[6]->l))<a2 && a1>a3)
        {
/*
   The object is recognized as the symbol 'E'. Perform model-scene
       segment      matching by geometric hashing.
*/
        E_vote();

        }
        }
   }

   else return; /* Return in case the object is recognized as
                   as not being a binary symbol.*/

}


void quicksort(int first,int last)
/*
   Quick sort algorithm.
*/

{
   int i,j,pivot;


   if(first<last)
   {
   pivot=A[first]->l;
   i=first;
   j=last;

   while(i<j)
        {
        while(A[i]->l>=pivot && i!=last) i++;
        while(A[j]->l<=pivot && j!=first) j--;
        if(i<j) exchange(i,j);
        }

   exchange(first,j);
   quicksort(first,j-1);
   quicksort(j+1,last);

   }
   return;
```

```
}

void exchange(int i,int j)

{
   POLY *a;

   a=A[i];
   A[i]=A[j];
   A[j]=a;

}
```

```
/*
                ************************************
                  MODEL-SCENE POINT MATCHING MODULE
                ************************************


   In this module model and scene points are matched using the
   geometric hashing technique. The matched model and scene points
   are used for the determination the 3D position of the symbol.
*/
   #include <stdlib.h>
   #include <tracer1.h>
   #include <math.h>


   void E_vote();
   void H_vote();
   void E_model();
   void H_model();

   extern void pose_from_perspective();


   SEGMENT E1[12];
   SEGMENT H1[12];
   SEGMENT1 E2[12];
   SEGMENT1 H2[12];
   extern POLY *A[30];
   SEGMENT1 MS[12][2];


   void E_vote()
/*
   This function performs point matching for the letter 'E'.
*/

{
   int x0,y0,v1x,v1y,v2x,v2y,xp,yp,x,y;
   int i,j;
   int ind;
   float xp1,yp1,x1,y1;
   float d,d0,det,a;

/*
   The model of the symbol is loaded into memory.
*/

   E_model();
/*
```

```
   Define frame related to the scene (three points are required).
*/
   v2x=A[0]->x1-A[0]->x2;
   v2y=A[0]->y1-A[0]->y2;
   v1x=A[1]->x2-A[1]->x1;
   v1y=A[1]->y2-A[1]->y1;

   if((A[0]->x2-A[1]->x1)*(A[0]->x2-A[1]->x1)>10)
   {
   v2x=-v2x;v2y=-v2y;
   }

   x0=A[0]->x2;y0=A[0]->y2;
   det=(float) (v1x*v2y-v1y*v2x);

   for(i=0;i<=6;i++)

   {
/*
   Compute the coordinates of each scene vertex in this frame
*/
         xp= (A[i]->x1+A[i]->x2)/2;
         yp= (A[i]->y1+A[i]->y2)/2;

         xp1= dot( xp-x0 , v2y , y0-yp , v2x ) /det;
         yp1= dot( x0-xp , v1y , yp-y0 , v1x ) /det;
/*
   Find, in the model, the vertex that has the closest coordinates.
*/
         d0=2.0;
         for(j=0;j<=6;j++)
               {
               d=dot(xp1-E1[j].x,xp1-E1[j].x,
                     yp1-E1[j].y,yp1-E1[j].y);

               if(d<d0) { d0=d ; ind=j; }

               }

/*
   Store the matching model and scene points in the array MS.
*/

         x=A[i]->x1;y=A[i]->y1;

         x1= dot( x-x0 , v2y , y0-y , v2x ) /det;
         y1= dot( x0-x , v1y , y-y0 , v1x ) /det;

         if(dot(a=x1-E2[ind].x1/10,a,a=y1-E2[ind].x1/15,a) <
            dot(a=x1-xp1,a,a=y1-yp1,a))
```

```
                {
                MS[i][0].x1= E2[ind].x1;
                MS[i][0].y1= E2[ind].y1;

                MS[i][0].x2= E2[ind].x2;
                MS[i][0].y2= E2[ind].y2;

                }

        else
                {
                MS[i][0].x1= E2[ind].x2;
                MS[i][0].y1= E2[ind].y2;

                MS[i][0].x2= E2[ind].x1;
                MS[i][0].y2= E2[ind].y1;
                }

        MS[i][1].x1= A[i]->x1;
        MS[i][1].y1= A[i]->y1;
        MS[i][1].x2= A[i]->x2;
        MS[i][1].y2= A[i]->y2;

   }

   MS[i][1].x1=MS[i][1].x2=MS[i][1].y1=MS[i][1].y2=-1;
/*
   Perform pose estimation.
*/
   pose_from_perspective();

}

   void H_vote()
/*
   This function performs point matching for the letter 'H'.
*/

{
   int x0,y0,v1x,v1y,v2x,v2y,xp,yp,x,y;
   int i,j,ind;
   float xp1,yp1,x1,y1;
   float d0,d,det,a;

/*
   The model of the symbol is loaded into memory.
*/
   H_model();
/*
   Define frame related to the scene (three points are required).
*/
```

```
   v2x=(A[0]->x1-A[0]->x2);
   v2y=(A[0]->y1-A[0]->y2);
   v1x=(A[1]->x1-A[0]->x2);
   v1y=(A[1]->y1- A[0]->y2);


   x0=A[0]->x2 ; y0= A[0]->y2;
   det=v1x*v2y-v1y*v2x;
   for(i=0;i<=5;i++)
         {
/*
   Compute the coordinates of each scene vertex in this frame.
*/
         xp= (A[i]->x1+A[i]->x2)/2;
         yp= (A[i]->y1+A[i]->y2)/2;
         xp1= dot( xp-x0 , v2y , y0-yp , v2x ) /det;
         yp1= dot( x0-xp , v1y , yp-y0 , v1x ) /det;
/*
   Find, in the model, the vertex that has the closest coordinates.
*/

         d0=2.0;
         for(j=0;j<=5;j++)
               {
               d=dot(xp1-H1[j].x,xp1-H1[j].x,
                     yp1-H1[j].y, yp1-H1[j].y);

               if(d<d0)  { d0=d ; ind=j; }

               }

/*
   Store the model and scene matching points in the array MS.
*/

         x=A[i]->x1;y=A[i]->y1;

         x1= dot( x-x0 , v2y , y0-y , v2x ) /det;
         y1= dot( x0-x , v1y , y-y0 , v1x ) /det;

         if(dot(a=x1-H2[ind].x1/10,a,a=y1-H2[ind].y1/15,a) <
            dot(a=x1-xp1,a,a=y1-yp1,a))
               {
               MS[i][0].x1= H2[ind].x1;
               MS[i][0].y1= H2[ind].y1;

               MS[i][0].x2= H2[ind].x2;
               MS[i][0].y2= H2[ind].y2;
```

```
                }

        else
                {
                MS[i][0].x1= H2[ind].x2;
                MS[i][0].y1= H2[ind].y2;

                MS[i][0].x2= H2[ind].x1;
                MS[i][0].y2= H2[ind].y1;
                }

        MS[i][1].x1= A[i]->x1;
        MS[i][1].y1= A[i]->y1;
        MS[i][1].x2= A[i]->x2;
        MS[i][1].y2= A[i]->y2;

    }

    MS[i][1].y2=MS[i][1].x2=MS[i][1].y1=MS[i][1].x1=-1;

/*
    Perform pose estimation
*/
    pose_from_perspective();

}

void E_model()
/*
    This function loads the model of the symbol 'E'into memory.
    Note that the function is executed only once.
*/

{
    static int flag=0;
    if(flag)return;
    flag=1;

    E1[0].x= .65 ; E1[0].y= .6 ;
    E1[1].x= .65 ; E1[1].y= .8 ;
    E1[2].x= .5  ; E1[2].y=  1 ;
    E1[3].x= .0  ; E1[3].y= .5 ;
    E1[4].x= .5  ; E1[4].y= .0 ;
    E1[5].x= .65 ; E1[5].y= .2 ;
    E1[6].x= .65 ; E1[6].y= .4 ;

    E2[0].x1= 10.0 ; E2[0].y1= 9.0 ;
    E2[0].x2= 3.0  ; E2[0].y2= 9.0 ;

    E2[1].x1= 3.0  ; E2[1].y1= 12.0 ;
    E2[1].x2= 10.0 ; E2[1].y2= 12.0 ;
```

```
   E2[2].x1= 10.0 ; E2[2].y1= 15.0 ;
   E2[2].x2= 0.0 ;  E2[2].y2= 15.0 ;

   E2[3].x1= 0.0  ; E2[3].y1= 15.0 ;
   E2[3].x2= 0.0  ; E2[3].y2=  0.0 ;

   E2[4].x1= 0.0  ; E2[4].y1=  0.0 ;
   E2[4].x2= 10.0 ; E2[4].y2=  0.0 ;

   E2[5].x1= 10.0 ; E2[5].y1=  3.0 ;
   E2[5].x2= 3.0  ; E2[5].y2=  3.0 ;

   E2[6].x1= 3.0  ; E2[6].y1=  6.0 ;
   E2[6].x2= 10.0 ; E2[6].y2=  0.0 ;

}

   void H_model()
/*
   This function loads the model of the symbol 'H into memory.
   Note that the function is executed only once.
*/

{
   static int flag=0;
   if(flag)return;
   flag=1;

   H1[0].x= 1.0 ; H1[0].y= .5 ;
   H1[1].x= .7  ; H1[1].y= .8 ;
   H1[2].x= .3  ; H1[2].y= .8 ;
   H1[3].x= .0  ; H1[3].y= .5 ;
   H1[4].x= .3  ; H1[4].y= .2 ;
   H1[5].x= .7  ; H1[5].y= .2 ;

   H2[0].x1= 10.0 ; H2[0].y1=0.0 ;
   H2[0].x2= 10.0 ; H2[0].y2=15.0 ;

   H2[1].x1= 7.0  ; H2[1].y1= 15.0 ;
   H2[1].x2= 7.0  ; H2[1].y2= 9.0  ;

   H2[2].x1= 3.0  ; H2[2].y1= 9.0  ;
   H2[2].x2= 3.0  ; H2[2].y2= 15.0 ;

   H2[3].x1= 0.0  ; H2[3].y1= 15.0 ;
   H2[3].x2= 0.0  ; H2[3].y2= 0.0  ;

   H2[4].x1= 3.0  ; H2[4].y1= 0.0  ;
   H2[4].x2= 3.0  ; H2[4].y2= 6.0  ;
```

```
   H2[5].x1= 7.0  ; H2[5].y1= 6.0  ;
   H2[5].x2= 7.0  ; H2[5].y2= 0.0  ;

}
```

```
}/*

            ****************************************
               MODEL BASED 3d POSITION ESTIMATION.
            ****************************************

   This module deals with model based 3D pose estimation.
*/


   #include <tracerl.h>
   #include <stdlib.h>
   #include <math.h>


   extern SEGMENT1 MS[12][2];
   float d[25];
   float Y1[25][2];
   int N;
   int nc=0;
   float f=500;
   float sig;
   BINARYCODE *BC[40];
   float R[3][3],T[3][1];


   XY preprocessing();
   void d_initilzation();
   XYZ average_dnvn();
   void HH(XYZ,float ho[2][3]);
   void TT(float Ro[3][3],float To[3][1],XY,XYZ);
   void svd(float ho[2][3],float Ro[3][3]);
   void d_updating(float RO[3][3],XYZ);


void pose_from_perspective()

/*
   This routine performs the model based pose estimation as
   indicated in algorithm IV.1.
*/

{


   float thresh;
```

```
XYZ cs;
XY cm;
float h[2][3],sig1;
float x[3][1];
int i,j,count=0;
float x0,y0,z0;


sig=20000;

cm=preprocessing();/* Compute the coordinates of the
           model centroid and the standard
           deviation from this centroid.*/


/*1*/    d_initilzation();/* Initialazation of the depth of each
                  scene point.*/

do    /*Iterate until convergence.*/
{

sig1=sig;/* "sig" represents the value of thecurrent  estimation
      error, and "sig1" represents that of the preceeding
      iteration.*/


/*2*/
   /*2.1*/
      cs=average_dnvn();/* Compute the coordinates of the
                 symbol centroid in its current estimated
                 3D position.*/

      HH(cs,h);    /* Compute of the model-scene
                 intercorrelation matrix.*/

      svd(h,R);      /* Singular value decomposition of the
                 model-scene intercorrelation matrix.*/


      d_updating(R,cs); /* Updating of each point's depth.*/

      count++;
}

while((sig1-sig)/sig >.01);/* Convergence when no noticeable
                        improvement in the estimation
                        accuracy can be achieved.*/


TT(R,T,cm,cs);
```

```
   x0=R[0][0]*cm.x+R[0][1]*cm.y+T[0][0];
   y0=R[1][0]*cm.x+R[1][1]*cm.y+T[1][0];
   z0=R[2][0]*cm.x+R[2][1]*cm.y+T[2][0];


   BC[nc]=malloc(sizeof(BINARYCODE));
   BC[nc]->x=x0;
   BC[nc]->y=y0;
   BC[nc]->z=z0;
   Bc[nc]->order=0;
   nc++;


}


   XY preprocessing()

/*
   This function computes the coordinates of the model centroid as
   well as the standard deviation from this centroid.
*/

{
   register int x,y;
   register int i,k;
   XY c;
   float a,b;


   x=y=0;
   i=0;
/*
   Computation of the coordinates of the model centroid.
*/
   while(MS[i][1].y1!=-1)
   {
   x+=MS[i][0].x1+ MS[i][0].x2;
   y+=MS[i][0].y1+MS[i][0].y2;
   i++;
   }

   N=2*i;
   a=1/(float)N;
   c.x=(float) x * a;
   c.y=(float) y * a;

   a=c.x;b=c.y;
/*
   Comptation of the deviation of each model point from
   the centroid.
```

```
*/

   for(k=0;k< N/2;k++)
   {
   Y1[2*k][0]=(float)MS[k][0].x1-a;
   Y1[2*k][1]=(float)MS[k][0].y1-b;
   Y1[2*k+1][0]=(float)MS[k][0].x2-a;
   Y1[2*k+1][1]=(float)MS[k][0].y2-b;

   }

   return c;.

}

   XYZ average_dnvn()

/*      This function computes the coordinates of the symbol
centroid
   in its current estimated 3D position .*/


{
   register int i;
   float x,y,z;
   float a;
   XYZ c;

/*
   Computation of the coordinates of the symbol centroid in its
   current estimated 3D position.
*/

   x=y=z=0;

   for(i=0;i< N/2;i++)
   {
   x+= d[2*i]* (float) MS[i][1].x1+
      d[2*i+1]*(float) MS[i][1].x2;

   y+= d[2*i]* (float) MS[i][1].y1+
      d[2*i+1]*(float) MS[i][1].y2;

   z+= d[2*i] + d[2*i+1];
   }

   i=N;
   a=(float) i * f;
   x=x/a ; y=y/a ; z=z/(float) i ;

   c.x=x;c.y=y;c.z=z;
```

```
   return c;

}

   void HH(XYZ dv,float h[2][3])

/*
   This function computes the model-scene intercorrelation function

   using equation IV.48.
*/

{
   int h00,h01,h02,h10,h11,h12;
   int a11,a21,a31,a12,a22,a32;
   register int i,j;
   float dv1,dv2,dv3;

   dv1=dv.x;dv2=dv.y;dv3=dv.z;


   h00=h01=h02=h10=h11=h12=0;

   for(i=0;i< N/2;i++)
   {
   a11=(int)  ( d[2*i] * (float) MS[i][1].x1/ f )  - (int) dv1 ;
   a12=(int)  ( d[2*i+1] *(float) MS[i][1].x2/ f)  - (int) dv1 ;

   a21= (int) ( d[2*i]  *(float) MS[i][1].y1 / f ) - (int) dv2;
   a22= (int) ( d[2*i+1]*(float) MS[i][1].y2 / f ) - (int) dv2;


   a31= (int)d[2*i]   - (int) dv3;
   a32= (int)d[2*i+1] - (int) dv3;


   h00+=(int) Y1[2*i][0]   *  a11 + (int) Y1[2*i+1][0] * a12 ;
   h01+=(int) Y1[2*i][0]   *  a21 + (int) Y1[2*i+1][0] * a22 ;
   h02+=(int) Y1[2*i][0]   *  a31 + (int) Y1[2*i+1][0] * a32 ;

   h10+=(int) Y1[2*i][1]   *  a11 + (int) Y1[2*i+1][1] * a12 ;
   h11+=(int) Y1[2*i][1]   *  a21 + (int) Y1[2*i+1][1] * a22 ;
   h12+=(int) Y1[2*i][1]   *  a31 + (int) Y1[2*i+1][1] * a32 ;


   }

   h[0][0]=h00;h[0][1]=h01;h[0][2]=h02;
   h[1][0]=h10;h[1][1]=h11;h[1][2]=h12;
```

```
}

   void svd(float h[2][3],float R[3][3])
/*
   This function performs the sigular value decomposition of the
   model-scene intercorrelation matrix as indicated in algorithm
   IV.2.
*/

{
   register int i,j;
   float U[3][3],V[3][3];

   float a,b,c;
   float delta,a1,a2,a3;
   float h11,h12,h13,h21,h22,h23;
   float lamda1,lamda2,srdelta,srlamda1,srlamda2;
   float u1,u2,u3;

   h11=h[0][0];h12=h[0][1];h13=h[0][2];
   h21=h[1][0];h22=h[1][1];h23=h[1][2];

   a=h11*h11+h12*h12+h13*h13;
   b=h21*h21+h22*h22+h23*h23;
   c=h11*h21+h12*h22+h13*h23;

/*
   Eigenvalues computation.
*/

   delta= (a1=(a-b))*a1+4*c*c;
   srdelta=sqrt(delta);

   lamda1=  ( (a2=(a+b) ) - srdelta)/2;
   lamda2=  ( a2 + srdelta)/2;

   if(c!=0)
   {
   a3=sqrt(( delta+srdelta * a1)/2);
   u1= (float) c/a3;
   u2= (lamda1- a)/a3;
   }

   else
   {
   u1=1;u2=0;
   }

/*
   Computation of the U matrix.
*/
```

```
   U[0][0]=u1;U[0][1]=-u2;U[0][2]=0;
   U[1][0]=u2;U[1][1]= u1;U[1][2]=0;
   U[2][0]= 0;U[2][1]=  0;U[2][2]=1.0;

   srlamda1=1/sqrt(lamda1);
   srlamda2=1/sqrt(lamda2);
/*
   Computation of the V matrix.
*/

   V[0][0]= ( (float) h11*u1+ (float) h21 *u2 ) *srlamda1;
   V[1][0]= ( (float) h12*u1+ (float) h22 *u2 ) *srlamda1;
   V[2][0]= ( (float) h13*u1+ (float) h23 *u2 ) *srlamda1;

   u3=-u2;
   V[0][1]= ( (float) h11*u3+ (float) h21 *u1 ) *srlamda2;
   V[1][1]= ( (float) h12*u3+ (float) h22 *u1 ) *srlamda2;
   V[2][1]= ( (float) h13*u3+ (float) h23 *u1 ) *srlamda2;

   a1=h12*h23-h22*h13;
   a3=srlamda1*srlamda2;
   V[0][2]= ( (float) a1)*a3;

   a1=h21*h13-h11*h23;
   V[1][2]= ( (float) a1)*a3;

   a1=h11*h22-h21*h12;
   V[2][2]= ( (float) a1)*a3;

/*
   Computation of the rotation matrix.
*/
   R[0][0]=V[0][0]*U[0][0]+V[0][1]*U[0][1];
   R[0][1]=V[0][0]*U[1][0]+V[0][1]*U[1][1];
   R[1][0]=V[1][0]*U[0][0]+V[1][1]*U[0][1];
   R[1][1]=V[1][0]*U[1][0]+V[1][1]*U[1][1];
   R[2][0]=V[2][0]*U[0][0]+V[2][1]*U[0][1];
   R[2][1]=V[2][0]*U[1][0]+V[2][1]*U[1][1];
   R[0][2]=V[0][2];
   R[1][2]=V[1][2];
   R[2][2]=V[2][2];


}.

   void TT(float R[3][3],float T[3][1],XY cm,XYZ cs)
/*
   This function calculates the translation matrix using equation
   IV.44.
*/
```

```
{

   T[0][0]=cs.x-R[0][0]*cm.x - R[0][1]*cm.y;
   T[1][0]=cs.y-R[1][0]*cm.x - R[1][1]*cm.y;
   T[2][0]=cs.z-R[2][0]*cm.x - R[2][1]*cm.y;

}

   void d_initilzation()
/*
   This function initializes the depth of each scene point.
   The same constant value is taken for all points.
*/

{
   register int i;
   float a,b;
   float cs,cm,c;


   c=0.0;
/*
   stimation of the scene perimeter.
*/

   for(i=0;i< N/2;i++)
   {
   b= (a=(float)MS[i][1].x2-(float) MS[i][1].x1) * a +
     (a=(float)MS[i][1].y2-(float) MS[i][1].y1) * a ;
   c+=sqrt(b);
   }

/*
   Estimation of the model perimeter.
*/

   cs=c;
   c=0.0;

   for(i=0;i< N/2;i++)
   {
   b= (a=(float)MS[i][0].x2-(float)MS[i][0].x1) * a +
     (a=(float)MS[i][0].y2-(float)MS[i][0].y1) * a ;
   c+=sqrt(b);
   }

   cm=c;

/*
   Equation IV.29.
*/
```

```
   a= ( f * cm/cs);

   for(i=0;i< N;i++)
   d[i]=a;

}


   void d_updating(float R[3][3], XYZ c)
/*
   This function updates the depths of the scene points using
   equation IV.27 and calculates the estimation error using
   equation IV.25.
*/
{

   float xm,ym,zm;
   float a,b,a1,a2;
   int i;
   float dd;


   sig=0;
   for(i=0;i< N/2;i++)
   {
   xm=R[0][0]*Y1[2*i][0]+R[0][1]*Y1[2*i][1]+ c.x;
   ym=R[1][0]*Y1[2*i][0]+R[1][1]*Y1[2*i][1]+ c.y;
   zm=R[2][0]*Y1[2*i][0]+R[2][1]*Y1[2*i][1]+ c.z;

   a=xm*(a1=(float) MS[i][1].x1/ f) +
     ym*(a2=(float) MS[i][1].y1/ f) +
     zm;
   b=a1*a1+a2*a2+1;
   dd= a/b;

   d[2*i]=dd;
   sig+=(a=xm-(float) dd*a1)*a +
        (b=ym-(float) dd*a2)*b +
      ` (b=zm-(float)  dd  )*b ;


   xm=R[0][0]*Y1[2*i+1][0]+R[0][1]*Y1[2*i+1][1]+c.x;
   ym=R[1][0]*Y1[2*i+1][0]+R[1][1]*Y1[2*i+1][1]+c.y;
   zm=R[2][0]*Y1[2*i+1][0]+R[2][1]*Y1[2*i+1][1]+c.z;

   a=xm*(a1= MS[i][1].x2/ f) +
     ym*(a2= MS[i][1].y2/ f) +
     zm;
   b=a1*a1+a2*a2+1;
   dd= a/b;
```

```
        d[2*l+1]=dd;
        sig+=(a=xm- dd*a1)*a +
             (b=ym- dd*a2)*b +
             (b=zm-  dd  )*b ;
        }

    }
```

```
/*
        ******************************************
          BYNARY SYMBOL 3D POSITION DETERMINATION
            USING THE FLAT-EARTH GEOMETRIC MODEL
        ******************************************


   This module deals with the use of the flat earth model
   to determine the 3D position of each binary symbol.
*/


   #include <stdlib.h>
   #include <tracer1.h>
   #include <conio.h>


   extern POLY *A[30];


   XYZ flat_earth()
{
   float x=0,y=0;
   float a=22.34,b=5.56,c=34.0,d=2.65,f=500;
   int i;
   XYZ ct;

/*
   Computation of the symbol centroid coodinates in
   the image frame.
*/
   for(i=0;i<=6;i++)
   {
   x+=A[i]->x1;
   y+=A[i]->y1;
   }
   x=x/7;y=y/7;

/*
   Computation of the 3D position of the symbol centroid using
   the system IV.7.
*/

   ct.x= d*x/(-a*x-b*y+f*c);
   ct.y= -d*y/(-a*x-b*y+f*c);
   ct.z=  f*d/(-a*x-b*y+f*c);

   return ct;
}
```

```
/*
                ************************************
                   PSEUDO-RANDOM WINDOM POSITIONING
                ************************************

/*
   This module sorts the list of the binary symbols in the
   order they follow each other on the guide-path and extracts
   the pseudo-random window.
*/

   #include <stdlib.h>
   #include <tracer1.h>

   typedef struct bc { char code;
               float x,y,z;
               int order;
               struct bc *next,*prior;} BINARYCODE;


   extern BINARYCODE *BC[40];
   extern int nc;
   extern R[3][3],T[3][1];
   BINARYCODE *tr0;
   float d0=5;
   int n=15;

   BINARYCODE *find_neighbor(BINARYCODE *no);
   void binary_window_positioning();


void binary_code_sort()
{
/*
   This function sorts the list of binary symbols in the order
   they follow on the guide-path.
*/

   int i,im;
   float xa,ya,za,sc,scm;
   BINARYCODE *B,*B1;

/*
.  Find the closest code to the border.
*/

/*1*/    scm=0;

   xa=T[0][0];ya=T[1][0];za=T[2][0];
```

```
   for(i=0;i<=nc;i++)
   {
   sc=(BC[i]->x-xa)*R[0][0]+(BC[i]->y-ya)*R[1][0]
      +(BC[i]->z-za)*R[2][0];

   if(abs(sc)>scm)
         {
         scm=abs(sc);
         im=i;
         }
   }

   BC[i]->order=1;

   store(BC[im]);
   B=BC[im];

/*2*/
/*
   Sort binary code list by finding successively each code's neighbor.

*/

   while((B1=find_neighbor(B))!=NULL)
   {
   store(B1);
   B=B1;
   }

/* Binary window positioning.*/

   binary_window_positioning();
}


   BINARYCODE *find_neighbor(BINARYCODE *C)

/* This function finds the neighbor along the guide path
   of the binary code C.
*/

{

   int i;
   float d,dm=200000,a;
   BINARYCODE *C1=NULL;

/* Find the binary code for which the order is equal zero,
   that minimizes the absolute value of d-d0; d being the
   distance between C and Ci and d0 a constant indicating
```

```
   the distance between two neighboring codes on the track.
*/

   for(i=0;i<=nc;i++)
   {
   if(BC[i]->order==0)
         {
         d= (a=BC[i]->x-C->x)*a + (a=BC[i]->y-C->y)*a
           +(a=BC[i]->z-C->z)*a ;
         if ( abs(d-d0)<dm) { dm=d;C1=BC[i];}

         }
   }

/*
   If the minimum value of d-d0 is small set order of C1 to 1
   else set it to 2.
*/
   if (dm<1) C1->order=1;

   else C1->order=2;

   return(C1);
}

   void binary_window_positioning()

/* This function finds the binary window position.*/

{
   BINARYCODE *C,*C0,*C1;
   int counter=0;

/*
   C0 and C1 indicate the first and the last codes of the binary
   window.
*/

   C=C0=tr0;

   while(C1=NULL)
   {
   if(C->order==1)
         {
         counter++;
         if(counter==n) {C1=C;C->next=NULL;}
         }

   else{ counter=0; C0=C;}
   C=C->next;
   }
```

```
}

   void store( BINARYCODE *C)
{

   static BINARYCODE *last=NULL;

   if(!last){tr0=C;}
   else last->next=C;
   C->next=NULL;
   C->prior=last;
   last=C;

}
```

```
/*
                    ***************************
                       IMAGE SEGMENTATION BY
                        THRESHOLDING MODULE.
                    ***************************
*/


/*
   This module includes the threshold selection
   as well as the image segmentation by thresholding
   routines.
*/


   #include <stdio.h>
   #include <tracerl.h>

   void segmentation(int);


   int thresh()
{

/* This is the threshold selection routine.
*/

   unsigned char buf[256];
   register int min, imin, i;

/* The function fg_histo() calculates the histogram
   of the grey level in the image.
*/

   fg_histo(buf);

/*
   Search of the minimum in the histogram
   graph.
*/

   min=2000;
   imin=-10;

   for(i=20;i<=240;i++)

   {
   if (buf[i]<min)
         {
         imin=i;
         min=buf[i];
```

```
        }
    }

    segmentation(imin);

}

    void segmentation(int thre)

{

/*
    This is function segments the image by thresholding.
*/

    register int i;
    unsigned char buf[256];


    for(i=0;i<=(255-thre);i++)buf[i]=100;
    for(i=(256-thre);i<=255;i++)buf[i]=0;

/*
    The function fg_lutd() using the values stored
    in the buffer, buf, initializes the lookup table.
    The lookup table is a divice part of the frame board
    that maps the incoming data to values set up by the
    function fg-lutd().

*/

    fg_lutd(0,0,0,255,buf);

}

    void threshold(int thre)
{

    register int x,y;

    for(y=0;y<=511;y++)
    for(x=0;x<=511;x++)
    {
    if(fg_pixr(x,y)<thre) fg_pixw(x,y,0);
    else fg_pixw(x,y,100);
    }

}
```