Autonomous Mobile Robot Navigation using Fuzzy Logic Control

Author

Michael Hunt

Supervisor

Charlie Daly

Submitted to
The School of Computer Applications
Dublin City University
for the degree of
Master of Science

June 1998

This is based on the candidate's own work

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science degree, is entirely my own work and has not been taken from the work of others save to the extent that such work has been cited and acknowledged within the text of my work.

Signed : Michael Hunt **ID No. :** 92700721

Michael Hunt

Date : 11/6/98

Acknowledgements

I would like to express my gratitude to my supervisor Charlie Daly whose help and guidance was invaluable during the course of this thesis, my friends and fellow postgraduates who provided a pleasant atmosphere that made the whole experience worthwhile. I also thank Dermot Hannan for his collaboration in getting the robot up and running. I would also like to thank Simon Hayter for his help and guidance beyond the call of duty.

Table of Contents

ABSTRACT

Traditionally the type of robot used in the workplace consisted mainly of the fixed arm variety. Any mobile robots that were commercially available required that the environment be altered to accommodate them. This involved the installation of guide lanes or some form of sensor units placed at various locations around the workplace to facilitate the robot in determining its position within the environment.

Such approaches are costly and limit the use of robots to environments where these methods are feasible. The inadequacies in this technology has led to research into autonomous mobile robots that offer greater flexibility and do not require changes in the environment. There are many technical issues to be addressed in designing such a robot. These stem from the necessity that the robot must be able to navigate through an environment unaided. Other problems such as the cost of the vehicle must be considered so that prospective customers will not be put off.

This thesis discusses the strategies taken in addressing the problems associated with navigation in an obstacle strewn environment. Such issues include position estimation, path planning, obstacle avoidance and the acquisition and interpretation of sensor information. It also discusses the suitability of fuzzy logic for controlling a robot.

A graphical user interface runs on the PC which communicates with the robot over a radio link. The robot uses a fuzzy logic controller to follow a planned path and avoid unknown obstacles by controlling the velocity and steering angle of the drive unit. It is a tracked vehicle which is suitable for indoor use only. The results of path planning and the robots attempts at following the paths and avoiding obstacles are illustrated and discussed.

Chapter 1

Introduction

1. Introduction to Thesis

There is the need in industry and in space exploration for robots that have greater autonomy. Traditionally robot arms worked in a fixed environment with a fixed co-ordinate reference frame. Mobile robots that operated in the environment did so by using techniques that involved altering the environment. An Autonomous vehicle is a self propelled vehicle that provides a means for transporting materials in the workplace. The vehicles are designed to have the capability of negotiating their own way around the environment through some on board intelligence that can also enable them to find the way around dynamic or static obstructions. These vehicles may receive their instructions from some off-board controller.

There are a number of considerations to be dealt with, when designing such a vehicle. The navigation system must be reliable and have sufficient accuracy to ensure the safety of nearby humans, the environment and the vehicle itself. The vehicle must be able to move with sufficient speed to make it a practical conveyance. The cost may also become a prohibitive factor if the vehicle has a high degree of complexity. If the robot is to be capable of performing tasks successfully in a dynamic environment it must have the ability to respond to changes in the perceived world that affects its operation. As such, it must be able to acquire and interpret information obtained from the surrounding environment. Multiple sensors utilising appropriate sensor fusion techniques can help ameliorate problems arising from the differences between the real and perceived world.

A method often used for the guidance of mobile robots is the use of guidepaths. In particular, vehicles used in heavy industrial environments usually follow a wire buried in the factory floor. The wire is used as a transmitting antennae and is energised by an alternating electrical current oscillating at a few kilohertz. A pair of receiving antennae is

mounted on the base of the vehicle which are used to straddle the buried transmitting wire. This enables the vehicle to track the wire by maintaining a balance in the strength of the signals received by each of its sensor antennae. High frequency communication signals can also be sent on the transmission wire allowing two-way communication with the vehicle.

Another approach taken uses photofluorescent or reflective materials applied in stripes on the floor to define the path network. A light source and a video adapter is then mounted on the base of the vehicle which operates in a similar manner to the guide wire system by keeping the detector positioned over the path. A radio is used to communicate with the robot or alternatively communication can be achieved through a link at a queuing station. The optical or chemical stripes are cheaper to lay down than the guidewire systems but require periodic maintenance and still offers no communication link.

These systems are all classed as fixed path systems and while they are reliable, the paths are difficult and expensive to change particularly in the case of guidewire systems. Guidewires cannot be installed in metal or wood block floors, under metal grates, in elevators or in sterile environments. Optical paths suffer from abrasion or obstructions in dirty environments and are as such, unsuitable for outdoor or heavy industrial applications. Another problem with the hardware configuration is that vehicles produced by one manufacturer are often incapable of following the guidewire or communicating with a controller produced by another manufacturer. If the system has to be updated or expanded, the same manufacturer's vehicles must be purchased or parallel guidepaths must be laid down to accommodate incompatible equipment. In dynamic environments, the fixed path vehicles may not be practical and this combined with the need for a more flexible approach has led to research into autonomous mobile robots.

Autonomous mobile robots offer greater flexibility and can plan their own path internally or receive electronic instructions that can be interpreted through software.

With this process any vehicle is capable of communicating over the same medium. Sometimes passive landmarks are used to facilitate the robot in negotiating its path. Even though such methods involve altering the environment, it is far easier to insert landmarks than it is to lay down guidewires.

There are a number of ways control can be distributed in autonomous robots, from hierarchical to distributed, or indeed a mixture of both. The hierarchical structure is commonly used and while robots may have a particular control structure in common, typically they can appear totally different in terms of functionality, complexity and design.

An important aspect to be considered in discussing Autonomous vehicles is the type of path planner that is to be used. There are several methods of path planning that can be employed for use with robots. Every planner has its own characteristics. Some are more suitable for use with dynamic environments than others. This is due to the type of planning representation used which in some cases cannot easily be altered to accommodate new data without having to be rebuilt from the start. Other planners will attempt to find the shortest path to a destination, but at a very high computational expense. Some planners are developed to minimise the computational time incurred when finding a path through a series of obstacles.

The choice of algorithm is dependent on the application. However, if a vehicle is to have the capacity to work in a dynamic environment, it should be able to integrate new data representing possible obstacles with the global map for the purposes of re-planning. Re-planning is necessary if the robot fails to pass some unmapped obstacle. This approach differs from the planners that are used for fixed arm robots. Such planners safely assume that no obstacles are encountered during the course of planning that would warrant the necessity to re-plan.

The accuracy with which the robot can maintain an estimate of its position is very important because it determines to a certain extent, the reliability and effectiveness of the robot within a working environment. An autonomous robot must be able to determine where it is in order to perform effectively. This is a difficult task if the environment is not to be altered to provide information that could enable the robot to determine its position. There are two approaches used in tandem that may be used to update the robot's position. One will acquire information from the sensors and the other will use Dead Reckoning.

The integration of data from sensors to form maps for use in position estimation is a complex issue. The type of problems that arise in this area are sensor dependent. Some sensors provide excellent directional information relating to where an object is in the robots path, but may offer a poor estimate of the robot's distance from the object. Other sensors can be better at obtaining distance measurements but offer poor directional information. In many implementations within robotics, the designers utilise the advantages of various sensors by integrating the information obtained from each sensor to compensate for the limitations of any particular type of sensor. Such sensors include lasers, ultrasonics and vision.

Information acquired from the sensors may be used to facilitate the process of position estimation. Dead Reckoning is commonly used as a 'local' method for maintaining an estimate of the position but is subject to a number of problems which result in an accumulation of errors if not corrected over time. Sensor information may be used to periodically update and correct the information obtained using Dead Reckoning. Sensors are also used to avoid obstacles that are not known to the global map. Obstacle identification and avoidance is imperative for an autonomous mobile robot and only possible through the acquisition of information from sensors. It is therefore important to consider what sensors to use with a robot as this can greatly enhance/limit the robot's capabilities.

The mobility of a robot is dependent on the wheel configuration used in its construction. There are a number of possible configurations that may be used. Some types offer greater flexibility of movement than others and can prove useful when operating in a cluttered environment. Most configurations cannot allow movement in the direction of the wheel axle. However, the omni-directional wheel has been developed to remove this limitation.

2. Objective of Research

It is evident that if a vehicle is to exhibit a high degree of autonomy there will be a number of distinct issues that must be dealt with. The following chapters discuss those outlined above. Chapter two provides a detailed discussion of Fuzzy Logic. Its suitability for use as a controller that is capable of avoiding obstacles or following a path is discussed. Chapter three follows with a breakdown and analysis of the control structures used in the design of autonomous mobile vehicles. Also considered are the various approaches taken in path planning and navigation. Chapter four discusses wheel configurations and sensor characteristics, describing the advantages and disadvantages of each. It also discusses the integration or fusion of information acquired from sensors as an aid to position estimation and obstacle avoidance. Chapter five illustrates the work completed with an indoor mobile robot. It describes the hardware and the operation of the path planner and navigator. Results are presented of the path planner and fuzzy controller in operation. Chapter six contains the conclusions.

Chapter 2.

Fuzzy Logic

1. Introduction

Fuzzy set theory was first developed in 1965 by Prof. Lotfi Zadeh as an extension to bilevel (boolean) logic. Since its introduction it has emerged as a powerful approach to reasoning when using uncertain data. Classical set theory is useful in problems where the output can easily be defined from knowledge of the input. Unfortunately many problems do not fall into this domain. The relation between input and output may not always be readily defined and it is to such problems that fuzzy logic can prove useful.

Fuzzy logic provides a powerful and straightforward means of problem solving where mathematical models are not easily definable or are difficult to develop. It has been successfully applied in areas such as process control, pattern recognition, expert systems and linguistics. In process control it obviates the need to develop a mathematical model for the system while in decision making it is able to draw conclusions from vague, incomplete or imprecise information.

The behaviour of a fuzzy system is described linguistically, which provides these systems with a reasoning capability similar to humans. Ambiguous statements such as *more or less* or *fairly fast* are easily dealt with using fuzzy reasoning. This is because any element in a fuzzy set has a given membership of that set. Using fuzzy logic, it is possible to devise a vocabulary of terms which define on a mathematical scale the linguistic terms used in expressing the control rules. Specific applications includes washing machines, camcorder autofocusing of lenses, controlling subway systems.

The construction of a fuzzy logic control (FLC) system is faster than traditional methods of control. It is understandable, robust and maintainable and in general requires much less memory and computing power than conventional methods.

Traditional control technology is based on a mathematical model that describes the control process. A set of model equations is derived from physical laws or identification algorithms and then a set of feedback control laws is generated to ensure these models behave as desired. Although this is perfectly satisfactory for simple processes, it gets more difficult as the process becomes more complex. Fuzzy logic on the other hand is very suitable for dealing with non-linear processes having noisy input data and has emerged in industry as one of the most promising techniques available.

It is easy to design fast control systems with fuzzy logic. However, there is no general procedure for deciding on the optimal number of fuzzy control rules since a number of factors are involved in the decision, e.g., performance of the controller, satisfactory system performance, the choice of fuzzy sets and the variables.

In Japan, there has been a great deal of research in the use of fuzzy logic. Several fuzzy logic implementations on VLSI chips and fuzzy logic software development systems have become available recently, allowing an engineer to quickly design, prototype and implement a system without worrying about the low level implementation of the fuzzy calculations, and allowing flexible choice of target system processor and other design details after the top level design is completed.

Like conventional industrial programmable logic controllers (PLC), Fuzzy Logic Controllers (FLC) also need hardware and software support for information processing. Information processing is performed within a Fuzzy Logic Controller by the Digital Fuzzy Processor (DFP). Presently, several companies in Europe and the USA are devoted to the manufacture of these processors.

A fuzzy logic controller is a knowledge based controller that uses fuzzy set theory and fuzzy logic for knowledge representation and inference. One advantage over other knowledge based controllers lies in the interpolative nature of the control rules. The overlapping fuzzy antecedents to the control rules provide smooth transitions between control actions of different rules. Because of this interpolative quality, fuzzy

logic controllers often require an order of magnitude fewer rules than other knowledge based controllers.

2. Fuzzy sets and permissible operations

Traditional bilevel logic provides for variables with values that are either TRUE or FALSE. No value that lies between these two extremes is allowed. The most common logic operations are AND, OR and NOT and when applied they return a crisp value that is either TRUE or FALSE. Traditional logic is based on Classical set theory which states that a given element is either entirely within a set or entirely outside a set i.e. either it is a member of that set or it is not. The two dominant operators available in classical set theory are INTERSECTION (the basis for the logical AND operation) and UNION (the basis for the logical OR operation).

Fuzzy set theory also has associated operations as in Classical set theory. However a fuzzy set allows values to be partial members of a set. Membership of a set ranges from 0 to 1, i.e. from no membership to full membership respectively. A comparison of a traditional crisp set and a fuzzy set is shown in fig. 1. Fuzzy sets have similar primary operations: intersection, union and complement although they are defined differently. Other operations such as concentration, dilation and intensification are used as hedges for modifying the base sets.

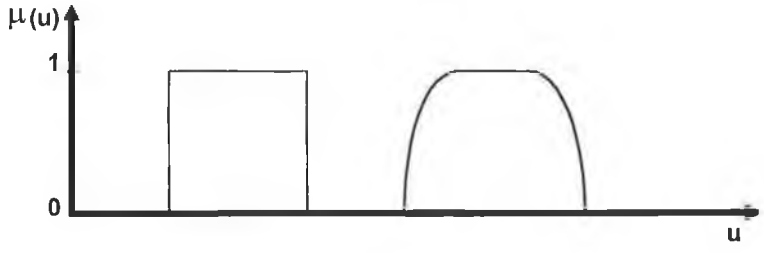


Figure 1.

A fuzzy set A of a universe of discourse U is characterised by a membership function $\mu_A(u)$ which assigns to every element $u \in U$ a number $\mu_A(u)$ in the interval 0 to 1 that represents the grade of membership of the element in the set A. This may be written as follows:

$$A = \{ u, \mu_A(u) \mid u \in U \}$$

When U is a discrete set this may be represented as follows:

$$A=\Sigma\mu_A(u_i)/u_i$$

where the summation denotes the set theory union operator rather than the arithmetic sum and the / relates a particular membership function to an element of the set.

Degree of membership

In fuzzy logic an element is assigned a degree of membership within a given set. The degree to which an element belongs to a set is given by its degree of membership within that set. The degree μ to which an element u belongs to a fuzzy set A is given as

$$\mu_A(u) \rightarrow [0,1]$$

This reads as "the degree of membership of u in the fuzzy set A ranges from 0 to 1 inclusive". The number of fuzzy set *membership functions* and the shapes you choose depend on such things as required accuracy, responsiveness and stability of the system, ease of manipulation and maintenance. The most commonly used forms are the:

- triangular type
- trapezoid type
- exponential type
- s-function
- Π-function

The S-function is defined as follows:

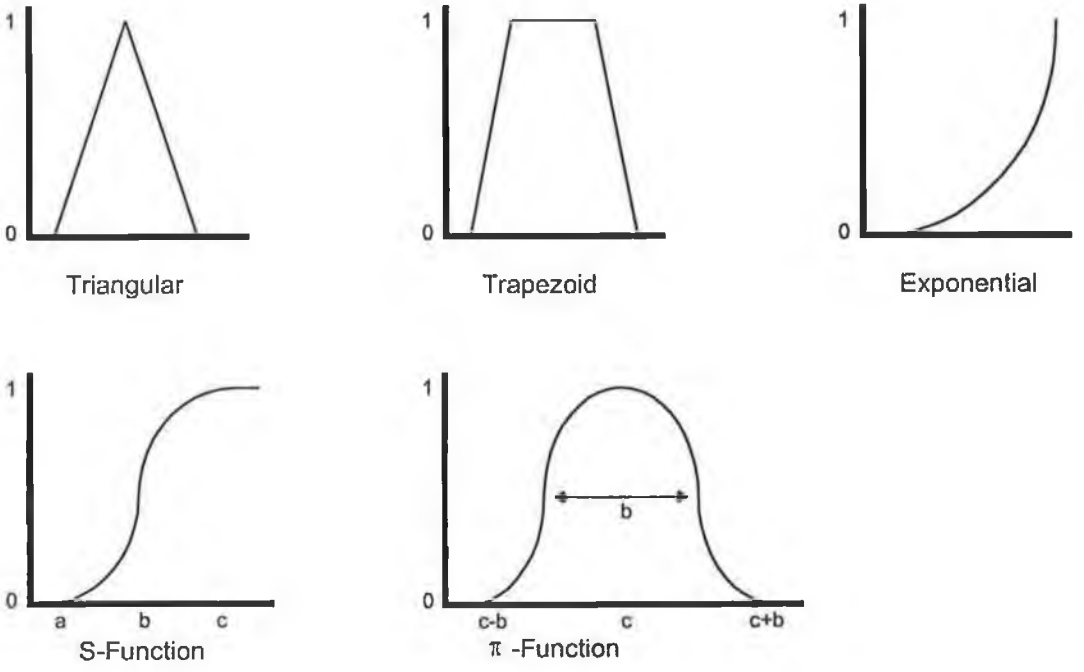
$$S(x;a,b,c) = \begin{cases} 0 & \text{for} \quad x \le a \\ 2\left[\dfrac{(x-a)}{(c-a)}\right]^2 & \text{for} \quad a \le x \le b \\ 1-2\left[\dfrac{(x-a)}{(c-a)}\right]^2 & \text{for} \quad b \le x \le c \\ 1 & \text{for} \quad x \ge c \end{cases} \qquad \text{where } b = (a+c)/2$$

The $\prod$-function is defined as follows:

$$\prod(x;a,c) = \begin{cases} S(x;c-b,c-b/2,c) & for \quad x \le c \\ 1-S(x;c,c+b/2,c+b) & for \quad x \ge c \end{cases} \quad \textit{where b is the bandwidth}$$

These functions when plotted have the following appearance:

Fuzzy set Membership Functions



The following is a list of the operations available in fuzzy set theory.

Let A and B be two fuzzy sets in U with membership functions μA and μB, respectively.

Set union

$\mu_{A\cup B}(u) = \max\{\mu_A(u),\mu_B(u)\}$ for all $u \in U$

Set intersection

$\mu_{A\cap B}(u) = \min\{\mu_A(u),\mu_B(u)\}$ for all $u \in U$

Set complement

$\mu_{-A}(u) = 1-\mu_A(u)$ for all $u \in U$

Set concentration

$\mu_{CON(A)}(u) = \mu_A^2(u)$ for all $u \in U$

Set dilation

$\mu_{DIL(A)}(u) = \mu_A^{1/2}(u)$ for all $u \in U$

Set intensification

$$\mu_{INT(A)}(u) = \begin{cases} 2\mu_A^2(u) & for \quad 0 \le \mu_A(u) \le 0.5 \\ 1-2[1-\mu_A(u)]^2 & for \quad 0.5 \le \mu_A(u) \le 1 \end{cases}$$

Some examples of fuzzy operators are shown in Figure 2 when applied to the fuzzy set A.

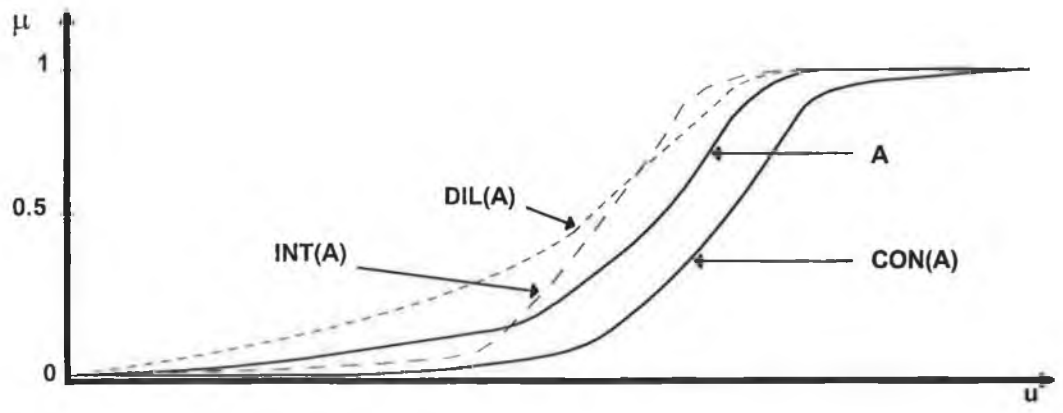


Figure 2.

Hedges

Terms such as **VERY** and **SLIGHTLY** are called hedges. Generally a hedge is a word that when applied to a fuzzy set modifies that fuzzy set relative to its original form. An example is shown in fig.3 where two fuzzy sets **HOT** and **VERY HOT** are illustrated. The term **VERY** when applied to the set **HOT** creates a new fuzzy set **VERY HOT** that is the original set shifted to the right. The use of a hedge provides a relational capability between the base set and the hedge set. This means that should the set **HOT** be modified then a corresponding shift should occur in the set **VERY HOT**. The definition of a hedge will be strictly coupled to the base set[1]. For example the hedge **VERY** would have a very different effect on the set **COLD** where **VERY COLD** is shifted to the left.

Hedges are formed by applying fuzzy operators to a fuzzy set. In Fig. 3, the hedge set **VERY HOT** is formed from the set **HOT** by performing the **CON(HOT)** to create the new set. This set is similar in form to the original one but shifted to the right. Fig. 4 shows the effects of same hedge when applied to the fuzzy set **COLD**. The new hedge set **VERY COLD** is obtained by performing the **CON(COLD)**. Notice that the hedge shifts the set to the left in this case.

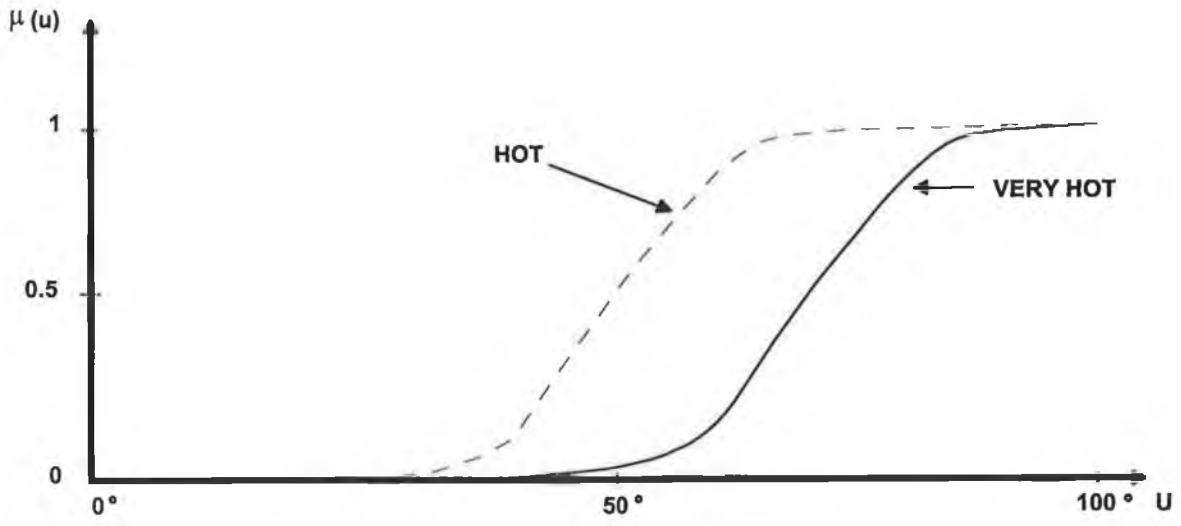


Figure 3.

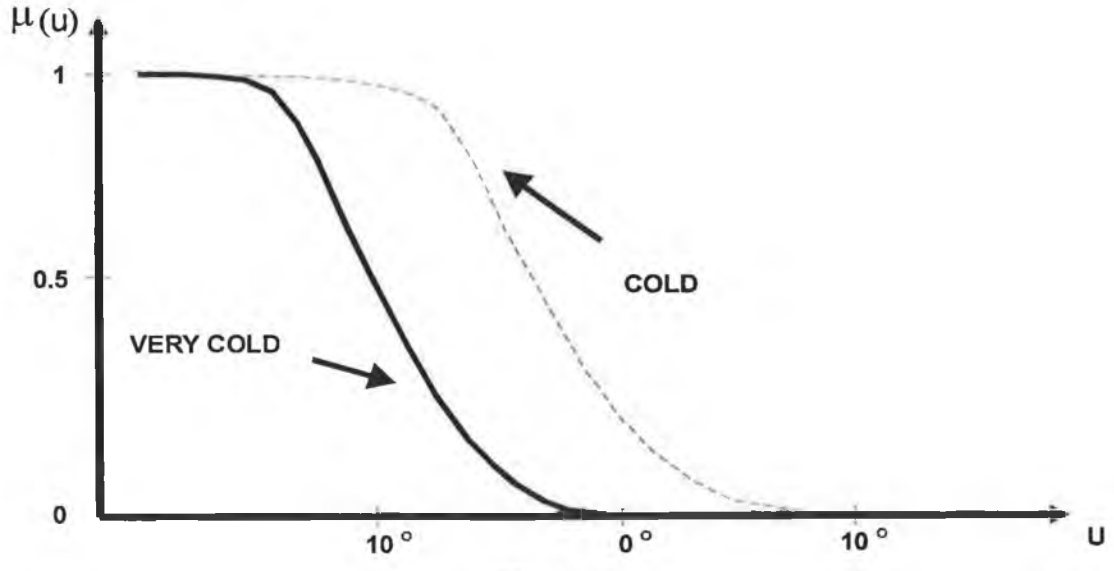


Figure 4.

3. The Fuzzy Control System

Fuzzy based systems are constructed so that generated outputs change in a smooth and continuous manner, regardless of inputs crossing set boundaries. System inputs undergo three transformations to become system outputs. First a fuzzification process uses predefined membership functions to map each system input into one or more degrees of membership. Then the rules in the rule base are evaluated by combining degrees of membership to form output strengths. And lastly the defuzzification process computes system outputs based on these output strengths and membership functions. A block diagram of a fuzzy control system[2] is shown in fig. 5.

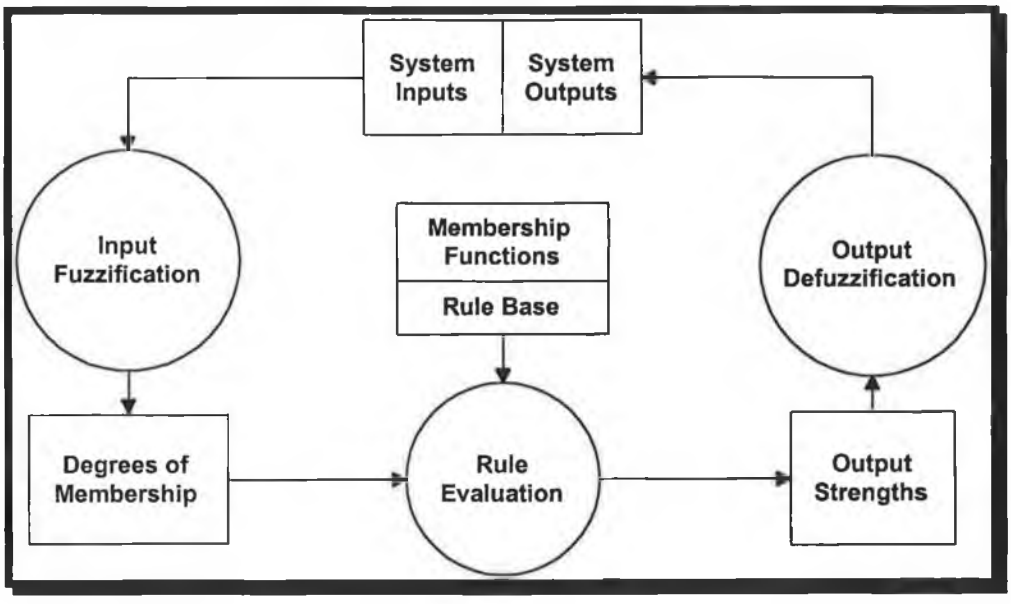


Figure 5.

3.1 Fuzzification

Fuzzification is the process of calculating the values to represent an input's *degree of membership* within one or more fuzzy sets. These values can then be used to determine the degree of truth for each rule premise. Fig. 6 illustrates an example where the fuzzy sets are cold, cool, warm and hot. Each temperature has a degree of membership within each of these sets. The degree of membership is determined by a membership function which is defined based on experience or intuition.

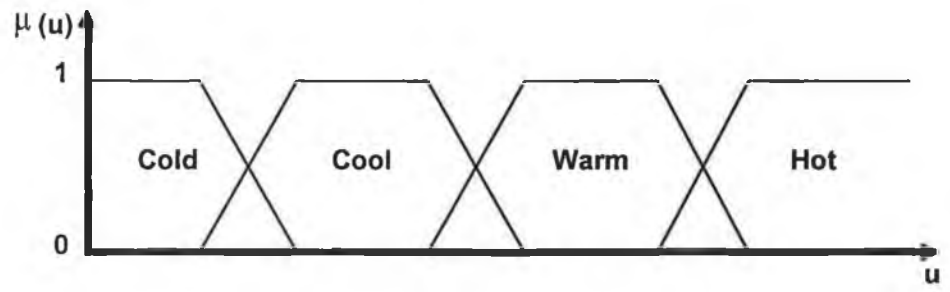


Figure 6.

3.2 Inference

The inference mechanism is that which processes the rules and determines which actions are performed by the control process based on a specific input condition. A rule is composed of an input and output. The input part of a rule is called the premise and it is composed of one or more antecedents related by fuzzy operators. The output part of the rule is called the consequence. When a rule fires, it fires to the degree dependent on the truth level in each antecedent in the premise of the rule. Under inference, the truth value for the premise of each rule is computed, and applied to the conclusion part of each rule. This results in one fuzzy subset being assigned to each output variable for each rule.

The antecedents of a rule are evaluated using membership functions to produce truth levels which are combined using Fuzzy operators to produce a final degree of fulfilment for that rule. The antecedents correspond directly to degrees of membership calculated during the fuzzification process. Generally a minimum function is used so that the strength of a rule is assigned the value of its weakest or least true antecedent. The degree of fulfilment (dof) of a rule is used to determine the strength or weighting of the consequent in the rule. The two methods most commonly used in calculating the weighting or degree of fulfilment of a rule are **Max-Min Inference** and **Max-Dot Inference**.

3.2.1 Max-Min Inference

In Max-Min inferencing, the output membership function is clipped off at a height corresponding to the rule premise's computed degree of truth. The fuzzy logic AND operator which takes the *Minimum* of the input sets, is used in computing the degree

of truth of the premise. When this is performed, all of the fuzzy subsets assigned to each output variable are combined together to form a single fuzzy subset for each output variable. This involves taking the pointwise *Maximum* over all of the fuzzy subsets assigned to the output variable by the inference rule, which corresponds to a UNION of the fuzzy subsets or a fuzzy logic OR operation. Hence the name Max-Min inference.

As an example assume a fuzzy control rule base has only two rules as follows:

Rule 1: IF x is A1 and y is B1 THEN z is C1

Rule 2: IF x is A2 and y is B2 THEN z is C2

The first step in Max-Min inference involves clipping the membership function of the rule output variable at the degree of fulfilment level of the premise[3]. This may be written as:

$$\mu_{C'}(u) = \text{MIN}\{\text{dof}, \mu_C(u)\}$$

where $\mu_C(u)$ represents the unclipped membership function of the output Fuzzy set and $\mu_{C'}(u)$ represents the clipped membership function.

If the strength of the i^{th} rule is denoted by α_i, the control decision led by the i^{th} rule can be expressed by:

$$\mu_{ci'}(w)=\min[\alpha_i, \mu_{ci}(w)]$$

The second step in Max-Min inference process involves computing a single fuzzy subset for the output variable. This is attained by performing a fuzzy OR (Max) operation on the output sets. The membership of the inferred consequence C is thus point-wise given by:

$$\mu_c(w)=\max[\mu_{c1'}(w), \mu_{c2'}(w)]$$

$$\Rightarrow \mu_c(w)=\max[\min[\alpha_1, \mu_{c1}(w)], \min[\alpha_2, \mu_{c2}(w)]]$$

Figure 7 shows the max-min inference processes with respect to the different types of inputs. In fig. 7a the fuzzy sets A' and B' have been taken as the inputs, while in fig. 7b crisp values x_o and y_o are taken as the inputs.

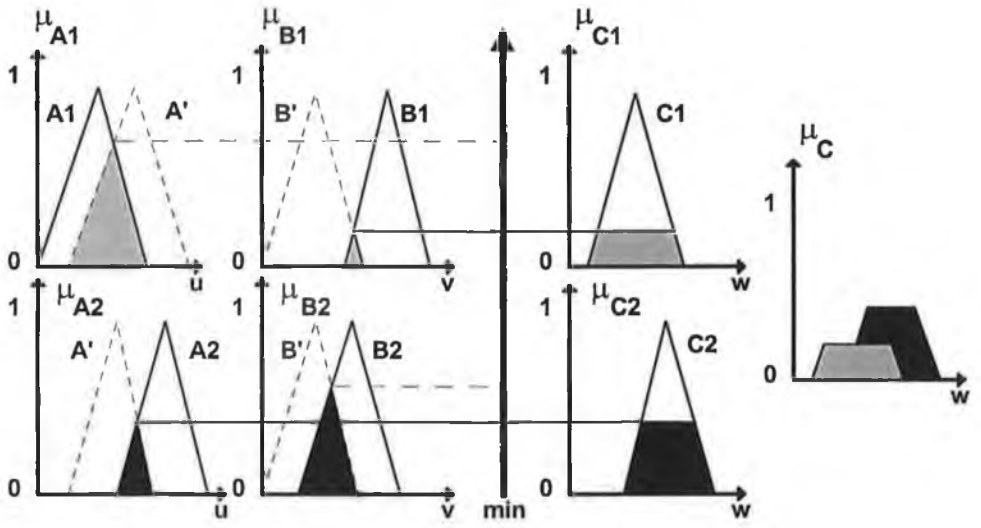


Figure 7a: max-min inference under fuzzy inputs

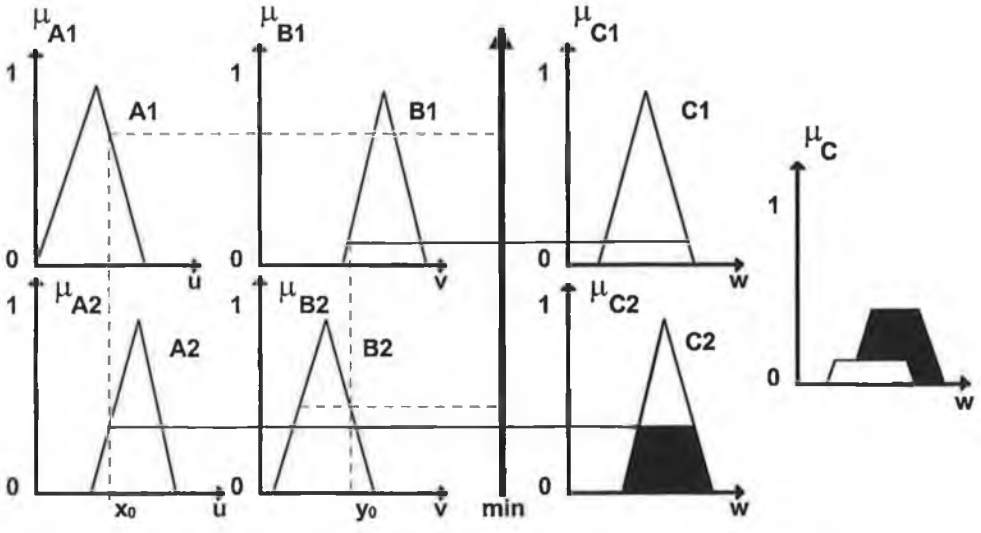


Figure 7b: max-min inference under crisp inputs

3.2.2 Max-Dot Inference

In Max-Dot Inferencing the output membership function is scaled (using the *Dot* product) by the rule premise's computed degree of truth. Then the output subsets are combined for each output variable using the OR operator (*Maximum*). Hence the name Max-Dot Inference.

The following procedure demonstrates the process which is applied to the rule base listed for the last example.

The first step of the Max-Dot Inference process involves scaling the membership function of the rule output variable by the dof of the premise. This may be written as:

$$\mu_{C'}(u) = \text{dof}.\mu_C(u)$$

where $\mu_C(u)$ represents the unscaled membership function of the output Fuzzy set and $\mu_{C'}(u)$ represents the scaled membership function. The control decision led by the i^{th} rule can be expressed as:

$$\mu_{ci'}(w)=\alpha_i \bullet \mu_{ci}(w)$$

The second step of the inference process involves computing a single fuzzy subset for the output variable. The membership of the inferred consequence C is therefore given by:

$$\mu_c(w)=\max[\mu_{c1'}(w),\mu_{c2'}(w)]$$
$$\Rightarrow \mu_c(w)=\max[(\alpha_1 \bullet \mu_{c1}(w)), (\alpha_2 \bullet \mu_{c2}(w))]$$

Fig 8 illustrates the MAX-DOT inference processes with respect to the different types of inputs. In fig. 8a the fuzzy sets A' and B' have been taken as the inputs. In fig. 8b crisp values x_0 and y_0 have been taken as inputs.

Under the crisp input conditions, it can be noted that the fire strength α_1 and α_2 of the rule base may be denoted by:

$$\alpha_1=\min[\mu_{A1}(x_0),\mu_{B1}(y_0)]$$
$$\alpha_2=\min[\mu_{A2}(x_0),\mu_{B2}(y_0)]$$

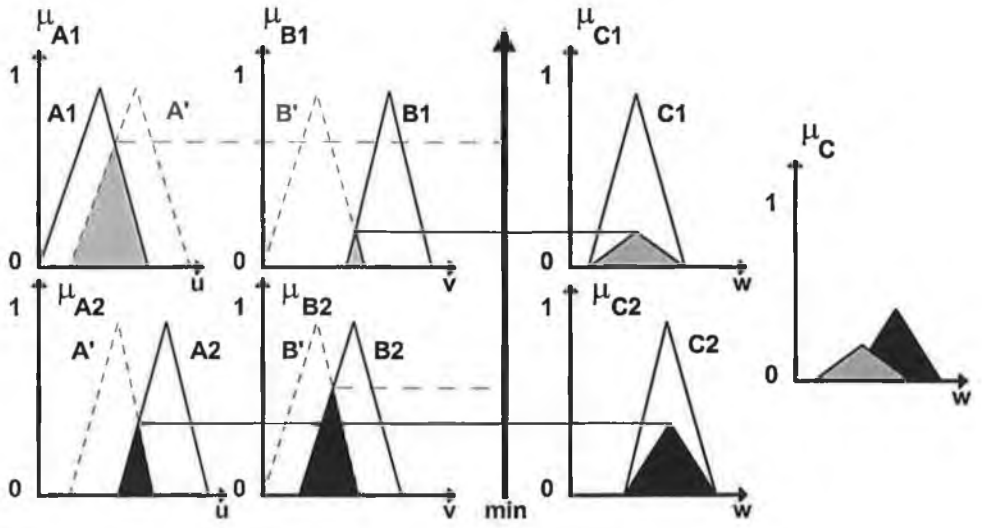


Figure 8a MAX-DOT inference under fuzzy inputs

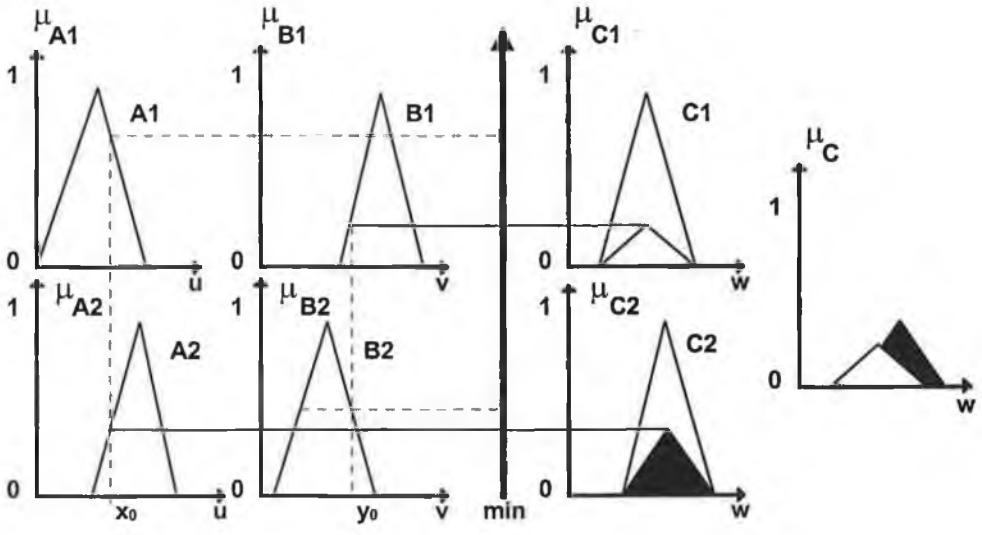


Figure 8b MAX-DOT inference under crisp inputs

3.2.3 Cartesian Product

The cartesian product has a direct application in inference where the products of more than one universe of discourse is being considered. The operations union, intersection and complement all operate only on a single universe of discourse. The cartesian product of A and B in the product plane is defined as

$$\mu_P(u,v) = \mu_{A\times B}(u,v) = \text{MIN}[\mu_A(u), \mu_B(v)] \text{ for all } u \varepsilon \text{ U}, v \varepsilon \text{ V}$$

An example of where this would be used is as follows. Consider the universe of discourse U of an input fuzzy set A and V the universe of discourse of a fuzzy set B. If these fuzzy sets have a rule associated with them of the form:

IF (input1 is A) AND (input2 is B) THEN output is C

then the truth value of the premise is given by:

$$MIN[\mu_A(u),\mu_B(v)] \text{ for all } u \varepsilon U, v \varepsilon V$$

3.3 Defuzzification

The rule-evaluation process assigns weights to each action that is activated. Further processing is required in order to decipher the meaning of the vague or fuzzy actions using membership functions and to also resolve conflicting actions. A number of methods[4] may be employed in order to reach a compromise between the various actions that have been triggered during rule evaluation. The most common of these are:

- The centroid method
- The centre of sums method

3.3.1 Centroid method

The composite output fuzzy set is first built by taking the union of all clipped or scaled output fuzzy sets. Then a crisp output value is obtained by deriving the centroid of the composite output fuzzy set. Thus given fuzzy sets A, B and C the membership function of the composite fuzzy set would be given by

$$\mu_{comp}(u) = \max\{\mu_{A'}(u)\mu_{B'}(u)\mu_{C'}(u)\}$$

for all $u \in U$, where U is the output Universe of Discourse and A', B' and C' are the qualified output fuzzy sets corresponding to the output sets A, B and C. The centroid or centre of gravity is now obtained for the discrete case using the following formula.

$$u^* = \frac{\sum_{k=1}^{n} \mu_{comp}(u_k).u_k}{\sum_{k=1}^{n} \mu_{comp}(u_k)}$$

For the continuous case the following formula applies.

$$u^* = \frac{\int \mu_{comp}(u).u.du}{\int \mu_{comp}(u).du}$$

In summary a centroid point on the x-axis is determined for each output membership function. Then the membership functions are limited in height by the applied rule strength and the areas of the union of the output membership functions are computed. Then the crisp output is derived by a weighted average of the x-axis centroid points and the computed areas, with the areas serving as weights.

The centroid method is computationally intensive and will therefore result in slow defuzzification cycles. As the centroid method takes into account the area of the composite output set as a whole, should two qualified output sets overlap then the overlapping is not taken into account. This means that the crisp output does not constitute a complete representation of the qualified output sets that were used as input to the defuzzifier. One other point regarding this method is that for a vertically symmetric membership function when only a single rule fires, the centroid method will always return a constant value that is independent of the input degree of fulfilment.

3.3.2 Centre of Sums

This method is similar to the centroid method but it takes into account overlapping membership functions in the qualified output set. Instead of building the output composite using the union operator, the centre of sums method takes the SUM of the output fuzzy sets. The formula for the discrete case is given by:

$$u^* = \frac{\sum_{k=1}^{m} u_k \sum_{i=1}^{n} \mu_{Ai}(u_k)}{\sum_{k=1}^{m} \sum_{i=1}^{n} \mu_{Ai}(u_k)}$$

And in the continuous case:

$$u^* = \frac{\int u_k \sum_{i=1}^{n} \mu_{Ai}(u_k) du}{\int \sum_{i=1}^{n} \mu_{Ai}(u_k) du}$$

Sometimes "Singletons" are used to simplify the defuzzification process. A singleton is an output membership function represented by a single vertical line. Since a singleton intersects the x-axis at only one point, the centre of gravity calculation reduces to just a weighted average calculation of x-axis points and rule strengths, i.e a weighted average calculation of all the qualified output sets. This approach is widely used in hardware implementations.

4. Derivation of Control Rules.

There are three possible modes of deriving fuzzy control rules[5] based on:

1. The operator's experience (verbalisation).
2. Fuzzification.
3. Identification.

1. This is the most subjective method, and directly akin to expert systems rule acquisition, whereby a rule set is generated by interrogating an experienced operator through structured questions. The vast majority of application studies have used this method to synthesise fuzzy logic controllers. The rules are derived by placing the controller in parallel with a human expert and learning or imitating the control actions for particular input/output situations.
2. The second method involves formulating a set of rules from a mathematical expression that represents the system. This system can then be validated against the known analytical model. An example is shown below where a decelerating

vehicle with initial velocity v_0 and constant deceleration $-a$ comes to a stop at a distance d from where the breaks were initially applied.

Using Newton's equation of motion: $v^2 = u^2 + 2\,a\,d$ (where v is the final velocity, u is the initial velocity, a is the acceleration and d is the distance travelled) the relationship between the velocity, deceleration and distance is:

$$d = v_0^2/2a \quad (1)$$

The *velocity* and *acceleration* are input variables that are mapped through equation 1 into *distance*. For acceleration control, this relationship needs to be inverted to $\langle velocity, distance \rangle \rightarrow \langle required\ acceleration \rangle$. In mathematical terms this is:

$$a = v_0^2/2d. \quad (2)$$

In general however, the causal relationship may not be invertible. The linguistic qualifiers for velocity, distance and deceleration may be defined as follows:

Velocity=(Very Slow, Slow, Medium, Medium Fast, Fast, Very Fast)

Distance=(Almost Zero, Very Close, Close, Medium, Medium Far, Far, Very Far)

Deceleration=(Almost Zero, Braking, Hard Braking, Very Hard Braking)

Using equation 2 as a guide, the braking rule base of Table 1 may be obtained. An example of how two of the rules were derived is shown below.

Rule 1

Velocity = VS

Distance = VF

Deceleration = $(VS)^2/(2*VF)$ = (small number)/(large number) = AZ

Rule 2

Velocity = VF

Distance = AZ

Deceleration = $(VF)^2/(2*AZ)$ = (largel number)/(small number) = VHB

Table 1 Static breaking fuzzy rule base

distance	velocity					
	VS	S	M	MF	F	VF
AZ	B	HB	HB	VHB	VHB	VHB
VC	B	B	HB	HB	VHB	VHB
C	AZ	B	B	HB	HB	VHB
M	AZ	B	B	B	HB	VHB
MF	AZ	B	B	B	HB	VHB
F	AZ	AZ	B	B	B	HB
VF	AZ	AZ	AZ	B	B	B

3. The third approach is used to provide a system model for either the purpose of output prediction and simulation or for the design of a feedback controller. This estimation method requires adequate representative input/output signal pairs to provide a relational matrix which represents all, or nearly all possible input/output situations.

   Consider a causal and a time-invariant process, represented by a finite-dimensional fuzzy relation R that maps current states S(t) into future states S(t+T) (T is the sample period) for some input U(t), i.e.

$$S(t)xU(t) \xrightarrow{R} S(t+T).$$

   The fuzzy relation R is a collection of statements or fuzzy rules of the form:

$$R_i: \quad \text{IF } S_i(t) \text{ AND } U_i(t) \text{ THEN } S_i(t+T); \quad i=1,2,....,N.$$

   R may be described as a collection of N Fuzzy rules mapping the current state S(t) and input U(t) into the future state S(t+T). If $S_i(t)$,$U_i(t)$ and $S_i(t+T)$ are described by fuzzy sets with membership functions $\mu_{Si(t)}$, $\mu_{Ui(t)}$, $\mu_{Si(t+T)}$ respectively then the membership function for this rule is:

$$\mu_{R_i}(S(t),U(t),S(t+T)) = \min\{\mu_{S_i(t)}(S(t)),\mu_{U_i(t)}(U(t)),\mu_{S_i(t+T)}(S(t+T))\}$$

The fuzzy sets are described on finite discrete universes for practical applications. Thus both R_i and the relational matrix R are finite discrete fuzzy relations. To construct R, collect all N rules, R_i, as R_1 OR R_2 OR R_3 OR ... R_N. Hence the membership function of R is:

$$\mu_R(S(t), U(t), S(t+T)) = \max_i \{\mu_{R_i}(S(t), U(t), S(t+T))\}$$

$$\mu_R(S(t), U(t), S(t+T)) = \max_i \{\min\{\mu_{S_i(t)}(S(t)), \mu_{U_i(t)}(U(t)), \mu_{S_i(t+T)}(S(t+T))\}\}$$

The relational matrix R must be evaluated from system input/output data, its dimensionality N being unknown a priori. Therefore every data set ($S_i(t)$, $U_i(t)$,...,$S_i(t+T)$) is considered as a possible rule, and is used to update the relational matrix R.

It is necessary however, to track systems with time-varying parameters and those with catastrophic changes through system faults, and as such the relational matrix must have learning and forgetting capabilities. This is accomplished through a forgetting operator D. At each update a new version say R', is computed from the previous version R through:

$$\mu_{R'}(S(t), U(t), S(t+T)) = \max\{(D \times \mu_R(S(t), U(t), S(t+T))), \max_i \{\min\{\mu_{S_i(t)}(S(t)), \mu_{U_i(t)}(U(t)), \mu_{S_i(t+T)}(S(t+T))\}\}\} \quad (3)$$

where D<1 is the forgetting factor causing old rules to decay slowly as new ones are added. The value of D determines the speed of adaptation - the slower the rate the less susceptible the modelling process is to noise. However the forgetting factor mechanism introduces a problem due to the uneven distribution of the input signal space over the relational matrix. If data is heavily biased to a particular region of R, then the forgetting factor will reduce all rules external to this region to zero. To avoid this, the updating procedure in equation (3) only applies to those $\{R_i\}$ for which the confidence in the data set $\{S_i(t), U_i(t)$ and $S_i(t+T)\}$ being relevant to {R} is greater than a prescribed threshold θ, i.e.

$$\min\{\mu_{S_i(t)}(S(t)), \mu_{U_i(t)}(U(t))\} > \theta$$

Having derived a relational matrix R, the system rule base can be used with specific measured values of S(t) and U(t) (say, S(t)' and U(t)') to predict the corresponding output value S(t+T)' through the composition operator -

$$S(t+T)'=(U(t)'xS(t)' \circ R)$$

or as a membership function:

$$\mu_{S(t+T)'}S(t+T) = \max\{\min\{\mu_{S'(t)}(S(t)), \mu_{U'(t)}(U(t)), \mu_R(S(t),U(t),S(t+T))\}\}$$

which must be defuzzified to get the deterministic output S(t+T)'.

5. Tuning a Fuzzy Logic Controller

A number of approaches have been investigated for tuning fuzzy controllers, some involve the use of neural nets[6], others use a cell state space algorithm[7] or use the controller's experience to adjust the membership functions. However a general methodology for tuning controllers to attain optimum values seems unlikely because any optimum values always depend on specific models of the process and the control objectives. Thus tuning controllers is more likely to be done based on the expert's knowledge of the controlled process and not by computation.

When tuning a fuzzy controller parameters should be tuned in order of their significance[8]. Thus any parameters with a global effect should be adjusted first, followed by those with reduced degrees of local effect.

If the scaling factor of a fuzzy variable is changed the definition of each membership function will be changed by the same ratio. This means that changing the scaling factor affects all of the control rules. If the peak value of a membership function is changed the only rules that will be effected are those that use the changed fuzzy label. Changing the width value of a membership function affects the interpolation between the peak value of that function and its adjacent membership function. From this it may be observed that any adjustments to be made towards the tuning of a controller, should follow the sequence below:

1. Scaling factors of variables.
2. Peak values of membership functions.
3. Rules.

Rules of thumb

In the design of control systems, it is usual to take an odd number of sets and to place the centre one in a position where it coincides with the desired value.

An overlap between membership functions of 25% is normally chosen.

Triangular waveforms are used in the case of a set that represents the desired value of a control system in order to obtain a very accurate setting. Examples of how not to subdivide input and output signals are shown in fig. 9. The curves in fig. 9a and fig 9b do not overlap which means that there is no defined μ for a number of values. Values that would fall into these undefined regions cause unpredictable behaviour. In fig. 9c the edges of the various curves spill over into various other sets which leads to instability in the system.

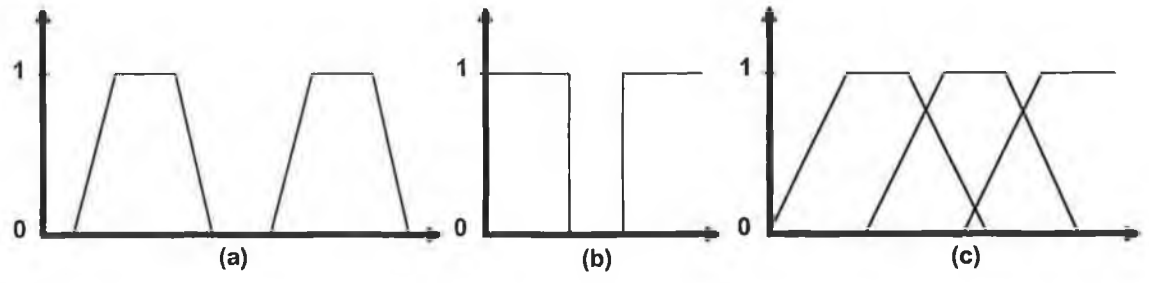


Figure 9.

6. Fuzzy implementations in Mobile Robots.

In [9], a simple sensor based F.L. controller was developed to track a wall. This was simulated on the Vax Station II/GPX. The car was represented as a geometrical point. It was equipped with two sensors placed at right angles to each other. One sensor pointing forward and the other at the side. Their simulated range was two metres.

The fuzzy logic control rules were derived by taking into account the linguistic control policy used by an experienced driver. The walls with which the simulation was tested had convex and concave turns and the results of the simulation were successful. The authors found that the trajectory progressively deteriorates as the sensors' orientation shifted from the (90°,0°) directions. In order to keep track of a corner a new rule was added to those that dealt primarily with the wall following.

In [10], a fuzzy and neural system was constructed and compared in a simulation for backing a truck. The fuzzy system was found to be reliable and showed optimal performance despite rough definition of fuzzy membership functions and a small number of rules. The neural controller when it did work often followed a non-optimal path and was computationally more expensive. Fuzzy Logic control proved to be efficient and robust while the neural system had a high failure rate and required hours of training time on a SUN 3 workstation. Training the neural network with the backpropogation algorithm required thousands of backups and in some cases the training did not converge. This compounded with the fact that insufficient training data may be available for the neural system led to fuzzy logic being the more promising approach for encoding the structured knowledge of a control system.

In [11], a rule based motion controller was developed for an autonomous mobile robot. A set of rules which were experimentally derived from a generic minimum time control rule were used to form the fuzzy logic controller.

In [12], a very reliable control system was developed for an automated guided vehicle using fuzzy inference. The system however, employed a semi-autonomous movement technique in which the vehicle moved partially autonomously but basically used the guide lane. This system is limited owing to the dependence on guide lanes.

7. Conclusions

There have been a number of strategies employed using fuzzy logic for controlling a vehicle along a defined path, avoidance of obstacles, manoeuvring vehicles for parking etc. One of the advantages of the use of fuzzy control resides in the possibility of averaging the conclusions of several rules to obtain an intermediary action. The basic idea behind a fuzzy logic controller is to avoid the design of a strategy based on a detailed dynamic model, by taking the approach of a human operator to an ill-defined system. Fuzzy logic offers a new approach to robot control, circumventing the need to develop accurate dynamic models. It has very good noise rejection capabilities because of the way it averages the output from several decisions. There are also processors designed specifically to carry out fuzzy logic computations that provide a single op code for defuzzification.

Chapter 3.

A Review of Mobile Robot Navigation

1. Introduction

The task of developing a Navigation system where the robot must find its way to the destination through acquiring information from the environment raises a number of issues. The design of any Navigation system for an Autonomous Mobile Robot is largely dependent on the real time demands placed on the robot and the limitations placed on its operation within the environment. Some designs are developed for indoor use where the environment may be altered to facilitate position update or to restrict the mobility of the robot using guide lanes which will simplify the navigation issue.

A great deal of consideration must be given to the planning problem, if the vehicle is to have full autonomous capabilities. The robot has to react swiftly to changes in the environment that affect its operation, which means that information must be processed and interpreted quickly and efficiently. Section 2 discusses the type of control strategies that may be employed. Section 3 discusses the various approaches to path planning and section 4 illustrates different Navigation systems.

2. Control Strategies

A number of strategies exist concerning the ways in which data is exchanged between modules and the ways in which control is applied among the various components of a mobile robot navigation system. The mobile robot system has a number of tasks with which it must deal with such as path planning, position estimation, landmark recognition, obstacle avoidance and possibly world model map building in some systems. The method of interaction between these components in the system gives rise to a number of possible implementations being chosen. There are three basic approaches that can be seen in systems to date:

- *Monolithic*
- *Hierarchical*
- *Distributed*

Each of these is discussed in the following section.

2.1 Monolithic Control Systems

A monolithic control system is simplistic in outline. The planning and navigation is conducted at a single representational level. There is no feedback after the initial path is generated. This means that if a path leads down an alley that is completely blocked (eg. box canyon), an alternative path will not be found. This is because real-time information about the environment is not entered into the map used by the planner and as such, there is no point in calling the planner a second time - it would return the original solution again. A local obstacle avoidance routine is provided to deal with dynamic situations.

This system is not suitable for a commercial robotic system. It is far too simplistic and is not capable of dealing with the demands of a dynamically changing environment because of the lack of feedback to the path generator and the over simplified structure. The main advantage of this control strategy is that it can respond very quickly to changes in the environment that dont necessitate re-planning, such as minor obstacles that the navigator can deal with. However it would only be considered for systems that are constrained by limited computing power and require minimum autonomy.

2.2 Hierarchical Control Systems

Systems that are designed in a hierarchical manner exhibit a clear subdivision of functionality. This functionality is relegated to distinct program modules which communicate with each other in a defined manner. Hierarchical control systems may be very different in structural implementation but the clear segmentation of control is always apparent These systems are easier to develop, debug and implement than distributed systems and can therefore be built more rapidly while still maintaining a high degree of functionality. A typical characteristic of the hierarchical control structure is that for every reduction in the level of the hierarchy, there is a corresponding reduction in the controllers' intelligence and scope, along with an increase in the resolution of the controllers' scope.

2.3 Distributed Control Systems

Distributed control systems operate in an asynchronous manner. The functional modules co-operate with each other via global data structures. These systems lend themselves extremely well to multiprocessing. One major advantage of this type of control is that if its properly designed, adding a new component into the system is possible with the minimum of effort. This leads to greater longevity for properly developed distributed systems. Real-time processing demands are more likely to be satisfied because of the possibility of transporting it to parallel processors. However it is difficult to develop a framework for these systems in which the individual components can execute and communicate effectively. Debugging can also be time consuming when trying to trace the execution of asynchronous processes.

2.4 Assessment of Control Systems

For all but the most basic mobile robot systems, a distributed or hierarchical system should be chosen. The lack of flexibility in monolithic control systems renders them incapable of dealing with some of the major problems that manifest themselves in autonomous systems. The distributed and hierarchical control systems are more suitable for systems that require responses to situations typical of dynamic environments, such as paths being blocked by static or dynamic obstacles that are not known by the global map. Real time responses can be dealt with interactively on a local level while global plans can be dealt with a priori, using a map of the environment which contains information about the environment beyond the range of the sensors.

Whether a system should be hierarchical or distributed is somewhat dependent on the application in question. Hierarchical control systems are easier to develop, debug and implement than distributed systems. A distributed system can be harder to develop and tracing through the instructions of an asynchronous operation can be difficult. In essence a hierarchical system can be built more rapidly while still having a high degree of functionality.

The distributed method if properly designed, is in theory easier to extend because individual components can be integrated into the original system with a minimum of effort and disturbance. Therefore a well designed distributed system will have greater longevity owing to the possibility of extensions being incorporated easily. Another advantage of distributed systems is that they are more easily converted to work on parallel processors which means that real time demands are more likely to be satisfied.

This effectively means that if rapid development of a system and high functional capabilities are required with no extensions or additions being expected, a hierarchical control system is to be preferred. If high functionality with the potential for growth is required, then a distributed system is preferable. Systems may encompass both distributed and hierarchical control as they are not mutually exclusive. The hierarchical structure may be applied to the global planner where it may be broken into the mission planner, navigator and pilot with the lower level (pilot) being distributed into various components all running asynchronously.

2.5 Subsumption Architecture

The traditional approach to robot programming handles data in a manner known as sensor fusion which is very computationally intensive. This method involves decomposing a robot program into an ordered sequence of functional components. World modelling and planning are broken down into their individual parts. Data is collected from the sensors, then noise and conflicts in the data are resolved in such a manner that a consistent model of the world can be constructed. The world model must include geometric details of all objects in the robot's world.

Given a goal which is usually provided by the programmer, the robot uses its model of the world to plan a series of actions that will achieve the goal. This plan is finally formulated and executed by sending appropriate commands to the actuators. This plan may be optimised before any motions are made by the robot. The robot will not go down a dead end and then have to backtrack because it will have the global

information necessary to take the correct path. However the drawback of this method is the large amount of data storage and the intense computation required which leads to problems in dynamic environments owing to the time required for the path planning process.

Professor Rodney Brooks and the Mobile Robot Group at the MIT Artificial Intelligence Laboratory[13] developed a new approach to the architectural structure of robots called subsumption architecture. This method does not resort to sensor fusion but rather utilises the notion of behaviour fusion. It provides a way of combining distributed real-time control with sensor-triggered behaviours. Instead of analysing the validity of the data obtained from a sensor, the sensor initiates a behaviour. Behaviours are layers of control systems that all run in parallel whenever appropriate sensors fire. If there is conflicting data then it is sent to the "conflicting behaviours" problem solver which deals with these type of issues. Fusion is then performed at the output of the behaviours rather than at the output of the sensors.

Behaviours do not call other behaviours as subroutines as they are all running in parallel. However a lower-level behaviour may be suppressed or inhibited by a higher-level behaviour. When the higher-level behaviour is not being triggered by a sensor condition they will stop suppressing the lower-level behaviour and the lower-level behaviour resumes control. Thus sensors cause behaviours to interject themselves at all levels and there is as such no global world mode. A behaviour is run as a process which may be thought of as a piece of code that runs simultaneously with other processes or programs.

Processes are run in parallel, but on a processor that is inherently a sequential machine they must be multitasked. This means that each process gets the processor for a small amount of time before the next process is then given control. Brooks used a round-robin schedular to switch between processes. G. Stoney et al[14], designed an Autonomous Mobile Robot control system using this architecture. The robot was able to wander around a room and avoid obstacles and slowly moving humans.

3. Path Planning

The task of automatic path planning introduces problems that are central to mobile robot applications. In this context, given the mobile robot's initial and final goal locations and an obstacle map, the problem is to find the appropriate path from the initial position to the goal position such that the mobile robot can smoothly travel through the area without colliding with the obstacles.

Unlike manipulators that generally work in a fixed environment performing a set number of movements, when planning for mobile robots, it is more important to develop a negotiable path quickly than to develop an "optimal" path, which is usually a costly operation. A mobile robot may be following some previously computed path when it finds it must modify the path to bypass some obstacle. Path planning for a robot is therefore a continuous on-line process rather than a single off-line process.

Often attempts in planning algorithms are made to optimise the path in terms of Euclidean distance. The time saved in travelling a shortest distance path may not be justified by the long period of time required in planning the path and no efficient algorithms currently exist for finding optimal paths among three-dimensional obstacles. A compromise is to find a path that is not optimal in absolute Euclidean distance but is optimal (shortest) using the primitive path segments. Therefore the resultant path will not deviate significantly from the optimal path, but the time saved justifies the simplification. The hierarchical search at different levels allows a search at the level sufficient to find a solution path and therefore avoids excess details at the lower levels.

A number of representations are commonly used to represent the environment for the purpose of path planning. These are:

- *Pure free-space methods*
- *Configuration Space (Vertex graphs)*

- *Hybrid free space vertex graphs*
- *Potential fields*
- *Regular grid*
- *Quadtree*

3.1 Pure free space methods

Using this strategy the space between the obstacles is represented as opposed to the obstacles themselves. There are two main methods that fall under this category: These are Voronoi diagrams and generalised cylinders.

3.1.1 Voronoi Diagrams

A voronoi diagram is produced by generating a set of polygons, each representing an enclosed region in which all contained points are closer to one particular point in a given point set than to any other point in the set. The plane is partitioned into a net of non-overlapping convex polygons. The free space representation resulting from this diagram represents space as a series of connected straight line segments that typically split the distance between the closest pair of line segments of surrounding obstacles.

This strategy is unsuitable when attempting to use it for a dynamic environment as moving obstacles cannot be dealt with in an efficient manner. All information about the obstacles is discarded i.e. descriptions of the obstacles are not represented by the graph and therefore this information is not available to high level processes.

3.1.2 Generalised Cones

This approach was developed by Brooks[15] where free space is subdivided into generalised cones or freeways. Figure 1 shows an environment broken down into generalised cones. The axes of these generalised cones determine the natural "freeways" for moving an object through the space. The path planning problem is solved by searching through the connectivity graph where each node corresponds to the intersection of two generalised cone axes, and each link corresponds to the cone axis. The generalised cones are constructed such that the object translation and rotation are permitted along the cone axes. The paths found tend to generously avoid the obstacles rather than barely avoid them.

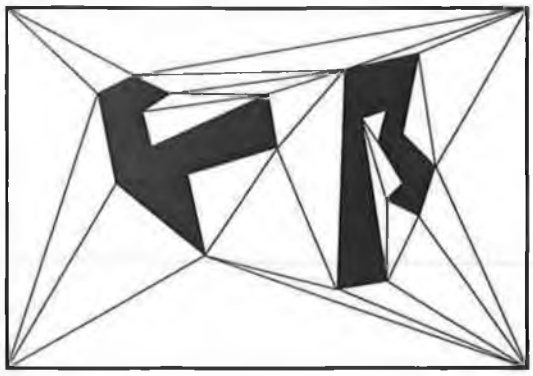


Figure 1: Generalised Cones

Chatila[16] uses convex polygons to represent the free space. This leads to a connectivity graph where each node is a convex polygon and each link corresponds to the common edge segments shared by adjacent polygons. Within each polygon, any two points can be linked with a straight line segment without intersecting the polygon boundary. An example is shown in figure 2.

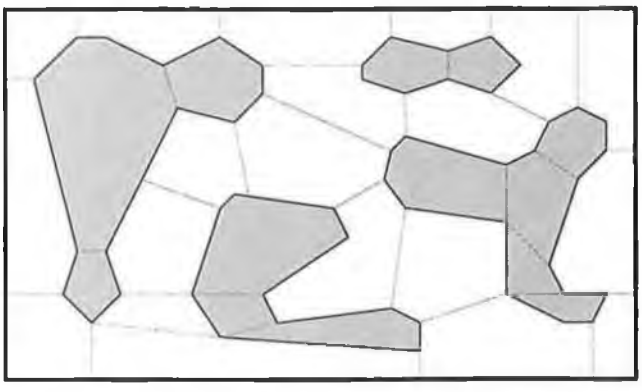


Figure 2: Convex polygon representation of free space.

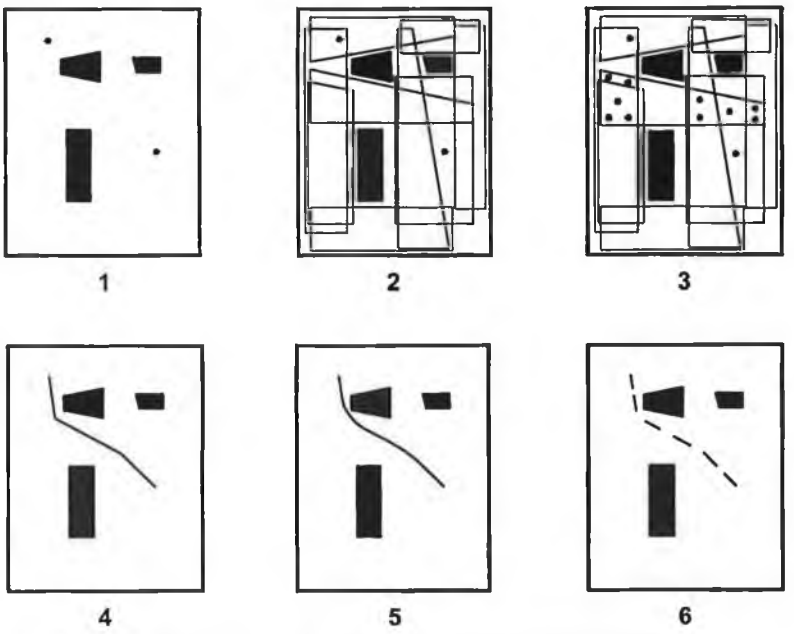
Hubert A. Vasseur et al [17] used a similar method for the Navigation of a mobile robot within a mapped environment. The environment was decomposed into convex polygonal cells. The paths through these cells consisted of tangents and arcs only.

In [18], the authors discuss the importance of planning a smooth path (a path that has no curvature discontinuity between one tangent and the next) enabling robots to have smooth flowing movement rather than the stop-turn and move strategy. The problem with generalised cones is that movement is supposed to occur along the spines which allows the path optimisation algorithm very little room to make smooth turns. They

propose *primary convex regions* (pcr) which are unobstructed convex regions where each boundary covers some portion of an obstacle wall.

This method has its advantages over the others when path smoothing. Free space is decomposed into overlapping pcrs. The overlapped regions are also convex which is a desirable property when planning the curves that traverse them. When the pcrs are found, a connectivity graph is built. Special points within the overlapped regions must be chosen as candidate turning corners for use with the cubic spirals. The nodes of the connectivity graph connect the line segments within the pcrs. An end point of such a line segment is either a candidate corner inside an overlap with another pcr or the initial or goal position of the robot. Then a search algorithm e.g. A* can be used to find the path. The process is illustrated in Fig 3.

This type of representation however, suffers from the same problems as the Voronoi diagram.



Steps involved in path planning. (1) Initial and goal position is given. (2) Identify PCRs. (3) Identify candidate turning points in overlap regions (only those on the solution path are shown). (4) Find least cost path in connectivity graph, consistent with maximum curvature constraint. (5) Create smooth path by inserting cubic spirals. (6) Identify subgoals as start/end points of turns of the path.

Figure 3

3.1.3 Mixed representation

Brooks[19] developed an extension to the generalised cones in an attempt to counter some of the problems that occur in the other method. A mixed representation is used, where free space is decomposed into non-overlapping geometric-shaped primitives that consist of two types convex polygons and generalised cones. The algorithm attempts to use the best of both representations as generalised cones provide a good means for representing the narrow free space between two neighbouring obstacles. However, they are awkward for representing a large open space because the cone axis artificially constrains the trajectory of the path inside the cone. On the other hand a convex polygon is a good choice for representing large open spaces, but not as a compact representation for narrow regions.

By providing the set of polygonal obstacles in space, the algorithm first decomposes concave obstacles into connecting convex obstacles in order to have a uniform obstacle representation, which facilitates later processing. The neighbourhood relations among these convex obstacles are identified and then used to locate critical "channels" and "passage regions" in the free space, and to localise the influence of obstacles on free space description. The free space is then decomposed into non-overlapping geometric shaped primitives where channels connect passage regions. The channels are similar to the generalised cones presented in Brooks[15]. The passage regions are represented as convex polygons. Based on this mixed representation a path planning algorithm can plan trajectories inside the channels and passage regions. The perimeter of the channels and passage regions associated with the planned path provides the boundaries of the path that the mobile robot must stay within during path execution.

The idea behind constructing the connectivity graph is to have an abstract graph that facilitates the construction of channels and passage regions. Essentially the connectivity graph identifies areas to construct channels and passage regions and it provides a high level connectivity representation of the free space. It is a compact representation and reduces the computation of free space representation.

At each link of the connectivity graph a generalised cone is constructed to represent a channel, and at each node a convex polygon is constructed to describe the passage region.

The A* search algorithm was used to find the minimum cost path. The cost function used is the distance between two nodes.

3.2 Configuration Space (Vertex Graphs)

A feature of vertex graphs is that the space between the obstacles is not explicitly represented. Lozano-Perez's configuration space approach[20] is an example that illustrates this category of representation. The vertices of an obstacle are first modelled by a polygon. They are then grown by a distance equal to the radius of a circle enclosing the robot plus a small safety margin. The idea is to shrink the robot into a single reference point, while at the same time expanding the obstacle regions according to the object shape. The minimum distance path is found by searching the visibility graph, which indicates all collision-free, straight-line paths among the expanded polygonal obstacle vertices.

Moravec[21] used a representation similar to this. Obstacles were modelled as circles rather than polygons and paths were constructed as a series of tangents to the circular obstacles. With this representation, planning for the optimal path is more computationally intensive than the previous strategy. Therefore he developed a sub-optimal algorithm that approximated the shortest path and produced the optimal solution most of the time.

Wang[68] considered the planning problem in configuration space and assumed the robot was a point that moved in a two dimensional environment where the obstacles were enlarged by half the maximum size of the robot. The environment consisted only of convex polygons represented by line segments. Moving obstacles were dealt with but were assumed to move in a fixed direction at a constant speed which was known a priori to the robot. An accessibility graph which is a generalisation of the

visibility graph was used to plan the robots path. The accessibility graph becomes a visibility graph if all moving obstacles in the environment move with zero speed i.e. when all obstacles are static.

The robot need not be circular and it is not required that it must be modelled by a circle, however most systems do this in order to simplify the computations.

This method is suitable for high level planning because the robot can be treated as a point and therefore consideration to the space occupied by the robot is not necessary. The major disadvantage of the configuration space approach however, is that it is difficult to deal with object rotation, and the minimum distance path is very close to the actual obstacles, which means that the robot is frequently in close proximity with obstacles and constantly changing direction in tandem with the obstacle boundaries leading to slower operation in the environment.

3.3 Hybrid free space and vertex graphs

Giralt[22] and Crowley[23] represented both free space and the obstacles by vertex graphs thus combining aspects from free space and vertex graphs. Free space is divided into convex regions referred to as meadows. A characteristic of a convex region is that any point can be reached from another point within the same convex region without traversing a border i.e. colliding with an object. Therefore path planning is reduced to finding a sequence of line segments traversing the meadows. This is easily found from a connectivity graph.

3.4 Potential Fields

This representation consists of a map where obstacles are converted into peaks and clear paths into valleys. The robot moves in a field of forces. The destination is an attractive pole for the end-effector, and obstacles are repulsive surfaces for the robot where the robots' initial position is placed at a higher elevation than its destination so that it is attracted to the destination and repulsed from any obstacles. Therefore the algorithm views the robot as a ball rolling towards a hole. A collision-free path, if attainable, is found by linking the absolute minima of the potential.

This technique is useful for representing uncertainty because uncertainty may be represented by changing the slope of the obstacle peaks. For example, high confidence in an obstacles whereabouts will result in very steep cliffs, while uncertain obstacles will result in slowly rising slopes. The algorithm does not attempt to find an optimal solution and is subject to problems of local potential energy minima which can occur resulting in the robot becoming stable before it ever reaches its goal. This results in extensions being added to ensure the robot does not get trapped in box canyons.

In [24], the Virtual Force Field (VFF) method was developed and implemented on a mobile robot (CARMEL). This method uses a 2-D Cartesian grid for obstacle representation. Every cell (i,j) in the grid holds a certainty value c_{ij} that represents the confidence of the algorithm in the existence of an obstacle at that location. The c_{ij} values are incremented when an ultrasonic sensor indicates the presence of an obstacle at that location (cell). As the vehicle moves, a window of 33x33 cells is taken momentarily to belong to the active region with the robot in the centre. Each cell (10cm x 10cm) in this region is an active cell. The active cell exerts a virtual repulsive force F_{ij} toward the robot. The magnitude of this force is given by:

$$F_{ij} = \frac{F_{cr} W^n C_{ij}}{d^n(i,j)} \left[\frac{x_i - x_0}{d(i,j)} \hat{x} + \frac{y_i - y_0}{d(i,j)} \hat{y} \right]$$ where:

F_{cr}	=	Repulsive force constant.
d(i,j)	=	Distance between active cell (i,j) and the robot.
C_{ij}	=	Certainty value of active cell (i,j).
W	=	Width of mobile the robot.
x_0, y_0	=	Robots present co-ordinates.
x_i, y_i	=	Co-ordinates of active cell (i,j).
n	=	2

All virtual repulsive forces add up to yield the resultant repulsive force F_r, i.e.

$$F_r = \sum_{ij} F_{ij}$$

Simultaneously, a virtual attractive force F_t is applied to the vehicle, pulling it toward the target.

$$F_t = F_{ct}\left[\frac{x_t - x_0}{d_t}\hat{x} + \frac{y_t - y_0}{d_t}\hat{y}\right] \text{ where:}$$

F_{ct} is the target (attraction) force constant; d_t is the distance between the target and the robot; and x_i, y_i are the target co-ordinates. The summation of F_r and F_t yields the resultant force vector R, where $R = F_r + F_t$. An example is illustrated in Figure 4, where the robot is in the vicinity of two obstacles. The certainty values in the cells all contribute to the total repulsive force which causes the robot to be pushed away from the obstacles but the attraction force to the target ensures it continues towards the target.

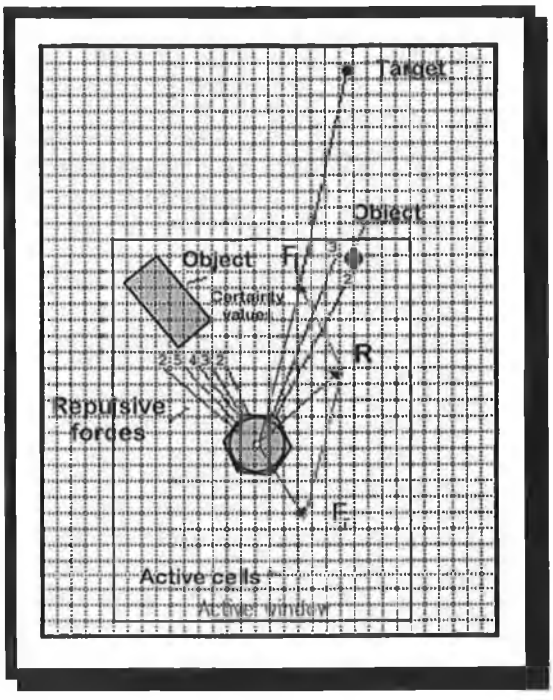


Figure 4: The Virtual Force Field concept.

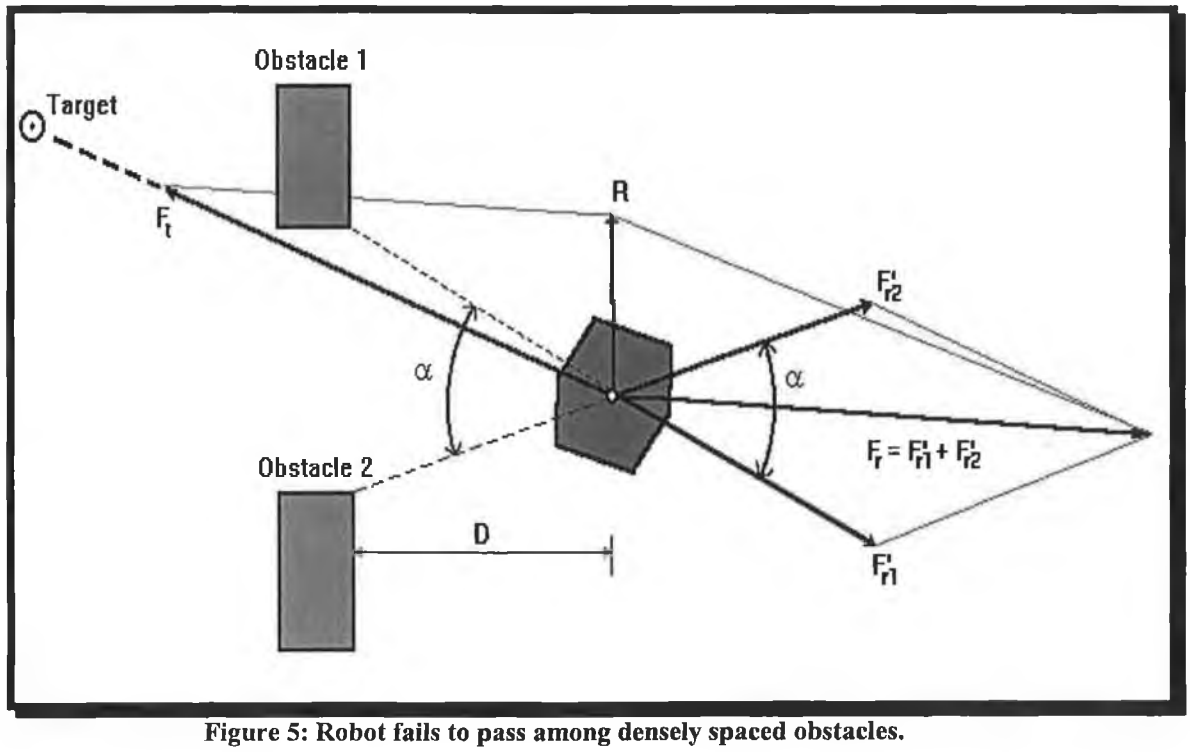


Figure 5: Robot fails to pass among densely spaced obstacles.

There were four problems inherent in the use of this method when tested on their robot:

1. Trap situations due to local minima. This occurs when the robot runs into a dead end, for example, a box canyon.
2. No passage between closely spaced obstacles. This is illustrated in Figure 5 where the repulsive forces from both obstacles combine to force the robot away from the gap between the obstacles.
3. Oscillations in narrow passages. This occurs when both walls are trying to push the robot away from them.
4. In the presence of obstacles the robot's motion is oscillatory.

As such, these authors recommended that alternative measures be taken for navigation that did not exhibit these pitfalls.

Work on high level collision-free path planning based on the potential field concept has been investigated by C. Buckley[25]. Such strategies taken for robot collision avoidance attempt to use it as a component in the higher levels of control of the hierarchical robot control systems where it is treated as a planning problem with research in this area being focused on the development of collision free path planning

algorithms. These algorithms aim at providing the low level control with a path that will enable the robot to accomplish its assigned task free of any risk of collision. Krogh and Thorpe[26] suggest a *generalised potential field* method that combines global and local planning.

However, the potential fields are more suitable for use in the lower levels of a multi level representation where they can prove useful in providing the robot with a set of capabilities needed for local or short range navigation such as goal seeking, obstacle avoidance and following paths. It is limited in use for complex situations due to the occurrence of local minima and the high computational cost that results in detecting and avoiding such pitfalls.

3.5 Regular Grids

This representation consists of a two dimensional cartesian grid that represents the environment. Mitchell et al[27] developed a system based on this approach where each pixel represented a certain area and connectivity was maintained through eight arcs to each of the pixels nearest neighbours. A path from source to destination then consisted of a series of nodes connected by arcs. The problem with this representation is that the arcs restrict movement to one of eight directions when traversing from one node to another. This is known as digitisation bias and would result in paths that were excessively long and thus non optimal.

Both four and eight neighbour connectivity has been used by Thorpe[28] and a path relaxation algorithm has been developed for overcoming the digitisation bias discussed above.

Parodi [29] uses a similar representation to Mitchell but the global planner accepts a rather large number of cost criteria such as energy consumption, travel time, potential hazard.

This method has the disadvantage of being computationally intensive and would require very powerful processors for practical applications.

3.6 Quadtree

Research at the University of Maryland[30] resulted in a representation suitable for planning purposes. Space is recursively decomposed into 2^i by 2^i areas. A binary value of one in a block represents the presence of an obstacle in that block and 0 represents free space (see example Fig 6). Decomposition stops before the lowest level of resolution is encountered. A path generated from a quadtree is a sequence of blocks through which it is possible for the robot to move. The detailed motion within any single block is not determined at this level; a default assumption of straight line motion through the block is assumed. Although this will not ordinarily be an optimal path, it will be negotiable. In developing the search only the horizontal and vertical neighbours of any block are considered for building extensions of paths as the use of diagonal neighbours which share only single points with the current node could cause the clipping of obstacles during path traversal. This method shares the problem of digitisation bias with regular grids.

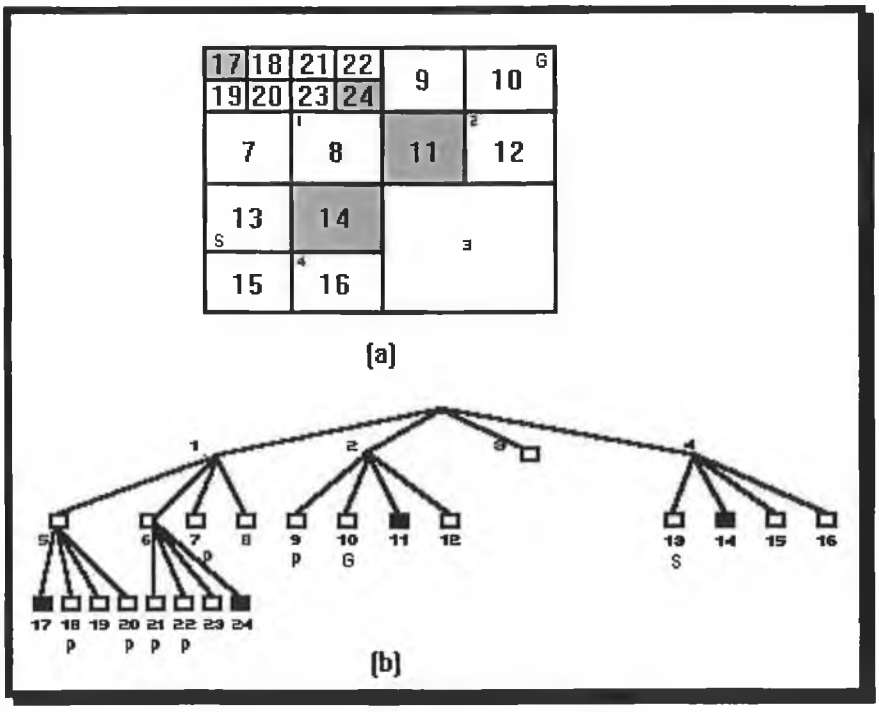


Figure 6: Quadtree representation of an Environment

The quadtree representation offers a compromise between the regular grid representation (which is straightforward to construct but then computationally costly to analyse) and a free space region representation (e.g. Brooks[15]) which is more costly to construct, but on the other hand more efficient to analyse. The main

disadvantages of this strategy is the loss of uncertainty information due to a binary encoding of occupancy and the varying resolution for path construction. No information on the nature of the obstacles is contained in this representation.

This multi-resolution representation of the robots immediate environment was used in [30] where the Hough transform [31] located landmarks of known image orientation and scale for position estimation. The A* search algorithm was implemented.

3.7 Path planning using resistive grids (Hardware approach)

An interesting alternative was presented by Tarassenko et al[32], where the robot was assumed to operate in a structured indoor environment. The robot's working environment could change rapidly as people moved around in the workspace so a neural network was used to detect the obstacles and they were mapped into a resistive grid of rectangular or hexagonal cells. The grid is composed of MOS transistors and implemented in VLSI (Very Large Scale Integration) technology. The robot's working environment is also mapped into the grid and planning is accomplished by connecting the start position to the power rails and grounding the goal position. Each transistor can be programmed to be open (R_∞) or closed (R_0) by a ram cell connected to its gate.

As any obstacle is represented by open transistors, most of the current will follow the lowest resistance path to ground avoiding the obstacles. The path from start to goal can be found from local voltage measurements. For each node, the next node is found by measuring the voltage drop ΔV between that node and each of its nearest neighbours (six measurements for a hexagonal grid) and taking the node with the maximum voltage drop. Any point on the grid may represent the robots current position or the goal, so it must be possible to connect every node in the grid to the power rails or to ground. This approach does not fall prey to the problem of local minima and can be very fast. It can alternatively be implemented in α-Si technology.

Another implementation is described by Alexandre M. Parodi[29]. The planner is designed for an autonomous land vehicle and is capable of dealing with complex problems where a specialised processor is designed to deal with the graph search.

3.8 3-D Path Planning

In [33], a hierarchical-orthogonal space approach was proposed for 3-D collision free path planning. Using this method, the path is obtained by planning in the 3-orthogonal 2-D projections of a 3-D environment. Each of the three orthogonal 2-D subspaces are searched for sub-optimal paths consisting of primitive path segments. Two dimensional path searching is easy using the quad-tree representation. Thus, the idea was extended for use in the 3-D path planner where a modified quadtree representation was built for each one of the orthogonal subspaces.

Octrees

Octrees are used in many 3-D representation problems because they provide a compact data structure, allowing rapid access to information, and implement efficient data manipulation algorithms. In [34], an algorithm was presented to construct the octree representation of a 3-D object from silhouette images of the object. The execution time was found to be linear in the number of nodes in the octree.

4. Navigation

The purpose of navigation is to plan a path to a specified destination and then to execute this plan, modifying it as necessary to avoid unexpected obstacles. Path planning may be broken down into a global planning problem and a local planning problem. Global path planning requires a pre-learned model of the environment which may be a simplified description of the real world. Local path planning executes the steps in the global plan, correcting for changes in the world that are encountered due to static or dynamic obstacles, during the plan execution. While global navigation operates on a pre-stored model, local navigation requires a model that reflects the current state of the environment as the plan is being executed.

Many early works have concerned themselves with locating paths in known environments or among unknown static obstacles. Obstacles are however, not always static and while some methods deal with path planning among obstacles that move along known trajectories[68], such methods may be unsuitable in real situations where an ALV (Autonomous Land Vechicle) may face obstacles such as human beings. If a flexible navigation system is to be developed such problems must be considered.

A navigation system must be dynamic because each action the vehicle performs in a real time dynamic environment will provide new information which may in turn require modifications to its actions. Collision-free path planning is one of the basic requirements in navigation. For example, if a path has been planned at the global level, local modifications to avoid obstacles may be processed at the local path planning level by the pilot. However, if the path is completely blocked, the global path may have to be replanned. In this case, the blockade has to be added to the map, and the action planner may request a new global plan[29]. The replanning process must be fast enough to eliminate long idle periods for the vehicle. The complete autonomous vehicle capabilities must also include situation assessment, perception, map making, communication, etc.

Primitive systems placed huge demands on the higher levels of planning where the function of low level control was reduced simply to the execution of a limited set of operations that served to maintain the robot along a specified path. The result was that the robot's interaction with its environment became dependent on the time cycle of high level control, which was generally too slow to meet the real time demands of a typical robot. This of course limited the capabilities of the robot when fast, precise and highly interactive operations were needed when used in a cluttered and evolving environment.

As such, later proposals suggested that the capability of low level control should be increased to cater for real-time obstacle avoidance. Collision avoidance at the local level does not attempt to replace high level planning problems. The purpose is to

utilise low level control capabilities in performing real-time operations. However the level of intelligence still remains much less than that of high level control.

Computationally efficient methods for real time obstacle avoidance are desirable and serve to provide the lower level of control with the necessary real time capabilities. The high level planning problem is distributed between different levels of control, allowing efficient real-time robot operations in a complex environment. In [35], such an attempt is made to try to extend the function of low level control and to carry out more complex operations by coupling environment sensing feedback with the lowest level of control.

The global path planning is based on a pre-learned "network of places" (see Fig 7). The network of places is learned in a special "active learning mode" in which the robot explores its environment. Each place in the network is connected to a set of adjacent places by "legal highways".

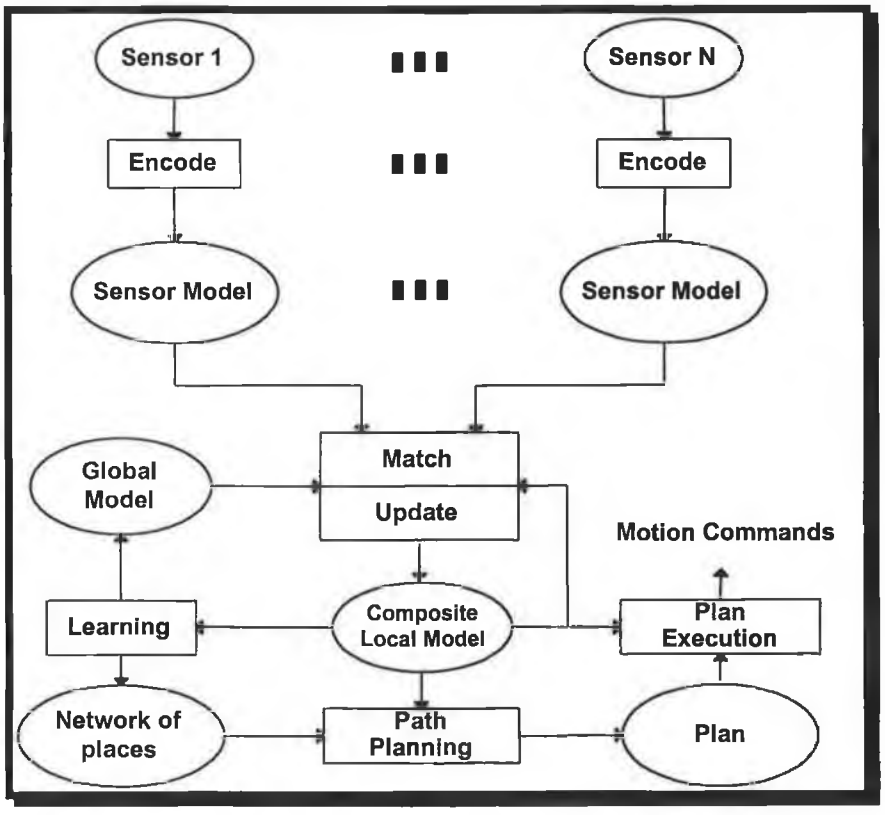


Figure 7: Framework for an Intelligent Navigation System

Global navigation is viewed as the process of choosing a set of "legal highways" which will carry the robot from its current location to the specified destination. A legal highway consists of a straight line path and local navigation is accomplished by means of a finite state process. Traversing each legal highway is the job of the navigator. Each path is tested for blocking obstacles using the raw sensor data, the Sensor Model and the Composite Local Model. If an obstacle is detected, a recursive obstacle avoidance procedure plans a new sequence of straight line paths to the next local goal. The recursive obstacle avoidance procedure is based on the current contents of the Composite Local Model.

An example of an autonomous mobile robot that is designed for use in structured or office environments as opposed to unstructured natural environments is discussed in [36]. It uses two sensors, an optical rangefinder and odometry. It is assumed that an off-line path planner provides the vehicle with a series of collision-free manoeuvres, consisting of line and arc segments, to move the vehicle from a current to a desired position.

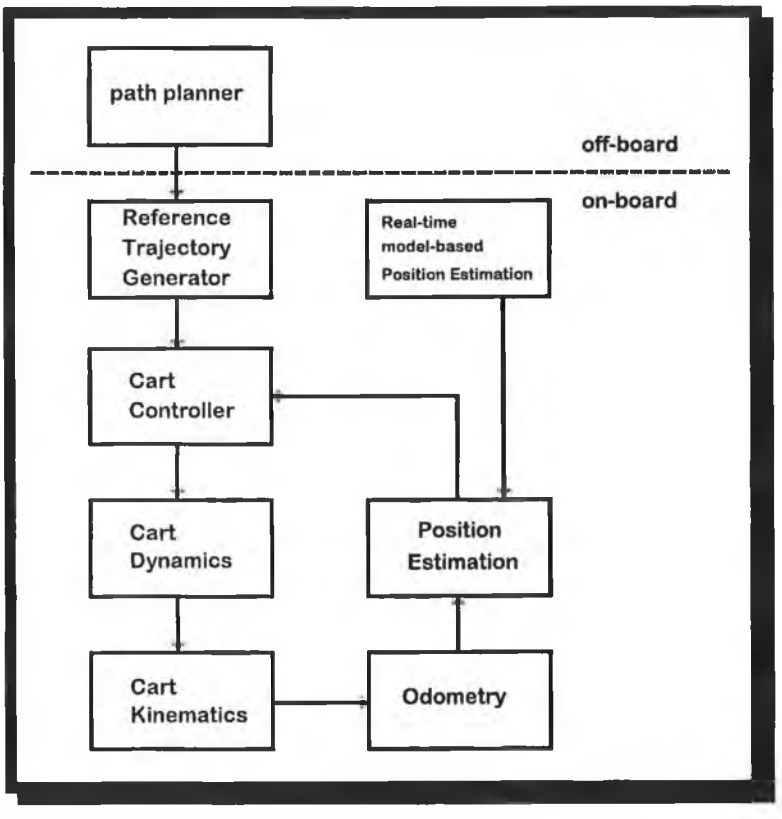


Figure 8: Block diagram of the low-level control system

On board the vehicle (see Fig 8), the line and arc segment specifications are sent to control software consisting of low-level trajectory generation and closed loop motion control. The trajectory generator takes each segment specification and generates a reference vector at each control update cycle. The cart controller controls the front steering angle and drive speed using conventional feedback compensation to minimise the errors between the reference and measured states.

Connell [37] proposed a scheme for navigation that did not rely on a detailed world map or a precise inertial guidance system. The approach is based on a multiple-agent control system in which there are a variety of local navigation behaviours that run in parallel. These behaviours arbitrate among themselves to select the most applicable action in any given situation.

The control system used was based on the Subsumption Architecture [13]. The purpose of the robot was to wander around the laboratory and collect soda cans. The visual localisation was achieved using a laser light-striper to find the cans. When the robot travelled along a path it recorded the distance and the turn so that it could retrace the same path on its return journey. This implementation was rather limited for practical applications.

Another method of navigation was presented in [38], of an Intelligent Mobile Autonomous System. A hierarchical control structure is used in which all motion planning is distributed between three levels of decision making as shown in Fig 9. World representation is in 2-D with obstacles represented as polygons. A subset of the Cartographer performs updating of the Navigator's 2-D map based on range data received from the sensor. The A* search algorithm was used.

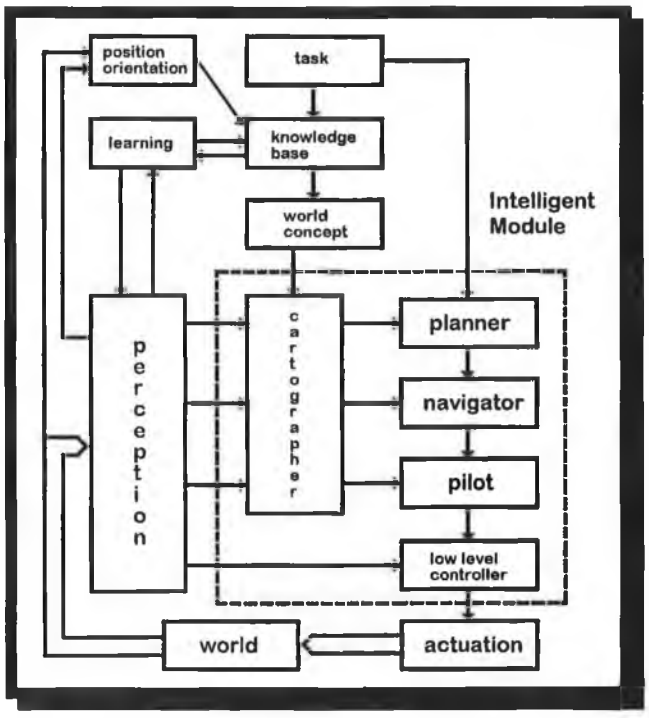


Figure 9: Conceptual structure of Intelligent Mobile Autonomous System

The Cartographer is the central data-base and manages all passing and updating of information to each subsystem. The planner is the highest level in the control hierarchy which performs path planning on a global scale. The planner produces a list of subgoals which become goals for the Navigator. The Navigator receives these subgoals and must find the optimum path in an obstacle strewn environment in which the "map" of the world may be completely known, partially known, or completely unknown. The output of the Navigator is a list of nodes between the milestones which becomes the input of the pilot. The pilot is responsible for calculating real motion trajectories between the path nodes generated by the Navigator while performing low level collision and obstacle avoidance. Their approach utilised a knowledge-based controller with constant feedback from a camera system.

A vision system for autonomous ground navigation was presented by the Computer Vision Laboratory at the University of Maryland[30]. They employ a hierarchical control system where they segment the visual navigation task into three levels, called long range, intermediate range and short range navigation. Decision making capabilities among levels generally flows from levels of greater abstraction to levels

of lesser abstraction while status information concerning the achievement of these goals flows in the opposite direction.

At the long range a plan is generated, identifying the start location, the goal, and a low resolution path for moving from the start to the goal. This path is chosen based on basic considerations of the terrain to be crossed and the capabilities of the robot. In order to maintain the position, landmarks are identified and by triangulation the current position is updated. At the intermediate range, generally safe directions of travel are examined. This involves assessing the nature of the terrain in the robots immediate visual environment and identifying those portions of that environment through which it is feasible to move. These navigable portions of the environment which correspond to segments of road in the vehicle's field of view are referred to as corridors of free space. With short range navigation any decisions made are based on a detailed topographic analysis of the immediate environment, and enables the robot to determine safe positions for travelling, and to navigate around obstacles in the immediate environment.

In [39] a new collision avoidance scheme was proposed for autonomous land vehicle navigation in indoor corridors. The global path for normal ALV navigation in a corridor was assumed to be known in advance. As the corridor environment could change due to moving obstacles a local navigation path was recalculated in every navigation cycle in order to conduct indoor collision free navigation. A three-wheeled vehicle was used and the scheme dealt with static obstacles with no a priori position information as well as moving obstacles with unknown trajectories.

Based on the predicted positions of obstacles, a local collision-free path was computed by the use of a modified version of the least-mean-square-error (LMSE) classifier used in pattern recognition. Wall and obstacle boundaries were sampled as a set of 2D co-ordinates, which were then viewed as feature points to the ALV location to reflect the locality of path planning. The trajectory of each obstacle was predicted by a real-time LMSE estimation method. The manoeuvring board technique used for nautical navigation was employed to determine the speed of the

ALV for each navigation cycle. Smooth collision-free paths were found in the simulation results.

Steele and Starr [40] decomposed the navigation process of a robot into two supporting processes: (1) using a graph search method to plan a path for avoiding all known static obstacles, and (2) using a field potential method to control the motion of the robot when unexpected obstacles were encountered.

5. Conclusions

Generally for the design of an Intelligent Autonomous System, a global planner will have a map available to it for the purpose of generating a global plan. The manner in which the data is made available to the path planner will depend on a number of considerations. These will take into account the pros and cons specific to the representation (e.g. memory requirements, time constraints, extendibility to moving obstacles). Control will be distributed using the hierarchical (more common) or the distributed approach. The only commonality between the hierarchical representations discussed is the clear flow of control that may be observed as one descends the hierarchy. At the top level, high intelligence and low resolution of the environment gives way to low intelligence and high resolution at the bottom level of the hierarchy.

Chapter 4

Position Estimation and Data acquisition

1. Introduction

This chapter first discusses the mechanical configurations that are available in robotic vehicles (section 2). This is an important issue because some configurations will impose limits on the robot's capabilities that others will not. After considering how the vehicle will move it seems logical to discuss how it can perceive the environment through the use of one or more different types of sensors. Hence section 3 discusses various sensor types including their drawbacks. If a robot is to be of any use it must be able to maintain an accurate estimate of its current position within the environment. Position estimation is discussed in section 4 and following that, section 5 provides an insight into the various techniques used in map building. Map building requires the availability of sensor data and is used in autonomous vehicles to provide information about the current environment to facilitate position estimation and obstacle avoidance.

2. Mechanical Configurations

The mechanical configuration of the most common wheeled vehicles falls into one of two categories, steered-wheeled vehicles or differential-drive vehicles. The following sections discuss each of these along with a novel development using the Ilonator wheel.

2.1 Steered Wheeled Configuration

The configuration for the steered wheel type typically consists of a single-steered front wheel with a fixed rear axle. The most common example is the tricycle configuration that's used in many robots. However, four wheeled vehicles can also be considered to be within this category if the two steered wheels are approximated by a single wheel at the centre. The steered wheel vehicles are incapable of turning on a point due to the physical limits associated with the maximum steering angle.

This may cause the vehicle to have a large turning radius and thus make the task of manoeuvring more difficult in a cluttered environment.

2.2 Differential Drive Configuration

The most common example of a differentially driven vehicle is the military tank, in which differences between the speed of the two tracked wheels determines or controls the steering. Differential drive vehicles have the advantage over the steered wheeled configurations of possessing a zero turning radius.

2.3 Omnidirectional vehicles

A disadvantage common to both of these vehicle configurations is that they do not permit arbitrary rigid body trajectories. For example, neither vehicle can move parallel to its fixed axle. This is not true of other mechanical configurations. The development of the Ilonator wheel has allowed the construction of a further category of vehicle. The wheel itself consists of a hub about which are mounted twelve rollers, each making a 45° angle to the wheel orientation. This development provides the wheel with the capabilities to move in any direction and not just tangential to the wheel direction, without slipping (see fig 1). A very detailed discussion of the kinematics of steered wheeled, differential drive, omnidirectional, and balled wheeled vehicles can be found in a paper by Muir and Neuman[41].

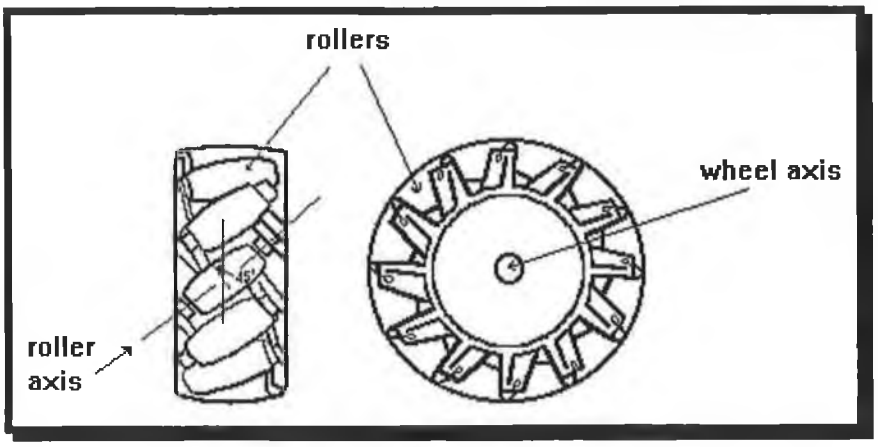


Figure 1: Omnidirectional Wheel

3. Sensors for Perception of the Environment

The range of applications for robotic devices both in industrial and research applications is very extensive. Robotic systems with high levels of autonomy that are capable of working in unstructured environments with little a priori information can be applied to many areas of industry. If a robot is to manoeuvre through its environment and execute any sort of reasonably intelligent task, it should be able to navigate through its world based on sensory information. In contrast with stationary arm robots that are fixed to a global co-ordinate frame, a mobile robot may have little or no knowledge about its environment and due to errors that accumulate when using odometry for position estimation (shaft encoders are subject to integral error owing to wheel slippage and the finite resolution inherent in its use), sensing the environment is necessary.

To achieve this degree of independence, the robot system must acquire an understanding of its surroundings through manipulating a rich model of its environment. A variety of sensors are available that allow robots to interact with the real world as they provide mechanisms for which meaningful information can be extracted from the data provided. Systems with little or no sensing capability are usually limited to fixed operations in highly structured working areas and cannot provide any substantial degree of autonomy or adaptability.

There are many methods available for obtaining range data. Jarvis [42] discusses among others, contrived lighting techniques (including striped lighting and grid coding), depth from occlusion, texture gradient and focusing, some methods range from stereo or motion, and triangulation-based and time-of-flight rangefinders.

Of these, contrived lighting techniques are not generally useful to systems operating in unstructured environments, while stereo vision systems and active rangefinding devices prove useful within unstructured environments. Cameras however, require computationally intensive processing which can prohibit their use. Triangulation-based rangefinders suffer from "gaps" in the data due to obstructions. In the time-of-

flight category, the main two representatives are ultrasonic and laser rangefinders. Sonar systems have lower resolution but are much less expensive than laser-based sensors. Phase-shift-based laser rangefinders are subject to a 2π uncertainty. Both sensors suffer from absorption and specular reflection problems, but measurement precision is much higher with laser rangefinders.

Sensors most commonly used for perception of the environment are bumper switches, shaft encoders, sonar transducers, photocells and infrared proximity sensors, cameras, infrared beacons and laser range finders.

3.1 Sonar Sensors

Ultrasonic range sensing has attracted attention in the robotics community because of its simplicity in construction and low cost. They also provide an inexpensive means of obtaining three dimensional information about the surrounding environment. While they are cheap, flexible and robust, they do however have their drawbacks as is discussed in detail within this section.

3.1.1 Construction

An ultrasonic range sensor is composed of a capacative transducer consisting of a very thin metalized diaphragm supported over a specially machined backplate. The simplest arrangement causes a short burst of constant-frequency signal to be emitted from the transducer. The transducer then switches to receive mode and waits for the return echo. The distance to an object is then found by measuring the time of flight between a transmitted pulse and a returned echo and multiplying by the speed of sound. The distance measure, however, is not necessarily the distance in the direction the sensor is pointing, since the width of the beam may cause an echo from one edge to be returned before the echo from the centreline.

Other more sophisticated methods can yield equivalent narrow pulse effects by using wide pulse and wide bandwidth techniques which have the advantage of additional signal energy. Ultrasonic energy, particularly at very high frequencies is quickly

attenuated in air. Consequently many practical macroscopic ranging systems operate in the frequency range below about 250 kHz.

3.1.2 Characteristic Problems with Sonar.

Ultrasonic sensors have their limitations. Due to the expanding signal beam of the sensor, determining the direction of the detected object poses problems because the acoustic beam expansion of the transmitted signal allows return echoes to be obtained from any angle inside the cone of propagation. The direction ambiguity can be reduced to some extent by increasing the operating frequency or diameter of the sensor, but some ambiguity will still remain. Related to this problem is the inability of an ultrasonic sensor to see an opening in front of it if the gap is less than the beamwidth. In fig. 2 an object is assumed to be directly in the path of the sensor, because the edges are smeared by the ultrasonic sensor and an echo is returned.

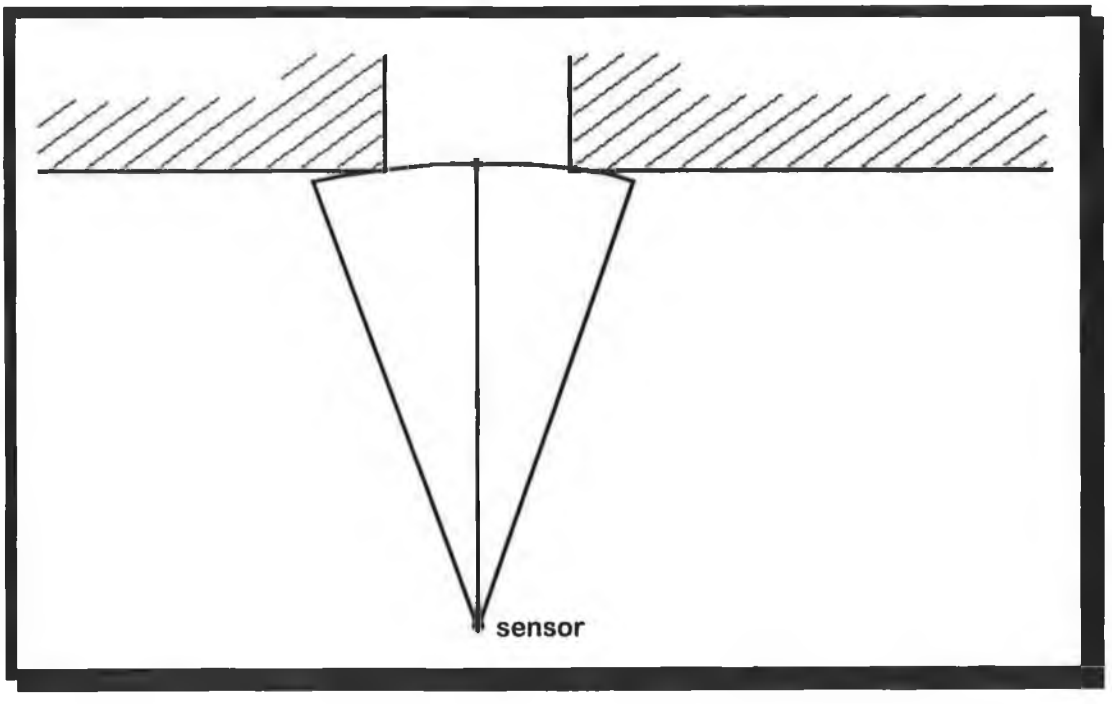


Figure 2: Smearing out of edges

Other problems arise due to the acoustic impedance of air being quite low compared to the typical acoustic impedance's of solid objects which are much larger (in both medium density and propagation velocity), so that all solid surfaces appear as

acoustic reflectors. Also because of the large wavelength of sound (in the order of 3-5mm), most surfaces appear to be acoustic mirrors i.e. many surfaces seem smooth and small surface detail will be undetectable.

Surfaces that are not orthogonal to the direction of acoustic propagation will reflect signal energy away from the source, and the surface may not be detected i.e. if the object surface is oblique with respect to the sensor axis the surface may not be seen, particularly with a narrow beam sensor. A sonar beam incident on such a surface bounces off at an angle equal to the angle of incidence, and can be reflected off other objects before being detected. Hence, the transducer can measure a much longer distance than it should. Fig. 3a is an illustration of the effects of specular reflections where the mirror image of another object is seen rather than the mirror (object) itself. Fig. 3b shows how a ghost object is seen due to the sensor being positioned at an oblique angle to the wall which acts as a reflector.

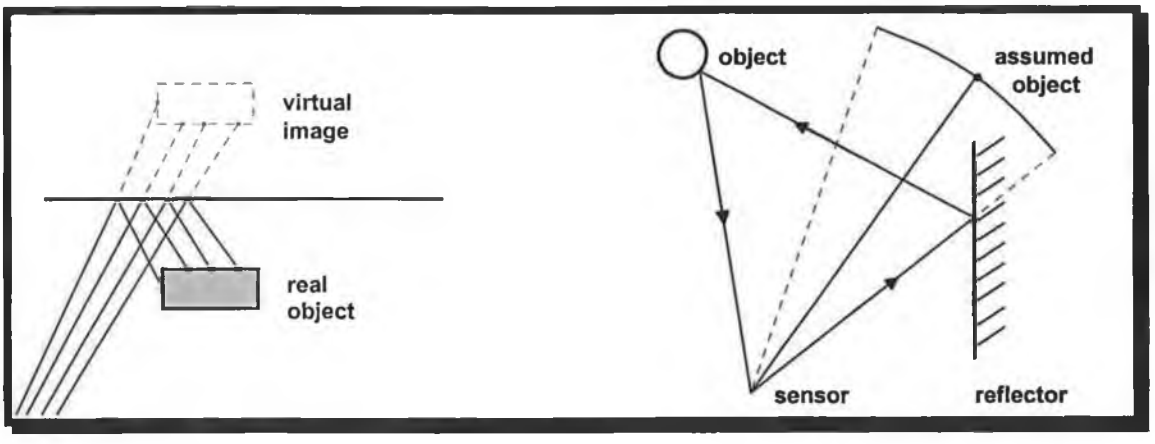


Figure 3: (a) Specular Reflections **(b) Ghost Object**

There may be additional error when using multiple frequencies to compensate for different types of surfaces that absorb more energy at different frequencies. The time of flight measurement begins with the rising edge of the first pulse transmitted and ends with the detection of the first echo. However that echo is not necessarily associated with the first pulse, the one from which the time of flight is measured, hence additional error can be introduced. Errors also occur when one sensor receives

the pulses emitted by another (crosstalk) which can lead to objects being observed in empty regions.

Other errors arise because of acoustic amplitude fluctuations. The magnitude of the range is dependent on the velocity of sound in air, however changes in the medium properties due to atmospheric effects, such as temperature, air currents and humidity variations or gaseous composition affects the speed of sound introducing errors. Under these conditions smooth flat surfaces can appear to be quite rough to a scanning ultrasonic sensor. Some advantage can be taken of acoustic scatter from textured surfaces which may allow them to be detected (but not accurately located) at oblique surface angles that would not otherwise be detectable. For large object surfaces, the effects of diffraction is negligible. For ranges in the order of 1m, the predominant source of range error is thermal variation in the air.

Problems of signal distortion of the reflected (received) signal occurs with curved surfaces (both convex and concave) due to multi-path effects of the finite diameter acoustic beam interacting with the reflection surface. Planar surfaces reflect signals of nearly the same shape as the transmitted signal at any target range, particularly at far range. This does not change significantly at close range. However the curved surfaces reflect distorted signals and the amount of distortion is range dependent. The mechanism is illustrated in fig 4. Because of the many acoustic flight path lengths over the reflection surface the composite reflected signal will be smeared in time. This phenomenon is also present with a planar target at close range but the effect is less significant. This has the effect of stretching the reflected pulse and making the apparent round-trip time longer for curved surfaces than for planar surfaces. Thus timing errors are introduced due to signal distortion.

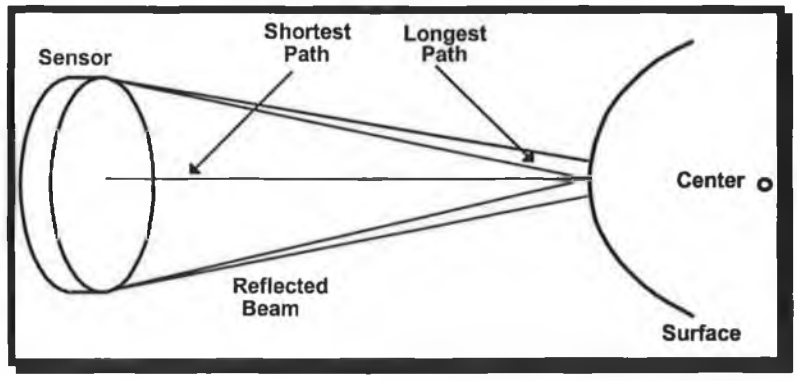


Figure 4: Mechanism for Reflected Signal Distortion

This problem may be decreased by reducing the diameter of the transducer or focusing the acoustic beam on the reflection surface. In this way the effective target surface coverage is small and the multi-path variation is reduced. Focused beams can be obtained with transducers operating at high frequencies (see fig. 5 which depicts a focused beam). The second way of reducing the effect is to transmit pulses with sharper leading edges and detect the return of the first edge thus getting the range to the closest reflection surface. The result is that surfaces with cavities or holes will cause problems for an ultrasonic transducer.

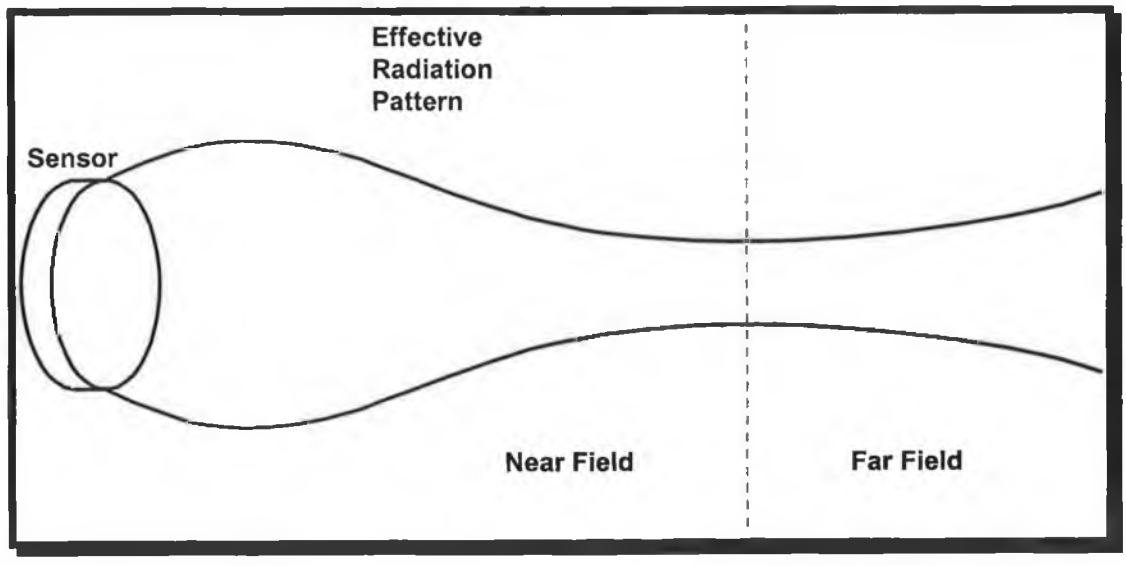


Figure 5: Radiation Pattern at 140 Khz

3.1.3 Current use of Sonar Sensors

Traditionally, active and passive sonar systems have been used for military intelligence applications in marine environments. Active sonar systems are used for communications, navigation, detection, and tracking, while passive sonar systems are used in surveillance. However non military marine sonar systems have spread. Typical applications include navigation, charting, and fishing.

Applications also extend to the medical field for use in medical imaging [43]. Active ultrasonic systems are used in medicine to build maps for human inspection. Other applications include rangefmding devices for industrial control applications, as well as camera autofocus systems [44]. Specific applications of sonar sensors in robotics include simple distance measurements [45], localising object surfaces [46], determining the position of a robot in a given environment, and some ad hoc navigation schemes. Research has also been conducted in the simulation of sonic sensors. This was investigated by Daly[69] who developed a simulator for a mobile robot using ultrasonic sensors. This work allows an individual to test controllers for a robot without worrying about hardware considerations.

3.2 Infrared Sensors

Infrared sensors also have their own problems. Unlike sonar, the time of flight of the light pulse from an infrared sensor cannot easily be measured. The sensor can only indicate whether any returned pulse was detected. So although distance to an object cannot be ascertained, the absence or presence of an object can be determined with very good angular resolution. Smooth surfaces do not have the same problems with specular reflection as they do with sonars, because of the much shorter wavelength of infrared light. However the more expensive laser range finders can determine both range from the sensor and location of the reflection surface due to the narrow beam used.

3.3 Combining different Sensors

Multiple sensors can be used on a mobile robot so that it can perceive its environment with better accuracy than if either sensor were used alone. Sonar and

infrared sensors have different properties that can be used in a complementary fashion, where the advantages of one compensate for the disadvantages of the other. A robot may then combine the information from the two sensors to build a more accurate map. A sonar range finder can provide an estimate of the distance to an object with a high degree of resolution, but has poor angular resolution due to its wide beamwidth. In contrast, the infrared sensor, though not able to measure distance accurately, has good angular resolution in detecting the absence or presence of an object. By using both sensors it is possible to build a better map of ones surroundings. The infrared sensor, can find the edges of doorways and narrow passages that would otherwise be blurred by the sonar's. Flynn[47] uses such an approach to obtain a more accurate map of the environment. Guidelines are also set when to ignore the sonar data given infra-red readings resulting in a more reliable map.

4. Position Estimation

An autonomous vehicle must be capable of determining its current position so that it may travel through an environment and perform its tasks. Many methods have been used in order to determine the position of a vehicle. A vehicle will always have some onboard odometric navigation system. These systems are very accurate but inevitably they suffer from drift which increases, if not corrected, over long distances. The vehicle may also have access to a satellite positioning system which provides it with a global estimate of its location. The global positioning approach does not suffer from drift but is not as accurate as an inertial system. Furthermore since such systems rely on components that are external to the vehicle there is no guarantee that they will be available when needed.

4.1 Overview of approaches taken in determining position

It is possible to compute the position of the vehicle by visually identifying objects of known appearance, scale and location and then triangulating to determine the current position. This involves using high-level computationally intensive vision systems such as stereo vision as a means of navigation. However the computation involved is not always practical for real-time systems. These systems are then used to

update/correct the current position estimate obtained from some local means such as odometry.

However due to noise and difficulties in interpreting the sensory information it is not always easy to recognise naturally occurring reference points. Alternative means have been proposed to counteract these problems. By placing easily recognisable beacons in the robot's workspace the robot may be able to determine its position by means of triangulation. A number of beacons have been investigated [48] for this purpose including corner cubes and laser scanning systems, bar-code, spot mark, or infrared diodes, and associated vision recognition systems, and sonic or laser beacon beams. However this means that the environment has to be modified which may not always be practical or desirable.

The Dead Reckoning Navigation System is based on the measurement of the rotations of the wheels and/or some directional information. The advantages of dead reckoning using odometry sensors is that it is both simple and inexpensive. However, it is prone to a number of errors. Wheel slippage can cause distance to be underestimated, while surface roughness and undulations may cause the distance to be overestimated. Variations in load can distort the odometer wheels and introduce additional errors. If the load can be measured then distortion can be modelled and accounted for. A more accurate solution is to provide a pair of knife-edge nonload-bearing wheels solely for odometry. However very small errors will still accumulate and eventually render the systems useless if the vehicle's environment is not sensed for key landmarks in order to annul these problems.

4.2 Specific implementations for updating the position of autonomous vehicles

In [49], a stereo vision system was used to locate obstacles, plan a path around them and track the motion of the robot as it moved. A path planner based on a grid combinatorial search and incremental path smoothing was implemented. Interest operators were used to pick points to be tracked from image to image. The motion solver determined the motion that minimised the error between where points were seen and where they should have been seen given that motion. The predicted vehicle

position was included as one of the points in the least squared process, weighted more or less depending on the assumed precision of the prediction. In none of the runs was vision as accurate at calculating the translation error, as was straight dead reckoning based on motor commands, though in the best runs vision determined the required rotations more accurately.

A system proposed by [30] comprised of three modules, called the MATCHER, the FINDER, and the SELECTOR. These interacted to establish the vehicle's position with a new level of uncertainty. The MATCHER located likely positions for one or more landmarks in an image and rated these positions to some measure of confidence. The FINDER controlled the focal length and the pointing direction of the camera to acquire specified images for a set of landmarks and directed the MATCHER to find possible positions for these landmarks in the images. It then eliminated possible positions for individual landmarks which were inconsistent with the positions found for other landmarks. The FINDER then evaluated the remaining possible positions to determine the actual positions of the given landmarks. The SELECTOR identified a set of landmarks whose recognition in images of appropriate angular resolution would improve the position estimate of the vehicle by the desired amount. It then directed the FINDER to establish likely positions in such images for subsets of those landmarks. With these positions, the SELECTOR then computed new estimates of the vehicle position and position uncertainty and directed the FINDER, if necessary, to locate additional subsets of landmarks.

A system proposed for position estimation by [36] and tested on Blanche (a mobile robot) consisted of an a priori map of its environment, represented as a collection of discrete line segments in the plane; a matching algorithm that registered range data with the map, and then estimated the precision of the corresponding match/correction which was optimally combined with the current odometric position to provide an improved estimate of the vehicle's position (see fig 6). The entire autonomous vehicle was self contained with all processing being performed on board except for the global path planner which was external to the vehicle.

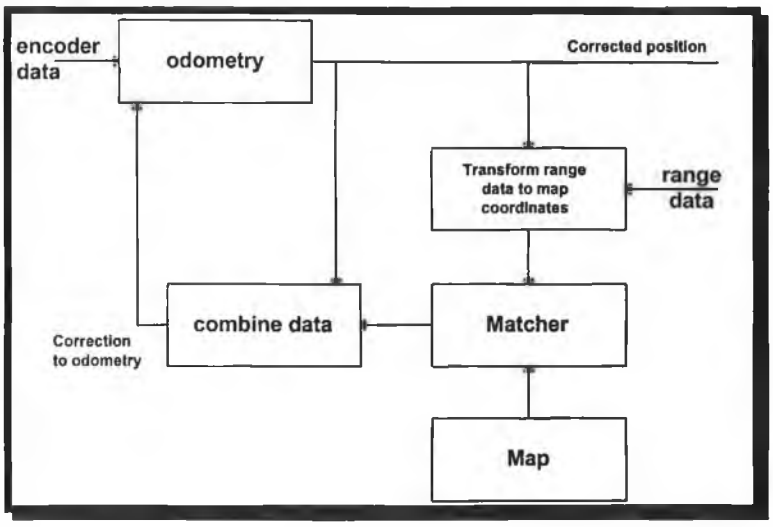


Figure 6: Position estimation subsystem

A rotating optical rangefinder returned a series of points after scanning the room. These points were matched to their corresponding line segments in the stored map using a least square solution that found a congruence that minimised the total squared distance between the image points and their target lines on the map (see fig 7). The solution of the least squares linear regression was a displacement and rotation which was then applied to update and correct the robots' current position. One difficulty with this approach is that the geometric interpretation is done very early and is therefore going to be greatly effected by noise and uncertainty in the data.

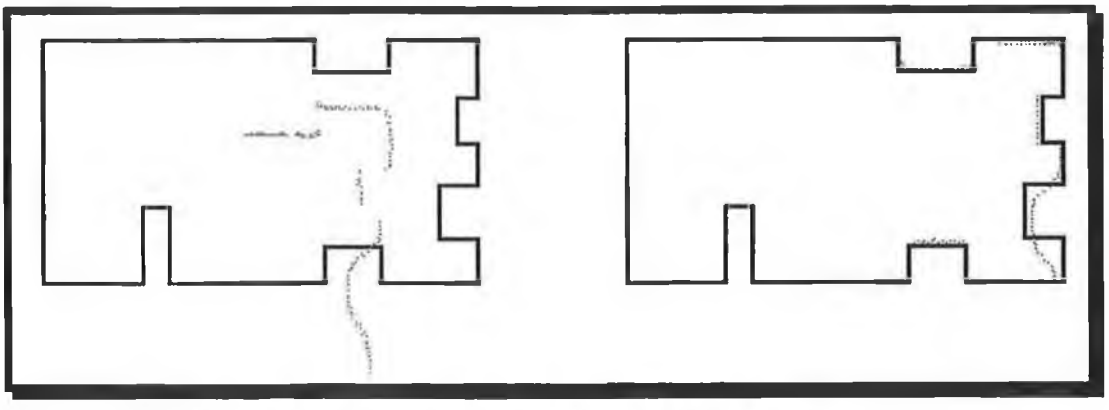


Figure 7: Registration of range data with stored map

Edlinger et al[50] used a laser rangefinder on an autonomous vehicle for navigation in unknown or partially known environments. The vehicle had a tricycle configuration. Dead reckoning was accomplished by using two shaft encoders that

updated the position. This was then fused with information acquired by using the sensors which consisted of four crosswise arranged triangulation laser range finders mounted on a rotating device. By updating the position using the sensors the translational and rotational errors were removed.

Correlation was achieved using a statistical analysis of the environment. This involved computing an angle histogram which is defined as the distribution frequency of the angles of the lines between neighbouring points. The highest peak in the histogram gave the angle α of the main orientation of the robot. The whole laser scan was subsequently rotated by the angle α and the x/y histograms (defined as the distribution frequency of the x/y values of the rotated laser points) were calculated. The highest peaks in these histograms marked the main borders of the environment. This method however assumed that a high percentage of objects in the environment had orthogonal surfaces.

A more primitive system was used in [51], where the vehicle measured its own position by using measuring wheels and corrected the accumulated error at regular intervals using its current position relative to a mark board. The correction interval was about every 50m at which mark boards had to be placed in pre-determined positions. The programmed course consisted of straight lines and arcs. The position and heading were calculated from the rotations of the measuring wheels (not the drive wheels) using rotary encoders. When the current position of the robot was calculated relative to the mark board, the robot referred to the location data of the mark board in memory and then calculated the position and heading to correct the accumulated error of the location unit. Two optical rangefinders were attached to the vehicle for this purpose.

Miller[52] uses sonar sensors to determine the position of a robot. It assumes that an accurate map of the environment is known and performs a search to determine where the robot would have to be to explain a given set of distance readings. The method does not take into account the errors that occur in actual sonar data. A similar

approach is used by Drumheller [53] who also assumes an accurate map of the environment is available, but is able to cope with noisy data.

5. Mapbuilding

It is important for manipulators and mobile robots to be capable of acquiring and handling information about the presence and localisation of obstacles and empty spaces in their working environment. Operations that involve spatial and geometric reasoning require a knowledge of the surrounding environment. Building up such a description involves the complex task of extracting range information from the real world.

There are numerous range measurement systems that have been proposed in the literature [42]. Of these, the most suitable for mobile robots are stereo vision systems[54], and active rangefinding devices [55], because these do not require artificial environments or contrived lighting. However, due to the inherent limitations in any kind of sensor, it is advantageous to aggregate information coming from multiple readings, so a coherent world-model can be constructed that reflects the information acquired. This world model may then facilitate the execution of essential operations such as path planning, obstacle avoidance, landmark identification, position and motion estimation.

New data must quickly be integrated into a model if the demands of a real time environment are to be met. The robot should be capable of building the model while navigating through the environment. The choice of representation to model the environment is therefore very important, because it determines the efficiency with which new measurements can be integrated into the map. Some strategies employed use a grid that is projected over the environment model where the resulting pixels are the basic elements. Other methods use high-level geometric basic elements such as lines or planes described by parameters. With these representations generality is being traded for speed.

It is often necessary to resolve conflicts between conflicting measurements when updating the global model with new data. A heuristic or a statistical approach may be used for integrating the measurements. Using the statistical method, a Bayesian or a maximum likelihood estimator is used to find the environment model that gives the best explanation of the measurements.

5.1 Specific Implementations taken for Mapbuilding

In [56], a grid representation of the environment is used where every pixel is assumed either occupied or empty, independent of its neighbour's status. This assumption makes it impossible for this strategy to take into account the interference effects caused by multiple reflecting points such as specular reflections.

Thorpe [28], presented a sonar-based mapping and navigation system for an autonomous mobile robot operating in unknown and unstructured environments. The system had no a priori map of its surroundings. It acquired data from the real world through a set of sonar sensors and used the interpreted data to build a sonar map of the environment. High resolution maps consisting of a grid of cells with regions classified as empty, occupied, and unknown were generated. Empty space was given high suitability while regions in the immediate vicinity of obstacles had low suitability. A relaxation algorithm was used to find locally optimum paths. The model was used in a sonar mapper, map matcher and path planner developed for navigating the Denning Sentry.

Mapping was divided into a number of separate stages using the sonar sensors. The sensor data was first pre-processed, screened, and associated with the current sensor position. Each reading was interpreted using probability density functions (see fig 8), where every sonar reading added a thirty degree cone of empty space, and a thirty degree arc of occupancy. Readings observed from one position of the robot being used to build a view, which stored empty, occupied, and unknown areas. This view was then combined with the sonar map. For the recognition of previously mapped areas and positional update, they developed a way of matching two sonar maps by convolving them. The method gave the displacement and rotation that resulted in the

successful integration of one map with the other, along with a degree of confidence of the match.

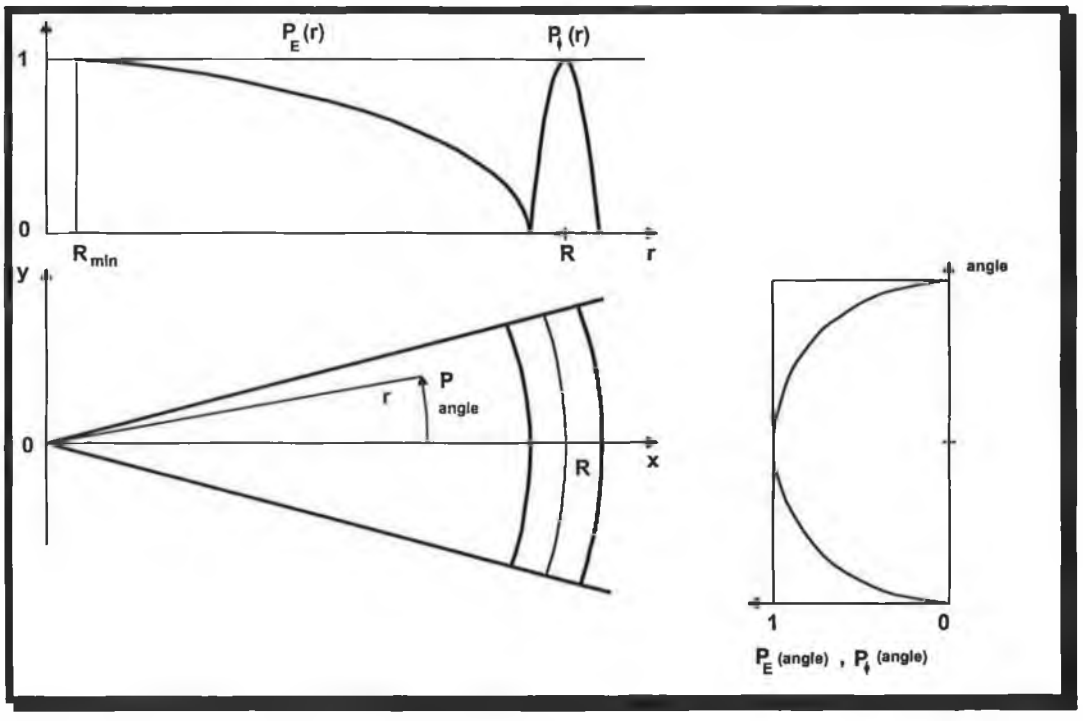


Figure 8.

Every sonar reading provided partial evidence about a map cell being occupied or empty. Different overlapping readings taken that supported each other increased the certainty of the cells in question, while evidence that a cell was empty would weaken the certainty of it being occupied and vice versa. Thus correct information was incrementally enhanced and wrong data was progressively cancelled out.
Results attained after taking several hundred readings produced an image with a resolution often better than 15 centimetres despite spurious data from individual readings. This method proved successful in dealing very reliably with uncertainty, surviving disturbances such as humans around the robot and was tested on the Denning robots as they went around long trajectories in cluttered environments that took a few hours to complete. Three second map building and three second map matching pauses were used at important points to repeatedly correct their position.

The following method was used to match two maps giving the rotation and displacement that best integrated one with the other. Cell values were negative if the cell was empty, positive if occupied and zero if unknown. A measure of the goodness of the match between two maps at a trial displacement and rotation was found by computing the sum of products of corresponding cells in both maps. An occupied cell falling on an occupied cell contributed a positive increment to the sum, as did an empty cell falling on an empty cell (the product of two negatives). An empty cell falling on an occupied one reduced the sum, and any comparison involving an unknown value caused no change to the sum. Faster methods just consider the occupied cells as the above approach is very slow. A hierarchy of reduced resolutions of each map can be generated in order to speed up the computation a bit further.

A three dimensional version of the sonar map builder was implemented by Ken Steward[57] of MIT and Woods Hole for use with submersible craft. The program was tested under large simulated errors and proved reliable in its reconstruction of large scale terrain in a 128x128x64 array using about 60,00 readings taken from a sonar transducer with a seven degree beam. The program ran on a Sun computer and processed the sonar data fast enough to keep up with the approximately one second pulse rate of the transducers on the two available submersibles. Another test was performed using real sonar data from a scanning transducer on an underwater robot that swam over the remains of a civil war battleship providing impressive results.

Serey and Matthies[58] adapted the grid representations for a stereo vision based navigator running on their "Neptune" mobile robot. A dynamic programming method was used to match any edges that crossed a particular scanline in the two stereo images, in order to produce a range profile. The space from the camera to the range profile formed a wedge shape which was marked empty while cells along the profile itself were marked occupied. This map was then used to plan obstacle avoiding paths. Matthies and Elfes[59] combined the sonar and stereo vision programs to builds maps that integrated data from both sensors. This was also run on the Neptune.

In [47], the robot remained stationary while it scanned the area of interest with its head-mounted and near-infrared sensors, taking 256 separate readings. This procedure was repeated in a number of locations. These readings were then merged with the infrared data where edges and large depth discontinuities such as doorways were integrated using information taken from the infrared data (open doorways were not discernible to the ultrasonic sensor because of beam divergence). The refined map was then converted to a representation suitable for planning intelligent tasks. This was achieved by first transforming the refined data to an intermediate representation, a modified version of the curvature primal sketch[60], which is convenient for merging separate views between robot moves. The curvature primal sketch representation can then easily be converted into a polygonal representation of the world suitable for path planners.

One of the traditional methods in mobile robot research has been the use of stereo vision systems to extract range information from pairs of images[61]. A major problem with this technique in real-world navigation is the intrinsic computational expense of extracting three-dimensional information from stereo pairs of images limiting the number of points that can be tracked. Another problem is that traditional stereo vision required features that contained high-contrast edges or points that could be easily tracked along several images [62]. As a result, practical stereo vision navigation systems such as [61], only build sparse depth maps of their surroundings, selecting points to be matched and tracked using an interest operator. They handle in the order of 30-50 points and generating the 3D map can be time consuming.

6. Conclusion

One of the most important functions of a mobile robot is the ability to compute its position accurately in order that it can perform a specific task. The use of Dead Reckoning is not sufficient to maintain an accurate estimate of the vehicle position due to the continuous accumulation of errors that is characteristic of this method. Another method must be used in conjunction with it to offset the problem. These methods can vary - the environment may be altered in some way (e.g. contrived lighting techniques) to facilitate the robot update its position. Guide wires may be

laid down which simplifies the problem. However altering the environment is costly and may not always be a viable alternative. In order to support full autonomous capabilities the robot must be able to determine its position using sensory information.

Many groups building mobile robots rely on sonar sensors for mapping the environment because they seem to require little processing to produce a two dimensional map of the environment. As they are relatively cheap an array of them can be hung around the robot or alternatively they can be mounted on a rotational device to provide 360° sensing. However as discussed earlier in the chapter there is a wide range of errors inherent in using ultrasonic readings. It is also evident that a good deal of effort must be spent on overcoming them. To correct ultrasonic data requires a large number of readings which may not be possible in the limited time span a robot has in performing its task. Therefore while ultrasonic readings may seem to require little processing, much time will be spent on cleaning up the errors which offsets the original reasons for first choosing them.

Sonar sensors cannot be used over long distances without much higher energy outputs than those of the standard cheap sensor being required. These sensors will be expensive and will still be subject to atmospheric effects. Stereo vision systems are expensive and require intensive computation to acquire any information about the environment. Laser rangefinders do not suffer from the problems that ultrasonics have and while they are more expensive, information can readily be obtained from them. Ultrasonic sensors are more suitable as backup sensors for avoiding obstacles rather than for use with position estimation.

Chapter 5

The DCU Autonomous Robot

1. Introduction

This chapter describes the work completed with a mobile robot purchased from Tag[63], by the department of Computer Applications. The robot is controlled from a PC where a map of the environment is entered through a graphical interface and the path is then generated from the source to the destination. Following this, the path is downloaded to the robot via a radio link. The navigator software on the robot, attempts to maintain the vehicle along this path and avoids any obstacles not known to the planner should they be encountered.

The operation of the path planner and navigator are discussed in detail. The path planner will quickly find a path through a cluttered environment because it only needs to consider the obstacles that lie directly within its path as opposed to all of those present in the environment, however it was found to have some limitations that are discussed in section three. The navigator consists mainly of the pilot which is implemented using fuzzy logic. Fuzzy logic is used to determine whether the robot should avoid an obstacle or follow the desired path. Fuzzy rules are also used to maintain the robot along a path or to avoid obstacles depending on the situation. The results from actual tests are presented and they illustrate the effectiveness of the pilot software.

2. System overview

This section gives an outline of the overall system by providing a brief description of the hardware used and the way the software is structured. It also discusses the sensors and the motors used with the robot. The main processor card along with the D/A and A/D cards is also described. These cards all communicate over an STE bus within the robot.

2.1 General description of the robot and radio communications hardware

The Tag robot is a tracked vehicle suitable for indoor use only. Two stepper motors are used to drive the vehicle which has three pods, each containing an ultrasonic, infrared and tactile sensor. Cards slotted into the robot communicate over an STE bus. It has a sixteen bit 80188 microprocessor along with two A/D and D/A cards used for interfacing with the sensors and motors respectively. Two light sensitive sensors attached to the left and right axle of the robot are used to determine its position. Pulses from these sensors are clocked into two timer counters which interrupt the processor to update the position, when a certain number of pulses have been detected. On every control cycle the timer is also read to update the position.

The Robot

Communication with the robot is accomplished using a two-way FM radio link consisting of a modem, transmitter, receiver and line driver. Data is transmitted using frequency-shift keying (fsk), a method which requires small bandwidth. For more detailed information see Appendix A.

2.2 Block overview of software structure

The navigator software endeavours to maintain the robot along a path that is downloaded from the PC. The navigator serves to inform the pilot of the next desired position and orientation of the robot as it progresses from one path segment to the next, along a planned path. The pilot consists of a Fuzzy logic controller, which based on sensor information, avoids unmapped obstacles that lies within the robot's

path. The PC contains a graphical user interface (GUI) that enables the static obstacles to be entered and recorded in a map. The user first enters the destination and start co-ordinates. The start co-ordinates are required only upon initialisation and are sent to the robot along with the final destination. The path planner is then run and based on its knowledge of the environment, a collision free path is constructed around any obstacles that would obstruct the robots passage to its destination. This information is then downloaded to the robot. The software for the path planner, navigator and the communication link between the PC and the robot is shown in Appendix C.

2.3 Using the Robot's Sensors and Motors

In order for the controller to perform correctly, it was necessary to obtain detailed information on the input sensors and on the output motors. This section first discusses how the returned readings from the sonar sensors were interpreted. It explains how a graph of the sonar readings versus distance was plotted. Tests that were used to acquire information on the motor's speed are then discussed.

2.3.1 The Sonic Sensors

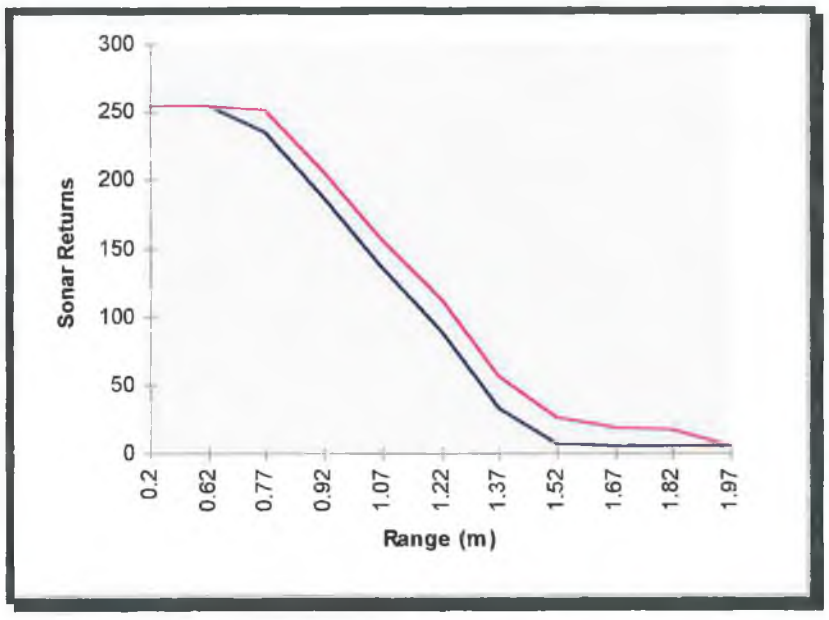
In order to plot the sonar readings versus distance, a program on the PC sent requests to the robot via serial link to send back the sensor readings. A box was placed in front of the robot at the minimum range and was moved progressively out towards the maximum range at which the sonars could detect activity. The box was moved in steps of 15cm and the sonar readings were noted.

The sonar sensors can output anything from 0-10 volts and their maximum range was tuned to 2 metres. This maximum range setting can be extended to 2.5 metres but at the cost of a reduction in the accuracy of the detected object. The Analog to Digital (A/D) card has a range of 0-255 corresponding to an input ranging from 0-10 volts. Therefore when the A/D was connected to the sonar sensors, the returned readings from the sensors varied from 0-255. These values were supplied directly to the fuzzy controller without converting them back to voltages or distance readings. Table 1 shows the relationship between these readings versus distance.

Range (m)	Sonar Readings
0.20	255
0.62	255
0.77	236-252
0.92	187-206
1.07	136-156
1.22	89-113
1.37	33-57
1.52	7-26
1.67	6-19
1.82	6-17
1.97	6

Table 1: Distance versus Sonar Readings

From Table 1 it may be observed that for objects beyond a distance of 0.62 metres, the sonar readings vary between upper and lower limits. The readings for any given distance vary within a bound of 20 units i.e. 0.094 volts. The following plot shows a graph of the maximum and minimum readings plotted against distance. It is evident from the graph that the plot is linear between the range 0.77-1.40 metres.



Graph of Sonar Readings versus Distance

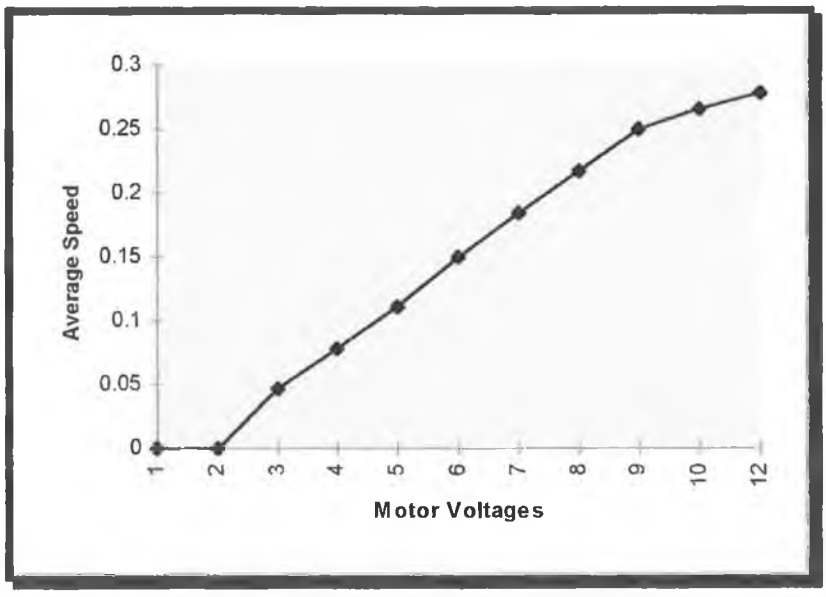
2.3.2 The Stepper Motors

A pair of stepper motors are used to drive the robot. They take voltages from 0-12 volts and are capable of a maximum speed of 0.28 m/s. Tests were performed to obtain a graph of speed versus motor voltages. This involved timing the robot as it traversed a five metres stretch within a lab. The average speed of the robot was then calculated for different voltage levels sent to the robot. The speed could then be found by using the equation *speed = distance / time.* This procedure was repeated three times for every voltage level. Table 2 shows the results of the tests.

Voltage	Speed1 (m/s)	Speed2 (m/s)	Speed3 (m/s)	Average Speed (m/s)
1	0	0	0	**0**
2	0	0	0	**0**
3	0.0466	0.0466	0.0467	**0.0466**
4	0.0781	0.0781	0.0781	**0.0781**
5	0.1111	0.1111	0.1111	**0.1111**
6	0.1515	0.1515	0.1471	**0.15**
7	0.1851	0.1851	0.1851	**0.1851**
8	0.2174	0.2174	0.2174	**0.2174**
9	0.25	0.25	0.25	**0.25**
10	0.2632	0.2703	0.2632	**0.2654**
12	0.2778	0.2778	0.2778	**0.2778**

Table 2: Motor Voltage versus Average Speed

The plot below shows the graph of these results. It is evident from the graph that the speed is linearly dependent on the voltage levels applied to the motors for voltages less than nine volts. Above nine volts the rate of increase in speed starts to diminish. It is also clear that the speed of the robot is sensitive to the voltage applied to the motors. This is due to the weight of the battery which being the heaviest part of the robot, limits the maximum speed substantially.



Graph of Motor voltages versus Average Speed

3. The Path Planner

The planning algorithm implemented in this thesis was developed by Charles W. Warren[64]. It combines some features of graph search and potential fields in its approach to path planning. The algorithm can plan a path around irregularly shaped objects in two or more dimensions. The method quickly finds a path even in very cluttered environments which is why it was chosen.

A simple explanation of how the algorithm may be applied follows. This is then further developed to cater for concave obstacles. Assuming that the obstacle is convex then the following approach will suffice.

An example is illustrated in fig. 1 where an object is shown with the start and goal positions outlined. The centre point within the obstacle refers to its centre of gravity. A trial vector is drawn from the start to the goal position. This vector intersects the object so an intermediate goal is needed. This goal is chosen such that the path will be forced away from the centre of the object. To determine the immediate goal, the point m_1 must first be found. M_1 is the midpoint of the line segment L_1 that passes through the obstacle. A vector is drawn from the centroid through this point and is

projected until free of the object plus a small distance ε. This new point becomes the intermediate goal IG1.

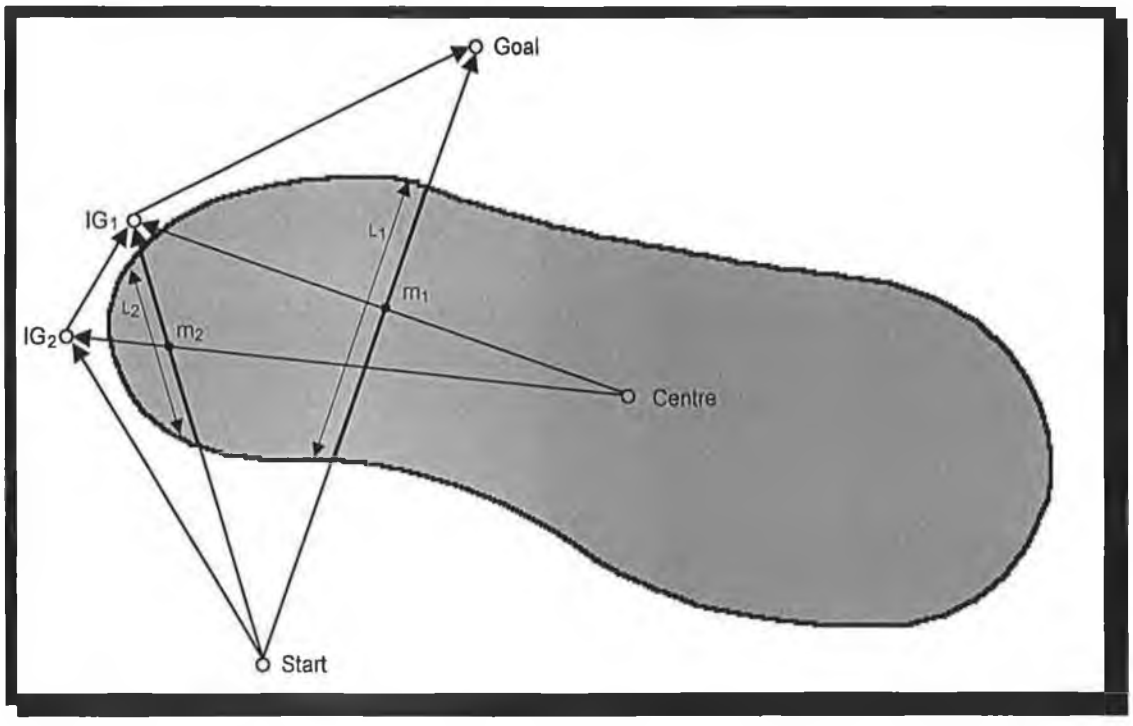


Figure 1: Illustration of procedure for path planning

The previous goal is then pushed onto a FILO (first in last out) stack. The process is then repeated by drawing a vector from the start position to the intermediate goal IG1. This also passes through a forbidden region so a new intermediate goal IG2 is needed. This is determined by drawing a vector from the centroid through m_2 the midpoint of the segment L_2 crossing the obstacle. IG1 is pushed onto the stack and a new vector is drawn from the start to IG2. In this case the path is clear, and IG2 becomes the new starting position. Then an intermediate goal is popped off the stack and the whole procedure is run again until the goal is reached.

This procedure works well with convex obstacles because the centroid lies within the obstacle. If it is used with a concave obstacle the centroid does not necessarily lie within the object and the algorithm can end in an infinite search. An illustration of one scenario is shown in fig. 2.

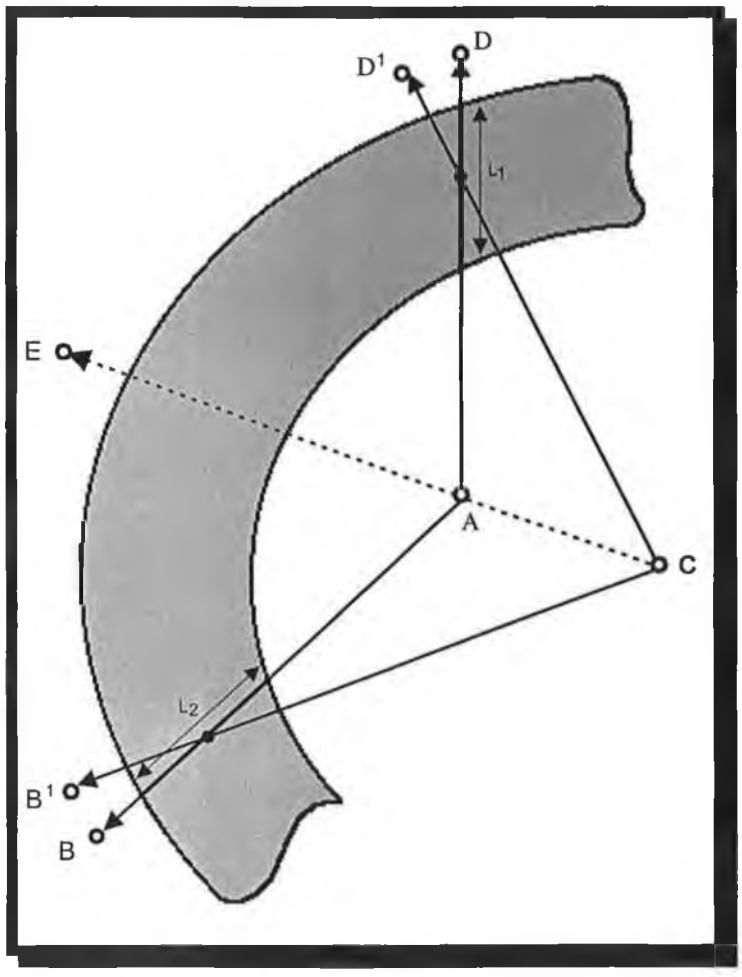


Figure 2: Problem with concave obstacles using the planning approach outlined

This is an example of an object where the centre of gravity lies outside the obstacle. If the start position is A and the destination is B, then when the first trial vector is drawn it will intersect the obstacle along the segment L_2. A vector from C is drawn through the midpoint of the segment L_2 and produced to B^1. This process is continued until the intermediate goal converges at the point E at which point the algorithm will hang. Similarly, if A is produced to D, the resulting intermediate goal will be in the opposite direction and repetition of the process will again result in convergence at the point E. To circumvent this situation a new technique for determining the intermediate goal is used. On the first iteration of the procedure, a path is forced in some direction. It is necessary to continue forcing the path in this direction. This is done by using the midpoint of the previous stage as the start vector for determining the intermediate goal.

An illustration of this procedure is shown in fig. 3. The vector is drawn from the start position to the destination and then a vector from the centroid through the midpoint of the segment crossing the forbidden region is produced to find the first intermediate goal IG1. A line from the start to this still results in traversing a forbidden region, so IG2 is found by creating a vector from m_1 through m_2 until clear of the obstacle. This procedure is continued except that the previous midpoint is used as the starting vector for finding the intermediate goal.

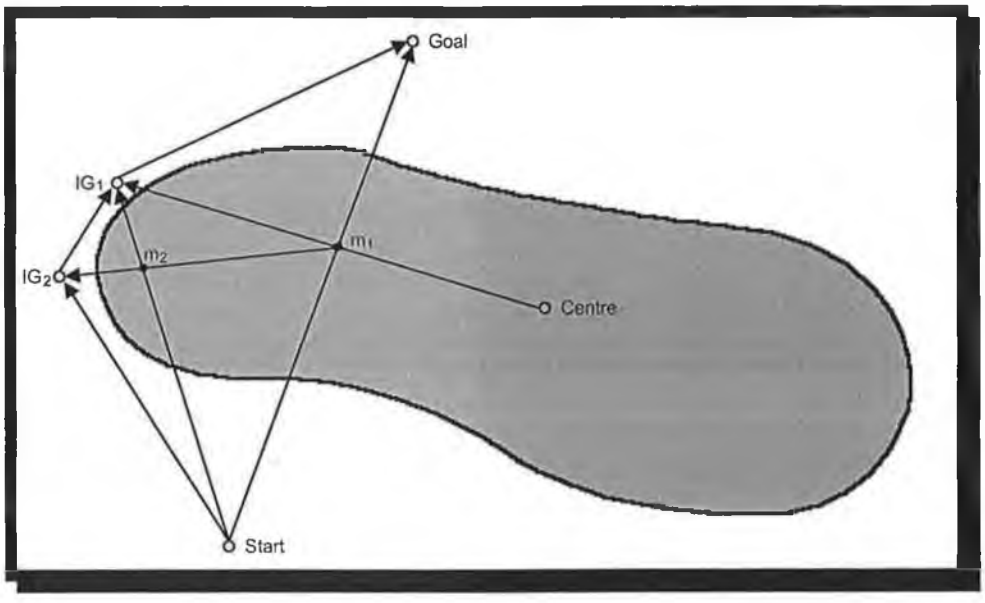


Figure 3: Actual method used for path planning

In fig. 4 this algorithm is applied to the concave obstacle that was previously discussed. The start position is A and the destination is B. When a line is drawn from A to B, it intersects the obstacle along the line segment L_1. A line is drawn from the centroid C through the midpoint of L_1 and produced to B^1. When A is joined to B^1 it still intersects the obstacle along the segment L_2, so the midpoint of L_1 is produced through the midpoint of L_2 and extended to B^2. This process is repeated until the point B^6 is reached. When a line is joined between A and B^6 it does not intersect the obstacle so this point becomes the new start position. Note that the points B^1 to B^5 are not used to connect B^6 to the destination B because it can not be assumed that B^5 can be reached from B^6 and B^4 from B^5 etc.

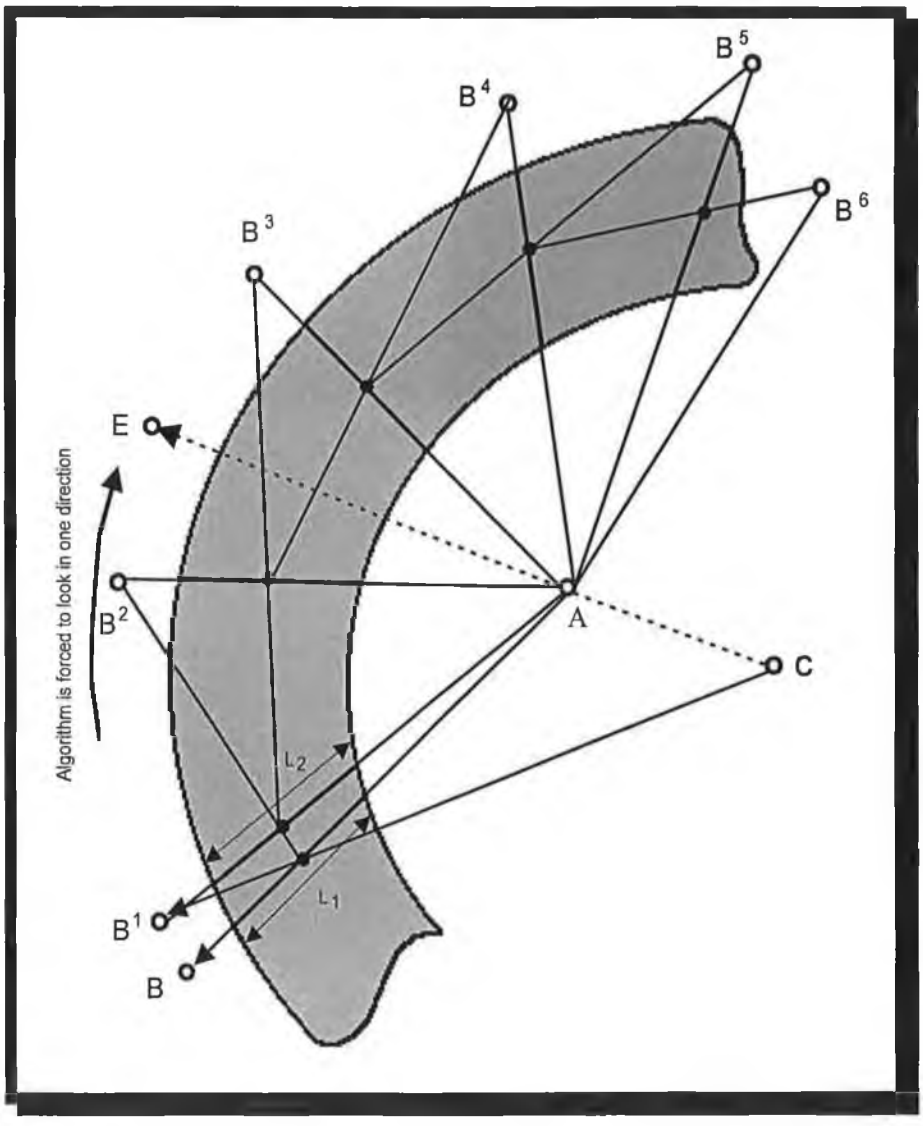


Figure 4: Algorithm applied to a concave obstacle where A is the start position and B is the destination and B^6 is the first point found that will form part of the path from A to B.

Fig. 5 shows the next stage of the algorithm. A line is drawn from A^1 (previously B^6) to the destination B. This intersects the obstacle along the line segment L_1, so a line is drawn from C through the midpoint of L_1 and produced to B^1. When a line is drawn from A^1 to B^1 it is still found to intersect the obstacle along L_2, so a line is drawn from the midpoint of L_1 through the midpoint of L_2 and produced to B^2. A line is then drawn from A^1 to B^2 which is still found to intersect the obstacle along L_3-, so a line is drawn from the midpoint of L_2 to the midpoint of L_3 and produced to B^3.

A line from A^1 is drawn to B^3 which is found to be clear of the obstacle so B^3 is the next point on the path and is renamed A^2 in figure 6.

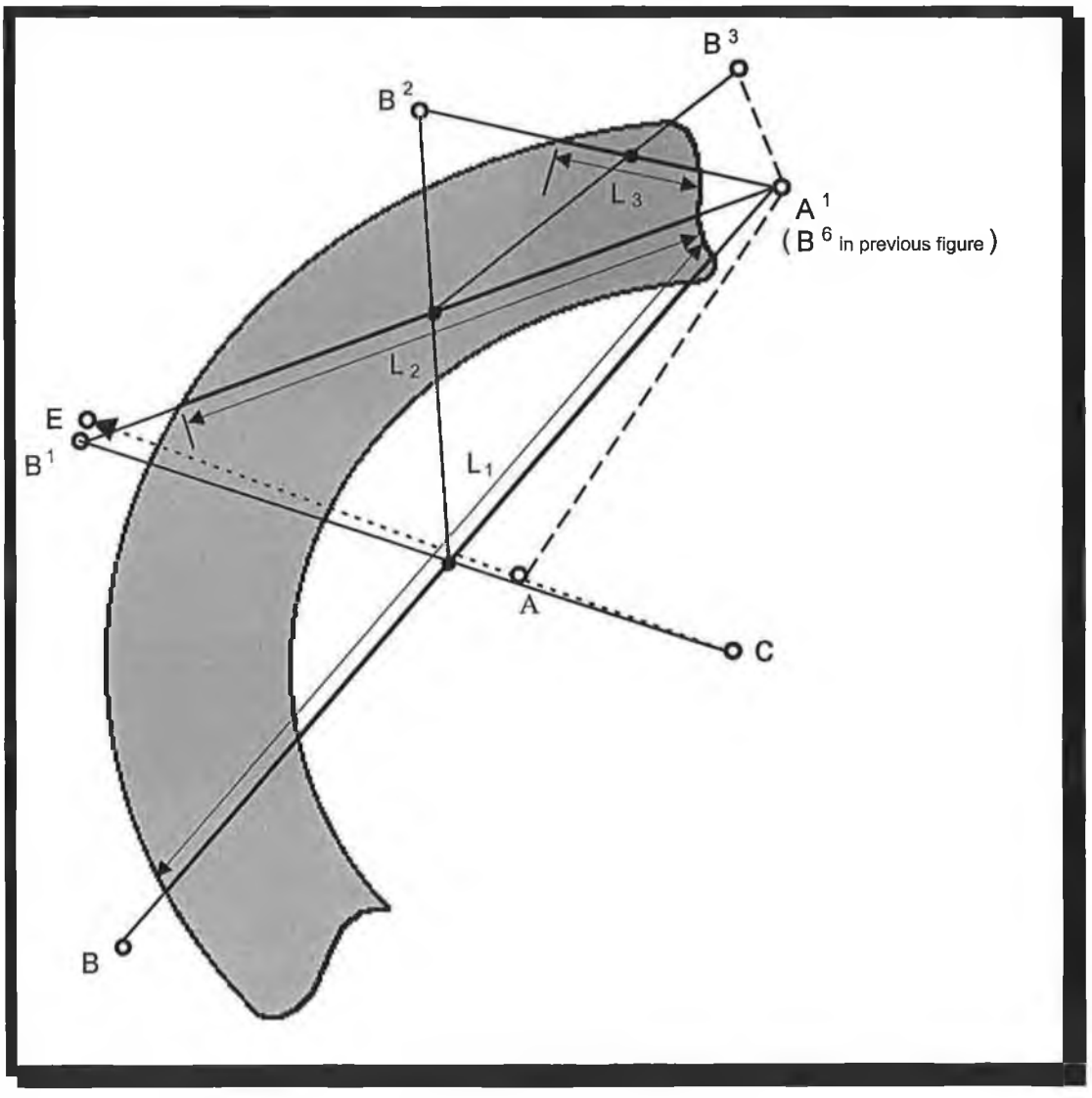


Figure 5: Algorithm applied to new start position A^1 (B^6)

In figure 6 the process is repeated using the point A^2 as the starting position. A^2 is joined to B and intersects the obstacle along L_1. A line is drawn from C through the midpoint of L_1 and produced to B^1. When A^2 is connected to B^1 it intersects the obstacle along L_2, so a line is drawn from the midpoint of L_1 through the midpoint of L_2 and produced to B^2. B^2 becomes the next point on the path because A^2 can be connected to it without touching the obstacle.

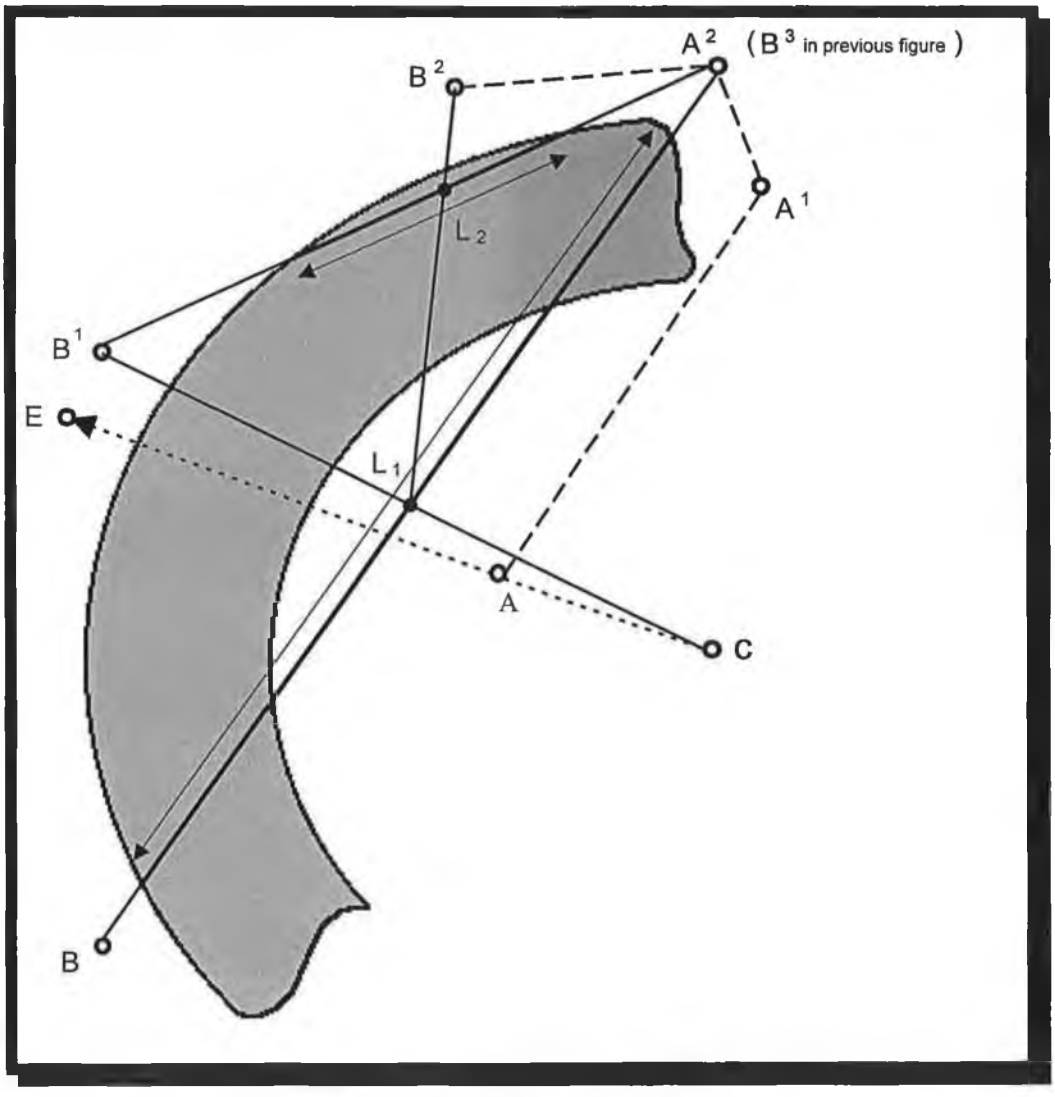


Figure 6: Algorithm applied to point A^2 (B^3)

Finally, in figure 7 the last of the points are found. A^3 is joined to B and intersects the obstacle along L_1. A line from the centroid C is drawn through the midpoint of L_1 and produced to B^1. A^3 can be joined to B^1 without intersecting the obstacle so B^1 becomes A^4, the next point on the path. A^4 can also be connected to B without intersecting the obstacle so all of the points may now be listed in sequence from start to finish: A - A^1 - A^2 - A^3 - A^4 - B.

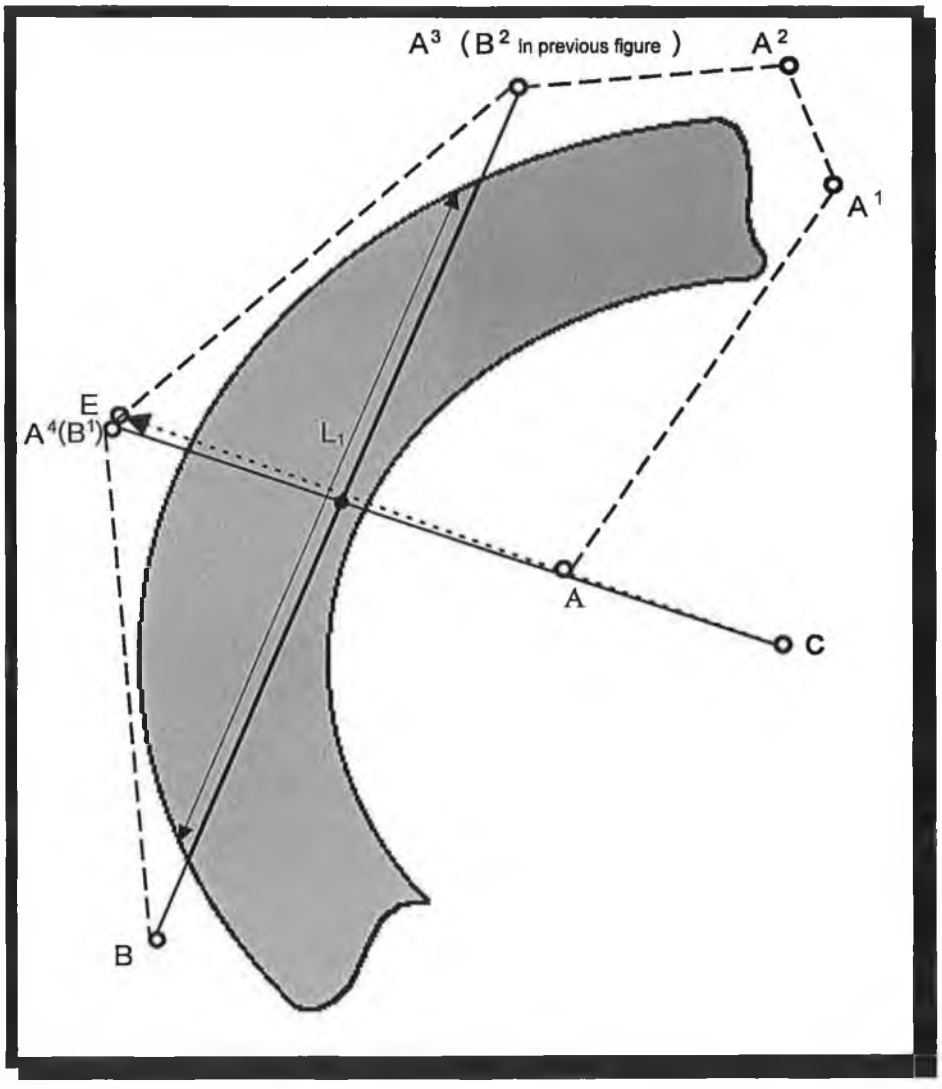


Figure 7: Algorithm applied to point A^3 (B^2)

This method was chosen because it allows a path to be generated quickly even in a very cluttered environment, facilitating quick decision making. But it has some problems that are outlined below:

- If the start and destination happen to have the centroid of the obstacle on the same line segment, then the algorithm will fail. This situation is not discussed in the paper, so it was dealt with by choosing at random a direction at 90° to the path from which the centroid is produced to find an intermediate goal.
- The objects are grown by an amount equal to the radius of a circle that will encapsulate the robot. But points are extended an epsilon distance beyond the object. This epsilon distance must not be such that an intermediate goal is found to be within another object. This means that for any objects that are grown to accommodate the robot and are within or equal to an epsilon distance of each other, these should be joined together to form one object. Figure 9 shows an example of this where two obstacles are first merged into one before planning commences.
- The algorithm does not necessarily get the shortest distance, as may be observed in figure 7.
- As is typical of graph search methods the found path tends to follow close to the obstacles and can mean that the robot has to go at a slower speed because of the continuously changing direction of the path. Free space methods on the other hand do not have this problem.
- The algorithm cannot deal with maze type obstacles such as the one illustrated in figure 8.

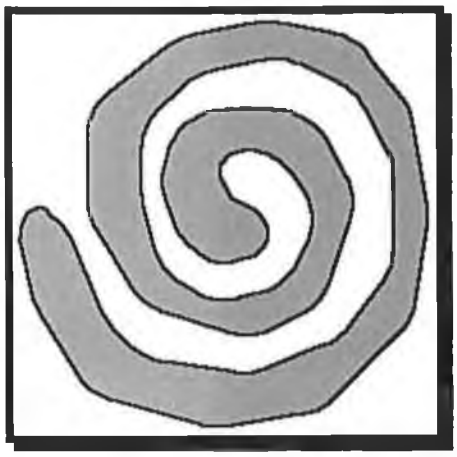


Figure 8: Maze type obstacle

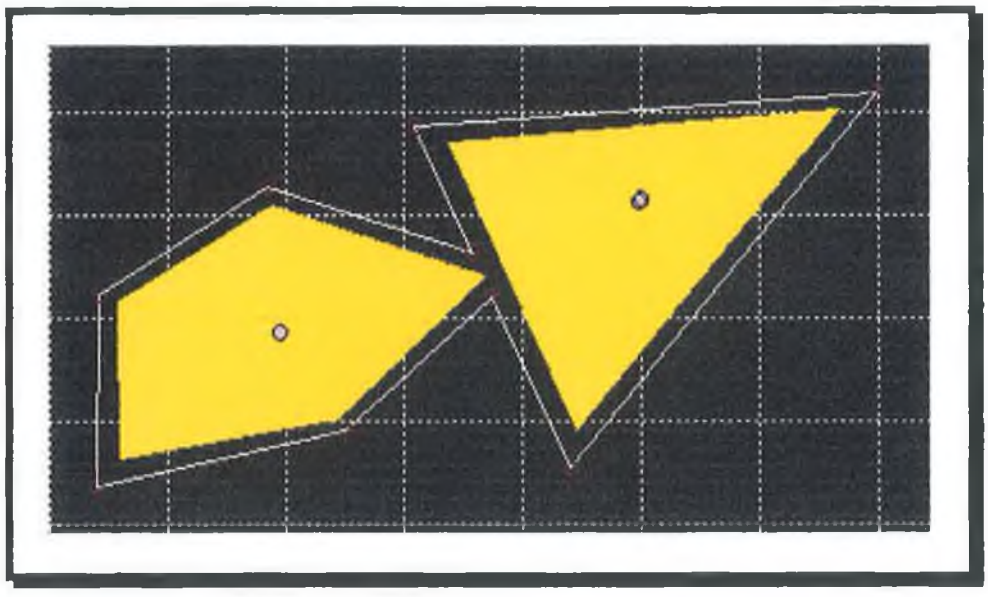


Figure 9: A pair of objects that are too close are merged together

Multiple objects

The method employed for dealing with multiple objects is quite simple. The objects are dealt with as they are encountered. An example is shown in fig. 10 where two objects are blocking the path to the destination. The first object is dealt with as was previously described and when this is accomplished, the second object is dealt with separately using the same procedure.

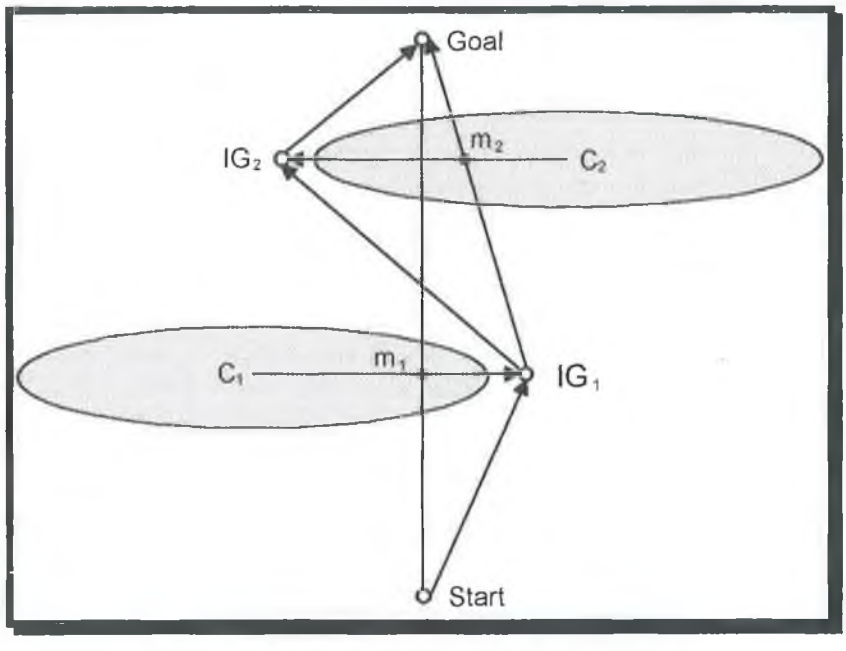


Figure 10: Path planning with multiple objects

A peculiarity of the algorithm is shown in fig. 11 where the path went one way but was subsequently forced back the other way to complete the path to the destination. This problem is easily rectified by using a path shortening heuristic.

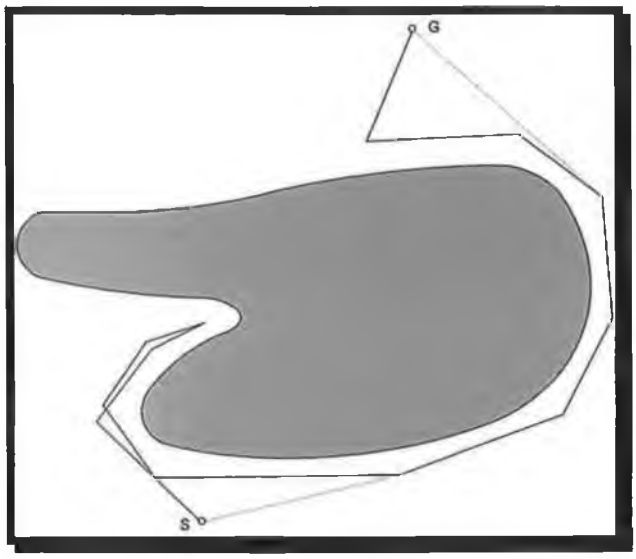


Figure 11: Path Reversal

The planning algorithm is implemented using a *recursive* function written in C (see Appendix C for software implementation). A graphical interface is used to enter and display the environment. Implementing this planner involved getting the centroid and area of an obstacle and being able to determine whether a point was within an object or not. The centroid[70][71] of an object whose mass is uniformly distributed is given by the following equations:

$$a \leq x \leq b,\ 0 \leq y \leq f(x)$$

$$M_x = \frac{1}{2}\int_a^b (f(x))^2 . dx$$

$$M_y = \int_a^b x . f(x) . dx$$

$$A = \int_a^b f(x) . dx$$

$$x_C = M_y/A \quad and \quad y_C = M_x/A$$

where A is the Area of the obstacle and x_c, and y_c are the co-ordinates of its centroid. The planner must be able to determine whether a point is within or outside an obstacle. The obstacle may be convex or highly concave. In [65], a convoluted

algorithm is described to determine this. Rather than using this method I used the following strategy as illustrated below in Figure 12.

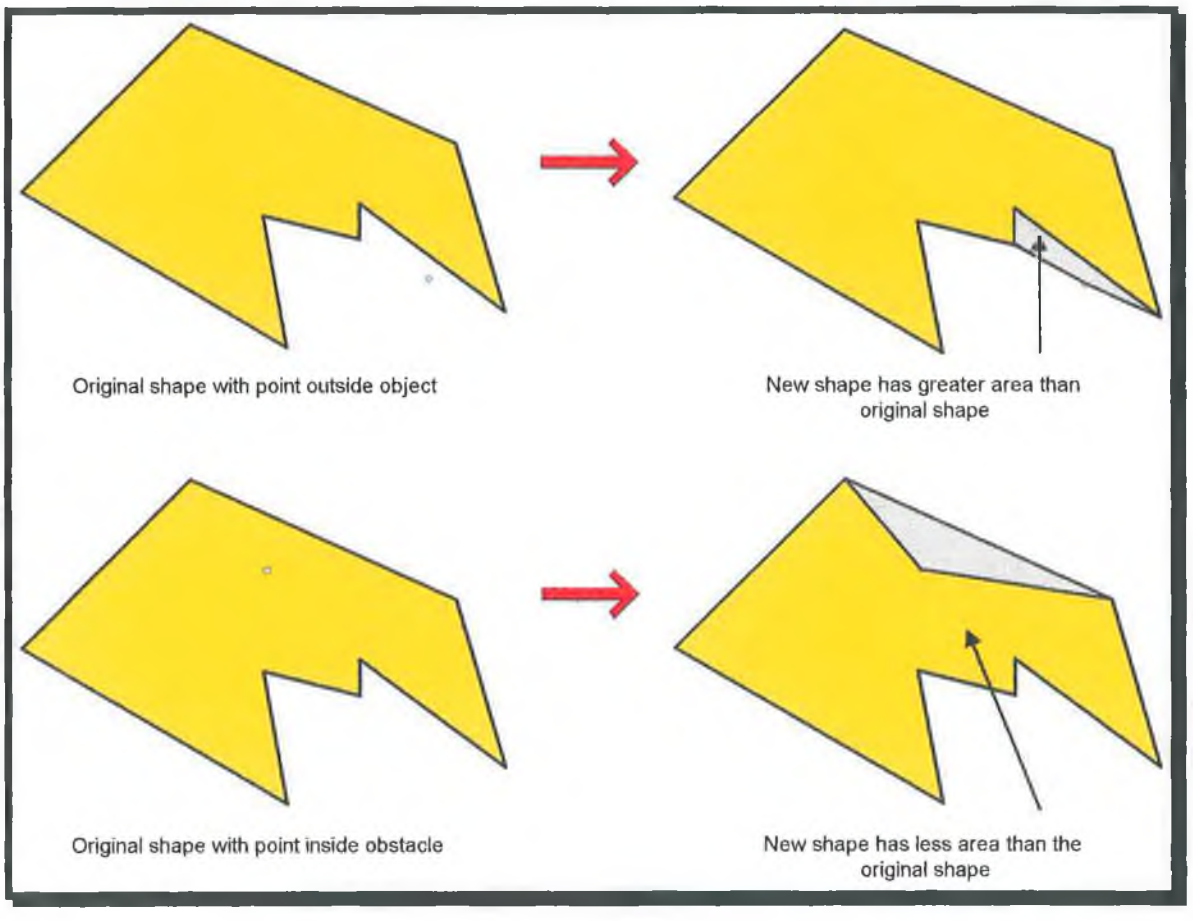


Figure 12: Determining points within an obstacle

The point is added as an extra point in the boundary of the obstacle and the area of the new obstacle is found. If the area is smaller than the area of the original object, then it is within the object, else if the area turns out to be larger, then the point is outside the object.

The area of a general polygon is calculated using the following technique:
The four extremities of the polygon are first found such that it can fit within a rectangle parallel to the X and Y axes. In figure 13 these points are **a, d, e,** and **h**. Starting from the point farthest left and moving anti-clockwise, *the area enclosed between each segment and the x-axis* is either added or subtracted from the polygon. The x co-ordinates of the points are used to determine whether the area is added or subtracted. If the next point has a greater value of x than the previous one, then the

area under the segment is added to the total area of the polygon, else it is subtracted. Note that the X and Y axes are translated to the base of the polygon such that **a** will be on the Y axis and **h** will be on the X axis.

This approach is now applied to figure 13. Let the variable **A** represent a running total of the area as it is being calculated. Let it be initially set to zero. Starting at point (a) and moving in an anti-clockwise direction, the points (a,b) are first encountered. Clearly $b_x > a_x$, so the area between the segment (a,b) and the x-axis is added to **A**. The next segment is (b,c) but in this case $c_x < b_x$ so the area under (c,b) is subtracted from **A**. The next segment has $d_x > c_x$, so the area under (d,c) is added to **A**. The next segment is (d,e) and $e_x > d_x$ so the area under this segment is added to **A**. This process is continued until the last point (a) is reached. In this example (h,a) is the last segment to be processed.

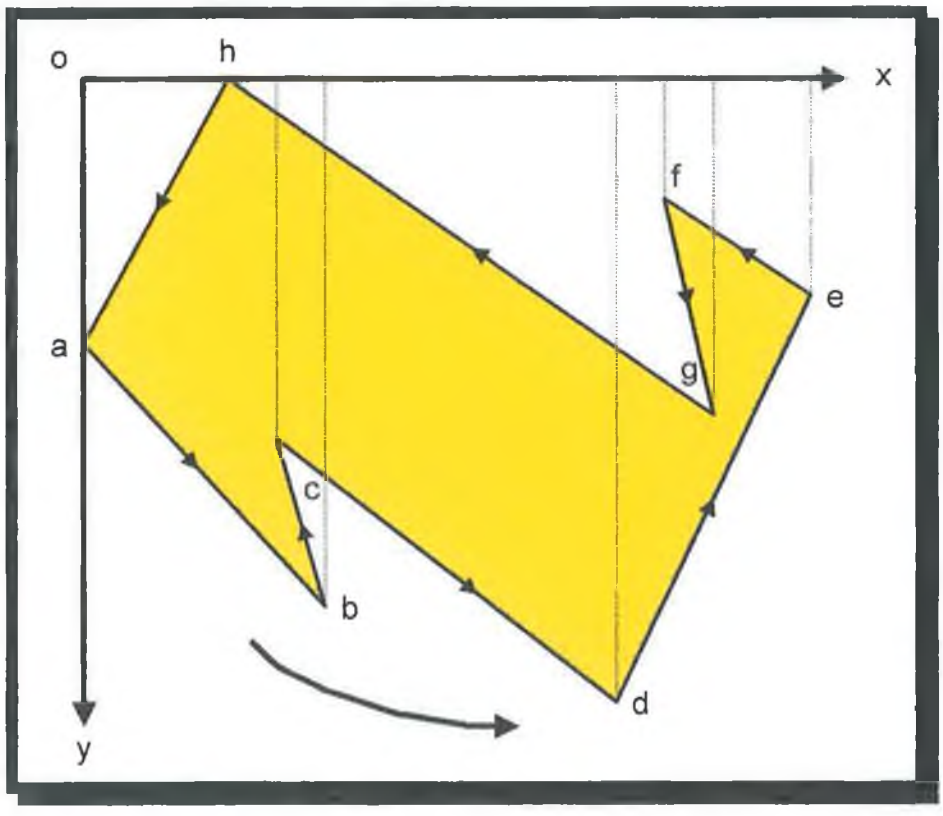


Figure 13: Calculating the area of a polygon

Two examples of the path planner in operation are shown below in figures 14 and 15. The objects within the environment were entered with the use of a mouse and the

start and destination points for the robot were selected. The planner then plans a path around the obstacles. These paths may be seen in both diagrams.

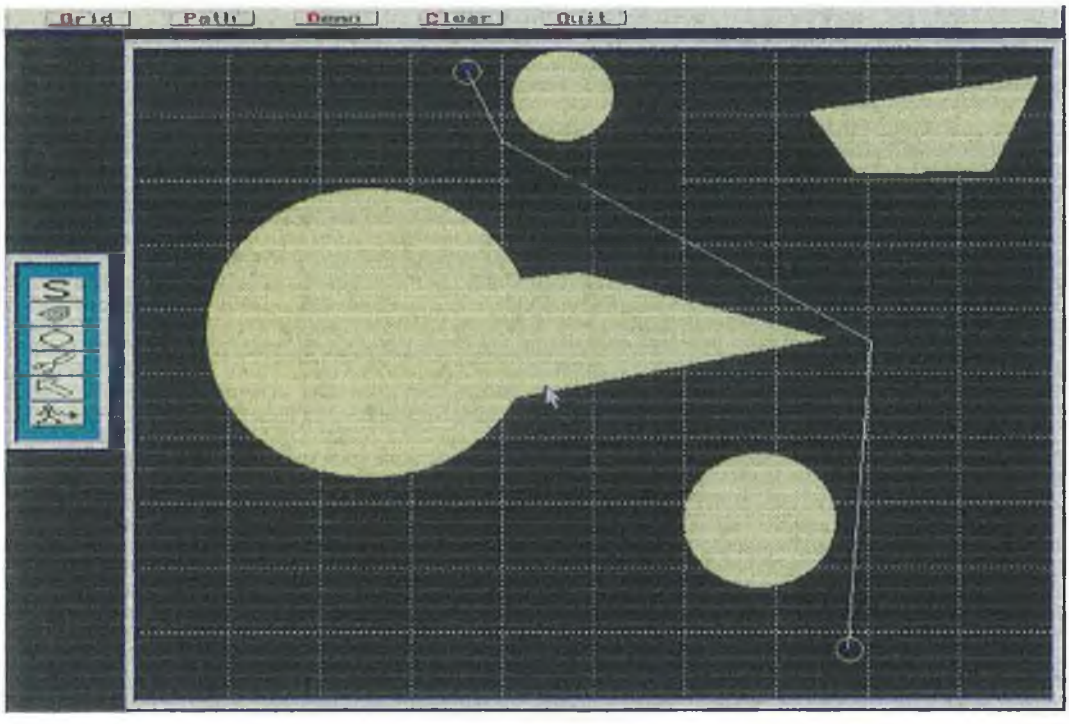


Figure 14: Path planning in object strewn environment

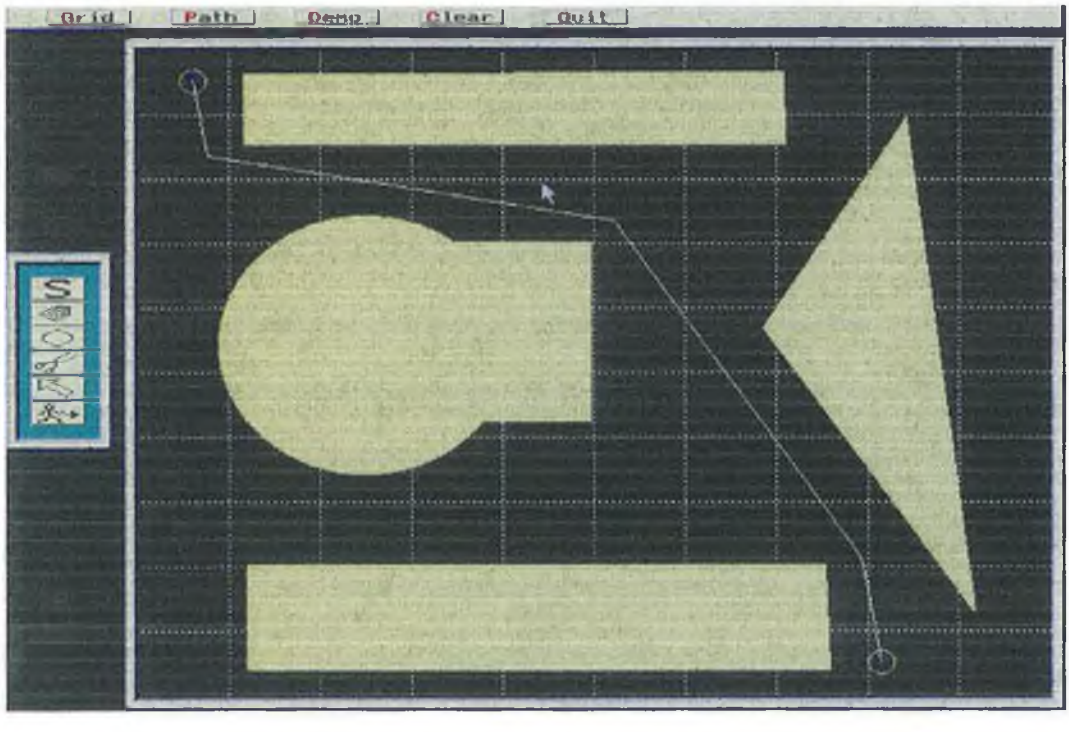


Figure 15: Another example of the path planner in operation

4. Navigator

This section describes the sensor-based navigation method used to control the robot in an indoor environment. When the planner sends the list of path segments to the navigator, the navigator will pass each of these points in turn to the pilot.

4.1 Block Structure of Pilot

The pilot then attempts to maintain the robot along these co-ordinates. If an obstacle impedes the path then the robot will try to pass it in order to complete its mission. The process of navigation is therefore split up into two main functions:

1. To maintain the robot along a planned path if it is not impeded.
2. To avoid detected obstacles and return to the path.

Each of these tasks are implemented independently using fuzzy logic. Fuzzy logic is also used to determine which task should be performed at any one time. Figure 16 shows a more detailed structure of the pilot. Fuzzy logic is used to select suitable rules for tracing a path or avoiding obstacles depending on the situation. If an obstacle is detected, the rules for avoiding obstacles become dominant over those for tracing a path until the obstacle is bypassed. This mechanism for choosing rules is necessary because it is not possible for the robot to avoid obstacles safely by combining 50% of each of the outputs. The final outputs for the steering (S) and speed (V) are the weighted sums of both outputs given by the equations below:

$$S = S_a * K + S_t * (1.0 - K)$$
$$V = V_a * K + V_t * (1.0 - K)$$

where S_t is the steering angle output for tracing a path and S_a is that for avoiding obstacles, V_t is the speed output for tracing a path and V_a is that for avoiding obstacles, and K is the weighted coefficient which varies between 0 and 1 inclusive. As an obstacle gets nearer, K tends towards *one* which means that the rules for tracing a path are removed and avoidance takes over. Similarly as K tends to *zero* it

may be seen that the rules for avoidance are removed from the final output while the rules for tracing become predominant.

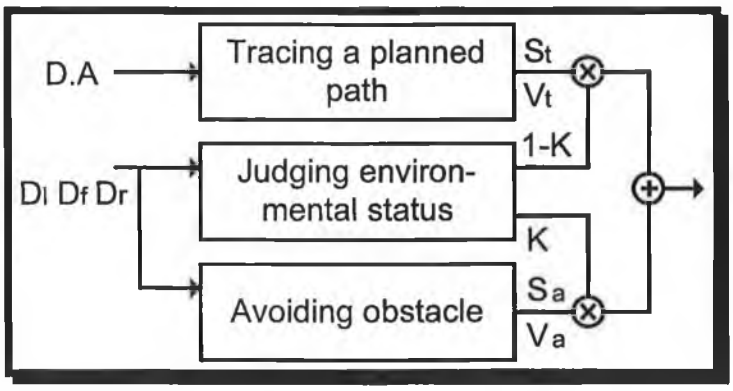


Figure 16: Structure of Pilot

Figure 17 shows the relationship between the fuzzy variables and the rulebases where the circles represent different rulebases and the rectangles represent the fuzzy variables. The Tracepath object serves to hide more detail where two rulebases are required. The contents of this object is shown in Figure 18. The arrows indicate whether the variables are used as inputs or outputs.

Note that Df, Df2, Dl and Dr are fuzzy input variables associated with the distance obstacles are from the front, left and right sensors. The fuzzy variables D and A are related to the distance and orientation of the robot from the path and are discussed in more detail later.

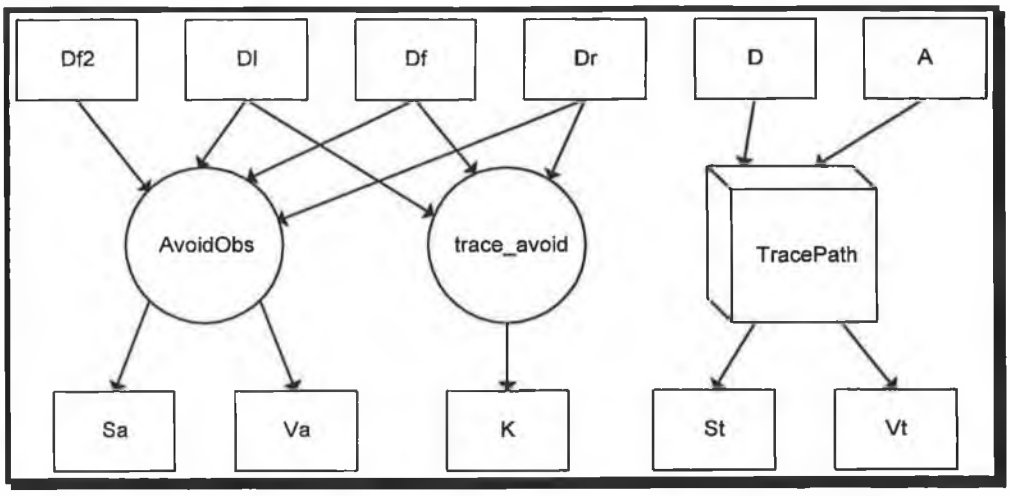


Figure 17: Block diagram of fuzzy controller

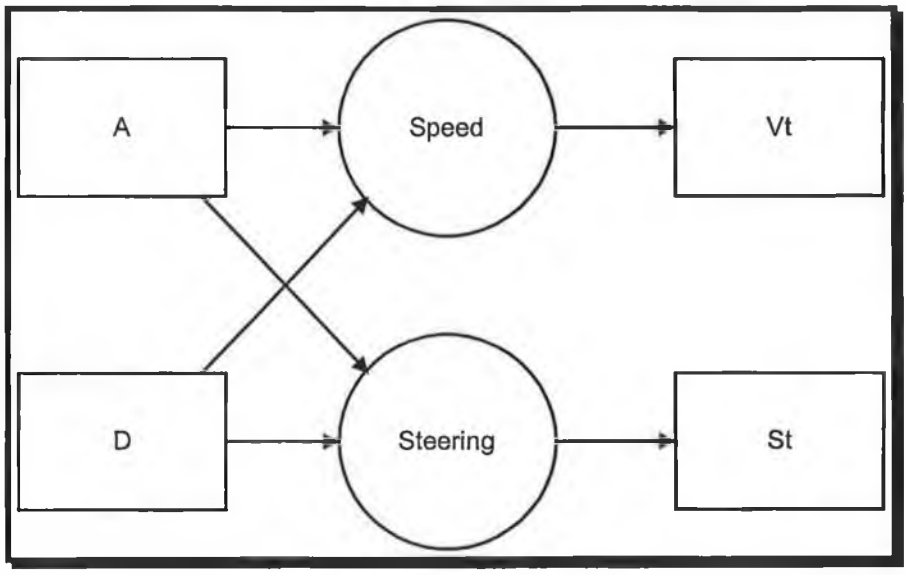


Figure 18: The TracePath object.

4.2 Position Estimation

The Navigator must know where the robot is currently located in order to guide the Robot on its journey and the pilot must also be aware of the current co-ordinates so that it may curtail any deviations from the path. Therefore position estimation is crucial to effective operation. The equations for maintaining the position of the robot are developed as follows:-

Figure 19 shows the path of the robot where it moved from the point o to the point p along the arc op. The robot was initially oriented at an angle of θ and was displaced through an arc of Δθ as it moved along the arc op. If the co-ordinates at o have the values (X_{n-1},Y_{n-1}) and the displacement to the new position p is $(\Delta X_n,\Delta Y_n)$, the problem therefore is to get the value of ΔX_n and ΔY_n where the new position at p (X_n,Y_n) becomes:

$$X_n = X_{n-1} + \Delta X_n$$

$$Y_n = Y_{n-1} + \Delta Y_{n-1}$$

where

$$\Delta X_n = |op|.\cos(pof) \quad (a)$$

$$\Delta Y_n = |op|.\sin(pof) \quad (b)$$

So we must find ∠pof and |op|

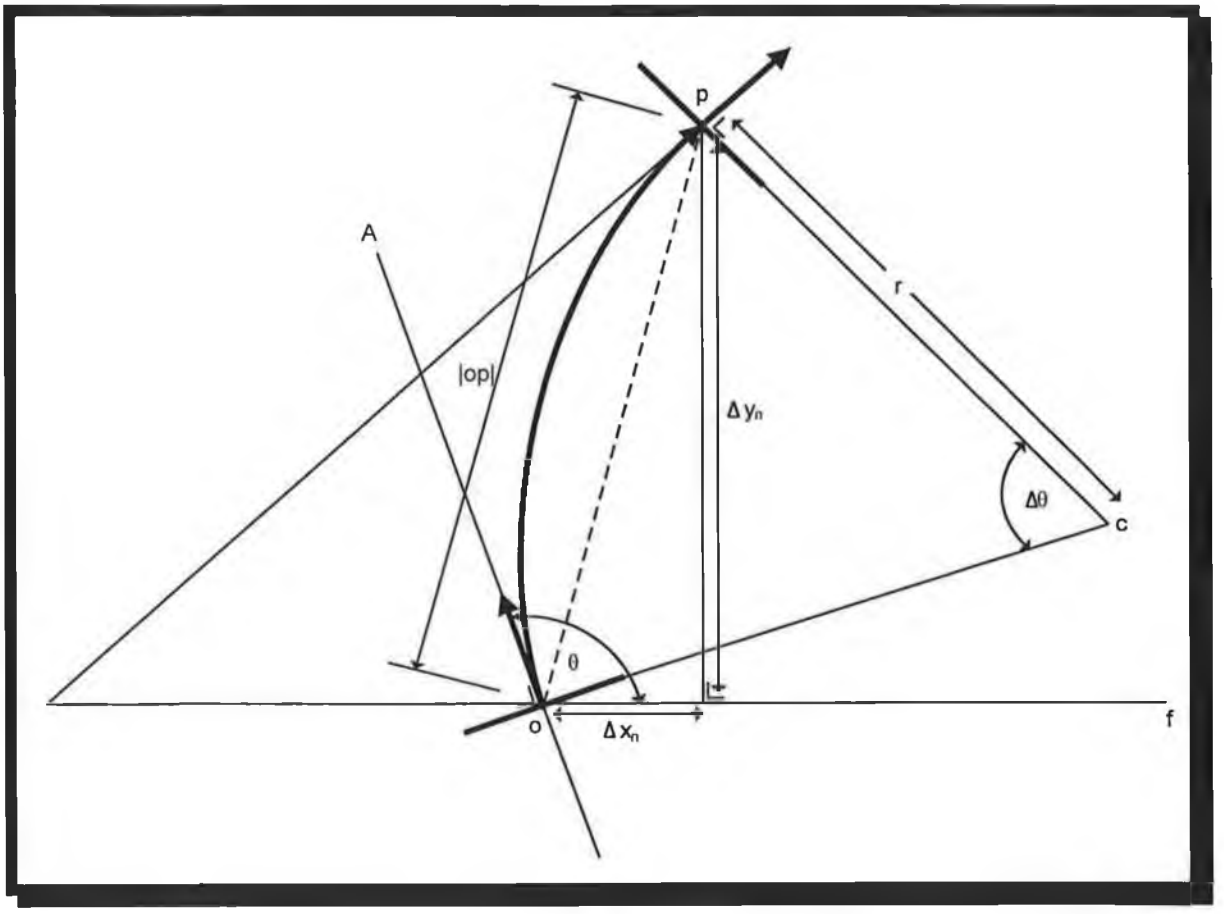


Figure 19: Initial and final position of the robot after moving through an arc of Δθ

First look at ∠pof

Using figure 20 it will be proven that ∠Aop = Δθ/2

Note that as |pc| = |co| = r.

$\Rightarrow \angle ecp = \angle eco = \Delta\theta/2$

$\therefore \angle eoc = 90° - \Delta\theta/2$

But $\angle Aoc = 90°$

Now $\angle Aop = \angle Aoc - \angle eoc$

$\therefore \angle Aop = 90 - (90° - \Delta\theta/2) = \Delta\theta/2$

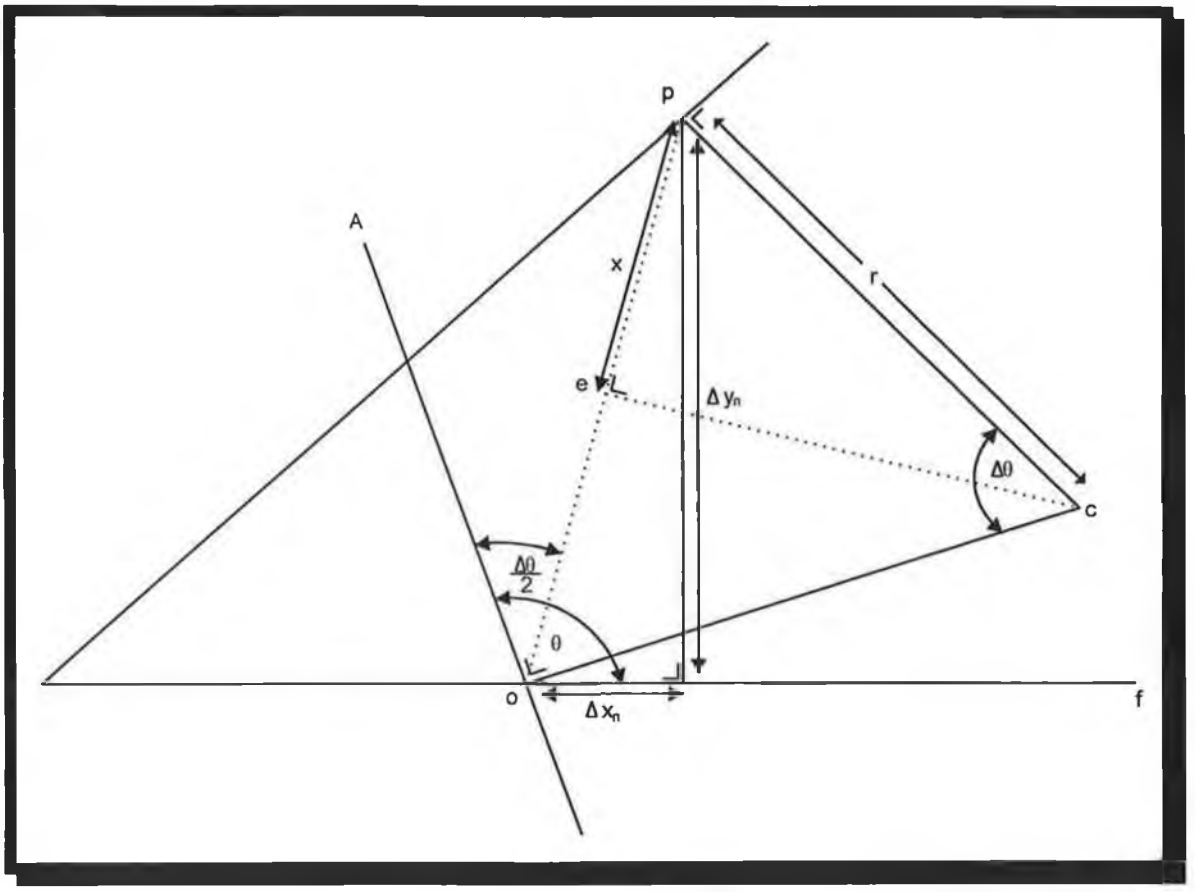


Figure 20: Illustration of the relationship between the present and previous position of the robot.

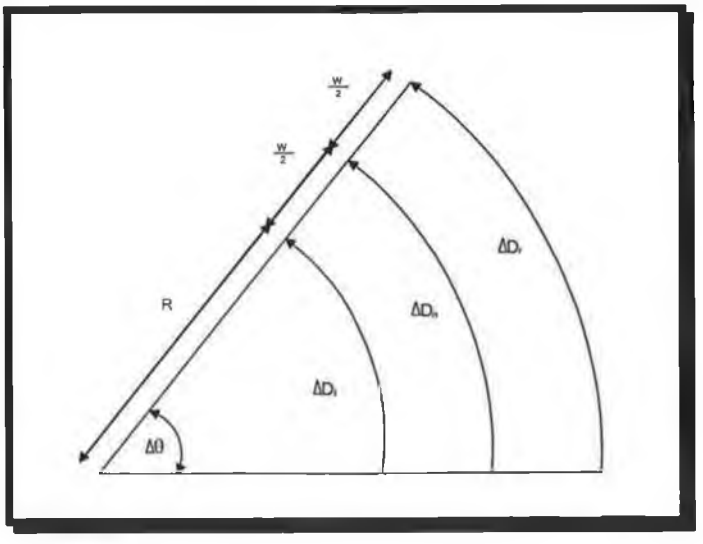


Figure 21: Illustration of the drive system of the robot

Figure 21 shows the robot going through an angle of $\Delta\theta$ where the left track moves a distance of ΔD_l and the right track moves a distance of ΔD_r. From this it may be observed that:

$$\Delta\theta.(R+W) = \Delta D_r$$

$$\Delta\theta.R = \Delta D_l$$

$$\Rightarrow \Delta\theta = (\Delta D_r - \Delta D_l)/W$$

From Fig 20

$$\angle Pof = \angle Aof - \angle Aop$$

i.e. $\angle Pof = \theta_{n-1} + \Delta\theta_n/2$ { if $\Delta\theta_n = (\Delta D_r - \Delta D_l)/W$ } (1) or alternatively

$$\angle Pof = \theta_{n-1} - \Delta\theta_n/2 \quad \{ \text{ if } \Delta\theta_n = (\Delta D_l - \Delta D_r)/W \} \quad (2)$$

We will stick with (1) in our analysis.

Using equation 1, it is worth noting that $\Delta\theta$ will have a negative value in figure 20 because as the robot moves from o to p it moves in a clockwise direction which means that $\Delta D_l > \Delta D_r$. Now to find |op|.

From fig 20:

$$\sin(\Delta\theta/2) = x/r$$

$$\Rightarrow r.\sin(\Delta\theta/2) = x$$

$$|op| = 2x$$

$$\therefore |op| = 2r.\sin(\Delta\theta/2)$$

$$\text{But } r = R + W/2$$

$$\Rightarrow |op| = 2(R + W/2)\sin(\Delta\theta/2) \quad (3)$$

Using the formula $s = r\theta$ to get the length of an arc s, of radius r and angle θ (radians), we will remove the unknown R from eqn (3):

$$\Delta D_n = (R + W/2)\,\Delta\theta$$

$$\Delta D_n/\Delta\theta = R + W/2 \text{ now substituting into (3)}$$

$$|op| = 2(\Delta D_n/\Delta\theta)\sin(\Delta\theta/2)$$

$$|op| = (\Delta D_n \sin(\Delta\theta/2))/(\Delta\theta/2)$$

Substituting for |op| and $\angle Pof$ in equations (a) and (b) we get:

$$\Delta X_n = (\sin(\Delta\theta/2) / (\Delta\theta/2))\,\Delta D_n \cos(\theta_{n-1} + \Delta\theta_n/2)$$

$$\Delta Y_n = (\sin(\Delta\theta/2) / (\Delta\theta/2))\,\Delta D_n \sin(\theta_{n-1} + \Delta\theta_n/2)$$

Thus the final equations become:

I. $X_n = X_{n-1} + (\sin(\Delta\theta/2) / (\Delta\theta/2))\,\Delta D_n \cos(\theta_{n-1} + \Delta\theta_n/2)$

II. $Y_n = Y_{n-1} + (\sin(\Delta\theta/2) / (\Delta\theta/2))\,\Delta D_n \sin(\theta_{n-1} + \Delta\theta_n/2)$

III. $\theta_n = \theta_{n-1} + \Delta\theta_n$

where:

ΔD_n = arc op = $(\Delta D_r + \Delta D_l)/2$

ΔD_r = (Rnumpulse * WheelRadius * 2 * Pi)/Numseg

ΔD_l = (Lnumpulse * WheelRadius * 2 * Pi)/Numseg

and $\Delta\theta_n$ is given in (1)

Note: Rnumpulse is the number of pulses counted by the Right Timer

Lnumpulse is the number of pulses counted by the Left Timer

Numseg is the number of segments on the wheel.

4.3 Processing of input parameters for the Pilot

The input parameters required by the pilot shown in fig. 22 (a) are the Distance (D) from the path and the angle (A) at which the robot is oriented from the path. These two parameters are used as input to the rulebase for tracing along a planned path. Positive values for the parameters D and A mean that the robot is to the left of the path (+D) and is oriented anti clockwise (+A). Similarly negative values for these parameters mean that the robot is to the right side of the path (-D) and oriented clockwise (-A).

The magnitude of D is the perpendicular distance from (*Xn, Yn*) to the path. To determine whether D is left or right (+/-) of the path the following equations may be used:-

$$Xl = Xn - ((W/2) * \sin(ang))$$
$$Xr = Xn + ((W/2) * \sin(ang))$$
$$Yl = Yn + ((W/2) * \cos(ang))$$
$$Yr = Yn - ((W/2) * \cos(ang))$$

where *ang* is the angle of the path, *W* the width of the robot, (*Xn, Yn*) the current position of the centre of the robot and (*Xl, Yl*), (*Xr, Yr*) are points to the left and right of the robot respectively. These equations are used to obtain points to the left and right side of the robot so it can determine which side is furthest from the path. If the left side of the robot is farther from the path than the right side, then it is to the left of

the path. Note that *ang* is the angle of the path segment - not the orientation of the robot, and that *ang* must range from -Pi to + Pi. The angle A is just the difference in angles of the path and orientation of the robot.

The parameters d_f, d_l and d_r, shown in fig 22 (b), denoting the distances measured by the front, left and right sensors respectively, are used in the rulebase by rules performing the obstacle avoidance procedures. These parameters are assigned the voltage readings read from the ultrasonic sensors which give an indication of the distance an obstacle is away from the robot. There is no need to translate the voltages directly to distance as there is a non-linear relationship between the two and unnecessary pre-processing of the information would therefore result. The fuzzy rule base translates the voltages automatically to represent distances that are FarAway, Far or Close etc. so the pre-processing stage is eliminated by just using the raw data.

When an object is detected by the sensors, the fuzzy variable K is altered to reflect this in the weighting of the rules. This is achieved again by using the voltages to represent object free distances that are categorised into a number of memberships, e.g. a high voltage is mapped to the VeryClose membership function (VeryClose means that the distance between some object and the robot is almost zero - indicating an eminent collision) and then using this information to weight K appropriately. The specific rules used to weight K are outlined towards the end of the next section.

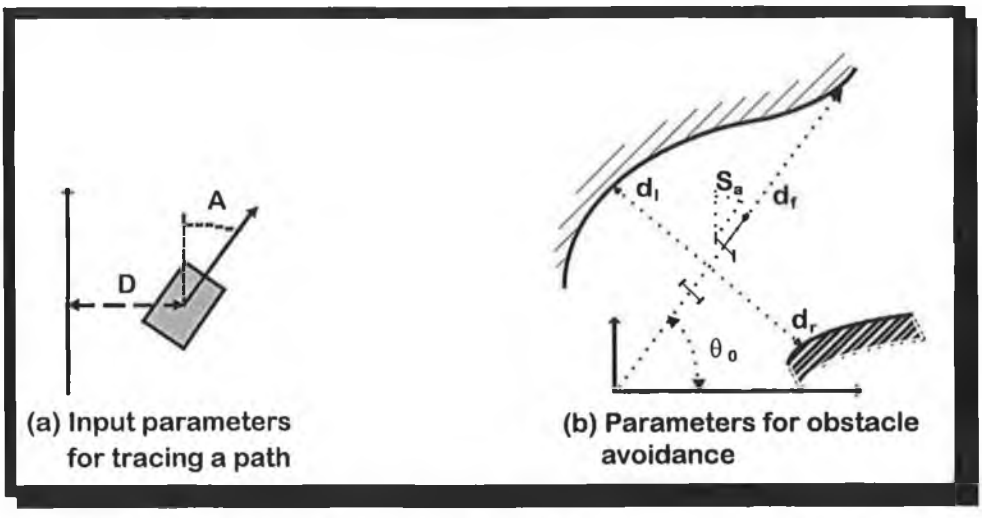


Figure 22: Illustration of the references used for the parameters required by the navigator

4.4 The Pilot's Fuzzy Rule Base and System Structure

This section illustrates the rules used for maintaining the robot along a planned path when no obstacle is encountered. These rules monitor the distance (D) and the angle (A) and strive to reduce these measurements to zero as the robot moves along the path.

The abbreviations used are NB, NM, NS, PS, PM, PB. This is the standard terminology for rulebases that require a large number of rules. The first letter of the abbreviation is an N or P which means negative or positive. This reflects whether an input is negative or positive or if an output is to be increased or decreased. The second letter is B, M or S for Big, Medium or Small respectively. This part reflects the magnitude of the inputs or the degree to which an output must be changed. Therefore PB means Positive Big and when associated with the speed variables V_t or V_a, it means increase the speed to a maximum. ZZ stands for Zero. DF, DL and DR are fuzzy input variables associated with the distance obstacles are from the front, left and right sensors. DF2 is another fuzzy variable associated with input from the front sensor. See Appendix B for the membership functions associated with the fuzzy variables.

These rules were chosen by first drawing numerous diagrams of the robot (see figure 22 (a)) in different positions by varying the values of A and D. The response chosen was based on the action a driver would take under the circumstances. A small level of experience helps in choosing what the drivers action will be but it is mostly common sense. If the robot is far away from the path and veering even further away then swift action is required by quickly changing the direction of steering towards the path and slowing down the speed:

IF (A is PM) AND (D is PB) THEN (S_t is NB)
IF (A is PB) OR (A is PM) THEN (V_t is PS)

If the robot is on the path and its orientation is in line with the path then increase the speed to a maximum and keep the steering angle at zero:

IF (A is ZZ) AND (D is ZZ) THEN (V_t is PB)
IF (A is ZZ) AND (D is ZZ) THEN (S_t is ZZ)

If the robot detects that an obstacle is in front of it and that there is an obstacle to its right side then turn left sharply and slow down in case it hits something:

IF (DF is VeryClose) AND (DR is Near) THEN (S_a is StrongTurnLeft)
IF (DF2 is Close) THEN (V_a is PS)

After the initial rules were chosen simulations were first run to test for different situations. At this stage it became apparent if there were more rules required to deal with certain situations. The same method was used to develop rules for the obstacle avoidance part of the controller.

Tuning the rules was first done through simulation. To test the tracing rulebase, different values of A and D were put into the controller to test each of the rules. Inputs were first chosen to trigger individual rules. If the response to the input was inadequate the relevant fuzzy variable had to be tuned by adjusting the peaks of the membership function. The second step involved choosing inputs that were members of more than one membership function in order to trigger a number of rules. If the outputs turned out to be incorrect, the overlap of the two membership functions was altered. The third step fixed faults in the rules for the controller. If the output of the controller was still not returning the required magnitude or the correct sign, the rules themselves were altered because conflicting rules caused indecision in the output.

This worked very well as long as sufficient thought went into considering the variety of inputs that could occur. I found it was very important to check ambiguous inputs that did not readily fall into one specific membership function . These inputs tested the transition stages between rules and were very useful in fine tuning the membership functions. Such inputs at first, returned miscellaneous outputs before the membership functions were properly tuned.

The same method was used for the avoidance rulebase. Values were fed into the controller that would normally be coming from the sensors. This was tested as before and after simultation returned satisfactory results it was tested in the physical environment. Changes were then made to the rulebase if the robot reacted too slowly or too quickly to a given situation or if it didnt stay far enough away from an unmapped obstacle when trying to avoid it.

The following rules are used to trace along a path:

Rules for Speed Control (Tracing)

IF (A is NB) OR (A is NM) THEN (V_t is PS)
IF (A is PB) OR (A is PM) THEN (V_t is PS)

IF (A is NS) AND (D is NB) THEN (V_t is PS)
IF (A is NS) AND (D is NM) THEN (V_t is PS)
IF (A is NS) AND (D is NS) THEN (V_t is PM)
IF (A is NS) AND (D is ZZ) THEN (V_t is PM)
IF (A is NS) AND (D is PS) THEN (V_t is PB)
IF (A is NS) AND (D is PM) THEN (V_t is PM)
IF (A is NS) AND (D is PB) THEN (V_t is PM)

IF (A is ZZ) AND (D is NB) THEN (V_t is PM)
IF (A is ZZ) AND (D is NM) THEN (V_t is PM)
IF (A is ZZ) AND (D is NS) THEN (V_t is PM)
IF (A is ZZ) AND (D is ZZ) THEN (V_t is PB)
IF (A is ZZ) AND (D is PS) THEN (V_t is PM)
IF (A is ZZ) AND (D is PM) THEN (V_t is PM)
IF (A is ZZ) AND (D is PB) THEN (V_t is PM)

IF (A is PS) AND (D is NB) THEN (V_t is PM)
IF (A is PS) AND (D is NM) THEN (V_t is PM)
IF (A is PS) AND (D is NS) THEN (V_t is PB)
IF (A is PS) AND (D is ZZ) THEN (V_t is PM)
IF (A is PS) AND (D is PS) THEN (V_t is PM)
IF (A is PS) AND (D is PM) THEN (V_t is PS)
IF (A is PS) AND (D is PB) THEN (V_t is PS)

Rules for Steering Control (Tracing)

IF (A is NB) THEN (S_t is PB)
IF (A is PB) THEN (S_t is NB)

IF (A is NM) AND (D is NB) THEN (S_t is PB)
IF (A is NM) AND (D is NM) THEN (S_t is PB)
IF (A is NM) AND (D is NS) THEN (S_t is PB)
IF (A is NM) AND (D is ZZ) THEN (S_t is PM)
IF (A is NM) AND (D is PS) THEN (S_t is PM)
IF (A is NM) AND (D is PM) THEN (S_t is PS)
IF (A is NM) AND (D is PB) THEN (S_t is PS)

IF (A is NS) AND (D is NB) THEN (S_t is PM)
IF (A is NS) AND (D is NM) THEN (S_t is PM)
IF (A is NS) AND (D is NS) THEN (S_t is PM)
IF (A is NS) AND (D is ZZ) THEN (S_t is PS)
IF (A is NS) AND (D is PS) THEN (S_t is ZZ)

IF (A is NS) AND (D is PM) THEN (S_t is NS)
IF (A is NS) AND (D is PB) THEN (S_t is NS)

IF (A is ZZ) AND (D is NB) THEN (S_t is PM)
IF (A is ZZ) AND (D is NM) THEN (S_t is PM)
IF (A is ZZ) AND (D is NS) THEN (S_t is PS)
IF (A is ZZ) AND (D is ZZ) THEN (S_t is ZZ)
IF (A is ZZ) AND (D is PS) THEN (S_t is NS)
IF (A is ZZ) AND (D is PM) THEN (S_t is NM)
IF (A is ZZ) AND (D is PB) THEN (S_t is NM)

IF (A is PS) AND (D is NB) THEN (S_t is PS)
IF (A is PS) AND (D is NM) THEN (S_t is PS)
IF (A is PS) AND (D is NS) THEN (S_t is ZZ)
IF (A is PS) AND (D is ZZ) THEN (S_t is NS)
IF (A is PS) AND (D is PS) THEN (S_t is NM)
IF (A is PS) AND (D is PM) THEN (S_t is NM)
IF (A is PS) AND (D is PB) THEN (S_t is NM)

IF (A is PM) AND (D is NB) THEN (S_t is NS)
IF (A is PM) AND (D is NM) THEN (S_t is NS)
IF (A is PM) AND (D is NS) THEN (S_t is NM)
IF (A is PM) AND (D is ZZ) THEN (S_t is NM)
IF (A is PM) AND (D is PS) THEN (S_t is NB)
IF (A is PM) AND (D is PM) THEN (S_t is NB)
IF (A is PM) AND (D is PB) THEN (S_t is NB)

The following rules are used to avoid an obstacle by tracking around its side:

Rules for Steering (Avoidance)

IF (DF is FarAway) AND (DL is Near) THEN (S_a is WeakTurnRight)
IF (DF is Close) AND (DL is Near) THEN (S_a is MedTurnRight)
IF (DF is VeryClose) AND (DL is Near) THEN (S_a is StrongTurnRight)

IF (DF is FarAway) AND (DR is Near) THEN (S_a is WeakTurnLeft)
IF (DF is Close) AND (DR is Near) THEN (S_a is MedTurnLeft)
IF (DF is VeryClose) AND (DR is Near) THEN (S_a is StrongTurnLeft)

IF (DL is Close) AND (DF is not VeryClose) THEN (S_a is Centre)
IF (DR is Close) AND (DF is not VeryClose) THEN (S_a is Centre)

IF (DL is VeryClose) AND (DF is not VeryClose) THEN (S_a is MedTurnRight)
IF (DL is FarAway) AND (DF is not VeryClose) THEN (S_a is MedTurnLeft)
IF (DR is VeryClose) AND (DF is not VeryClose) THEN (S_a is MedTurnLeft)
IF (DR is FarAway) AND (DF is not VeryClose) THEN (S_a is MedTurnRight)

Rules for Speed (Avoidance)

IF (DF2 is VeryClose) THEN (V_a is ZZ)
IF (DF2 is Close) THEN (V_a is PS)
IF (DF2 is Far) THEN (V_a is PM)
IF (DF2 is VeryFar) THEN (V_a is PM)

The following rules are used to weight K:

IF (DF is VeryClose) THEN (K is PB)
IF (DF is Close) THEN (K is PM)
IF (DF is FarAway) AND (DL is FarAway) AND (DR is FarAway) THEN (K is ZZ)
IF (DF is FarAway) AND (DL is Near) THEN (K is Near)
IF (DF is FarAway) AND (DR is Near) THEN (K is Near)
IF (DF is FarAway) AND (DL is Near) AND (DR is Near) THEN (K is ZZ)

The overall system is shown in figure 23 where the planner sends down the path points to the navigator. The navigator relies on the pilot to control the robot by reading the sensor information and determining the speed and steering angle that should be sent to the drive unit to maintain the robot along the desired trajectory.

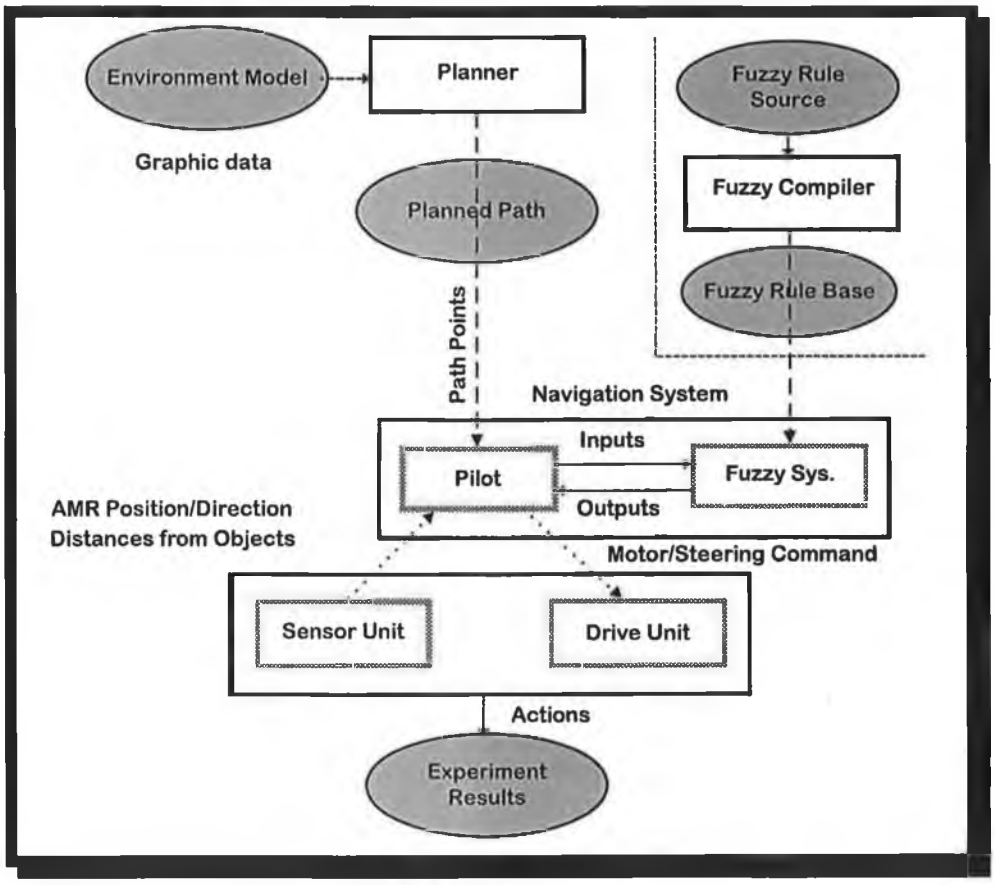


Figure 23: AMR system structure

5. Results

To get the following experimental results, the robot was set in a series of different surroundings. The physical environment was entered on the PC along with the robots' starting position. The robot was then given a destination point and the path planner generated a path around any obstacles lying between the robots' current position and the destination point. This information was then sent to the robot which duly followed this path if possible. The robot periodically updates the PC on its current position and this information is reflected on the graphical screen in real time. Some tests show what happens when the PCs environment was incomplete and did

not totally reflect the actual environment of the robot. In such cases a path was planned through an uncharted obstacle and the results show the robots ability to avoid it and get back onto the path.

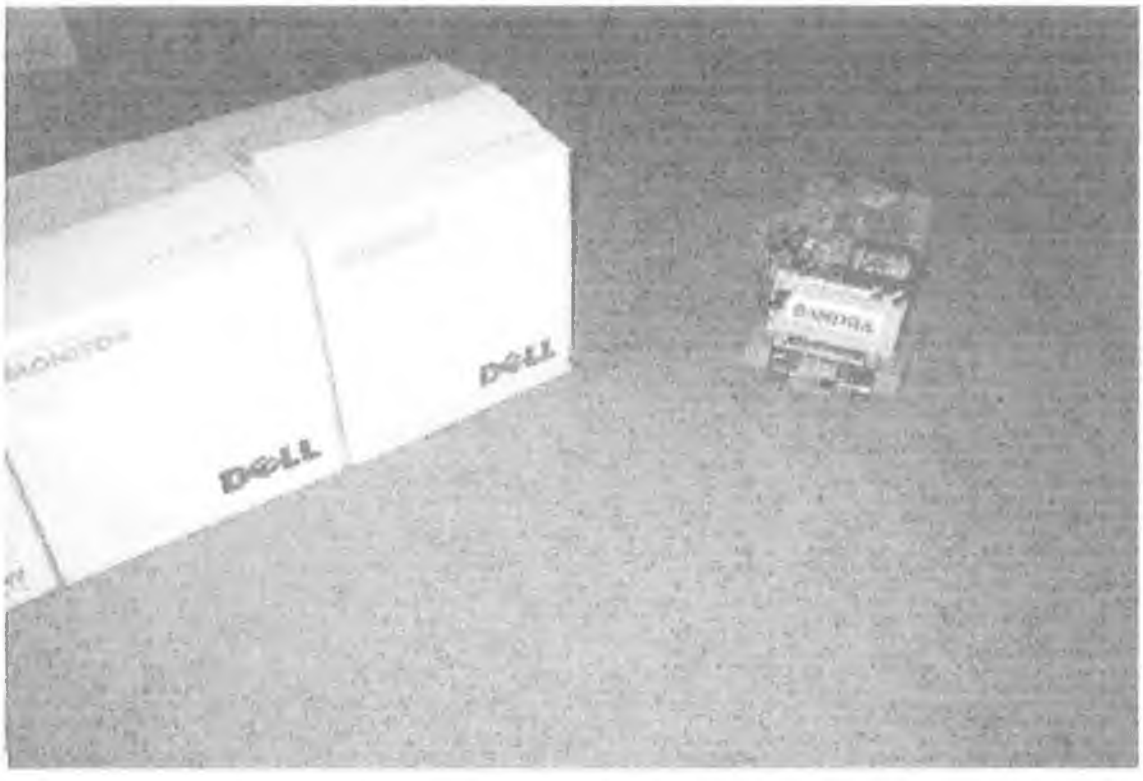
Note that where the *angle of the robot relative to the path was large* at the starting point the oscillations along the path are *greater* because it takes the robot more time to converge on the path as well as track it. This is the typical operation of a PI[72] (Proportional and Integral) controller where the oscillations are centred around the desired steady state output.

Figures 23 to 32 illustrate different situations within the robot's environment providing an insight into the operation of the planner and controller. The distance between each grid line is one meter and the line segments representing the robot's width are also scaled accordingly. Thus each segment represents the robots' position and orientation taken at that instant in time. The centre of a segment corresponds with the centre of the robot and the ends of the segment correspond with the robot's left and right sides.

The orientation of the robot at the start was always set to zero and it can be seen that in situations (fig 23 to fig 30) where the initial orientation of the path from the robot is less than about ninety degrees the controller converges on the path very quickly with little or no deviation from the path. In fig 31 and 32 the initial orientation of the path from the robot is *greater* than ninety degrees so *initially* there are much greater deviations around the path before the robot converges. The following picture shows a typical situation where the robot must track an obstacle.

Figures 24 to 27 show situations where the obstacle avoidance part of the controller takes over completely from the path following controller. These situations occurred where there were unexpected obstacles in the way of the path. When the robot passed the obstacles and the sensors detected nothing else in the way, control was returned to the path follower and the robot converged on the path once again. Figures 24 and 26 illustrate what happened when rectangular objects were placed in the path of the

path and Figure 27 illustrates the result of a cylindrical object being placed on the path. The other main point to note is that it can be seen that the robot slows down slightly when going around an unexpected obstacle. This can be observed from the proximity of the points taken over constant time intervals.



Sample manoeuvre

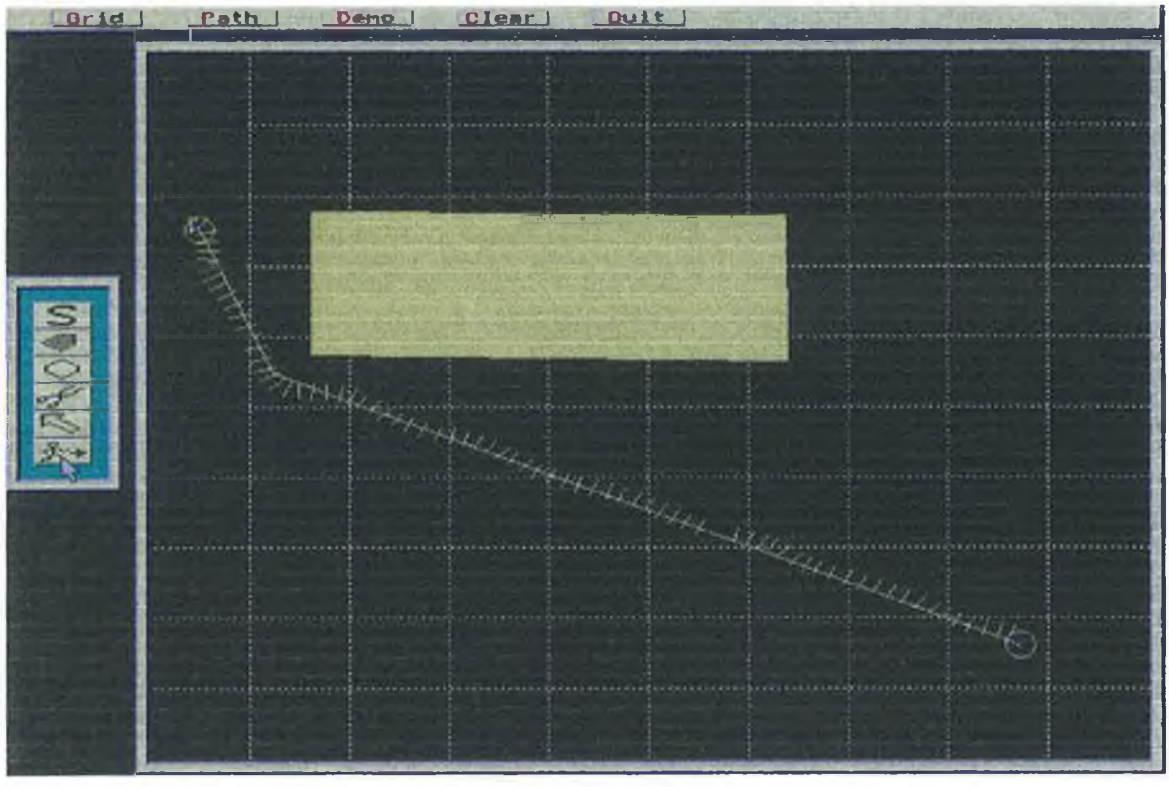


Figure 23: PathPlanner and Fuzzy Controller in operation

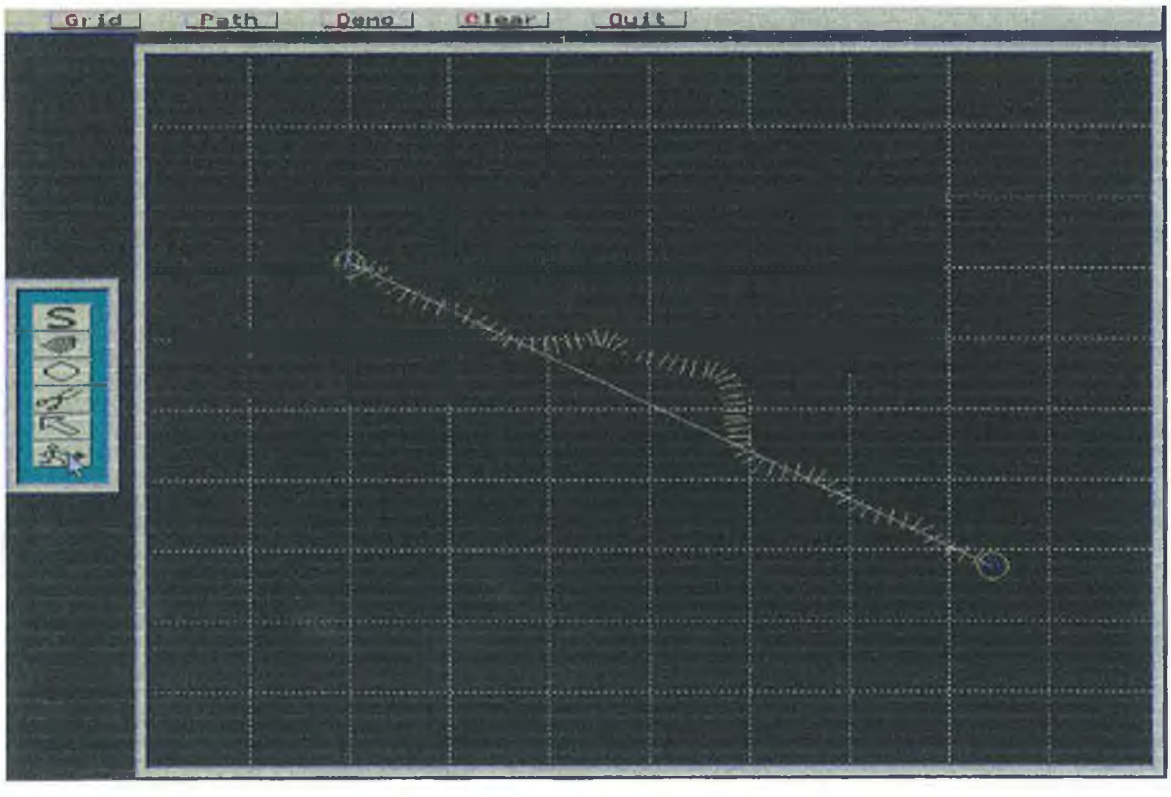


Figure 24: Robot encounters an unmapped obstacle

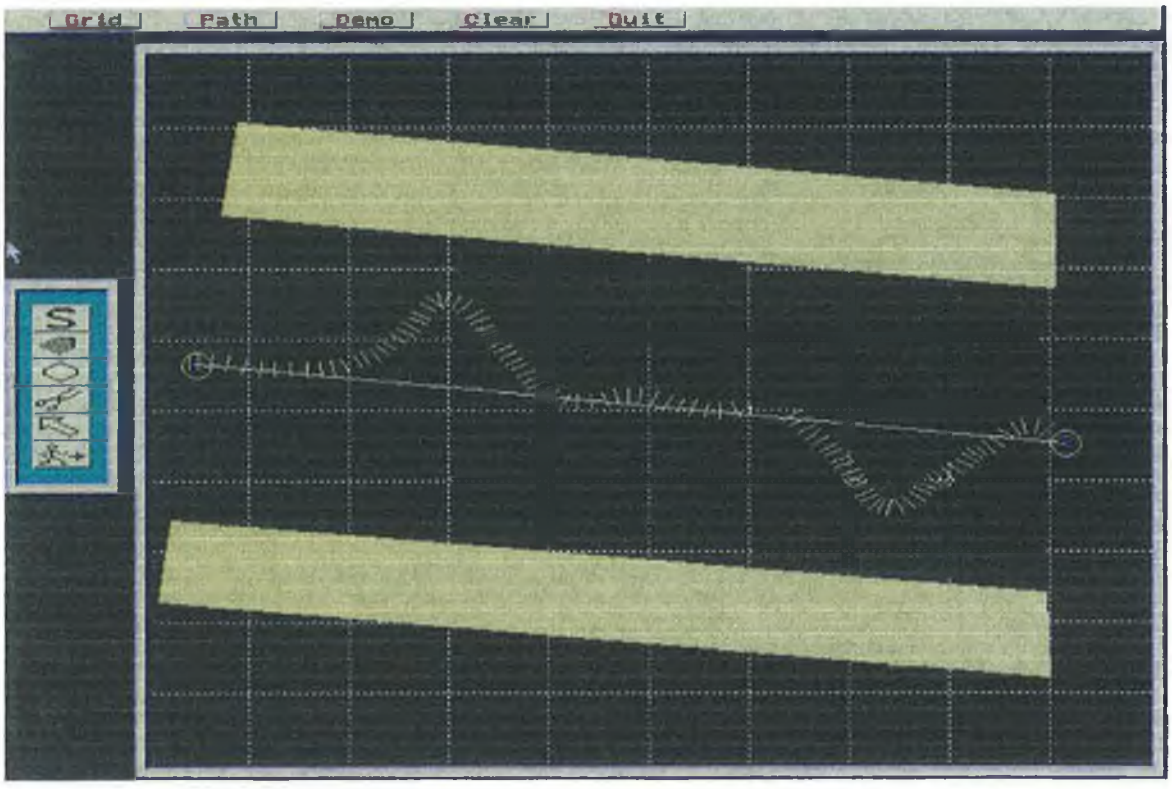


Figure 25: Robot encounters two unmapped obstacles

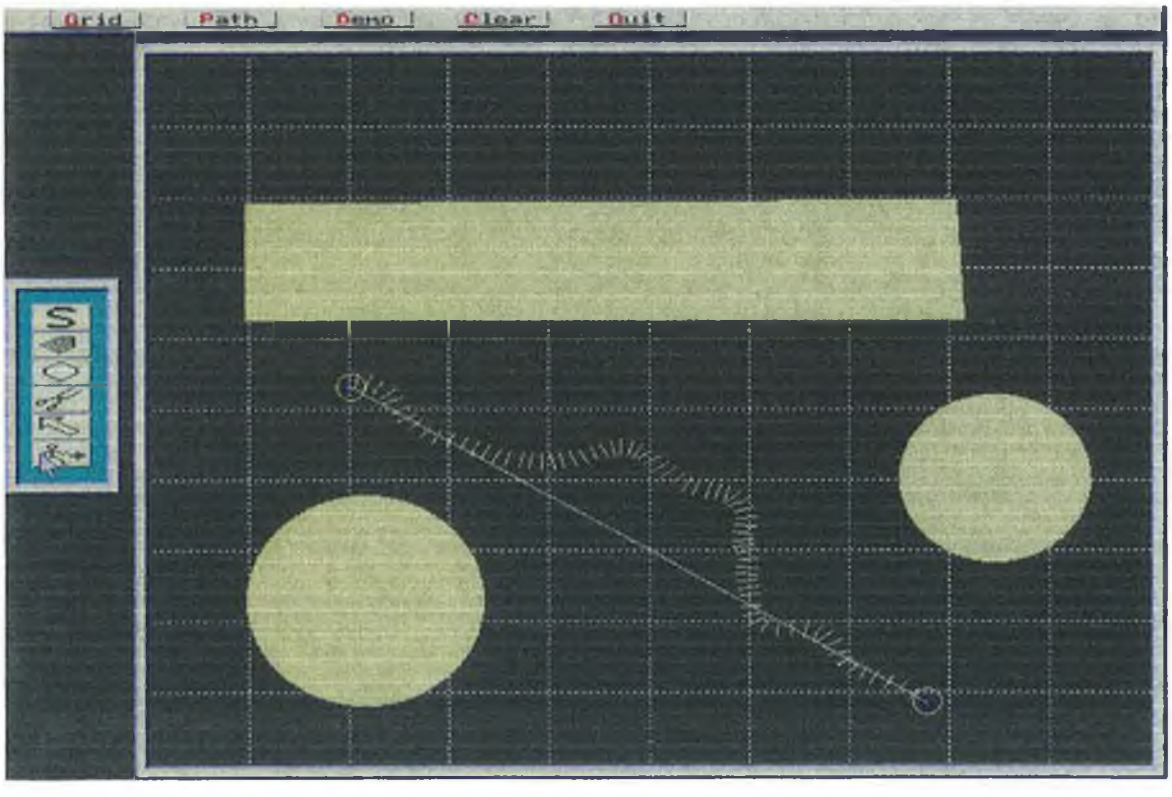


Figure 26: Robot encounters an unmapped obstacle

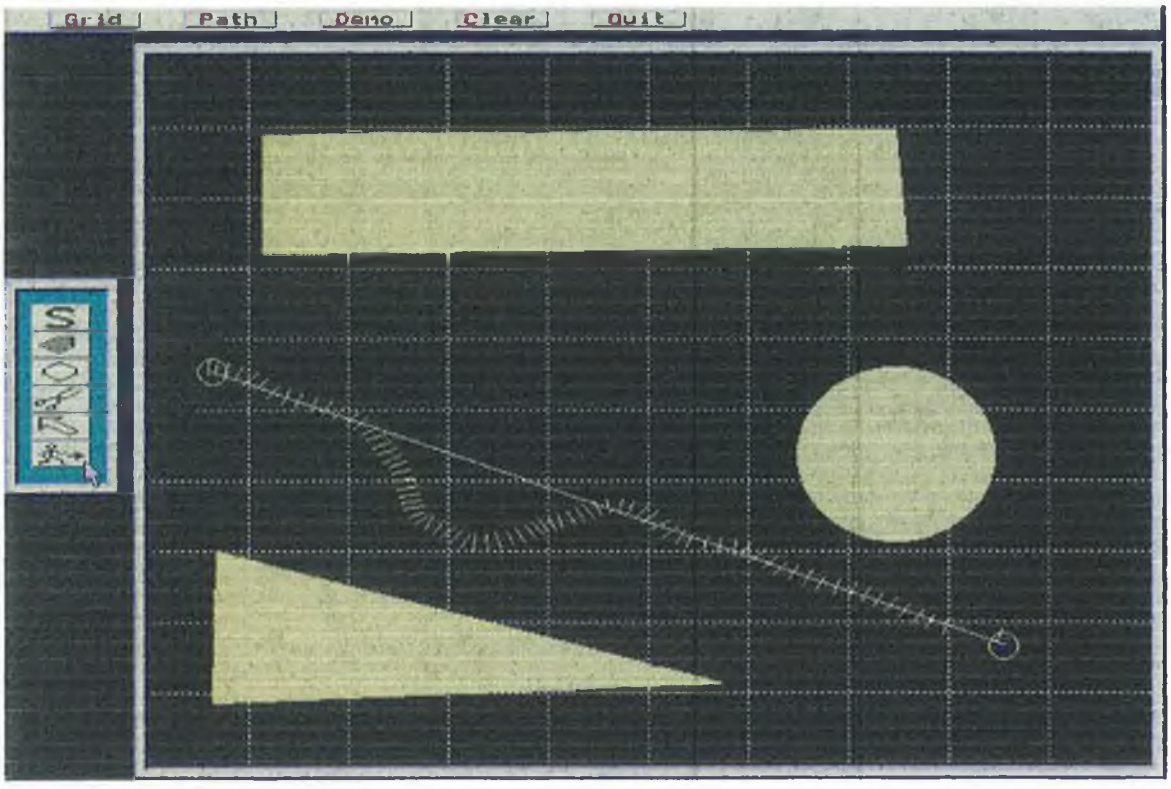


Figure 27: Robot encounters an unmapped obstacle

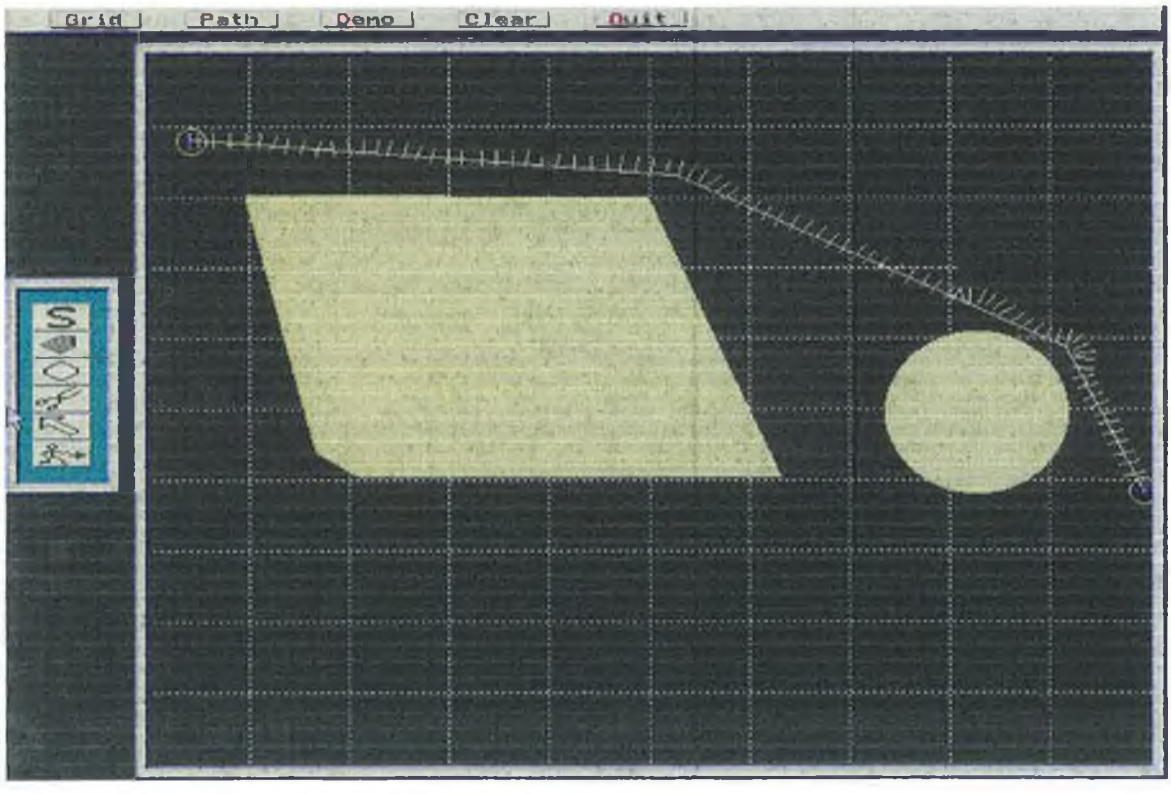


Figure 28: PathPlanner and Fuzzy Controller in operation

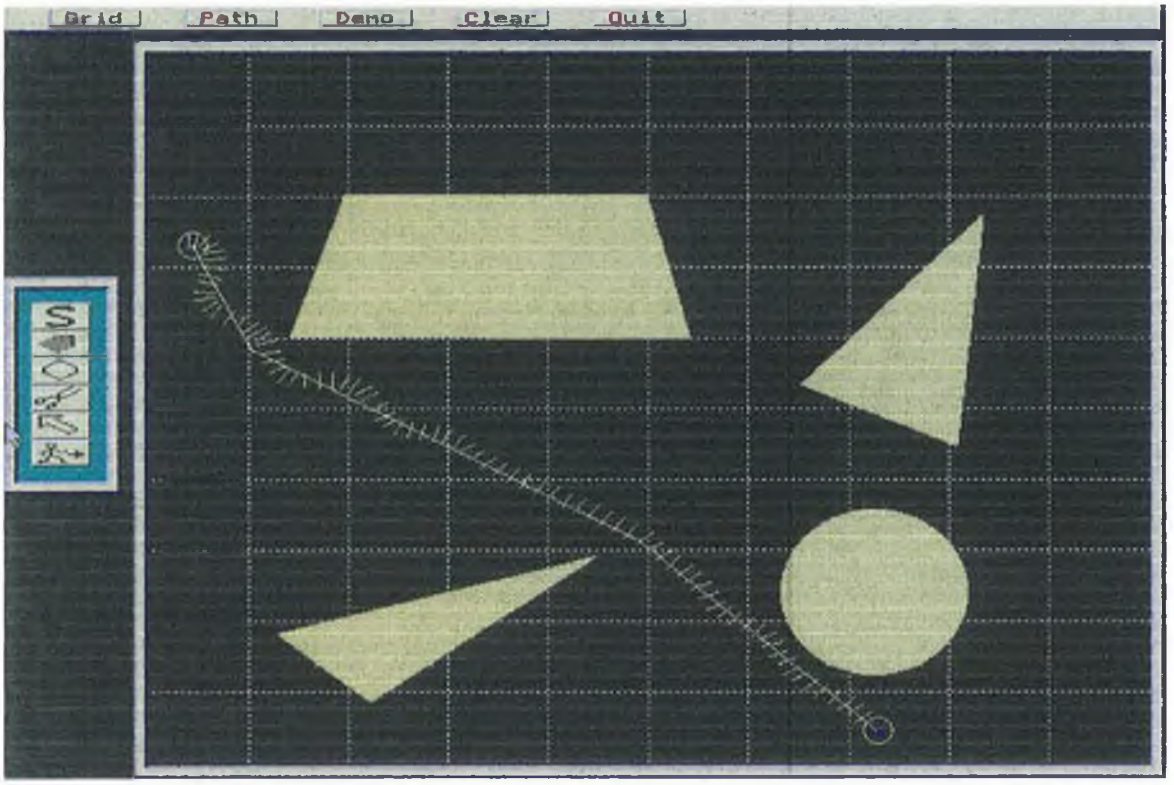


Figure 29: PathPlanner and Fuzzy Controller in operation

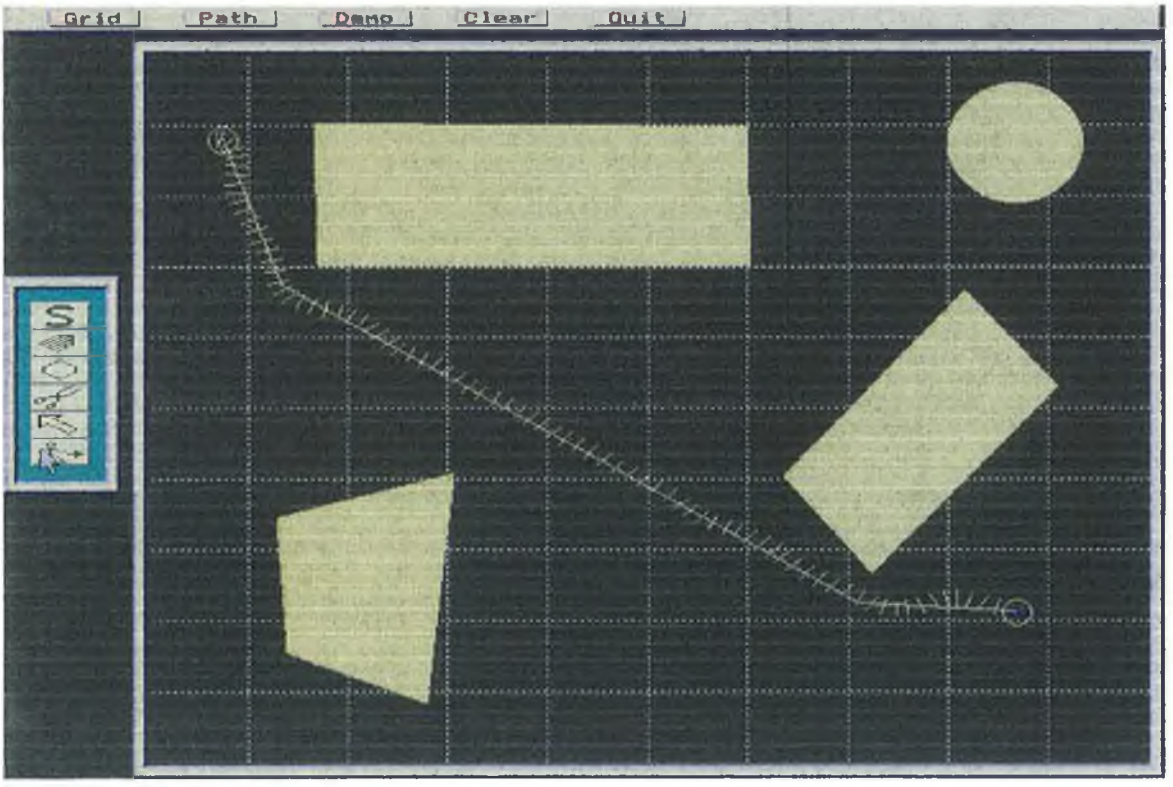


Figure 30: PathPlanner and Fuzzy Controller in operation

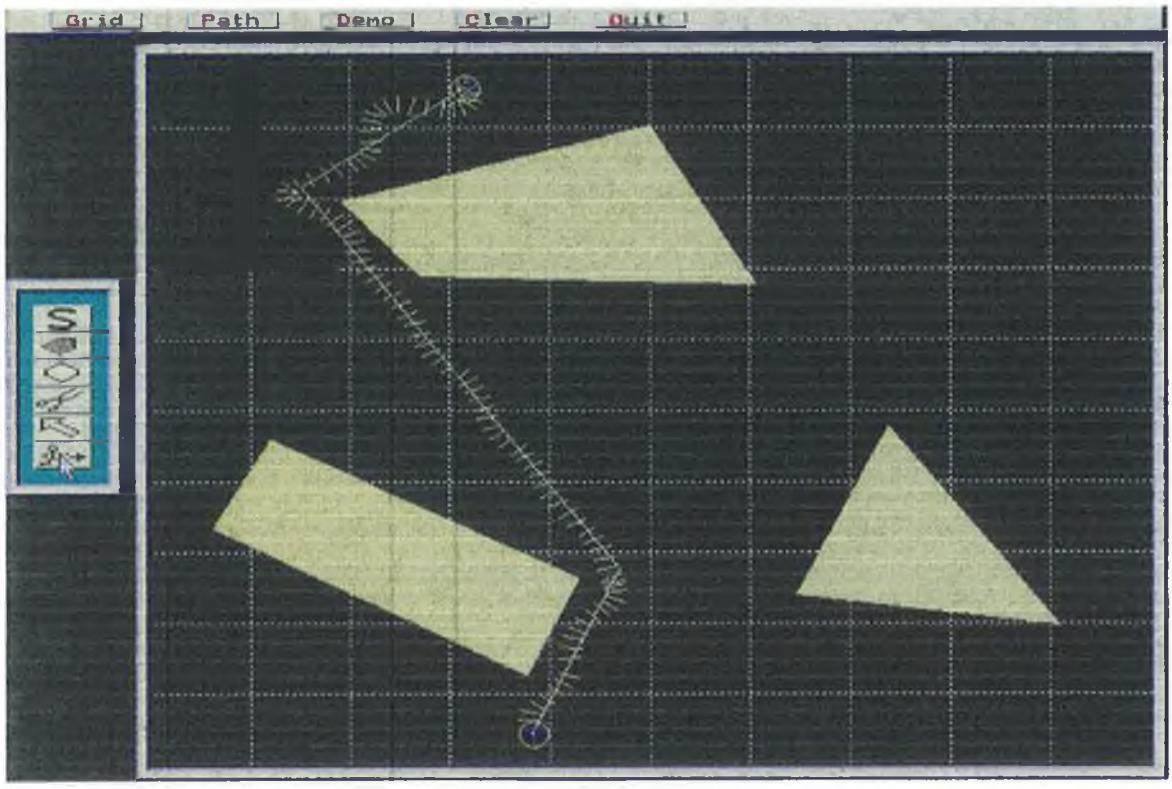


Figure 31: Initially the Robot was offset at 145° from the direction of the path

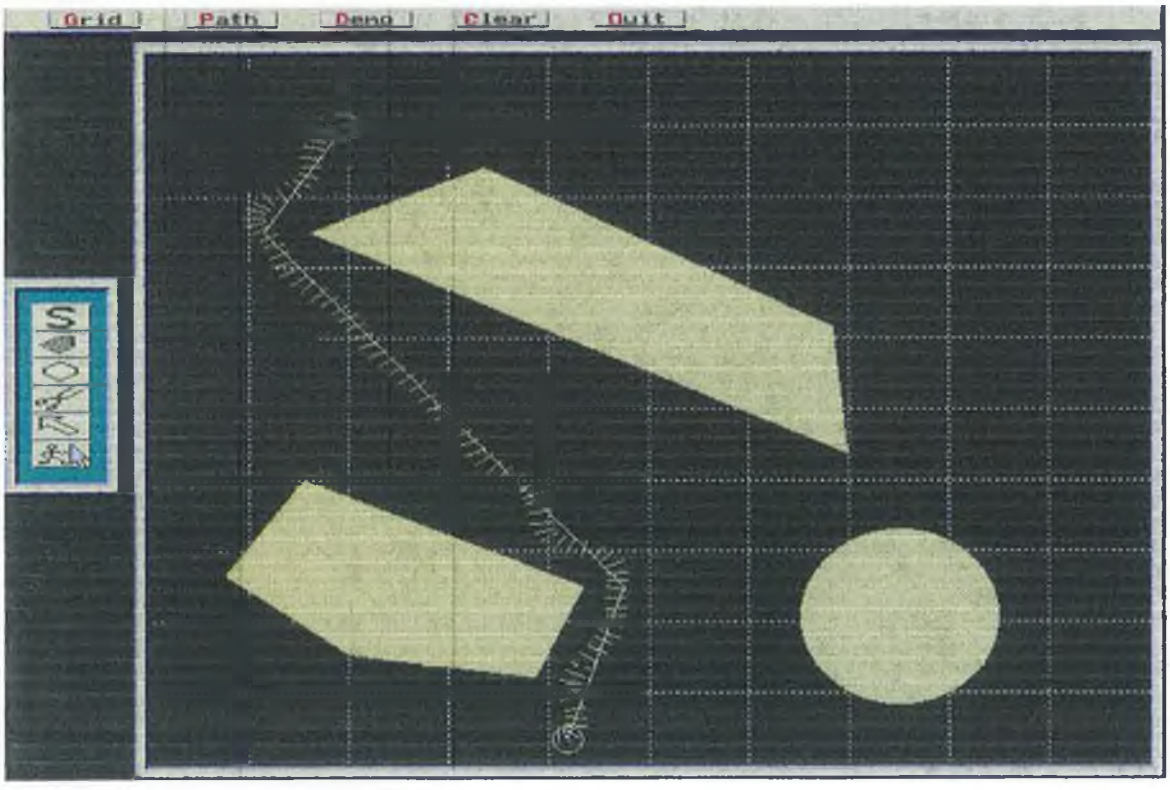


Figure 32: PathPlanner and Fuzzy Controller in operation

6. Conclusions

The path planner works very well in normal situations and is very quick at finding a solution even when there are a large number of obstacles in the room. This is because it need only concern itself with the obstacles that lie in the path of the robot and its destination. There are situations where the planner cannot be used. The algorithm breaks down when used in a maze and it is not possible to extend its use to this area because the algorithm becomes meaningless when dealing with situations where there may only be one object consisting of a spiral.

Attempts were made to extend the use of this planner to situations where it would normally break down. Obstacles lying too close together, preventing safe passage of the robot were merged into one. During planning if the centroid of an obstacle was found to be on the path, the path was pushed away from the centroid in a direction chosen at random. However the algorithm would not be suitable for use in a commercial vehicle because of its inability to deal with the situations outlined earlier.

The robot would have greater autonomy if the planner was run on the robot. Though more modifications are also necessary. The robot should contain a map of the local environment that is constantly updated by the sensors so that the robot can correlate its actual position with the calculated position from the shaft encoders. It would do this by comparing the global map to the local map until an optimal match is found. Using this information the robot can correct any errors that occur from relying on the shaft encoders over long periods of time. This process must be run regularly in the robot. This procedure emphasises the importance of suitable sensors.

If ultrasonic sensors are used, an array of twelve to sixteen sensors is common on many systems. The sensors should be long range to give the robot adequate warning of some hindrance to its safe passage. The Tag robot had only three ultrasonic sensors with a maximum possible range of two and a half metres. The sensors had reduced accuracy when tuned to this maximum range. This is unsatisfactory to

maintain a knowledge of the robots local domain, as possible landmarks will most likely be out of range.

The large number of sensors required is twofold. First enough must be present to reduce the errors inherent from using ultrasonic sensors. Secondly a commercial robot will not have the time to travel a path at leisure. Reading huge quantities of information about the surrounding environment to reduce errors using one sonar sensor is a time consuming process. Information must be readily available through the use of a large number of sensors. This allows a robot quick access to information on the surrounding environment in all directions. Other systems use one ultrasonic sensor which is rotated to find information on the surrounding environment. The robot is stopped while this information is being scanned. This procedure is too slow and information is available in only one direction at any given time.

Acquiring information on the surrounding environment has also another purpose apart from obstacle avoidance and position estimation. If a corridor is blocked another route must be found if one is available. The fuzzy controller will fail to pass the obstacle because there is no way through, so the planner must be run again. Thus the global map should be updated with this new information about the environment so that it will not consider the present corridor as a possible route. It is therefore clear that a local map is necessary for effective autonomy and that the quantity and quality of sonic sensors - when used, is an important issue.

The fuzzy logic controller turned out to be easy to tune and worked extremely well. It converged quickly on the path and circumvented the need for developing detailed mathematical models and equations to control the robot. The controller is also easy to extend or modify for different situations. It was possible to design the controller in a modular manner. Modules that performed different functions (e.g. for obstacle avoidance) could be written individually without worrying about what another module was doing. Each module could then be weighted or chosen as was appropriate to be part of the final output. Choosing the rules for the controller turned out to be straightforward as it was possible to translate ones own understanding of

the problem into a list of rules covering possible situations. The rules were possible to design in a tabular manner which made the process of design a relatively quick exercise.

Chapter 6

Conclusions

1. Research Summary

Fuzzy logic proved to be an easy and effective method for constructing a controller for the pilot. It provided a means to circumvent the 'necessity' of deriving a mathematical model of the desired controller which can be time consuming and difficult. The controller was designed quickly and once the rules were established, tuning them to improve the results was the only part of the entire procedure which involved testing on a trial and error basis.

The path planner turned out to have a number of problems. It did not necessarily work where an object displayed a high degree of concavity. Warren's paper states that the algorithm will not work for maze type obstacles. This was one of the problems encountered and the reason it fails is due to the fact that such obstacles end up with the centroid being calculated to be outside of the obstacles' perimeter, thus rendering the algorithm useless. The other problem arose from the merging of two concave obstacles. This is necessary in cases where their close proximity would prevent the robot from passing between them. In such circumstances the merged object is not entirely concave and one can end up with the centroid again being calculated to be outside the obstacle. The planner was extremely quick when it did provide a solution but is not acceptable for a commercial application because of the potential of running into an infinite loop should it encounter a local minima. The algorithm is therefore limited in it's use because it is not possible to rectify the problem except perhaps detect when a minima has occurred. A great deal of time was spent trying to increase the capabilities of this algorithm and should further attempts be made towards increased functionality, the programmer runs the risk of losing the initial advantage of the algorithm which aims at a fast solution to the path planning problem.

The ultrasonic sensors used in this development had low range and poor resolution and it became apparent that the ultrasonics were more suitable as backup sensors

for the avoidance of obstacles rather than for use with position estimation. The infra-red sensors had a low range, designed only to compensate for the blind-zone of the sonar sensors. As such it was not possible to fuse information in a manner that would correct the sonar readings at longer ranges.

Any attempt at removing the erroneous data from a sensor map which is derived from sonar readings, involves alot of computation and the data will still be uncertain as it isn't possible to remove the possibility of specular reflections causing incorrect readings. Moreover there is only limited time available to take the readings as the robot negotiates it's way along the path and therefore a sparse map can result that requires intensive computation to reduce the noise. The use of a stereo vision system means a high degree of computationally expensive processing will be required to extract the required information. The hardware required for such a system is also more expensive than for other sensors. Lasers can provide sufficient information from the environment with very little pre processing required to obtain an estimate of the position. The problem with these sensors is that for long distances a high power laser would be needed. High powered laser sensors are very dangerous to a persons eyes with the glare from such a sensor being enough to cause serious damage. This limits their use to fairly small distances. It would therefore seem that a reasonable approach for acquiring information from the environment is to use an array of twelve or more high quality ultrasonic sensors combined with infra-red sensors of similar range. The fusion of information between the two helps complement the weaknesses of both.

The control used should be hierarchical, distributed or perhaps a combination of the two depending on the nature of the project and the long and short term goals to be achieved. Subsumption architecture was used by Brooks to redefine the approach taken to autonomous vehicles which was subsequently used by Connell[37] and Stoney et al[14]. Connells' implementation was very simplistic however, demonstrating it's ability to collect cans. His implementation used path remembering to return to the base which was far too simplistic to be suitable for anything other than a demonstration. Stoney's attempt was more convincing. There

has been further development in this area with subsumption being used to plan paths. However this method inevitably falls into the same trap as the potential fields approach when no global search method is used for path planning, resulting in the robot travelling down box canyons in an effort to find it's destination.

2. Conclusions

Overall it became clear that fuzzy logic was useful where the control of robots is concerned. Designing a rulebase was relatively straightforward and proved even easier to tune. The fuzzy controller was successful at following a defined path and at avoiding obstacles. It enabled the robot to make it successfully back to the path after it had been forced off it by some obstacle unknown to the planner.

The sensors on this robot were not of the quality necessary to accurately get an idea of what type of obstacle was surrounding it. The planner would not have the ability to readily make use of this information because it would need to be able to deal with uncertain data. Because of the drawbacks associated with the planner, it is not suitable for use in a commercial environment. It is difficult to find a medium between a planner that works well with uncertain data and one that works efficiently, because inevitably if the inputs are uncertain, there is the need for vast quantities of data to reduce the uncertainty of the surrounding environment. It is better to combine different types of sensors that can remove a good deal of the uncertainty which means that there is less need for intensive processing.

The fuzzy controller was adept at utilising uncertain data to avoid obstacles but there is room for further research into a planner that would fit into this implementation more effectively. The planner should be able to integrate new data into a map of the environment should the need for re-planning occur. It could also be used to facilitate the robot in determining its whereabouts within the environment, with a view to correcting any drift that occurs with the dead reckoning system.

Bibliography

[1] The Huntington Group Technical Report; IEEE Control Systems '91

[2] Fuzzy Logic in C; Greg Viot, Dr. Dobb's Journal, February '93

[3] Fuzzy Logic and it's applications; Dr Jun Yan, D.C.U., Computer Applications, '92

[4] Fuzzy Logic; D P Burton & G M Lyons, University of Limerick, 17th December '93

[5] Intelligent identification and control for autonomous guided vehicles using adaptive fuzzy-based algorithms; C. J. Harris & C. G. Moore, Eng. Appli. of AI, Vol. 2, December '89

[6] Neural Networks & Fuzzy Systems; Bart Kosko, Prentice Hall '92

[7] Automated calibration of a fuzzy logic controller using a cell state space algorithm; Samuel M. Smith & David J. Comer, IEEE Control Systems '91

[8] A practical guide to tune fuzzy controllers; Li Zheng, IEEE '92

[9] Fuzzy logic strategies to control an autonomous mobile robot; Ricardo García Rosa & María C. García-alegre, Cybernetics and systems: An International Journal '90

[10] Comparison of fuzzy and neural truck backer-upper control system; Seong-Gon & Bart Kosko, University of Southern California '92

[11] Fuzzy rule-based motion controller for an autonomous mobile robot; M. Kemal Ciliz & Can Isik, Robotica '89, vol. 7, pp 37-42

[12] Highly reliable automated guided vehicle control with fuzzy inference; Tsuneo Tsukagoshi, Hiroo Wakaumi, NEC Res. & Develop., Vol. 32, No. 2, April '91

[13] A Robust Layered Control System for a Mobile Robot; R. Brooks, IEEE Journal of Robotics and Automation, RA-2, April '86, pp. 14-23

[14] An Autonomous Mobile Robot Control System using Subsumption Architecture; G. Stoney, C. R. Drone, J. Leaney, B. Walker, D. Semler, School of Electrical Engineering, University of Technology, Sydney, Australia, '91

[15] Solving the find-path problem by good representation of free space; IEEE transactions on System, Man, and Cybern., SMC Vol. 13, No. 3, '83

[16] Path planning and Environment Learning in a Mobile Robot System; Raja Chatila, Proc. of the European Conference on AI, Orlay, France, July '82

[17] Navigation of a car-like mobile robot using a decomposition of the environment in convex cells; Hubert A. Vasseur, François G. Pin, Jack R. Taylor, IEEE Proc. of the International Conference on Robotics and Automation, Apr. '91

[18] Planning a smooth path for autonomous vehicles using primary convex regions; Akira Hayashi, Benjamin J. Kuipers, Dept. of Computer Sciences, The University of Texas at Austin

[19] Natural Decomposition of free space for path planning; R. Brooks, J. Zamiska, D. T. Kuan, IEEE '85

[20] Automatic planning of Manipulator Transfer Movements; Thomas Lozano-Perez, IEEE Trans. on Sys., Man and Cybern., Vol. SMC-11, No. 10, Oct. '81

[21] Robot Rover Visual Navigation; H. Moravec, UMI Research Press, Ann Arbor, Michigan '81

[22] Mobile Robots; G. Giralt, Robotics and AI, NATO ASI Series '84, Vol. F11, pp 365-393

[23] Navigation for an intelligent mobile robot; J. Crowley, IEEE Conf. on Applications of AI, Dec. '84

[24] Potential Field Methods and their inherent limitations for mobile robot navigation; Yoram Koren, Johann Borenstein, International Conference on Robotics and Automation, IEEE Proceedings April '91

[25] The Application of Continuum Methods to Path Planning; C. Buckley, Stanford Univ., Dept. of Mechanical Engineering, Ph. D. thesis '85

[26] Integrated path planning and dynamic steering control for autonomous vehicles; B. H. Krogh, C. Thorpe, Proceedings of the IEEE International Conference on Robotics and Automation, CA, Apr. 7-10, '86, pp 1664-1669

[27] Planning Strategic paths through variable terrain data; J. Mitchell, D. Keirsey, Proceedings of the SPIE Conference on Applications of AI, VA, May '84

[28] Path Relaxation: Path Planning for a Mobile Robot; C. Thorpe, National Conference on AI, Austin Texas, Aug. 8, '84

[29] Multi-Goal Real-Time Global Path Planning for an Autonomous Land Vehicle using a high speed graph search processor; A. M. Parodi, IEEE '85

[30] Visual algorithms for Autonomous Navigation; Fred P. Andresen, Larry S. Davis, Roger D. Eastman, Subbarao Kambhampati, IEEE '85, Computer Vision Lab., Centre for Automation Research, University of Maryland

[31] Generalising the Hough Transform to detect arbitrary shapes; D. Ballard, Pattern Recognition, 13, 111-122, '81

[32] Parallel Analogue Computation for Real-Time Path Planning; Lionel Tarassenko, Gillian Marshall, Alan Murray, Felipe Gomez-Castaneda, VLSI for Artificial Intelligence and Neural Networks, New York, '91

[33] A Hierarchical-Orthogonal-Space approach to collision-free path planning; E. K. Wong, K. S. Fu, IEEE '85

[34] Octree Generation from silhouette views of an object; Jack Veenstra, Narendra Ahuja, IEEE Journal of Robotics and Automation, Vol. RA-2, No. 1, Mar. '86

[35] Dynamic world modelling for an intelligent mobile robot using a rotating ultra-sonic ranging device; James L. Crowley, IEEE '85, The Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania

[36] Blanche- An experiment in Guidance and Navigation of an Autonomous Robot Vehicle; Ingemar J. Cox, IEEE Transactions on Robotics and Automation, Vol. 7, No. 2, Apr. '91

[37] Navigation by path remembering; Jonathan Connell, SPIE Vol. 1007 Mobile Robots III '88

[38] Simulation of Path Planning for a system with Vision and Map Updating; E. Koch, C. Yeh, G. Hillel, A Meystel, C. Isik, IEEE '85

[39] Collision Avoidance by a Modified Least-Mean-Square-Error Classification Scheme for Indoor Autonomous Land Vehicle Navigation; Ling-Ling Wang, Journal of Robotic Systems '91

[40] Mobile Robot Path Planning in Dynamic Environments; J. P. H. Steele, G. P. Starr, Proc. IEEE Int. Conf. Systems, Man, Cybernet., Beijing and Shenyang, China, '88

[41] Kinematic modeling for feedback control of an omnidirectional wheeled mobile robot; Patrick F. Muir, Charles P. Neuman, '87 IEEE, Dept. of Electrical and Computer Eng., The Robotics Inst., Carnegie-Mellon University, Pittsburgh

[42] A Perspective on range finding techniques for computer vision; R. A. Jarvis, IEEE Trans. Pattern Anal. Machine Intell., Vol. PAMI-5, Mar. '83

[43] Ultrasound in Medical diagnosis; G. B. Devey, P. N. T. Wells, Sci. Amer., Vol. 238, May '78

[44] Ultrasonic range finders; Polaroid Corporation '82

[45] An integrated navigation and motion control system for autonomous multisensory robots; G. Giralt, R. Chatila, M. Vaisset, 1st International Symp. Robotics Research, Cambridge, '84

[46] Locating object surfaces with an ultrasonic rangefinder; M. K. Brown, Proc. IEEE Int. Conf. Robotics and Automation, St. Louis, MO, Mar. '85

[47] Combining Sonar and infrared sensors for mobile robot Navigation; A M. Flynn, The International Journal of Robotics Research, Vol. 7, No. 6, Dec. '88

[48] Survey of Automated Guided vehicles in Japanese factory; T. Tsumura, Proc. IEEE Int. Conf. Robotics and Automation, '86, pp. 1329-1334

[49] Experiments and thoughts on Visual Navigation; C. Thorpe, L. Matthies, H. Moravec, Carnegie-Mellon Univ., IEEE '85

[50] Accurate Position Estimation for an Autonomous Mobile Robot fusing shaft encoder values and laser range data; Thomas Edlinger, Ewald von Puttkamer, Rainer Trieb, Computer Science Dept., University of Kaiserslautern, Erwin-Schroedinger-Straβe, Germany

[51] An Autonomous Guidance system for a self-controlled vehicle; Hideo Arakawa, Takero Hongo, Gunji Sugimoto, Koichi Tange, Yuzo Yammota, 8th Jul. '86, IEEE log No. 8611671

[52] A spatial representation system for mobile robots; C. Miller, Proc. IEEE Int. Conf. Robotics and Automation, St. Louis, MO, Mar. '85

[53] Mobile robot localisation using sonar; AI Lab, Mass. Inst. Technol., AI-M-826, Jan '85

[54] Towards Autonomous Vehicles; H. P. Moravec, Robotics Research Review, The Robotics Inst., Carnegie-Mellon University, Pittsburgh, '85

[55] Outdoor scene Analysis using range data; M. Herbert, T. Kanade, Proc. IEEE Int. Conf. Robotics and Automation, San Francisco, CA, Apr. 7-10, '86

[56] Integration of Sonar and Stereo range data using a grid-based representation; A. Elfes, L. H. Matthies, IEEE International Conference on Robotics and Automation, pp. 727-733, '88

[57] A Non-Deterministic approach to 3-D Modelling Underwater; W. K. Stewart, 5th Symposium on unmanned untethered Submersible Technology, Univ. of New Hampshire, June '87

[58] Obstacle avoidance using 1-D Stereo Vision; B. Serey, L. H. Matthies, CMU Robotics Institute Report, Nov. '86

[59] Sensor Integration for Robot Navigation: Combining Sonar and Stereo Range Data in a Grid-Based Representation; IEEE Decision and Control Conference, Los Angeles, CA, Dec. '87

[60] The Curvature Primal Sketch; J. M. Brady, MIT AI Lab Memo 758, '84

[61] Experience with Visual Robot Navigation; L. H. Matthies, C. E. Thorpe, Proc. IEEE Oceans, Washington, D.C., August '84

[62] The Stanford Cart and the CMU Rover; Proc. IEEE, Vol. 71, Jul. '83

[63] The Technology Applications Group; Bolams Mill, Dispensary St., Alnwick, Northumberland

[64] A Vector Based Approach to Path Planning; Charles W. Warren, IEEE Proceedings, April '91

[65] Solving the Two-Dimensional Findpath Problem using a Line-Triangle Representation of the Robot; B. K. Bhattacharya, J. Zorbas, Journal of Algorithms 9, pp 449-469, '88

[66] Computer Control of Wireless links; Jan Axelson, Microcomputer journal, March/April '94

[67] Ramsey Electronics; 793 Canning Pkwy, Victor, NY 14564

[68] A Study of Mobile Robot Motion Planning; Bang Wang, 1994

[69] The Development of a Simulator for an Autonomous Mobile Robot; Keith Daly, 1995

[70] Calculus A Complete Course, 3rd edition; Robert A. Adams

[71] Modern Calculus and Analytic Geometry; Richard A. Silverman

[72] Control Systems Engineering & Design; S. Thompson

Appendix A

Communication Link

The wireless link sends data at 1,200 bits per second (bps). The transmitter's range is up to one quarter of a mile. On the transmit side, a modem chip receives the computer's digital data and converts it into audio tones. The transmitter is a low-cost stereo broadcaster, built from a kit, that frequency-modulates the tones onto a carrier frequency in the FM broadcast band. On the receive side, an ordinary FM radio detects the carrier, extracts the audio tones and makes them available at an earphone jack. A second modem chip translates the tones back into digital data for the receiving end. With two sets of transmitters and receivers, each tuned to a different frequency, it was possible to send and receive at the same time, for full-duplex communications. One advantage of using FM is that the FM receivers tend to automatically select the strongest signal and reject weaker interfering signals.

Theory of Operation

Shown in Fig. 1 is a block diagram of one end of the link. The other end contains identical components but is tuned to complementary transmitting and receiving frequencies. The transmitting computer sends the data to its' serial port. The PC can be any computer that has an asynchronous RS-232 serial port that transmits at 300 or 1,200 (bps). A MAX232 chip translates the RS-232 potentials into 5-volt logic levels. An AMD 7910 modem chip encodes the voltages as sine waves, with different frequencies that represent 1s and 0s.

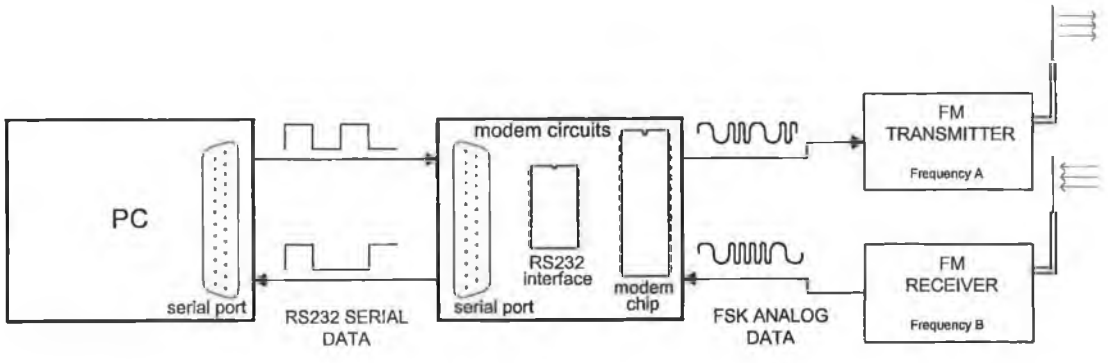


Figure 1: Block diagram of one end of the wireless link.

The FM transmitter sends the sine waves to a receiver tuned to the transmitter's carrier frequency. The receiving modem converts the tones back into digital data, and a second MAX232 chip converts the data to RS-232 voltages for the receiving end. The conversion of digital data into audio tones before transmission called frequency shift keying (fsk), is an important feature of this transmitter. The reason is that the stereo transmitter, and the FM broadcast band it uses, are intended for transmitting audio frequencies which consist of sine waves at 15 KHz or less. The modem's output falls within this band.

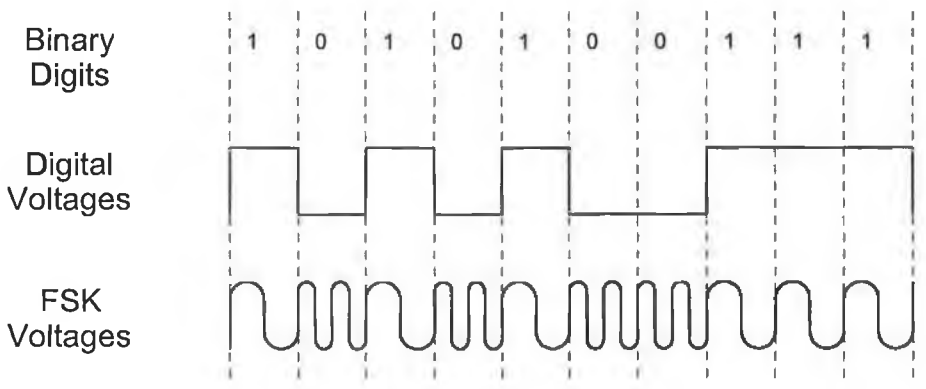


Figure 2: Frequency-shift keying (FSK) encodes binary data as frequencies, with different frequencies representing 0 and 1.

In contrast, a 1200 bps digital transmission consists of square pulses that are made up of sine waves of many different frequencies (the Fourier series shows the different components), many much greater in frequency than the 1,200 bps transmission rate. Fig. 2 shows how fsk works when converting digital data to sinusoids or vice versa. The modem chips also provide a means of detecting the transmitted tones and reject noise and interference that the receiver may pick up.

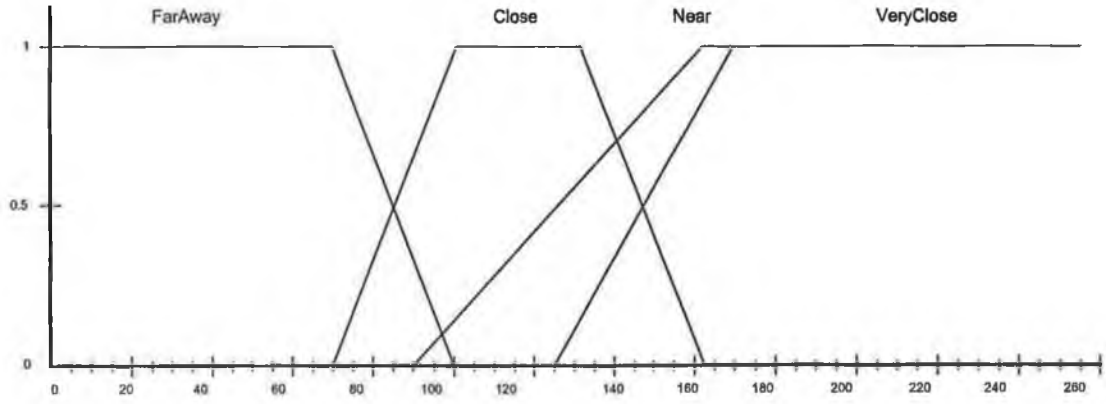
The transmitter uses a special BA1404 stereo transmitter chip, that contains a stereo modulator, FM modulator and r-f amplifier. The transmitter used [66][67] is one of many low power FM transmitters on the market. But as opposed to some others that were tried during the course of this thesis, this one was able so stay on frequency without drifting after a few minutes. A walkman, used as a receiver is tuned into the appropriate frequency and an Am7910 is used for the modem chip. For transmitting at 88 to 108 MHz, a ¼-wave antenna would range from 27 to 34

inches in length. In this case an antenna of one meter was used. Transmission was at 1,200 bps. The carrier frequencies were at 88 Mhz and 101 Mhz. There were a number of other possible frequencies that could be used each one having a 200 Khz bandwidth and 25 Khz guard bands.

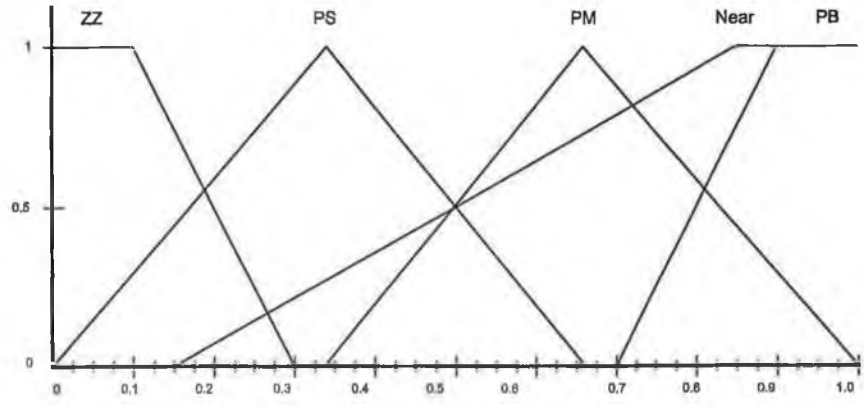
Appendix B

Fuzzy Controller Membership Functions

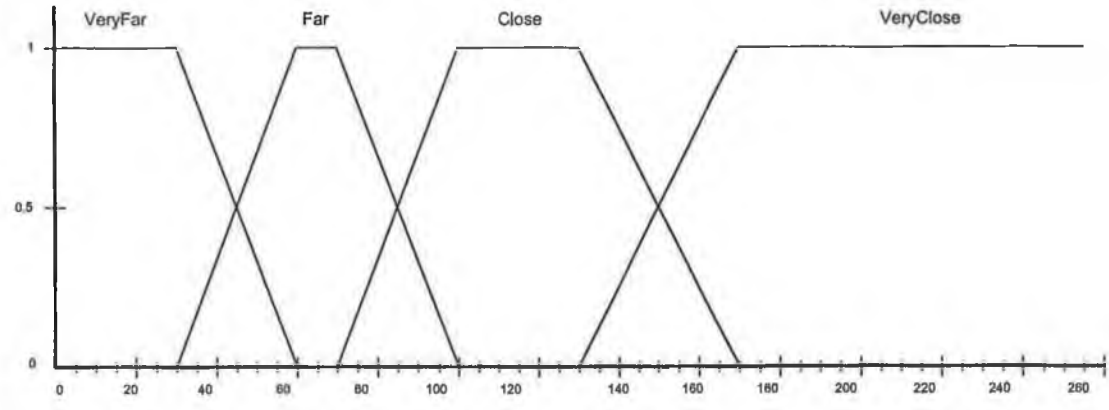
Dr Df Dl

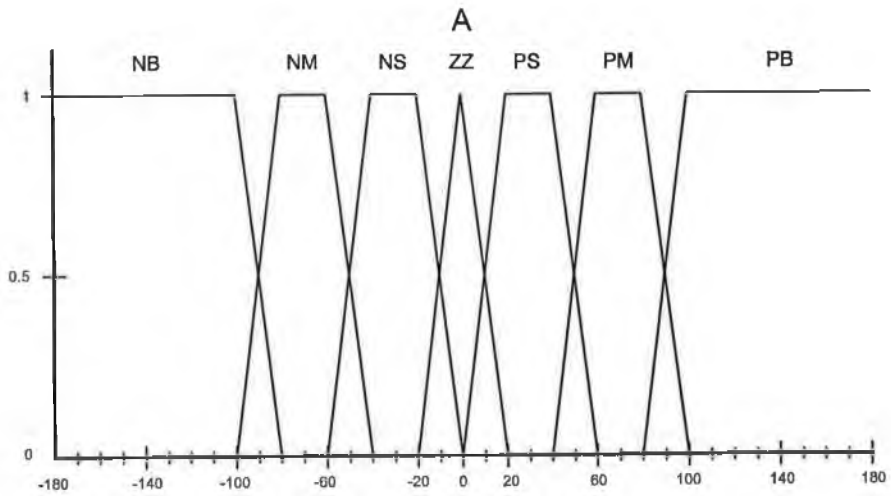


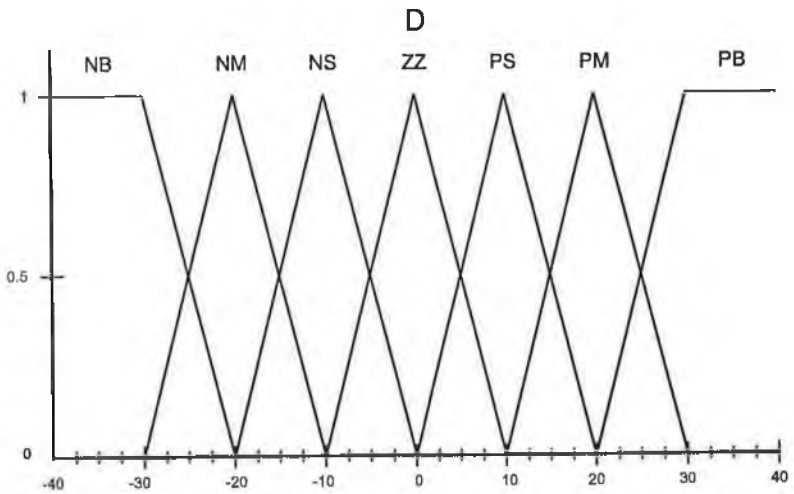
K

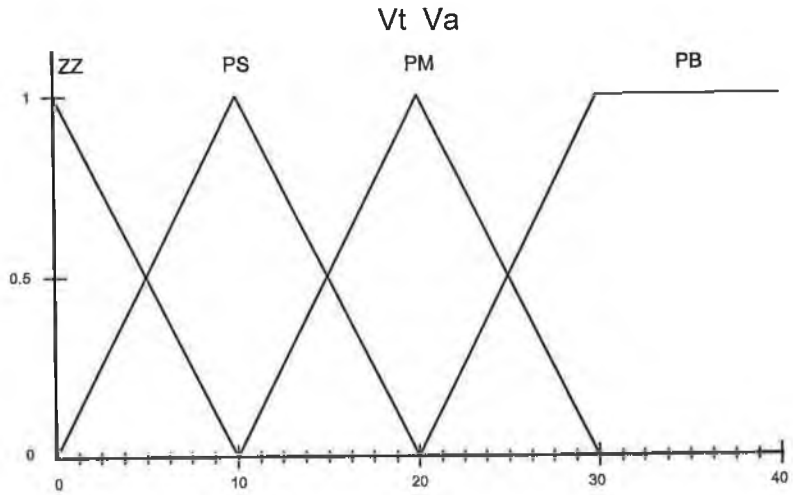


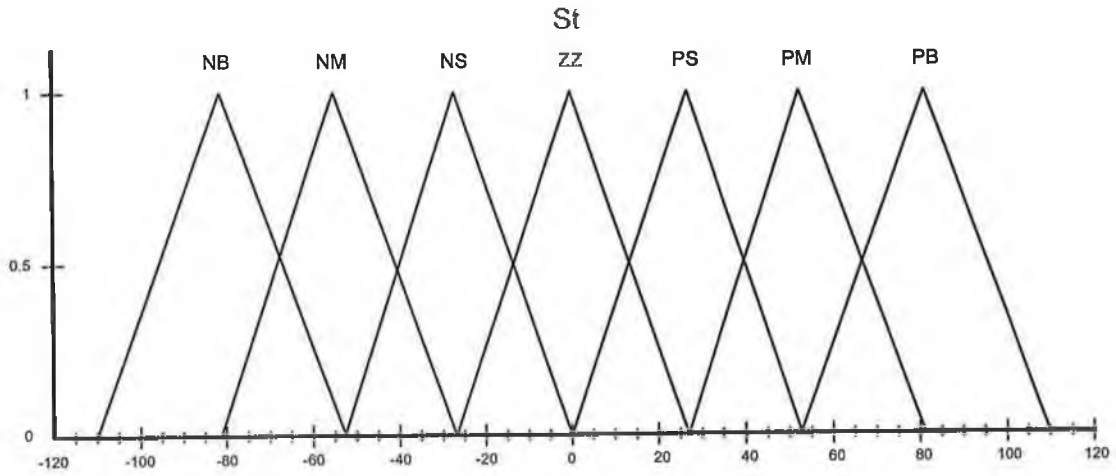
Df2

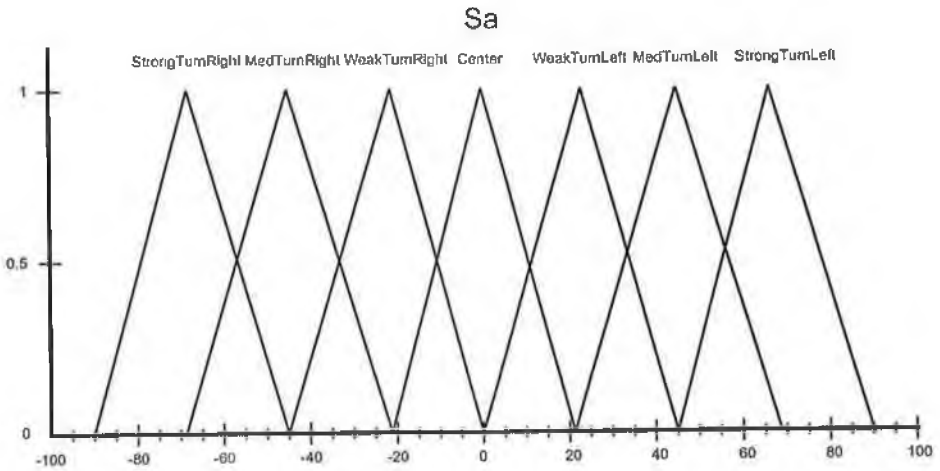



A
NB
NM
NS
ZZ
PS
PM
PB
1
0.5
0
-180
-140
-100
-60
-20
0
20
60
100
140
180


D
NB
NM
NS
ZZ
PS
PM
PB
1
0.5
0
-40
-30
-20
-10
0
10
20
30
40


Vt Va
ZZ
PS
PM
PB
1
0.5
0
0
10
20
30
40


St
NB
NM
NS
ZZ
PS
PM
PB
1
0.5
0
-120
-100
-80
-60
-40
-20
0
20
40
60
80
100
120


Sa
StrongTurnRight
MedTurnRight
WeakTurnRight
Center
WeakTurnLeft
MedTurnLeft
StrongTurnLeft
1
0.5
0
-100
-80
-60
-40
-20
0
20
40
60
80
100

Appendix C

Software for the Planner, Navigator and Communication link

```
/*****************************************************************/
/*                                                               */
/*              HEADER FILES FOR ROBOT SOFTWARE                  */
/*                                                               */
/*****************************************************************/



/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        Header          : d6int.h
        Function:  The addresses for the configuration of serial port
                        (muart) of the robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/

#ifndef D6INT

#define D6INT


/* 8626 MUART registers */
#define MUART                   0
#define MUART_COMM1             (MUART * 2)
#define MUART_COMM2             ((MUART + 1) * 2)
#define MUART_COMM3             ((MUART + 2) * 2)
#define MUART_MODE              ((MUART + 3) * 2)
#define MUART_P1_CTRL           ((MUART + 4) * 2)
#define MUART_INT_ENABLE        ((MUART + 5) * 2)
#define MUART_SET_INT           MUART_INT_ENABLE
#define MUART_RESET_INT         ((MUART + 6) * 2)
#define MUART_COMPORT           ((MUART + 7) * 2)
#define MUART_P1                ((MUART + 8) * 2)
#define MUART_P2                ((MUART + 9) * 2)
#define MUART_T1                ((MUART + 10) * 2)
#define MUART_T2                ((MUART + 11) * 2)
#define MUART_T3                ((MUART + 12) * 2)
#define MUART_T4                ((MUART + 13) * 2)
#define MUART_T5                ((MUART + 14) * 2)
#define MUART_STAT              ((MUART + 15) * 2)

#define comport                 MUART_COMPORT

#define TRUE                    1
#define FALSE                   0

#define tx_ints_on() outportb(MUART_INT_ENABLE,THREINT);

/* Two serial port macros */
#define xmit_ready()            (inportb(0x1e) & 0x20)
#define byte_ready()            (inportb(0x1e) & 0x40)

/* The 80188 PIC */
#define INT1                    0xff3a
#define EOI                     0xFF22

#define BAUD_96                 0x04
#define BAUD_12                 0x07
```

```
typedef struct _queue
{
  int count;
  int front;
  int rear;
  int maxsize;
  unsigned char *data;
} QUEUE;

#endif

/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header          : ada.h
        Function:  The addresses of the analog to digital and
                          digital to analog ports
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

#ifndef ADA

#define ADA
#define  BOARD1            0x40E0
#define  BOARD2            0x4FE0
#define  BOARD3            0x42C0

#define  DAC1_PORT         0x40E8  /* Board no + dac1 or dac2 */
#define  DAC2_PORT         0x40E9
#define  DAC3_PORT         0x4FE8
#define  DAC4_PORT         0x4FE9
#define  DAC5_PORT         BOARD3+0 /* 0x42C0 */
#define  DAC6_PORT         BOARD3+2 /* 0x42C2 */
#define  DAC7_PORT         BOARD3+4 /* 0x42C4 */
#define  DAC8_PORT         BOARD3+6 /* 0x42C6 */

#define  CHANNEL_1         0x08  /* Mux_no (1-2) + Channel_no (0-7) */
#define  CHANNEL_2         0x09
#define  CHANNEL_3         0x0A
#define  CHANNEL_4         0x0B
#define  CHANNEL_5         0x0C
#define  CHANNEL_6         0x0D
#define  CHANNEL_7         0x0E
#define  CHANNEL_8         0x0F
#define  CHANNEL_9         0x10
#define  CHANNEL_10        0x11
#define  CHANNEL_11        0x12
#define  CHANNEL_12        0x13
#define  CHANNEL_13        0x14
#define  CHANNEL_14        0x15
#define  CHANNEL_15        0x16
#define  CHANNEL_16        0x17

#define  MUX_OFF_ADR       0x000A
#define  EOC               0x80

#endif
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header      : sensors.h
        Function:  sets up the structure to contain all the sensor
                   information
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

#ifndef  SENSDATA

#define SENSDATA

struct sens{
        unsigned char Left_ultra;
        unsigned char Left_infra;
        unsigned char Left_touch;
        unsigned char Front_ultra;
        unsigned char Front_infra;
        unsigned char Front_touch;
        unsigned char Right_ultra;
        unsigned char Right_infra;
        unsigned char Right_touch;
};

struct motors {
        unsigned char rff;
        unsigned char rfr;
        unsigned char lff;
        unsigned char lfr;
};

#endif


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header      : packet.h
        Function:  sets up the variables used in setting up a packet
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

#ifndef  PACKETS
#define  PACKETS

#define  DOWNLOAD_PATH          0xF1
#define  DOWNLOAD_MAP           0xF2
#define  DOWNLOAD_MOTORS        0xF3
#define  RQST_SENSOR_INF        0x03
#define  ACKNOWLGE              0x01
#define  SEND_AGAIN             0x02
#define  PKT_SIZE               1024
#define  DLE                    0x10
#define  STX                    0x02
#define  RETRY                  4000
typedef struct _packet
{
        unsigned char type;
        unsigned int count;
        unsigned char data[PKT_SIZE];
        unsigned int checksum;
} PACKET;
#endif
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header          : pic.h
        Function:  sets up the addresses and values used to access and
                          program the PIC
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
#ifndef PIC

#define PIC


#define COUNT                 10
#define PIC_MASK              0xff28
#define TIMER0_CNT            0xFF50
#define TIMER0_CONTR          0xFF56
#define TIMER0_COUNT          0xFF52
#define TIMER1_CNT            0xFF58
#define TIMER1_CONTR          0xFF5E
#define TIMER1_COUNT          0xFF5A

#endif


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header          : fuzzy.h
        Function:  sets up the prototypes for the fuzzy control program
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

#include "tilcomp.h"

#ifndef  FUZZY_LOGIC

#define FUZZY_LOGIC

struct _UBY_b_point { UBYTE x; FUBYTE y; };

struct _SBY_b_point { SBYTE x; FUBYTE y; };

struct _SWO_b_point { SWORD x; FUBYTE y; };

void Controller (SWORD A, SBYTE D, UBYTE Df, UBYTE Dl, UBYTE Dr, FLOAT *K,
SBYTE *Sa, SBYTE *St, UBYTE *Va, UBYTE *Vt);

#endif
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Header          : proto.h
        Function:  function declarations
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
#include "fuzzy.h"
#include "tilcomp.h"
#include "sensors.h"

#ifndef PROTO
#define PROTO

double toradians(double d);
void initialisations(void);
int get_pkt(void);
void download_sensors(void);
void load_motors(void);
void test_controller(void);
void comm_init( unsigned char );
void load_path(void);
void load_map(void);
void init_arrays(void);
void init_ar(void);
void control_motors(void);
void test_controller(void);
int fndnearstseg(void);
void calc_motor_value( float S,unsigned int *lmotor,unsigned int *rmotor );
void updateposn(float *Xold, float *Yold);
void getangleA(SWORD *A, float Xold, float Yold);
void LeftorRight_disD(int *left, int *right, SBYTE *D );
void send_motors( struct motors  * motorvals );

int room_on(QUEUE *q);
int data_on(QUEUE *q);
static void put_on(QUEUE *, unsigned char );
static unsigned char get_from(QUEUE *);
void set_vector(unsigned int, void interrupt (*isr) () );
void comm_initl(unsigned char baud, int rxqsize, int txqsize);
int cinc(int x, int y);
int cincr(int x, int y,int incr);
static void interrupt tx_handler(void);
static void interrupt rx_handler();
void interrupt pic_ldist();
void interrupt pic_rdist();
void distance_ints(void);
static void put_on_pkt(unsigned char c);
void send(unsigned char type);
void loadrpkt(QUEUE *rxq);
int pktinteg(void);
void transmit(unsigned char );
int checkforpkt(QUEUE *rxq);
int headr(void);
static void pkt_initl(void);
static unsigned char read_from(QUEUE *q, int *front);

/* external read routine in sens.c and motors.c */
extern void read_sens(struct sens *);
extern void motor(unsigned int c,unsigned int dac_port);
#endif
```

```
/*************************************************************************/
/*                          The ROBOT SOFTWARE                           */
/*                                robot.c                                */
/*                This downloads the path + envirnment map from          */
/*                the PC to the robot and runs the fuzzy controller      */
/*************************************************************************/


#include <dos.h>
#include <stdlib.h>
//#include <conio.h>
//#include <alloc.h>
#include "d6int.h"
#include "pic.h"
#include "packet.h"
#include "ada.h"
#include "sensors.h"
#include "proto.h"
#include "fuzzy.h"
#include "tilcomp.h"


const unsigned THREINT    = 0x20;
const unsigned RXQ_SIZE = 1024;
const unsigned TXQ_SIZE = 1024;
const unsigned RX_INT     = 0x10;
const unsigned RX_INT_VECTOR = 0x44;
const unsigned TX_INT_VECTOR = 0x45;

static const unsigned MODE_8086 = 0x02;
static const unsigned FRQ_1KHz  = 0x01;

static const unsigned COMM3_SET = 0x80;
static const unsigned RX_ENABLE = 0x40;
static const unsigned INT_ACK   = 0x20;
static const unsigned NESTED_INT_ENABLE = 0x10;
static const unsigned MUART_MODIFICATION = MUART_STAT;

static QUEUE rxvq, *rxq = &rxvq;
static QUEUE txvq, *txq = &txvq;
static PACKET tpkt,rpkt;

volatile static unsigned int rightdist=0,leftdist=0;

#define MAXPATHLEN 200
#define X_ROW  0
#define Y_ROW  1
#define MAX_LEN  40
#define MAXOBJS  25

#define W 0.205
#define PI  3.14159
#define R_TO_D 57.29578
#define WheelRadius 0.015
#define NUMSEG      3.0

static struct motors motor_vals;
struct sens sensors;
```

```
static unsigned int ppath[MAXPATHLEN][2];

typedef struct obj
{
	int type;
	unsigned int numpoints;
	unsigned int points[2][MAX_LEN];
} OBJ;

OBJ map[MAXOBJS];

//extern void init_cios(void);
int numpoints=0;
float V=0.0;
float Xn=0.0,Yn=0.0,quita=0.0;
int cur=0,avoidance=0;
FLOAT K=0.0,K_OLD=0.0;




/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	main
	Function :	Initialise everything and constantly check for any
			requests coming from the PC
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void main(void)
{

    initialisations();
    // init_cios();
    while(1)
    {
        get_pkt(); /* listen for packets */
        run_controller();  /*will only run if the robot is following a path */
    }
}

void run_controller(void)
{
    Xn=ppath[cur][X_ROW];
    Yn=ppath[cur][Y_ROW];
    Xn=Xn/10.0; // convert to meters
    Yn=Yn/10.0;

    while (cur<numpoints-1)  /* While the robot hasnt reached its destination */
    {
        if ( sqrt( ((((float)Xn*10.0)-(float)ppath[cur+1][X_ROW])*(((float)Xn*10.0)-
                (float)ppath[cur+1][X_ROW])) + (((Yn*10.0)(float)ppath[cur+1][Y_ROW])*
                (((float)Yn*10.0)-(float)ppath[cur+1][Y_ROW])) ) < 2.0 )
            cur++;
        else if ( sqrt( ((((float)Xn*10.0)-(float)ppath[cur+1][X_ROW])*(((float)Xn*10.0)-
                        (float)ppath[cur+1][X_ROW])) + (((Yn*10.0) -
                        (float)ppath[cur+1][Y_ROW])*(((float)Yn*10.0)-
                        (float)ppath[cur+1][Y_ROW])) ) < \
```

```
                    sqrt( ((((float)Xn*10.0)-(float)ppath[cur][X_ROW])*(((float)Xn*10.0)-
                            (float)ppath[cur][X_ROW])) + (((Yn*10.0)-
                            (float)ppath[cur][Y_ROW]) * (((float)Yn*10.0)-
                            (float)ppath[cur][Y_ROW])) ))
            {
                  if ( sqrt( ((((float)Xn*10.0)-(float)ppath[cur][X_ROW])*(((float)Xn*10.0)-
                             (float)ppath[cur][X_ROW])) + (((Yn*10.0)-
                             (float)ppath[cur][Y_ROW]) * (((float)Yn*10.0)-
                             (float)ppath[cur][Y_ROW])) ) > \
                     sqrt( ((((float)ppath[cur+1][X_ROW])-(float)ppath[cur][X_ROW])*
                           (((float)ppath[cur+1][X_ROW])-(float)ppath[cur][X_ROW]))+
                           (((ppath[cur+1][Y_ROW]) -(float)ppath[cur][Y_ROW])*
                           (((float)ppath[cur+1][Y_ROW])-(float)ppath[cur][Y_ROW])) ))
                       cur ++;
            }
            if ( cur >= (numpoints-1) )
                  break;


            if ( avoidance == 1 )
            {
                  cur = fndnearstseg(); // Find the nearest path segment to the current position
                  if (K<0.4 )
                        avoidance=0;
            }
            test_controller(); /* One cycle of fuzzy logic controller */
            get_pkt(); /* want to listen for packets when following a path */
            delay(200);
      }
      stop_motors();
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
      Routine :      test_controller
      Function :     Processes the input parameters for the fuzzy logic
                     controller calls the fuzzy controller and then processes
                     the outputs and sends them to the motors
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void test_controller(void)
{

      SBYTE Sa=(SBYTE)0;
      SBYTE St=(SBYTE)0;
      UBYTE Vt=(UBYTE)0;
      UBYTE Va=(UBYTE)0;

      SWORD A=0;
      SBYTE D=(SBYTE)0;

      float Xold=0.0,Yold=0.0,S=0.0;
      int left=0,right=0,flag=0;
      unsigned int lmotor=0,rmotor=0;
      static unsigned int rightultra=0,leftultra=0;
```

```
/*************************************************************/
/*                  process the input for the controller                    */
/*************************************************************/
        read_sens(&sensors); /* Get the info from the sensors */
        rightultra = (unsigned int)sensors.Right_ultra;
        leftultra  = (unsigned int)sensors.Left_ultra;
        K_OLD = K;

        updateposn( &Xold,&Yold );
        LeftorRight_disD( &left,&right,&D );
        getangleA( &A,Xold,Yold );

        if ( K > 0.6 )   // Check for avoidance
        {
                if( (A>40) || (A<-40) )
                avoidance = 1;
        }


/*************************************************************/
/*                          Call the fuzzy logic controller                 */
/*************************************************************/


        Controller (A, D, sensors.Front_ultra, sensors.Left_ultra, sensors.Right_ultra, \
                        &K, &Sa, &St, &Va, &Vt);


/*************************************************************/
/*                  process the output for the controller                   */
/*************************************************************/

        V = (1.0-K)*(float)Vt + K*(float)Va;
        S = (1.0-K)*(float)St + K*(float)Sa;

        calc_motor_value( S,&lmotor,&rmotor );

        motor_vals.rff=(unsigned char)rmotor;
        motor_vals.rfr=0x00; /* reverse right disabled when going forward */
        motor_vals.lff=(unsigned char)lmotor;
        motor_vals.lfr=0x00; /* reverse left disabled when going forward */
        send_motors(&motor_vals); /* Output results to motors */
/*************************************************************/
/*           end process the output for the controller                      */
/*************************************************************/

}

void calc_motor_value( float S,unsigned int *lmotr,unsigned int *rmotr )
{
        float m=0.0,sigm=0.0;
        unsigned int rmotor=0,lmotor=0;

        // convert outputs to motor values
        m = 30.0/53.0; // (0x55-0x20)(max velocity)/(active voltage range incl zero vel)
                        // incorporate velocity into o/p
        lmotor = V/m + 0x20;
        rmotor = V/m + 0x20;
```

```
        sigm = 48.0/85.0*S; // (0x55-0x25)(active voltage range)/85(active steering range)
        if( S > 0 )             // incorporate steering into o/p
        {
                rmotor = rmotor + sigm/2.0;
                lmotor = lmotor - sigm/2.0;
                if ( lmotor > 0x55 )
                {
                        rmotor = rmotor - (lmotor - 0x55);
                        lmotor = 0x55;
                }
                else if ( rmotor < 0x20 )
                {
                        lmotor = lmotor + (0x20 - rmotor);
                        rmotor = 0x20;
                }
        }
        else
        {
                rmotor = rmotor + sigm/2.0;
                lmotor = lmotor - sigm/2.0;
                if ( rmotor > 0x55 )
                {
                        lmotor = lmotor - (rmotor - 0x55);
                        rmotor = 0x55;
                }
                else if ( lmotor < 0x20 )
                {
                        rmotor = rmotor + (0x20 - lmotor);
                        lmotor = 0x20;
                }
        }
        *lmotr = lmotor;
        *rmotr = rmotor;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        Routine :       getangleA
        Function :      Get the angle A of the robot relativce to the path
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void getangleA(SWORD *A,float Xold, float Yold)
{
        float ang1=0.0,ang2=0.0;
        float num=0.0,den=1.0;

        // get ang1
        if( (Yn==Yold)&&(Xn==Xold) )
                ang1=0;
        else
        {
                num = Yn - Yold;
                den = Xn - Xold;
                ang1 = atan2(num,den);

        }
```

```
	// get ang2
	num = (float)ppath[cur+1][Y_ROW] - (float)ppath[cur][Y_ROW];
	den = (float)ppath[cur+1][X_ROW] - (float)ppath[cur][X_ROW];
	ang2 = atan2(num,den);

	ang1 = ang1*R_TO_D;
	ang2 = ang2*R_TO_D;
	if (ang2>90)
	{
		if( ang1<-70 )
		{
			//ang1 = 180;
			*A = -180 - ang1;
			return;
		}
	}
	else if (ang2 < -90)
	{
		if( ang1>70 )
		{
			//ang1 = -180;
			*A = 180 - ang1;
			return;
		}
	}
	*A = ang2 - ang1;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	get_pkt
	Function :	This function gets a packet sent from the PC

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
int get_pkt(void)
{
	static int flag=0;
	if(checkforpkt(rxq)==FALSE) /* Wait for a pkt */
	{
		flag++;
		if (flag==200)
			flag=0;
		return FALSE; /* No packet in waiting */
	}

	if(pktinteg()==TRUE)	/* Is it good */
	{
		switch(rpkt.type)
		{
			case DOWNLOAD_PATH:	send(ACKNOWLGE);
						load_path();
						return 1;
			case DOWNLOAD_MAP:	send(ACKNOWLGE);
						load_map();
						return 2;
			case DOWNLOAD_MOTORS: send(ACKNOWLGE);
					//can be used to stop the robot from the pc
```

```
                                        load_motors();
                                        return 3;
                    case RQST_SENSOR_INF:  download_sensors();
                                        return 4;
            }
    }
    else
    {
            rxq->count=rxq->front=rxq->rear=0;
    }
    return 7;
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
    Routine :       load_path
    Function :      This routine downloads the planned path from the
                    PC
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void load_path(void)
{
    unsigned int i,j=0;
    unsigned char lo_byte,hi_byte;

    for(i=0;i<rpkt.count;i++);
    {
            lo_byte=rpkt.data[i++];
            hi_byte=rpkt.data[i++];
            ppath[j][X_ROW]= hi_byte<<8 + lo_byte;
            lo_byte=rpkt.data[i++];
            hi_byte=rpkt.data[i];
            ppath[j][Y_ROW]= hi_byte<<8 + lo_byte;
            j++;
    }
    numpoints = j
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
    Routine :       load_map
    Function :      This function downloads the map of the environment
                    into an array of obstacles

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void load_map(void)
{
    unsigned int numobjs,type,numpoints;
    int i,j,obj,offset;

    numobjs=(unsigned int)((rpkt.data[1]<<8)+rpkt.data[0]);
    offset=2;

    for(obj=0;obj<numobjs;obj++)
    {
            type=(unsigned int)((rpkt.data[offset+1]<<8)+rpkt.data[offset]);
            offset+=2;
            numpoints=(unsigned int)((rpkt.data[offset+1]<<8)+rpkt.data[offset]);
```

```
			offset+=2;

			map[obj].type=type;
			map[obj].numpoints=numpoints;

			for(i=0;i<numpoints;i++)
			{
				map[obj].points[X_ROW][i]=(unsigned int)((rpkt.data[offset+1]<<8)+
			rpkt.data[offset]);
				offset+=2;
				map[obj].points[Y_ROW][i]=(unsigned int)((rpkt.data[offset+1]<<8)+
			rpkt.data[offset]);
				offset+=2;
			}
		}
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :		download_sensors
	Function :		The following functions download information to the pc
				Both sensor information and current position are
				transmitted

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void download_sensors(void)
{
	unsigned char lo_byte,hi_byte;

	read_sens(&sensors);
	// This puts sensors ultra-sonic and infra-red and touch on queue

	put_on_pkt(sensors.Left_ultra);
	put_on_pkt(sensors.Left_infra);
	put_on_pkt(sensors.Left_touch);
	put_on_pkt(sensors.Front_ultra);
	put_on_pkt(sensors.Front_infra);
	put_on_pkt(sensors.Front_touch);
	put_on_pkt(sensors.Right_ultra);
	put_on_pkt(sensors.Right_infra);
	put_on_pkt(sensors.Right_touch);


	lo_byte = Xn & 0xFF;
	hi_byte = (Xn >>8) & 0xFF;
	put_on_pkt(lo_byte);
	put_on_pkt(hi_byte);

	lo_byte = Yn & 0xFF;
	hi_byte = (Yn >>8) & 0xFF;
	put_on_pkt(lo_byte);
	put_on_pkt(hi_byte);

	send('d');

}
```

```
/*=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=
        Routinee:       load_motors
        Function :      This is used to stop the robot from the PC
.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=*/
void load_motors(void)
{
        int i;

        for(i=0;i<rpkt.count;i++)
        {
                motor((unsigned int)rpkt.data[i],DAC5_PORT);
                i++;
                motor((unsigned int)rpkt.data[i],DAC6_PORT);
                i++;
                motor((unsigned int)rpkt.data[i],DAC7_PORT);
                i++;
                motor((unsigned int)rpkt.data[i],DAC8_PORT);
        }
}


/*=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=
        Routinee:       load_motors
        Function :      This is used to stop the robot from the PC
.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=*/
void send_motors( struct motors  * motorvals )
{
        motor((unsigned int) motorvals->rff,DAC5_PORT);
        motor((unsigned int) motorvals->rfr,DAC6_PORT);
        motor((unsigned int) motorvals->lff,DAC7_PORT);
        motor((unsigned int) motorvals->lfr,DAC8_PORT);
}

double toradians(double d)
{
        return ((double)(d) * PI / 180.0);
}

/*=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=
        Routine :       updateposn
        Function :      As the robot moves the position must be updated
.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=*/

void updateposn(float *Xold, float *Yold)
{
        float right_dist=0.0,left_dist=0.0;
        float del_quita=0.0,sign=0.0,del_Dn=0.0;

        // update the position

        leftdist = leftdist + inport(TIMER0_CNT);
        rightdist = rightdist + inport(TIMER1_CNT);

        //  Clear the variables
        outport(TIMER0_CNT,0);
        outport(TIMER1_CNT,0);

        right_dist=(rightdist*WheelRadius*2.0*PI)/NUMSEG;
```

```
        left_dist=(leftdist*WheelRadius*2.0*PI)/NUMSEG;

        del_Dn = (right_dist + left_dist)/2.0;
        del_quita = ((-right_dist + left_dist)*180.0)/(W*PI);

        if(del_quita!=0)
        {
                *Xold = Xn;
                *Yold = Yn;
                Xn = Xn + ( ((sin(toradians(del_quita/2.0))/(toradians(del_quita/2.0))) * del_Dn)
                                * cos(toradians(quita+(del_quita/2.0))) );
                Yn = Yn + ( ((sin(toradians(del_quita/2.0))/(toradians(del_quita/2.0))) * del_Dn)
                                * sin(toradians(quita+(del_quita/2.0))) );
                quita = quita + del_quita;
        }
        else
        {
                *Xold = Xn;
                *Yold = Yn;
                Xn = Xn + (del_Dn*cos(toradians(quita)));
                Yn = Yn + (del_Dn*sin(toradians(quita)));
        }
        sign=quita>0 ? 1:-1; // keep quita in range -360 -> 360 for neatness
        if (quita<-360.0 || quita>360.0)
                quita = quita-sign*360.0;

        leftdist=rightdist=0;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       LeftorRight_disD
        Function :      Find D the distance from the path
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void LeftorRight_disD(int *left, int *right, SBYTE *D )
{
        float num=0.0,den=1.0,rdist=0.0,ldist=0.0,Dis=0.0;
        float Xr=0.0,Yr=0.0,Xl=0.0,Yl=0.0;
        float m=0.0,a=0.0,b=0.0,c=0.0,ang=0.0;

        // find whether robot is left or right of path

        /*--------------------------------*/
        if(ppath[cur+1][X_ROW]!=ppath[cur][X_ROW]) // if not 90 or -90
        {
                num  = ((float)ppath[cur+1][Y_ROW] - (float)ppath[cur][Y_ROW]);
                den  = ((float)ppath[cur+1][X_ROW] - (float)ppath[cur][X_ROW]);
                m    = num/den;
                a  = m;
                b  = -1;
                c  = (float)ppath[cur][Y_ROW] - m*(float)ppath[cur][X_ROW];
                c  = c/10.0;
                /*--------------------------------*/
                ang = atan2(num,den);

                Xr = Xn - ((W/2.0) * sin(ang));
                Xl = Xn + ((W/2.0) * sin(ang));
```

```
            Yr = Yn + ((W/2.0) * cos(ang));
            Yl = Yn - ((W/2.0) * cos(ang));

            num = (a*Xr) + (b*Yr) + c;
            num = (num > 0) ? num : -num;
            den = sqrt((a*a) + (b*b));
            rdist = num/den;

            num = (a*Xl) + (b*Yl) + c;
            num = (num > 0) ? num : -num;
            den = sqrt((a*a) + (b*b));
            ldist = num/den;
}
else if(ppath[cur+1][Y_ROW]>ppath[cur][Y_ROW]) // +90
{
            Xr = Xn - (W/2.0);
            Yr = Yn;
            Xl = Xn + (W/2.0);
            Yl = Yn;

            ldist = Xl - (ppath[cur][X_ROW]/10.0);
            rdist = Xr - (ppath[cur][X_ROW]/10.0);
            ldist = (ldist>0) ? ldist : -1.0*ldist;
            rdist = (rdist>0) ? rdist : -1.0*rdist;
}
else // -90
{
            Xr = Xn + (W/2.0);
            Yr = Yn;
            Xl = Xn - (W/2.0);
            Yl = Yn;

            ldist = Xl - (ppath[cur][X_ROW]/10.0);
            rdist = Xr - (ppath[cur][X_ROW]/10.0);
            ldist = (ldist>0) ? ldist : -1.0*ldist;
            rdist = (rdist>0) ? rdist : -1.0*rdist;
}

if ( rdist < ldist )
{
            *left = 1;
            *right = 0;
}
else if ( rdist >ldist )
{
            *left = 0;
            *right = 1;
}
else *left = *right = 0;


// Get D the distance from the path
if(ppath[cur+1][X_ROW]!=ppath[cur][X_ROW])
{
            num = (a*Xn) + (b*Yn) + c;
            num = (num > 0) ? num : -num;
            den = sqrt((a*a) + (b*b));
            Dis = num/den;
```

```
        }
        else
        {
                Dis = Xn - (ppath[cur][X_ROW]/10.0);
        }
        Dis = Dis * 100; // Convert to cm
        Dis = (Dis>0) ? Dis : -Dis; //Dis should be positive at this point

        if (Dis > 40) Dis = 40; //clip
        if (*right == 1) Dis = -Dis;
        *D = (SBYTE) Dis;

}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       fndnearstseg
        Function :      It checks to see if the robot should try to follow
                        the next path segment
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int fndnearstseg(void)
{
        int min = 1000,i,tmp=cur;
        float dis = 0.0;

        for ( i=cur;i<numpoints-1;i++)
        {
                dis = (float)sqrt( (((Xn*10.0)-(float)(ppath[i][X_ROW]))*((Xn*10.0)-
                        (float)(ppath[i][X_ROW]))) + (((Yn*10.0)-(float)(ppath[i][Y_ROW]))*
                        ((Yn*10.0)-(float)(ppath[i][Y_ROW]))) );
                if ( dis < min)
                {
                        min = dis;
                        tmp = i;
                }
        }
        return tmp;
}


/************************************************************************/
/*                                                                      */
/*              Interrupt handling routines for the robot               */
/*                                                                      */
/************************************************************************/


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       tx_handler
        Function :      Interrupt routine, called when the serial chip generates an
                        interrupt. This could be to signify that some data has arrived
                        or that the transmitter section is ready to transmit another
                        character. To find out which, read the interrupt identification
                        register (IIR).
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

```
static void interrupt tx_handler(void)
{
	unsigned char c;

	/* If the transmit register is empty */
	if (data_on(txq))  /*  and there is someat on the Q ... */
	{
		c = get_from(txq);
		/* Dont need to check if buffer is empty as it interrupts only when it is */
		outportb(MUART_COMPORT, c);  /* ... transmit it */
	}
	outportb(MUART_COMM3, 0x88);
			/* End of interrupt command for the MUART controller */
			/* Required for nested interrupt mode */


	outport(EOI, 0x8000);  /* Issue an end of interrupt command to the internal PIC */

}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	rx_handler
	Function :	The isr that is called when a character is received. The
			character is subsequently queued and it then tells the
			PIC that the interrupt is finished.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void interrupt rx_handler()
{
	asm pushf
	if (byte_ready())
		if (room_on(rxq))
			put_on(rxq, inportb(MUART_COMPORT));
	asm popf
	/* End of interrupt command for the MUART controller */
	outportb(MUART_COMM3, 0x88);  /* ... required for nested interrupt mode */

	/* Issue an end of interrupt command to the internal PIC */
	outport(EOI, 0x8000);
}

/* Status of queue */
int room_on(QUEUE *q)
{
	return q->count < q->maxsize ;
}

int data_on(QUEUE *q)
{
	return q->count;
}

/* Some queue handling functions */
static unsigned char get_from(QUEUE *q)
{
	unsigned char c;

	_disable();
```

```
        c = q->data[q->front];
        q->front = cinc(q->front, q->maxsize);
        q->count--;
        _enable();

        return c;
}

static unsigned char read_from(QUEUE *q, int *front)
{
        unsigned char c;

        _disable();
        if (*front==-10)
                *front=q->front;

        c = q->data[*front];
        *front = cinc(*front, q->maxsize);
        _enable();

        return c;
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       put_on
        Function :      Put a character on the interrupt queue once it has been
                        received
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void put_on(QUEUE *q, unsigned char c)
{
        _disable();
        q->data[q->rear] = c;
        q->rear = cinc(q->rear, q->maxsize);
        q->count++;
        _enable();
}

/* Some queue handling macros */
int cinc(int x, int y)
{
        return (x + 1) % y;
}

/* Some queue handling macros */
int cincr(int x, int y,int incr)
{
        return (x + incr) % y;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       comm_initl
        Function :      Responsible for calling all the initailisation routines
                        that set up the interrupt queues + take over specific
                        interrupt vectors
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

```
static void comm_initl(unsigned char baud, int rxqsize, int txqsize)
{
        comm_init(baud);  // Baud rate etc

        // Allocate space for buffers
        if (rxqsize)
        {
                rxq->count=rxq->front=rxq->rear=0;
                rxq->maxsize = rxqsize;
                rxq->data = (unsigned char *) malloc(rxq->maxsize);

                set_vector(RX_INT_VECTOR, rx_handler);

                /* Enable receiver interrupts */
                outportb(MUART_INT_ENABLE, (inportb(MUART_INT_ENABLE) |
                        RX_INT));
        }

        if (txqsize)
        {
                txq->count=txq->front=txq->rear=0;
                txq->maxsize = txqsize;
                txq->data = (unsigned char *) malloc(txq->maxsize);

                set_vector(TX_INT_VECTOR, tx_handler);

                outportb(MUART_INT_ENABLE, (inportb(MUART_INT_ENABLE) |
                        THREINT));
                 //tx_ints_on();
        }

        /* Program the int_1 line on the 80188 for cascade ... */
        /*  ... and level triggered mode ... */
        /* ... 'cos it's connected to the 8256 MUART */
        /* ... and unmask int 1 (see 80188 data sheet p.32, fig 29) */
        outport(INT1, 0x30);

        _enable();
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       set_vector
        Function :      Take over an interrupt vector by pointing it to an isr
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void set_vector(unsigned intnum, void interrupt (*isr)())
{
        char far *far *ivt = (void far *) 0;

        char far *far *vector = ivt + intnum;

        _disable();
        *vector = (char far *) isr;
        _enable();
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	comms_init
	Function :	Initialise the muart to the correct baud rate and to
			generate an interrupt when a character arrives so
			that the character will be queued in a buffer and not
			lost
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void comm_init(unsigned char baud)
{
	outportb(MUART_COMM1, MODE_8086 | FRQ_1KHz);
	outportb(MUART_COMM3, COMM3_SET | RX_ENABLE | INT_ACK |
		NESTED_INT_ENABLE);
	outportb(MUART_RESET_INT,0xEF);
	outportb(MUART_MODIFICATION, 0);

	outportb(MUART_COMM2,baud);
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	distance_ints
	Function :	Initialise the timer counters to generate interrupts
			every ten pulses. Also enables the PIC to recognise
			the interrupt requests coming from the timer counters

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void distance_ints(void)
{

	disable();
	set_vector(8,pic_ldist);	/* set interrupt vector address */
	outport(TIMER0_CONTR,0xE005);	/* continuous ints */
	outport(TIMER0_COUNT,0x0A); /* the count at which the timer interrupts */

	set_vector(18,pic_rdist);	/* as above .... */
	outport(TIMER1_CONTR,0xE005);
	outport(TIMER1_COUNT,0x0A);
	outportb(PIC_MASK,inport(PIC_MASK) & 0xfc);
				/* Enable timer0 & timer1 in pic mask reg */
	enable();

}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	pic_rdist
	Function :	Updates the distance the right track has moved.
			This happens once every ten pulses when the timer
			counter counts to ten

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void interrupt pic_rdist()
{
	rightdist = rightdist + COUNT;

	/* Issue an end of interrupt command */
	/* (if you leave out this the PIC won't interrupt again) */
	outport(EOI, 0x8000);	/* Tell the internal PIC that we're finished */
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	pic_ldist
	Function :	Updates the distance the left track has moved.
			This happens once every ten pulses when the timer
			counter counts to ten

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void interrupt pic_ldist()
{
	leftdist = leftdist + COUNT;

	/* Issue an end of interrupt command */
	/* (if you leave out this the PIC won't interrupt again) */
	outport(EOI, 0x8000);	/* Tell the internal PIC that we're finished */
}


/*********************************************************************/
/*                                                                   */
/*		Packet handling routines for the robot               */
/*                                                                   */
/*********************************************************************/
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine:	Initialise the receive + transmit packets
	Function :	finds the centroid of a given obstacle

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void pkt_initl(void)
{
	int i;
	tpkt.count=tpkt.checksum=0;
	rpkt.count=rpkt.checksum=0;

	for(i=0;i<PKT_SIZE;i++)
	{
		rpkt.data[i]=tpkt.data[i]=0;
	}
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	put_on_pkt
	Function :	Puts a character on the transmit packet to be transmitted

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void put_on_pkt(unsigned char c)
{
	tpkt.data[tpkt.count]=c;
	tpkt.count++;
	tpkt.checksum=tpkt.checksum+c;
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       send
        Function :      Sends an entire packet

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void send(unsigned char type)
{
        int i;
        unsigned char lo_byte,hi_byte;

        transmit(DLE);
        transmit(STX);
        tpkt.type=type;

        transmit(tpkt.type);            /* 1. transmit the type */

        lo_byte=(unsigned char)(tpkt.count & 0x00ff);   /* 2. transmit the count low byte first */
        hi_byte=(unsigned char)((tpkt.count & 0xff00)/256);
        transmit(lo_byte);
        transmit(hi_byte);

        for(i=0;i<tpkt.count;i++)     /* 3. transmit the data */
                transmit(tpkt.data[i]);

        lo_byte=(unsigned char)(tpkt.checksum & 0x00ff);
                                        /* 4. transmit the checksum low byte first */
        hi_byte=(unsigned char)((tpkt.checksum & 0xff00)/256);
        transmit(lo_byte);
        transmit(hi_byte);

        tpkt.count=tpkt.checksum=0;

}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       pktinteg
        Function :      Checks the integrity of a packet

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int pktinteg(void)
{
        unsigned int i,sum=0;

        if (rpkt.type==ACKNOWLGE) return(TRUE);

        for(i=0;i<rpkt.count;i++)  /* see if the checksum sums up the data */
                sum=sum+(unsigned int)(rpkt.data[i]);   /* in the pkt */

        if(sum==rpkt.checksum) return(TRUE);
        else return(FALSE);
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
          Routine :          transmit
          Function :         Transmits a byte from the serial port using the muart
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void transmit(unsigned char val)
{
          while(!xmit_ready()); // Wait till you can transmit
          outportb(comport,val);
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
          Routine :          checkforpkt
          Function :         waits till an entire pkt has been received and then loads
                             it up as a pkt. The packet format is as follows:
                             1.   type
                             2.   count (int) no. of data bytes
                             3.   data
                             4.   checksum (int) sum of data bytes only

                             Note that if the type is a control pkt then the data may
                             contain a RQST or an ACK
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int checkforpkt(QUEUE *rxq)      /* receives an entire packet */
{
          unsigned char lo_byte,hi_byte;
          unsigned int num=0,cnt,i;
          static unsigned int check=0;

          if(headr()==FALSE)
          {         // must strip bad header else you'll be
                    // reading the same stuff each time
                    return(FALSE);
          }
          else
          {
                    num=data_on(rxq);
                    if(num >5)
                    {
                              cnt=0;
                              cnt=rxq->data[cincr(rxq->front,rxq->maxsize,4)]*256 & 0xff00;
                              cnt=cnt+rxq->data[cincr(rxq->front,rxq->maxsize,3)];

                              if(num>=cnt+7)
                              {
                                        get_from(rxq);
                                        get_from(rxq);
                                        rpkt.type=get_from(rxq);     /* 1. receive the type ...*/

                                        /* get integer count */     /* 2. receive the count ...*/
                                        lo_byte=get_from(rxq); /* received low byte of count first */
                                        hi_byte=get_from(rxq);     /* high byte of count next */
                                        rpkt.count=hi_byte*256;
                                        rpkt.count=rpkt.count & 0xff00;
                                        rpkt.count=rpkt.count | lo_byte;
                                                                         /* 3. receive the data ...*/
                                        for(i=0;i<rpkt.count;i++)
```

```
                                        /* count gives the number of data bytes */
                                {
                                        rpkt.data[i]=get_from(rxq);
                                }
                                        /* 4. receive the integer checksum ... */
                                lo_byte=get_from(rxq);
                                hi_byte=get_from(rxq);
                                        /* rx the low byte of the checksum first */
                                rpkt.checksum=hi_byte*256;      /* then the high byte */
                                rpkt.checksum=rpkt.checksum & 0xff00;
                                rpkt.checksum=rpkt.checksum | lo_byte;
                                return TRUE;
                        }
                }
                check++;
                if (check==1500)
                        rxq->count=rxq->front=rxq->rear=0;

        }
        return FALSE;
}


int headr(void)
{
        unsigned char c=FALSE;
        int front=-10;

        if(data_on(rxq)>=2)
        {
                c = read_from(rxq,&front);

                if (c == DLE)
                {
                        switch (read_from(rxq,&front))
                        {
                                case DLE :
                                        rxq->count=rxq->front=rxq->rear=0;
                                c=FALSE;
                                        break;
                                case STX :
                                        c = TRUE;
                                        break;
                                default :
                                        rxq->count=rxq->front=rxq->rear=0;
                                        c = FALSE;
                                        break;
                        }
                }
                else
                {
                        rxq->count=rxq->front=rxq->rear=0;
                        c=FALSE;
                }
        }
        return (int)c;
}
```

```
/*****************************************************************/
/*                    INITIALISATION Routines                    */
/*****************************************************************/
void initialisations(void)
{
        comm_initl(BAUD_12,RXQ_SIZE,0); /* comm port interrupts and queues */
        distance_ints();      /* distance sensor interrupts via 80188 internal PIC */
        pkt_initl();
        init_arrays();
        init_ar();
        outport(TIMER0_CNT,0);
        outport(TIMER1_CNT,0);
}

void init_arrays(void)
{
        int i;

        for(i=0;i<MAXPATHLEN;i++)
                ppath[i][0]=ppath[i][1]=0;
}

void init_ar(void)
{
        int obj,j;
        for(obj=0;obj<MAXOBJS;obj++)
        {
                for(j=0;j<MAX_LEN;j++)
                        map[obj].points[X_ROW][j]=map[obj].points[Y_ROW][j]=0;
        }
}
```

```
/**************************************************************/
/*                                                            */
/*                          Sensors.c                         */
/*                                                            */
/**************************************************************/

#include <dos.h>
#include "ada.h"
#include "packet.h"

void dac(long ,unsigned int );
unsigned char adc_chan(long , int );
void read_sens(struct sens *);

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       read_sens
        Function :      read the ultrasonic,infrared,and touch sensors

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void read_sens(struct sens *sensors)
{
        unsigned char channel;

        channel=CHANNEL_1;
        sensors->Left_ultra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Left_infra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Left_touch=adc_chan(BOARD2,channel);
        channel=channel+2;
        sensors->Front_ultra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Front_infra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Front_touch=adc_chan(BOARD2,channel);
        channel=channel+2;
        sensors->Right_ultra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Right_infra=adc_chan(BOARD2,channel);
        channel++;
        sensors->Right_touch=adc_chan(BOARD2,channel);
}

/* Analog - Digital Converter */
unsigned char adc_chan( long BOARD_NO, int channel )
{
        long STATUS, MULTIPLEXOR;

        MULTIPLEXOR=BOARD_NO+MUX_OFF_ADR;
        outportb(MULTIPLEXOR,channel);  // Tell the A/D to start
        outportb(MULTIPLEXOR,channel);

        STATUS=BOARD_NO+MUX_OFF_ADR;
        while(inportb(STATUS)&EOC);     // Wait till the value is converted
        return ( (inportb(BOARD_NO+0x0008)));
}
```

```
/*****************************************************************/
/*                                                               */
/*                          Motors.c                             */
/*                                                               */
/*****************************************************************/


#include <dos.h>
#include "ada.h"

void dac(long ,unsigned int );
void motor(unsigned int c,unsigned int dac_port);

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

        Routine :       motor
        Function :      write a value out to a motor port

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void motor(unsigned int c,unsigned int dac_port)
{
        dac(dac_port,c);
}

/* Digital - Analog Converter */
void dac(long ADRSS, unsigned int volt)
{
        unsigned int mvolt=0,hibyte=0,lobyte=0;

        mvolt = volt<<4;
        hibyte=mvolt&0xFF00;  /* Swap the high byte and the low byte */
        hibyte=hibyte/256;    /* as outport writes the high and low  */
        hibyte=hibyte&0x00FF; /* bytes the wrong way around for the  */
        lobyte=mvolt&0x00FF;  /* SDAC12-4 D/A board */
        lobyte=lobyte*256;    /* Note: this procedure is only required */
        lobyte=lobyte&0xFF00; /* for the SDAC12-4. The other boards    */
        mvolt=lobyte|hibyte;  /* require one byte (0-255) and range    */
        outport(ADRSS,mvolt);
}
```

```
/*****************************************************************/
/*                                                               */
/*              HEADER FILES FOR THE PC SOFTWARE                 */
/*                                                               */
/*****************************************************************/

/*****************************************************************************
 **                                                                         **
 **  MODULE          :       Packet.h                                       **
 **                                                                         **
 **  DESCRIPTION     :       Defines variables used in a packet             **
 **                                                                         **
 *****************************************************************************/

#ifndef  PACKETS

#define  PACKETS


#define  DOWNLOAD_PATH          0xF1
#define  DOWNLOAD_MAP           0xF2
#define  DOWNLOAD_MOTORS        0xF3
#define  RQST_SENSOR_INF        0x03
#define  ACKNOWLGE              0x01
#define  SEND_AGAIN             0x02
#define  PKT_SIZE               1024
#define  DLE                    0x10
#define  STX                    0x02
#define  RETRY                  64000

typedef struct _packet
{
     unsigned char type;
     unsigned int count;
     unsigned char data[PKT_SIZE];
     unsigned int checksum;
} PACKET;

#endif


/*****************************************************************************
 **                                                                         **
 **  MODULE          :       PcInt.h                                        **
 **                                                                         **
 **  DESCRIPTION     :       Defines the comm port addresses                **
 **                                                                         **
 *****************************************************************************/

#ifndef  PCINT

#define PCINT


#define  TRUE           1
#define  FALSE          0
```

```
#define  COM_PARAMS
(_COM_CHR8|_COM_STOP1|_COM_NOPARITY|_COM_1200)
#define  P8259_0          0x20
#define  P8259_1          0x21
#define  END_OF_INT       0x20
#define  BIOS_DATA        ((int far *) (0x400000L))

#define  IER              (comport+1)
#define  IIR              (comport+2)
#define  LCR              (comport+3)
#define  MCR              (comport+4)
#define  LSR              (comport+5)
#define  MSR              (comport+6)

#define  IEROFF           0
#define  MCROFF           0
#define  RDAINT           1
#define  OUT2             0x08

/* Two serial port macros */
#define xmit_ready()      (inp(LSR) & 0x20)
#define byte_ready()      (inportb(LSR) & 0x40)

typedef struct _queue
{
      int count;
      int front;
      int rear;
      int maxsize;
      unsigned char *data;
} QUEUE;

#ifdef __cplusplus
      #define __CPPARGS ...
#else
      #define __CPPARGS
#endif

#endif


/***************************************************************************
 **                                                                       **
 **  MODULE          :       DEFS.H                                       **
 **                                                                       **
 **  DESCRIPTION     :       Header file to be included in the three      **
 **  main source files, it contains all definitions necessary             **
 **  in thees files.                                                      **
 **                                                                       **
 ***************************************************************************/

#ifndef  DEFS

#define DEFS


#define FALSE                 0
#define TRUE                  1
```

```
/*
 * Icon and window information
 */
#define ICONW                   16
#define ICONH                   16
#define NUM_FILLS_WIDE          16
#define BORDER                  4
#define MAXPOLYSIZE             40
#define DIMENSION               8.0

#define OBJWINLEFT              510
#define OBJWINTOP               60
#define OBJWININCX              20
#define OBJWININCY              20
#define OBJWINOFFSET            80


/*
 * Path and motion planning definitions
 */
#define CORNER_R                0.2
#define BACK_R                  1.414213562  * CORNER_R
#define INCREMENT               0.1
#define PATH_APROX              10

/*
 * Definitions for the planning and drawing functions.
 */
#define NUMGOBJECTS 50   /* Max num of objects allowed */
#define MAXOBJPARTS             12
#define TXT                     0
#define POLY                    1
#define CIRCLE                  2
#define LINE                    3
#define PATH                    4
#define COS45                   0.707106781
#define SIN45                   0.707106781
#define NODE_RADIUS             8
#define GOBJECT_COLOR           YELLOW
#define MOBJECT_COLOR           LIGHTBLUE

#define PATH_COLOR              LIGHTGRAY

/*
 * Defines for speed entry function
 */
#define BUTUP_PTR               0       /* Instances of type button for speed */
#define BUTDOWN_PTR             1       /* entry window */
#define BUTOK_PTR               2


/*
 * Button and Menu definitions.
 */
#define OUTLINE                 6
#define BUT_WIDTH               50
#define BUT_HEIGHT              10
```

```
#define BUT_SPACE           25
#define MAX_NUM_BUTTONS     16
#define BUTTON_UP           0       /* Direction to push button */
#define BUTTON_DOWN         1

#define MENU_ITEM_SPACE     3
#define MENU1_OPTIONS       2
#define MENU2_OPTIONS       3

/*
 * KEYS !
 */
#define CTRL_Z              0x1a

#define ESC                 27
#define RETURN              13
#define CR                  0xOd
#define LF                  10
#define BS                  0x08
#define TAB_CHAR            0x09

#define LEFT_ARROW          203
#define RIGHT_ARROW         205

#define UP_ARROW            72
#define DOWN_ARROW          80

#define HOME_KEY            199
#define END_KEY             207
#define PGUP                201
#define PGDN                209
#define INS                 210
#define DEL                 211
#define BACKSPACE           8
#define TAB                 9
#define BACKTAB             143

#define FUNC_1              187
#define FUNC_2              188
#define FUNC_3              189
#define FUNC_4              190
#define FUNC_5              191
#define FUNC_6              192
#define FUNC_7              193
#define FUNC_8              194
#define FUNC_9              195
#define FUNC_10             196

#define RIGHT_SHIFT         1
#define LEFT_SHIFT          2
#define CTRL_LOCK           4
#define ALT_LOCK            8
#define SCROLL_LOCK         16
#define NUMLOCK             32
#define CAPSLOCK            64
#define INSERTLOCK          128

#define S_DIST              .2
```

```
#define EXT                     .2
#define MAXNUM                  99999
#define EPSOLON                 0.0000001

#endif


/*************************************************************************
**                                                                      **
**      MODULE          :       sensors.h                               **
**                                                                      **
**                              Sensor structure                        **
*************************************************************************/

#ifndef  SENSORS

#define  SENSORS


struct sens{
        unsigned char Left_ultra;
        unsigned char Left_infra;
        unsigned char Left_touch;
        unsigned char Front_ultra;
        unsigned char Front_infra;
        unsigned char Front_touch;
        unsigned char Right_ultra;
        unsigned char Right_infra;
        unsigned char Right_touch;
};

#endif

/*************************************************************************
**                                                                      **
**      MODULE          :       DATA.h                                  **
**                                                                      **
**      DESCRIPTION  :          Source file which contains all variable **
**              and structure declarations which are required to be     **
**              global to the system.                                   **
**                                                                      **
*************************************************************************/

#include "defs.h"

#ifndef  DATA

#define  DATA

/*
 * structure for objects in drawing window.
 */
                                /* Button structure */
extern struct BUTTON
{
        int left, top, right, bottom;
        int xcorrection, ycorrection;       /* Correction due to mouse using */
};
```

```
                                        /* full screen coordinates.     */
extern struct graphobj
{
        int type;
        int numpoints;
        double *points;
};

extern struct GRAPHICSOBJ
{
        int type;
        int oldtyp;
        int left, top;
        int right,bottom;
        int numpoints;
        int numparts;
        int oldnum;
        double *points;
        double *extent_points;
        struct graphobj *gobj[MAXOBJPARTS];
};

/*
 * This structure is used for passing lines and simplifying parameter lists.
 */
extern struct LINESTRUCT
{
        double x1,y1;
        double x2,y2;
};

/*
 * Used to stored the planned path on completion of its planning !
 */
extern struct PATHOBJ
{
        int numpoints;
        double *points;
};

typedef struct apoint
{
        int num;
        double x;
        double y;
        int PERM;
        int D_END;
}point;

extern struct centre_grav{
        int obj;
        double xc;
        double yc;
};

typedef struct acoord
{
        double x;
```

```
        double y;
}coord;

typedef struct pth
{
        double x;
        double y;
        struct pth *next;
} pathpoints;

extern struct OBSTRUCT
{
        double x;
        double y;
        int obj;
        struct OBSTRUCT *next;
};


/* Global variables */

extern int maxx, maxy;
extern int wl, wr, wt, wb;          /* Main window coordinates */
extern int mwl, mwr, mwt, mwb;      /* Message window coordinates */
extern int owl, owr, owt, owb;      /* Obstacle window coordinates */
extern int rwl, rwr, rwt, rwb;      /* Robot window coordinates */
extern int globalfillstyle;
extern int globallinestyle;
extern int globaltextstyle;
extern int gridon;
extern int TRAIL;                   /* Flag to tell if the robot trail is to be shown */

extern int PATH_PICKED;
extern int PATH_INVALID;
extern int globalwindowcolor;       /* Used for writing on windows */
extern int currentmovingobj;
extern int nextmovingobj;
extern int currentobj;
extern int nextobj;
extern int left_arrow;
extern int right_arrow;

extern int Buttoncolor;
extern int buttonptr;
extern int currbutton;

extern int sx,sy,gx,gy;             /* Source and goal nodes */
extern double worldsx,worldsy,worldgx,worldgy;

extern int num_enodes;              /* 0 & 1 are source & goal */

extern struct BUTTON Buttons[MAX_NUM_BUTTONS];
extern struct PATHOBJ path;
extern struct MOVINGOBJ  mobjects[2];
extern struct GRAPHICSOBJ gobjects[NUMGOBJECTS];
extern struct centre_grav centroid[NUMGOBJECTS];

#endif
```

```
/*****************************************************************************
**                                                                          **
**      MODULE              :       proto.h                                 **
**                                                                          **
**                          prototypes                                      **
*****************************************************************************/
#include "data.h"
#include "PcInt.h"


#ifndef  PROTO

#define PROTO


void stop_motors(void);
int upload_sens_struct(unsigned char);
int download_motors(unsigned char);
int download_path(unsigned char);
int download_map(unsigned char);
void download(void);
void initialisations(void);
void put_poly_on_pkt(int obj);
void put_circle_on_pkt(int obj);
void start(void);
void draw_robot(void);
double toradians(double d);

static void put_on(QUEUE *, unsigned char );
static unsigned char get_from(QUEUE *);
void SetRxQueue(int rxqsize);
void cleanup(void);
void setup(short port_number,unsigned commparams);
int cincr(int x, int y,int incr);
static void interrupt far  rx_handler(__CPPARGS);    /* interrupt prototype */
void interrupt (*old_handler)(__CPPARGS);            /* interrupt function pointer */
int room_on(QUEUE *q);
int data_on(QUEUE *q);
static unsigned char get_from(QUEUE *q);
int ccinc(int x, int y);
static void put_on_pkt(unsigned char c);
int send(unsigned char rqst);
void loadrpkt(QUEUE *rxq);
int pktinteg(void);
void transmit(unsigned char );
int send_pkt(unsigned char rqst);
int waitforpkt(QUEUE *rxq);
int headr(void);
static void pkt_initl(void);
int intersect(struct LINESTRUCT ln1,struct LINESTRUCT ln2,double *x,double *y);

#endif
```

```
/*****************************************************************/
/*                        The PC SOFTWARE                        */
/*                             client.c                          */
/*          This downloads the path + envirnment map to the      */
/*          ROBOT and requests information periodically          */
/*          from the ROBOT to update the GUI on the PC.          */
/*****************************************************************/

/*****************************************************************/
/*            The PC COMMUNICATIONS with the ROBOT               */
/*                          COMMS.c                              */
/*     This downloads the path + envirnment to the robot         */
/*****************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <math.h>
#include <graphics.h>

#include "sensors.h"
#include "data.h"
#include "PcInt.h"
#include "Packet.h"
#include "proto.h"


static unsigned int leftdist=0,rightdist=0;
static QUEUE rxvq, *rxq = &rxvq;
static QUEUE txvq, *txq = &txvq;
static short comport=0,
             port_number=0,
             int_number=12,
             int_enable_mask=0xef,
             int_disable_mask=0x10;
static PACKET tpkt,rpkt;
const unsigned RXQ_SIZE = 1024;
const unsigned TXQ_SIZE = 1024;
struct motor {
        unsigned char rff;
        unsigned char rfr;
        unsigned char lff;
        unsigned char lfr;
};
static struct motor motor_vals;

struct sens sensors;

#define MAXPATHLEN          200
#define X_ROW               0
#define Y_ROW               1
#define MAX_LEN             40
#define MAXOBJS             25
```

```
#define W                       0.205
#define PI                      3.14159
#define R_TO_D                  57.29578
#define WheelRadius             0.015
#define NUMSEG                  3.0

static unsigned int ppath[MAXPATHLEN][2];
static unsigned int  Xn=0,Yn=0,quita=0;


extern pathpoints gpath;
extern void messagebox(char *messagestr);
extern void calc_circ_points(double *points, double *EXTENTpoly);


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       start
        Function :      This is where the fun starts - The robot gets told about its
                        environment and informs the pc what it is at by keeping the
                        pc updated on its whereabouts. It is called when the person
                        clicks download from the GUI.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void start(void)
{

        initialisations();
        download();
        while(!kbhit())
        {
                while ( send_pkt(RQST_SENSOR_INF)!=4 );
                                        /* Rqst sensor + position info to update display */
                draw_robot();  /* Update the PC while the robot is moving */
                delay(200);
        }
        stop_motors() /* The motors are probably stopped anyway if the
                        robot completed its path else I interruped it */
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       download
        Function :      Downloads the path and environment map to the robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void download(void)
{
        while ( send_pkt(DOWNLOAD_PATH)!=1 ); /* Rqst to download the path */
        while ( send_pkt(DOWNLOAD_MAP)!=2 ); /* Rqst to download the map */
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       download_path
        Function :      Download the path to the robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

```
int download_path( unsigned char rqst )
{
	int i;
	unsigned int xval,yval;
	unsigned char lo_byte,hi_byte;

	pathpoints *pathptr;

	pathptr=&gpath;
	while(pathptr!=NULL)
	{
		xval=(unsigned int)(pathptr->x*10); // convert to cm
		yval=(unsigned int)(pathptr->y*10);
		lo_byte= (unsigned char)(xval & 0x00FF);
		hi_byte= (unsigned char)((xval>>8) & 0xFF);

		put_on_pkt(lo_byte); /* Put xval onto the packet */
		put_on_pkt(hi_byte);

		lo_byte= (unsigned char)(yval & 0x00FF);
		hi_byte= (unsigned char)((yval>>8) & 0xFF);

		put_on_pkt(lo_byte); /* Put yval onto the packet */
		put_on_pkt(hi_byte);

		pathptr=pathptr->next;
	}

	if(gpath.next!=NULL)	return send(rqst);
	else	messagebox("No Path has been generated");

	return FALSE;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	download_map
	Function :	Download a map of the environment to the robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int download_map( unsigned char rqst )
{
	int obj;

	put_on_pkt( (unsigned char)(nextobj & 0xFF));  /* lo byte */
	put_on_pkt( (unsigned char)((nextobj >>8) & 0xFF));  /* hi byte */

	for(obj=0; obj<nextobj; obj++)
	{
		switch(gobjects[obj].type)
		{
			case POLY:
				put_poly_on_pkt(obj);
				break;
			case CIRCLE:
				put_circle_on_pkt(obj);
				break;
```

```
                }
        }
        return send( rqst );
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       put_poly_on_pkt
        Function :      If the object is a polygon this function downloads it
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void put_poly_on_pkt(int obj)
{
        int j;

        put_on_pkt( (unsigned char)(gobjects[obj].type & 0xFF) ); /* lo byte */
        put_on_pkt( (unsigned char)((gobjects[obj].type>>8) & 0xFF) ); /* hi byte */

        put_on_pkt( (unsigned char)(gobjects[obj].numpoints & 0xFF) ); /* lo byte */
        put_on_pkt( (unsigned char)((gobjects[obj].numpoints>>8) & 0xFF) ); /* hi byte */

        for(j=0;j<gobjects[obj].numpoints*2;j++)
        {
                put_on_pkt( (unsigned char)((unsigned int)(gobjects[obj].points[j]) & 0xFF) );
                put_on_pkt( (unsigned char)(((unsigned int)(gobjects[obj].points[j])>>8) &
                        0xFF) );
        }
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       put_circle_on_pkt
        Function :      If the object is a circle this function downloads it
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void put_circle_on_pkt(int obj)
{
        int j;
        double circ_array[18];

        put_on_pkt( (unsigned char)(gobjects[obj].type & 0xFF) );
        put_on_pkt( (unsigned char)((gobjects[obj].type>>8) & 0xFF) );

        put_on_pkt(0x09);
        put_on_pkt(0x00);

        calc_circ_points(gobjects[obj].points,circ_array);
        for(j=0;j<18;j++)
        {
                put_on_pkt( (unsigned char)((unsigned int)(circ_array[j]) & 0xFF) );
                put_on_pkt( (unsigned char)(((unsigned int)(circ_array[j])>>8) & 0xFF) );
        }
}


double toradians(double d)
{
        return ((double)(d) * PI / 180.0);
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       draw_robot
        Function :      Draw the robot on the Pcs display
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void draw_robot(void)
{
        int Xll=0,Yll=0,Xrr=0,Yrr=0,savecolor=0;
        float Xra=0.0,Yra=0.0,Xla=0.0,Yla=0.0;

        Xra = (float)Xn - ((W/2.0) * sin(toradians((float)quita)));
        Xla = (float)Xn + ((W/2.0) * sin(toradians((float)quita)));
        Yra = (float)Yn + ((W/2.0) * cos(toradians((float)quita)));
        Yla = (float)Yn - ((W/2.0) * cos(toradians((float)quita)));

        WORLDtoPC((double)Xra,(double)Yra,&Xrr,&Yrr);
        WORLDtoPC((double)Xla,(double)Yla,&Xll,&Yll);
        hidemouse();
        savecolor = getcolor();
        setcolor(YELLOW);
        line(Xll,Yll,Xrr,Yrr);
        setcolor(savecolor);
        showmouse();
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       upload_sens_struct
        Function :      Gets position and sensor information from robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int upload_sens_struct(unsigned char rqst)
{

        unsigned char lo_byte,hi_byte;
        unsigned int tmp=0;


        if ( send(rqst)==TRUE )
        {
                sensors.Left_ultra=rpkt.data[0];
                sensors.Left_infra=rpkt.data[1];
                sensors.Left_touch=rpkt.data[2];
                sensors.Front_ultra=rpkt.data[3];
                sensors.Front_infra=rpkt.data[4];
                sensors.Front_touch=rpkt.data[5];
                sensors.Right_ultra=rpkt.data[6];
                sensors.Right_infra=rpkt.data[7];
                sensors.Right_touch=rpkt.data[8];

                // the code below is for the tachometer.......
                lo_byte = rpkt.data[9];
                hi_byte = rpkt.data[10];
                tmp     = hi_byte<<8;
                Xn= tmp + (lo_byte & 0xFF);

                lo_byte = rpkt.data[11];
                hi_byte = rpkt.data[12];
```

```
                tmp     = hi_byte<<8;
                Yn= tmp + (lo_byte & 0xFF);

                return (TRUE);
        }
        else return FALSE;
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       stop_motors
        Function :      Stop the robots motors from the pc
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void stop_motors(void)
{
        int i;

        tpkt.count=tpkt.checksum=0;
        motor_vals.rff=0x0;
        motor_vals.rfr=0x0;
        motor_vals.lff=0x0;
        motor_vals.lfr=0x0;
        while ( send_pkt(DOWNLOAD_MOTORS) !=3 );  /* send motor values */
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       download_motors
        Function :      called when we call stop_motors
                        downloads values to the motors in the following order:
                        right full forward
                        right full reverse
                        left full forward
                        left full reverse
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int download_motors(unsigned char rqst) /* Called when stop_motors is used */
{
        int i;

        tpkt.count=tpkt.checksum=0;
        put_on_pkt(motor_vals.rff);
        put_on_pkt(motor_vals.rfr);
        put_on_pkt(motor_vals.lff);
        put_on_pkt(motor_vals.lfr);

        return send(rqst);
}

void initialisations(void)
{
        setup(1,COM_PARAMS);
        SetRxQueue(RXQ_SIZE);
        pkt_initl();
}
```

```
/****************************************************************/
/*              intrpt_q.c          (interrupt_queue)          */
/****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <math.h>
#include <graphics.h>

#include "sensors.h"
#include "data.h"
#include "PcInt.h"
#include "Packet.h"
#include "proto.h"


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       rx_handler
        Function :      The handler thats called when a character is received by
                        the PC
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void interrupt far rx_handler(__CPPARGS)
{
        _disable();
        asm pushf

        if (byte_ready())
                if (room_on(rxq))
                        put_on(rxq, inp(comport));

        asm popf
        /* Issue an end of interrupt command to the internal PIC */

        outp(P8259_0, END_OF_INT);
        _enable();
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       room_on
        Function :      Status of queue
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int room_on(QUEUE *q)
{
        return q->count < q->maxsize ;
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       room_on
        Function :      Status of queue
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

```
int data_on(QUEUE *q)
{
        return q->count;
}

static unsigned char get_from(QUEUE *q)
{
        unsigned char c;

        _disable();
        c = q->data[q->front];
        q->front = ccinc(q->front, q->maxsize);
        q->count--;
        _enable();

        return c;
}

static unsigned char read_from(QUEUE *q, int *front)
{
        unsigned char c;

        _disable();
        if (*front==-10) *front=q->front;

        c = q->data[*front];
        *front = ccinc(*front, q->maxsize);
        _enable();

        return c;
}

static void put_on(QUEUE *q, unsigned char c)
{
        _disable();
        q->data[q->rear] = c;
        q->rear = ccinc(q->rear, q->maxsize);
        q->count++;
        _enable();
}

/* Some queue handling macros */
int ccinc(int x, int y)
{
        return (x + 1) % y;
}

/* Some queue handling macros */
int cincr(int x, int y,int incr)
{
        return (x + incr) % y;
}

/*-----------------------------------------------------------------------
        Routine :       SetRxQueue
        Function :      Set the size of the receive queue
-----------------------------------------------------------------------*/
```

```
void SetRxQueue(int rxqsize)
{
        // Allocate space for buffers
        if (rxqsize)
        {
                rxq->count=rxq->front=rxq->rear=0;
                rxq->maxsize = rxqsize;
                rxq->data = (unsigned char *) malloc(rxq->maxsize);
        }
}

void setup(short port_number,unsigned commparams)
{
        int intmask;
        comport=*(BIOS_DATA+port_number);
        if(port_number==0)
        {
                int_enable_mask=0xef;
                int_disable_mask=0x10;
                int_number=12;
        }
        if(port_number==1)
        {
                int_enable_mask=0xf7;
                int_disable_mask=8;
                int_number=11;
        }
        _disable();
        old_handler=_dos_getvect(int_number);
        _dos_setvect(int_number,rx_handler);
        _disable();
        _bios_serialcom(_COM_INIT,port_number,commparams);

        outp(MCR,OUT2);
        outp(IER,RDAINT);
        intmask=inp(P8259_1) & int_enable_mask;
        outp(P8259_1,intmask);

        /*************************************************/
        // These remove bug they arise owing to an
        // installed mouse driver ....
        outp(IIR,0x01);
        outp(LSR,0x60);
        /*************************************************/
        _enable();
}

void cleanup(void)
{
        int intmask;
        _disable();
        outp(IER,IEROFF);
        outp(MCR,MCROFF);
        intmask=inp(P8259_1) | int_disable_mask;
        outp(P8259_1,intmask);
        _dos_setvect(int_number,old_handler);
        _enable();
}
```

```
/****************************************************************/
/*              packet.c                                        */
/****************************************************************/


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       pkt_initl
        Function :      Initialise the receive and transmit pkts
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

static void pkt_initl(void)
{
        int i;
        tpkt.count=tpkt.checksum=0;
        rpkt.count=rpkt.checksum=0;
        for(i=0;i<PKT_SIZE;i++)
        {
                rpkt.data[i]=tpkt.data[i]=0;
        }
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       send_pkt
        Function :      send a packet to the robot
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
int send_pkt(unsigned char rqst)
{
                        /* If rqsting to send data ...*/
        switch (rqst)   /* download the appropriate data */
        {
                case DOWNLOAD_MOTORS:
                        if(download_motors(rqst)==TRUE)  return 3;
                        else return FALSE;

                case RQST_SENSOR_INF:
                        if(upload_sens_struct(rqst)==TRUE) return 4;
                        else return FALSE;

                case DOWNLOAD_PATH:
                        if(download_path(rqst)==TRUE) return 1;
                        else return FALSE;

                case DOWNLOAD_MAP:
                        if(download_map(rqst)==TRUE) return 2;
                        else return FALSE;
        }
        return FALSE;
}

static void put_on_pkt(unsigned char c)
{
        tpkt.data[tpkt.count]=c;
        tpkt.count++;
        tpkt.checksum=tpkt.checksum+c;
}
```

```
int send(unsigned char rqst) /* Sends an entire packet */
{
        int i;
        unsigned char lo_byte,hi_byte;

        transmit(DLE);
        transmit(STX);

        tpkt.type=rqst;

        transmit(tpkt.type);        /* 1. transmit the type */

        lo_byte=(unsigned char)(tpkt.count & 0x00ff);  /* 2. transmit the count low byte first */
        hi_byte=(unsigned char)((tpkt.count & 0xff00)/256);
        transmit(lo_byte);
        transmit(hi_byte);

        for(i=0;i<tpkt.count;i++)   /* 3. transmit the data */
                transmit(tpkt.data[i]);

        lo_byte=(unsigned char)(tpkt.checksum & 0x00ff);
                                    /* 4. transmit the checksum low byte first */
        hi_byte=(unsigned char)((tpkt.checksum & 0xff00)/256);
        transmit(lo_byte);
        transmit(hi_byte);

        tpkt.count=tpkt.checksum=0;

        /* Wait for the data */
        if(waitforpkt(rxq)==FALSE)
        {
                rxq->count=rxq->front=rxq->rear=0;
                _disable();
                outp(IIR,0x01);
                outp(LSR,0x60);
                _enable();

                return FALSE;
        }

        if(pktinteg()==TRUE)
        {

                switch(rqst)
                {
                        case DOWNLOAD_MOTORS:
                        case DOWNLOAD_PATH:
                        case DOWNLOAD_MAP:
                                                if(rpkt.type==ACKNOWLGE)
                                                    return(TRUE);
                                                else return FALSE;

                        case RQST_SENSOR_INF:
                                                if(rpkt.type=='d') return(TRUE);
                                                else return FALSE;

                }
```

```
	}
	else  //bad pkt
	{
		rxq->count=rxq->front=rxq->rear=0;
	}
	return(FALSE);
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	pkinteg
	Function :	Check the packet integrity
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int pktinteg(void)
{
	unsigned int i,sum=0;

	if (rpkt.type==ACKNOWLGE) return(TRUE);

	//if (rpkt.count==0) return(FALSE);
	for(i=0;i<rpkt.count;i++)  /* see if the checksum sums up the data */
		sum=sum+(unsigned int)(rpkt.data[i]);    /* in the pkt */

	if(sum==rpkt.checksum) return(TRUE);
	else  return(FALSE);

}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	transmit
	Function :	Transmits a byte from the serial port using the muart
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void transmit(unsigned char val)
{
	while(!xmit_ready()); // Wait till you can transmit
	outportb(comport,val);
}



/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	waitforpkt
	Function :	waits till an entire pkt has been received then loads it up
			as a pkt. The following describes the pkt format:
			1.   type
			2.   count (int) no. of data bytes
			3.   data
			4.   checksum (int) sum of data bytes only
			Note that if the type is a control pkt
			then the data may contain a RQST or an ACK
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int waitforpkt(QUEUE *rxq)      /* receives an entire packet */
{
	unsigned char lo_byte,hi_byte;
```

```
unsigned int num=0,cnt;
unsigned int i,retry=0,delay=0;

while(headr()==FALSE && delay<5)
{
        retry++;
        if(retry==RETRY)
        {
                retry=0;
                delay++;
        }
}
if(delay == 5 ) return FALSE;
retry=delay=0;
while(delay<5)
{
        num=data_on(rxq);
        if(num >5)
        {
                cnt=0;
                cnt=rxq->data[cincr(rxq->front,rxq->maxsize,4)]*256 & 0xff00;
                cnt=cnt+rxq->data[cincr(rxq->front,rxq->maxsize,3)];
                if(num>=cnt+7)
                {
                        get_from(rxq);
                        get_from(rxq);
                        rpkt.type=get_from(rxq);     /* 1. receive the type ...*/

                        /* get integer count */    /* 2. receive the count ...*/
                        lo_byte=get_from(rxq); /* received low byte of count first */
                        hi_byte=get_from(rxq);    /* high byte of count next */
                        rpkt.count=hi_byte*256;
                        rpkt.count=rpkt.count & 0xff00;
                        rpkt.count=rpkt.count | lo_byte;  /* 3. receive the data ...*/

                        for(i=0;i<rpkt.count;i++)
                                        /*count gives the number of data bytes*/
                                rpkt.data[i]=get_from(rxq);
                                        /* 4. receive the integer checksum ... */
                        lo_byte=get_from(rxq);
                        hi_byte=get_from(rxq);
                                        /* rx the low byte of the checksum first */
                        rpkt.checksum=hi_byte*256;        /* then the high byte */
                        rpkt.checksum=rpkt.checksum & 0xff00;
                        rpkt.checksum=rpkt.checksum | lo_byte;
                        return TRUE;
                }
        }
        retry++;
        if(retry==RETRY)
        {
                retry=0;
                delay++;
        }
}
return FALSE;
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	headr
	Function :	Looks at the header of a packet to check where it starts
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
int headr(void)
{
	unsigned char c=FALSE;
	int front=-10;

	if(data_on(rxq)>=2)
	{
		c = read_from(rxq,&front);

		if (c == DLE)
		{
			switch (read_from(rxq,&front))
			{
				case DLE :
					c=FALSE;
					break;
				case STX :
					c = TRUE;
					break;
					default :
				c = FALSE;
					break;
			}
		}
		else
		{
			rxq->count=rxq->front=rxq->rear=0;
			c=FALSE;
		}
	}
	return (int)c;
}
```

```
/*****************************************************************************/
/*      Centroid.c                                                           */
/*      These routines calculate the centroid of an obstacle. The supporting */
/*      routines for calculating the area of a polygon are included          */
/*      along with functions to test if points are witin obstacles           */
/*****************************************************************************/

#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <stdarg.h>
#include <alloc.h>
#include <math.h>
#include <time.h>
#include "mouse.h"
#include "calc.h"
#include "gtext.h"

#include "defs.h"
#include "data.h"
#include "proto.h"


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        Routine :       calc_centroid
        Function :      finds the centroid of a given obstacle

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/

void calc_centroid(int obj)
{
    double Area=0,Mx=0,My=0,xc=0,yc=0;
    int ir,it,il,ib,i;
    double left,right,top,bottom;
    int PCx,PCy,sign=0,numpoints=0;
    int delta=0,oldelta=0,tst=0,defsign=0;

    if(gobjects[obj].type==CIRCLE)
    {
        xc=gobjects[obj].points[0];
        yc=gobjects[obj].points[1];
    }
    else
    {
        numpoints=gobjects[obj].numpoints;
        getipolybounds(numpoints,&gobjects[obj].points[0],&il,&it,&ir,&ib);

        if(il<ir)
        {
            if( (it>=il) && (it<=ir) )
            {
                sign=defsign=1;
            }
            else sign=defsign =-1;

            for(i=il;i<ir;)
            {
```

```
            if(i==il||tst==TRUE)
            {
                if(gobjects[obj].points[cinc(i,numpoints)*2]==gobjects[obj].points[i*2])
                        tst=TRUE;
                else
                {
                        oldelta=delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
                                        gobjects[obj].points[i*2]) >=0 ? 1 : -1;
                        tst=FALSE;
                }
            }
            else
            {
                delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
                        gobjects[obj].points[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                        sign=defsign*-1;
                else sign=defsign;
            }


            calcarea(obj,i,&Area,&Mx,&My,sign);
            i=cinc(i,numpoints);
        }
        defsign=sign=(sign>0) ? -1 : 1;
        tst=FALSE;
        for(i=ir;i!=il;)
        {
            if(i==ir||tst==TRUE)
            {
                if(gobjects[obj].points[cinc(i,numpoints)*2]==gobjects[obj].points[i*2])
                        tst=TRUE;
                else
                {
                        oldelta=delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
                                gobjects[obj].points[i*2]) >=0 ? 1 : -1;
                        tst=FALSE;
                }
            }
            else
            {
                delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
                                gobjects[obj].points[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                        sign=defsign*-1;
                else sign=defsign;
            }

            calcarea(obj,i,&Area,&Mx,&My,sign);
            i=cinc(i,numpoints);
        }
    }
    else
    {
        if( (it>=ir) && (it<=il) )
        {
            sign=defsign=1;
        }
```

```
else
{
	sign=defsign=-1;
}

for(i=ir;i<il;)
{
	if(i==ir||tst==TRUE)
	{
		if(gobjects[obj].points[cinc(i,numpoints)*2]==gobjects[obj].points[i*2])
			tst=TRUE;
		else
		{
			oldelta=delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
					gobjects[obj].points[i*2]) >=0 ? 1 : -1;
			tst=FALSE;
		}
	}
	else
	{
		delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
			gobjects[obj].points[i*2]) >=0 ? 1 : -1;
		if(delta!=oldelta)
			sign=defsign*-1;
		else sign=defsign;
	}

	calcarea(obj,i,&Area,&Mx,&My,sign);
	i=cinc(i,numpoints);
}
defsign=sign=(sign>0) ? -1 : 1;
tst=FALSE;
for(i=il;i!=ir;)
{
	if(i==il||tst==TRUE)
	{
		if(gobjects[obj].points[cinc(i,numpoints)*2]==gobjects[obj].points[i*2])
			tst=TRUE;
		else
		{
			oldelta=delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
					gobjects[obj].points[i*2]) >=0 ? 1 : -1;
			tst=FALSE;
		}
	}
	else
	{
		delta=(gobjects[obj].points[cinc(i,numpoints)*2]-
			gobjects[obj].points[i*2]) >=0 ? 1 : -1;
		if(delta!=oldelta)
			sign=defsign*-1;
		else sign=defsign;
	}

	calcarea(obj,i,&Area,&Mx,&My,sign);
	i=cinc(i,numpoints);
}
```

```
            }
            xc=(My/Area);
            yc=(Mx/Area);

        }
        centroid[obj].obj=obj;
        centroid[obj].xc=xc;  /* Store the results in an array so I can access them quickly later */
        centroid[obj].yc=yc;
        WORLDtoPC(xc,yc,&PCx,&PCy); /* The pc screen coordinates of the centroid */
        hidemouse();
        setcolor(BLUE);
        circle(PCx-wl,PCy-wt,3);
        setcolor(YELLOW);
        showmouse();
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       cinc
        Function :      increments the variable i to point to the next
                        coordinate in the polygon
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int cinc(int i,int numpoints)
{
        return (i+=1)%(numpoints-1);
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       calcarea
        Function :      adds or subtracts an area to the overall area of the
                        polygon as its being calculated
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void calcarea(int obj, int i,double *area,double *Mx,double *My,double sign)
{
        double x1,x2,y1,y2,deltax=0,deltamx=0,deltamy=0,ym=0,deltaarea=0,xm=0;

        x1=gobjects[obj].points[i*2];
        y1=gobjects[obj].points[i*2+1];
        x2=gobjects[obj].points[cinc(i,gobjects[obj].numpoints)*2];
        y2=gobjects[obj].points[cinc(i,gobjects[obj].numpoints)*2+1];

        if(x1!=x2)
        {
                deltax=x2-x1;
                deltax=(deltax>0) ?deltax :-1*deltax;
                ym=(y1+y2)/2;
                deltaarea=deltax*ym;
                *area=*area+sign*deltaarea;

                xm=(x1+x2)/2;
                deltamy=deltax*xm*ym;
                *My=*My+sign*deltamy;

                deltamx=deltax*((ym*ym)/2);
                *Mx=*Mx+sign*deltamx;
        }
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	getipolybounds
	Function :	get the indexes to the boundary points of the
			polygon.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void getipolybounds(int numpoints, double *poly, int *il, int *it,
			int *ir, int *ib)
{
	int i;
	double left,top,right,bottom;
	left = MAXNUM;
	top = 0;
	right = 0;
	bottom = MAXNUM;

	for (i=0;i<numpoints;i++)
	{
		if (poly[i*2] < left ){ left  = poly[i*2]; *il=i;}
		if (poly[i*2] > right ){ right  = poly[i*2]; *ir=i;}
		if (poly[i*2+1] > top ){ top  = poly[i*2+1]; *it=i;}
		if (poly[i*2+1] < bottom ){ bottom  = poly[i*2+1]; *ib=i;}
	}
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	findig
	Function :	finds the point xg,yg as in algorithm
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void findig(int obj,double xc,double yc, double x1,double y1, double x2, double y2, double *xg,
double *yg, double *xm, double *ym)
{
	struct LINESTRUCT line1={0,0,0,0},line2={0,0,0,0};
	int i,j,maxi;
	double interx=0,intery=0;


	for(j=1;j<=2;j++)
	{
		findmidpnt(j,xc,yc,x1,y1,x2,y2,&line1,xm,ym);
		setviewport(wl,wt,wr,wb,1);

		if(gobjects[obj].type==CIRCLE)
			maxi=8;
		else	maxi=gobjects[obj].numpoints -1;

		for(i=0; i<maxi; i++)
		{
			line2.x1=gobjects[obj].extent_points[i*2];
			line2.y1=gobjects[obj].extent_points[i*2+1];
			line2.x2=gobjects[obj].extent_points[i*2+2];
			line2.y2=gobjects[obj].extent_points[i*2+3];

			if(intersect(line1,line2,&interx,&intery)) break;
		}
```

```
                calcig(xc,yc,*xm,*ym,interx,intery,xg,yg);
                if(!pointinobj(*xg,*yg)) break;
        }
        setviewport(0,0,maxx,maxy,1);

}

int intersect(struct LINESTRUCT ln1,struct LINESTRUCT ln2,double *x,double *y)
{
        double bigx=0,bigy=0,smallx=0,smally=0,xa=0,ya=0;

        if(calc_line_interc(ln1,ln2,&xa,&ya))
        {
                getbig(ln2.x1,ln2.y1,ln2.x2,ln2.y2,&bigx,&bigy,&smallx,&smally);
                if( (xa<=bigx) && (xa>=smallx) && (ya<=bigy) && (ya>=smally) )
                {
                        if(distance(xa,ya,ln1.x1,ln1.y1) > distance(xa,ya,ln1.x2,ln1.y2) )
                        {
                                *x=xa;
                                *y=ya;
                                return 1;
                        }
                }
        }
        return 0;
}

void findmidpnt(int j,double xc,double yc,double x1,double y1,double x2,double y2,struct
LINESTRUCT *ln1,double *xm,double *ym)
{
        struct LINESTRUCT line1={0,0,0,0};
        double m=0,m1=0,c=0,r=0,h=0,k=0,b=0,num=0,den=0;
        double xa=0,ya=0,xb=0,yb=0,d1=0,d2=0;

        line1.x1=xc;
        line1.y1=yc;

        *xm=line1.x2=(x1+x2)/2;
        *ym=line1.y2=(y1+y2)/2;

        /* This if statement calculates the value of xm,ym for special case */
        if( ((int)(xc*100)==(int)(*xm*100)) && ((int)(yc*100)==(int)(*ym*100)) )
        {
                /* If the path happens to cut through the center of gravity   */
                /* a line at 90 degrees to the path is drawn to intersect the */
                /* circle.                                                    */

                if((int)(x1*100)==(int)(x2*100))
                {
                        m=0;
                        c=yc;
                        *xm=line1.x2=xc+.1;
                        *ym=line1.y2=m*(*xm)+c;
                }
                else if((int)(y1*100)==(int)(y2*100))
                {
                        *xm=line1.x2=xc;
```

```
                *ym=line1.y2=yc+0.1;
        }
        else
        {
                m1=(y1-y2)/(x1-x2);
                m=-1/m1;
                c=yc-m*xc;

                *xm=line1.x2=xc+.1;
                *ym=line1.y2=m*(*xm)+c;
        }
}

if(j==2) /* If the first ig was inside another obj try going the other way */
{
        if((int)(xc*100)==(int)(*xm*100))
        {
                if(*ym>yc)
                        *ym=line1.y2=yc-EXT;
                else    *ym=line1.y2=yc+EXT;
        }
        else
        {
                m = (yc-*ym)/(xc-*xm);
                c = yc-m*xc;
                r = S_DIST;
                h = xc;
                k = yc;
                b =  2*(h-m*(c-k));
                num = ((2*(m*(c-k)-h))*(2*(m*(c-k)-h)))-4*
                        (1+m*m)*((h*h)+((c-k)*(c-k))-(r*r));
                den = 2*(1+m*m);

                xa = ( b+sqrt(num) )/den;
                xb = ( b-sqrt(num) )/den;
                ya = m*xa+c;
                yb = m*xb+c;
                d1 = distance(*xm,*ym,xa,ya);
                d2 = distance(*xm,*ym,xb,yb);
                if(d1<d2)
                {
                        *xm=line1.x2=xb;
                        *ym=line1.y2=yb;
                }
                else
                {
                        *xm=line1.x2=xa;
                        *ym=line1.y2=ya;
                }

        }

}
ln1->x1=line1.x1;
ln1->y1=line1.y1;
ln1->x2=line1.x2;
ln1->y2=line1.y2;
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       calcig
        Function :      finds the point xg,yg as in algorithm
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void calcig(double xc,double yc,double xm,double ym,double interx,double intery,double
*xg,double *yg)
{
        double xa=0,ya=0,xb=0,yb=0,d1=0,d2=0;
        double m=0,c=0,r=0,h=0,k=0,b=0,num=0,den=0;


        /* This if statement calculates the value of xg,yg */
        if((int)(xc*100)==(int)(xm*100))
        {
                *xg=interx;
                if(intery>yc)
                        *yg=intery+S_DIST;
                else    *yg=intery-S_DIST;
        }
        else
        {
                m = (yc-ym)/(xc-xm);
                c = yc-m*xc;

                r = S_DIST;
                h = interx;
                k = intery;

                b =  2*(h-m*(c-k));
                num = ((2*(m*(c-k)-h))*(2*(m*(c-k)-h)))-4*(1+m*m)*
                        ((h*h)+((c-k)*(c-k))-(r*r));

                den = 2*(1+m*m);

                xa = ( b+sqrt(num) )/den;
                xb = ( b-sqrt(num) )/den;
                ya = m*xa+c;
                yb = m*xb+c;
                d1 = distance(xc,yc,xa,ya);
                d2 = distance(xc,yc,xb,yb);
                                                /* Note */
                                /* If the path intersects the object */
                                /* on only one extension point then  */
                if(d1<d2)       /* xm,ym will be equal to that point */
                {                /* as x1==x2 and y1==y2.          */
                        *xg=xb;
                        *yg=yb;
                }
                else
                {
                        *xg=xa;
                        *yg=ya;
                }
        }
}
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :        check_visible
        Function :    checks if two points are visible to each other
                         by finding if a line joining them intersects with
                         any obstacle boundary.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int visible(double x1, double y1, double x2, double y2, coord *inter1, coord *inter2, int *objt)

{
    int i;
    struct LINESTRUCT polyline={0,0,0,0},pathline={0,0,0,0};
    struct OBSTRUCT *ptr,*head,root={0,0,0,NULL};
    double interx=0,intery=0;
    double bigx=0,bigy=0;
    double smallx=0,smally=0;
    int obj;
    int maxi;

    ptr=&root;
    root.next=NULL;

    pathline.x1 = x1;
    pathline.y1 = y1;
    pathline.x2 = x2;
    pathline.y2 = y2;

    setviewport(wl,wt,wr,wb,1);

    for (obj=0;obj<nextobj;obj++)
    {
        if (gobjects[obj].type == CIRCLE)
            maxi = 8;  /*  circle  */
        else  maxi = gobjects[obj].numpoints-1;

        for (i=0; i<maxi; i++)
        {
            polyline.x1 = gobjects[obj].extent_points[i*2];
            polyline.y1 = gobjects[obj].extent_points[i*2+1];
            polyline.x2 = gobjects[obj].extent_points[i*2+2];
            polyline.y2 = gobjects[obj].extent_points[i*2+3];

            if ( calc_line_interc(polyline,pathline,&interx,&intery) )
            {     /* Check if it's a valid intersection */
                getbig(polyline.x1, polyline.y1, polyline.x2, polyline.y2,
                    &bigx, &bigy, &smallx, &smally);

                if (  (interx <= bigx) && (interx >= smallx)
                && (intery <= bigy) && (intery >= smally)  )
                {
                    getbig(pathline.x1, pathline.y1, pathline.x2, pathline.y2,
                     &bigx, &bigy, &smallx, &smally);

                    if (  (interx <= bigx) && (interx >= smallx)
                     && (intery <= bigy) && (intery >= smally)  )
                     {
                     ptr->next = (struct OBSTRUCT *)malloc(sizeof(struct OBSTRUCT));
```

```
                        ptr=ptr->next;
                        ptr->x=interx;
                        ptr->y=intery;
                        ptr->obj=obj;
                        ptr->next=NULL;
                        }
                    }
                }
            }
        }
        if ( root.next!=NULL )
        {
            getinter(&root,inter1,inter2,objt);
            return(FALSE);  /* Not visible */
        }
        setviewport(0,0,maxx,maxy,1);
        return(TRUE);      /* Visible */
}


void getinter(struct OBSTRUCT *head,coord *inter1, coord *inter2,int *obj)
{
        double d=0,oldist=MAXNUM;
        struct OBSTRUCT *ptr,*ptr1;
        int flag=0;

        ptr=head->next;
        if(ptr!=NULL)
        {
                do{
                        if( (d=distance(ptr->x,ptr->y,curr->x,curr->y)) <oldist )
                        {
                                oldist=d;
                                inter1->x=ptr->x;
                                inter1->y=ptr->y;
                                *obj=ptr->obj;
                        }
                        ptr=ptr->next;
                }while(ptr!=NULL);
                ptr=head->next;
                do{
                        if( ptr->obj==*obj )
                        {
                                if( ((d=distance(ptr->x,ptr->y,curr->x,curr->y)) >oldist) &&
                                (!((ptr->x==inter1->x) && (ptr->y==inter1->y))) )
                                {
                                        oldist=d;
                                        inter2->x=ptr->x;
                                        inter2->y=ptr->y;
                                        flag=1;
                                }
                        }
                        ptr=ptr->next;
                }while(ptr != NULL );
                if(flag==0)
                {
                        inter2->x=inter1->x;
```

```
                        inter2->y=inter1->y;
                }
                ptr1=ptr=head->next;
                do{
                        ptr=ptr->next;
                        free(ptr1);
                        ptr1=ptr;
                }while(ptr!=NULL);
                head->next=NULL;
        }
}


/*-----------------------------------------------------------------------
        Routine :       pointinobj
        Function :      checks if a point x1,y1 is in any of the obstacles
-----------------------------------------------------------------------*/

int pointinobj(double x1, double y1)

{
    struct LINESTRUCT polyline={0,0,0,0},pathline={0,0,0,0};
    double interx=0,intery=0,xc=0,yc=0;
    double bigx=0,bigy=0;
    double smallx=0,smally=0,oldist=0,d=0;
    int i,obj,maxi,flag=0,j,endpt=FALSE;



    setviewport(wl,wt,wr,wb,1);

    for (obj=0;obj<nextobj;obj++)
    {
        if (gobjects[obj].type == CIRCLE)
            maxi = 8;  /*  circle  */
        else  maxi = gobjects[obj].numpoints-1;

        get_centroid(obj,&xc,&yc);
        pathline.x1 = x1;
        pathline.y1 = y1;
        pathline.x2 = xc;
        pathline.y2 = yc;

        for (i=0; i<maxi; i++)
        {
            polyline.x1 = gobjects[obj].extent_points[i*2];
            polyline.y1 = gobjects[obj].extent_points[i*2+1];
            polyline.x2 = gobjects[obj].extent_points[i*2+2];
            polyline.y2 = gobjects[obj].extent_points[i*2+3];

            if ( calc_line_interc(polyline,pathline,&interx,&intery) )
            {     /* Check if it's a valid intersection */
                getbig(polyline.x1, polyline.y1, polyline.x2, polyline.y2,
                        &bigx, &bigy, &smallx, &smally);

                if (  (interx <= bigx) && (interx >= smallx)
                      && (intery <= bigy) && (intery >= smally)  )
                {
```

```
                        getbig(pathline.x1, pathline.y1, pathline.x2, pathline.y2,
                          &bigx, &bigy, &smallx, &smally);

                        if ( (interx <= bigx) && (interx >= smallx)
                        && (intery <= bigy) && (intery >= smally) )
                        {
                              flag++;
                              d=distance(interx,intery,xc,yc);
                              if(d>oldist)
                              {
                                    j=i;
                                    oldist=d;
                                    if( (interx==polyline.x1)&&(intery==polyline.y1) )
                                          endpt=TRUE;
                                    if( (interx==polyline.x2)&&(intery==polyline.y2) )
                                          endpt=TRUE;

                              }
                        }
                  }
            }
      }
      if(flag==0)
            return(TRUE);
      else
      {
            if (flag==1) return(FALSE);
            else
            {
                  if(endpt==TRUE)  /* You'll get 2 intersections if the point */
                        return(FALSE); /* intersects one of the polygon points */

                  else   /* Else special case where point may be inside an obstacle
                        ie dont want to plan a path within an object!!!*/
                        return(isvisib(obj,x1,y1,j));
            }
      }
   }
   setviewport(0,0,maxx,maxy,1);
   return(FALSE);
}

/*-----------------------------------------------------------------
      Routine :      isvisib
      Function :     This decides if a point is within or outside an
                     obstacle, j is the number of points
-----------------------------------------------------------------*/

int isvisib(int obj,double x1,double y1,int j)
{
      double area1=0,area2=0,*ptr;
      int i=0,maxi;


      setviewport(wl,wt,wr,wb,1);

      if (gobjects[obj].type == CIRCLE)
            maxi = 8;  /*  circle  */
```

```
	else	maxi = gobjects[obj].numpoints-1;


	area1=getarea(gobjects[obj].extent_points,maxi);
	ptr=malloc(sizeof(double)*(gobjects[obj].numpoints+1)*2);
	for(i=0;i<=j;i++)
	{
		ptr[i*2]=gobjects[obj].extent_points[i*2];
		ptr[i*2+1]=gobjects[obj].extent_points[i*2+1];
	}
	ptr[i*2]=x1;  /* i==j+1 */
	ptr[i*2+1]=y1;
	for(i=j+1;i<maxi+1;i++)
	{
		ptr[(i+1)*2]=gobjects[obj].extent_points[i*2];
		ptr[(i+1)*2+1]=gobjects[obj].extent_points[i*2+1];
	}
	area2=getarea(ptr,maxi+1);
	free(ptr);



	if(area1>area2)
		return(TRUE);
	else return (FALSE);

}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	getarea
	Function :	To find the area of the polygon it finds the indexs to
			the polygons boundary points. Then starting from the base
			it will add on or subtract the area enclosed by the next
			segment depending on its slope.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

double getarea(double *ptr,int maxi)
{
   double Area=0;
   int ir,it,il,ib,i;
   int sign=0,defsign=0;
   int delta=0,oldelta=0,tst=0;

   getipolybounds(maxi+1,ptr,&il,&it,&ir,&ib);

   if(il<ir)
   {
	if( (it>=il) && (it<=ir) )
	{
	     sign=defsign=1;
	}
	else sign=defsign =-1;

	for(i=il;i<ir;)
	{
	     if(i==il||tst==TRUE)
	     {
```

```
                if(ptr[cinc(i,maxi+1)*2]==ptr[i*2])
                    tst=TRUE;
                else
                {
                    oldelta=delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                    tst=FALSE;
                }
            }
            else
            {
                delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                    sign=defsign*-1;
                else sign=defsign;
            }
            calarea(i,maxi,&Area,sign,ptr);
            i=cinc(i,maxi+1);
        }
        defsign=sign=(sign>0) ? -1 : 1;
        tst=FALSE;
        for(i=ir;i!=il;)
        {
            if(i==ir||tst==TRUE)
            {
                if(ptr[cinc(i,maxi+1)*2]==ptr[i*2])
                    tst=TRUE;
                else
                {
                    oldelta=delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                    tst=FALSE;
                }
            }
            else
            {
                delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                    sign=defsign*-1;
                else sign=defsign;
            }

            calarea(i,maxi,&Area,sign,ptr);
            i=cinc(i,maxi+1);
        }
    }
    else
    {
        if( (it>=ir) && (it<=il) )
        {
            sign=defsign=1;
        }
        else
        {
            sign=defsign=-1;
        }

        for(i=ir;i<il;)
        {
```

```
            if(i==ir||tst==TRUE)
            {
                if(ptr[cinc(i,maxi+1)*2]==ptr[i*2])
                    tst=TRUE;
                else
                {
                    oldelta=delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                    tst=FALSE;
                }
            }
            else
            {
                delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                    sign=defsign*-1;
                else sign=defsign;
            }

            calarea(i,maxi,&Area,sign,ptr);
            i=cinc(i,maxi+1);
        }
        defsign=sign=(sign>0) ? -1 : 1;
        tst=FALSE;
        for(i=il;i!=ir;)
        {
            if(i==il||tst==TRUE)
            {
                if(ptr[cinc(i,maxi+1)*2]==ptr[i*2])
                    tst=TRUE;
                else
                {
                    oldelta=delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                    tst=FALSE;
                }
            }
            else
            {
                delta=(ptr[cinc(i,maxi+1)*2]-ptr[i*2]) >=0 ? 1 : -1;
                if(delta!=oldelta)
                    sign=defsign*-1;
                else sign=defsign;
            }

            calarea(i,maxi,&Area,sign,ptr);
            i=cinc(i,maxi+1);
        }

    }
    return(Area);
}

/*-----------------------------------------------------------------
        Routine :       calarea
        Function :      It adds or subtracts the area enclosed by the points
                        depending on whether it encloses the polygon
-----------------------------------------------------------------*/
```

```
void calarea(int i,int maxi,double *area,double sign,double *ptr)
{
        double x1,x2,y1,y2,deltax=0,ym=0,deltaarea=0;

        x1=ptr[i*2];
        y1=ptr[i*2+1];
        x2=ptr[cinc(i,maxi+1)*2];
        y2=ptr[cinc(i,maxi+1)*2+1];

        if(x1!=x2)
        {
                deltax=x2-x1;
                deltax=(deltax>0) ?deltax :-1*deltax;
                ym=(y1+y2)/2;
                deltaarea=deltax*ym;
                *area=*area+sign*deltaarea;
        }
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       pointinobject
        Function :      Is a point within an obstacle. If it is when its added
                        to a polygon the polygon will have a smaller area
                        than the original polygon
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

int pointinobject(int obj, double x1, double y1)
{
    struct LINESTRUCT polyline={0,0,0,0},pathline={0,0,0,0};
    double interx=0,intery=0,xc=0,yc=0;
    double bigx=0,bigy=0;
    double smallx=0,smally=0,oldist=0,d=0;
    int i,maxi,flag=0,j,endpt=FALSE;

    if (gobjects[obj].type == CIRCLE)
        maxi = 8;  /*  circle  */
    else if( (gobjects[obj].oldtyp == CIRCLE) && (gobjects[obj].numparts == 1) )
        maxi = 8;
    else
        maxi = gobjects[obj].numpoints-1;

    get_centroid(obj,&xc,&yc);
    pathline.x1 = x1;
    pathline.y1 = y1;
    pathline.x2 = xc;
    pathline.y2 = yc;

    for (i=0; i<maxi; i++)
    {
        polyline.x1 = gobjects[obj].extent_points[i*2];
        polyline.y1 = gobjects[obj].extent_points[i*2+1];
        polyline.x2 = gobjects[obj].extent_points[i*2+2];
        polyline.y2 = gobjects[obj].extent_points[i*2+3];

        if ( calc_line_interc(polyline,pathline,&interx,&intery) )
        {   /* Check if it's a valid intersection */
            getbig(polyline.x1, polyline.y1, polyline.x2, polyline.y2,
                &bigx, &bigy, &smallx, &smally);
```

```
                if ( (interx <= bigx) && (interx >= smallx)
                && (intery <= bigy) && (intery >= smally) )
                {
                        getbig(pathline.x1, pathline.y1, pathline.x2, pathline.y2,
                                &bigx, &bigy, &smallx, &smally);

                        if ( (interx <= bigx) && (interx >= smallx)
                        && (intery <= bigy) && (intery >= smally) )
                        {
                                flag++;
                                d=distance(interx,intery,xc,yc);
                                if(d>oldist)
                                {
                                        j=i;
                                        oldist=d;
                                        if( (interx==polyline.x1)
                                                &&(intery==polyline.y1) )
                                                endpt=TRUE;
                                        if( (interx==polyline.x2)
                                                &&(intery==polyline.y2) )
                                                endpt=TRUE;

                                }
                        }
                }
        }
}
if(flag==0)
        return(TRUE);
else
{
        if (flag==1)
                return(FALSE);
        else
        {
                if(endpt==TRUE)  /* You'll get 2 intersections if the point */
                        return(FALSE); /* intersects one of the polygon points */

                else            /* Else special case where point may be inside */
                        return(isvisib(obj,x1,y1,j));
        }
}

}
```

```
/*************************************************************************
**                                                                      **
**   MODULE      :    PLANNING.C                                        **
**                                                                      **
**   DESCRIPTION   :                                                    **
**        This module contains the planning algorithm and a number      **
**        of supplimentay funcitions to support it. Such functions include **
**        thise for calculating free space extent nodes,                **
**                                                                      **
*************************************************************************/

#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <stdarg.h>
#include <alloc.h>
#include <math.h>
#include <time.h>
#include "mouse.h"
#include "calc.h"
#include "gtext.h"

#include "defs.h"
#include "data.h"
#include "proto.h"


/*
 * Variables global to this file.
 */
point s,g;
point extent_nodes[100];

double total_distance;
int MAXEXTENTS = 100;
pathpoints gpath={0,0,NULL},*end,*curr;



/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
    Routine         : path_motion
    Function        : controls the setting up and calculating of the
                              planned path
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void path_motion(void)
{
        get_nodes();
        if (plan_path())
                display_planned_path();
}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine       : get_nodes
        Function      : gets and marks start and end nodes for planned
                              path.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
```

```
void get_nodes(void)
{
        int START_CHOSEN = FALSE;
        struct textsettingstype savetext;

        gettextsettings(&savetext);

        messagebox("Select Start and end nodes");

        mousebuttonreleased(LEFT_BUTTON);
        mousebuttonreleased(RIGHT_BUTTON);
        setviewport(wl,wt,wr,wb,1);

        settextjustify(CENTER_TEXT,CENTER_TEXT);

        while(1)
        {
                while (!mousebuttonreleased (LEFT_BUTTON));
                if (!START_CHOSEN)
                {
                        getmousecoords(&sx,&sy);
                        if (sx < wl || sy < wt || sy > wb)
                        {
                                setviewport(0,0,maxx,maxy,1);
                                return;
                        }
                        START_CHOSEN = TRUE;
                        hidemouse();
                        setcolor(LIGHTBLUE);
                        outtextxy(sx-wl,sy-wt,"S");
                        setcolor(YELLOW);
                        circle(sx-wl,sy-wt,NODE_RADIUS);
                        showmouse();

                        PCtoWORLD(sx,sy,&worldsx,&worldsy);
                        s.x = worldsx;
                        s.y = worldsy;
                        s.num = 0;
                }
                else
                {
                        getmousecoords(&gx,&gy);
                        if (gx < wl || gy < wt || gy > wb)
                        {
                                setviewport(0,0,maxx,maxy,1);
                                return;
                        }
                        hidemouse();
                        setcolor(LIGHTBLUE);
                        outtextxy(gx-wl,gy-wt,"G");
                        setcolor(YELLOW);
                        circle(gx-wl,gy-wt,NODE_RADIUS);
                        showmouse();
                        PCtoWORLD(gx,gy,&worldgx,&worldgy);
                        g.x = worldgx;
                        g.y = worldgy;
                        g.num = 1;
```

```
                        break;
                }
        }  /* end while 1 */
        setviewport(0,0,maxx,maxy,1);
        START_CHOSEN = FALSE;
        settextjustify(savetext.horiz,savetext.vert);
        settextstyle(savetext.font, savetext.direction, savetext.charsize);

}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        Routine :       display_planned_path
        Function :      Sets up the path in the path structure using the
        preceed array to guide it. Note all points are converted to world
        coordinates. Lastly this function displays the path by drawing
        lines from start to goal nodes via the path.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void display_planned_path(void)
{
        int first;
        int pcx1,pcy1;
        int oldpcx1,oldpcy1;
        int i,j;
        pathpoints *tmp,*curr;
        char buf[50];

        total_distance = 0;
        setviewport(wl,wt,wr,wb,1);
        PATH_PICKED = TRUE;


        /* Loop for displaying the path */
        hidemouse();
        setcolor(PATH_COLOR);
        curr=tmp=&gpath;
        while(1)
        {
                WORLDtoPC(tmp->x,tmp->y,&oldpcx1,&oldpcy1);
                tmp=tmp->next;if(tmp==NULL) break;
                WORLDtoPC(tmp->x,tmp->y,&pcx1,&pcy1);
                line (pcx1-wl,pcy1-wt,oldpcx1-wl,oldpcy1-wt);
                total_distance += distance(curr->x,curr->y,tmp->x,tmp->y);
                curr=tmp;
        }
        setcolor(YELLOW);
        showmouse();
        sprintf(buf,"Path Length is %f meters",total_distance);
        messagebox(buf);

        setviewport(0,0,maxx,maxy,1);
}
```

```
/*-------------------------------------------------------------------
        Routine         : plan_path
        Function: sets up the variables for the pathplanner routine which
                        plans the path.

-------------------------------------------------------------------*/


int plan_path(void)
{
        coord s1={0,0},g1={0,0}; /* Start,goal */

        s1.x=s.x;
        s1.y=s.y;
        g1.x=g.x;
        g1.y=g.y;

        /*   growobjs(); */

        gpath.x=s.x;
        gpath.y=s.y;
        gpath.next=NULL;
        end=&gpath;

        if( pathplanner(s1,g1)) /* Find the path between start + destination */
        {
                end->next=(struct pth *)malloc(sizeof(struct pth));
                end=end->next;
                end->x=g.x;
                end->y=g.y;
                end->next=NULL;

                removeloops(); /* Cut out any loops */
                return 1;
        }
        else    return 0;
}

/*-------------------------------------------------------------------
        Routine     : pathplanner
        Function    : recursively calls itself to find its way around obstacles

-------------------------------------------------------------------*/

int pathplanner(coord p1, coord p2)
{
    coord inter1={0,0},inter2={0,0};
    double xc=0,yc=0,xm=0,ym=0,xg=0,yg=0,xmnew=0,ymnew=0;
    int obj=0,oldobj=0;

    curr=end;
    if ( visible(p1.x, p1.y, p2.x, p2.y, &inter1, &inter2, &obj) )
    {
        /* outtextxy(200,200,"No barriers between S and G ");*/
        return(TRUE);
    }

    get_centroid(obj,&xc,&yc);
```

```
	findig(obj,xc,yc,inter1.x,inter1.y,inter2.x,inter2.y,&xg,&yg,&xm,&ym);

	end->next=(struct pth *)malloc(sizeof(struct pth));
	end=end->next;
	end->x=xg;
	end->y=yg;
	end->next=NULL;
	oldobj=obj;

	do{
		if ( visible(p1.x,p1.y,xg,yg,&inter1,&inter2,&obj) ==0 )
		{
			if(oldobj!=obj)
			{
				get_centroid(obj,&xm,&ym);
				oldobj=obj;
			}
			findig(obj,xm,ym,inter1.x,inter1.y,inter2.x,inter2.y,&xg,&yg, &xmnew,&ymnew);
			addpath(xg,yg);
		}
		else break;
	}while(1);
	p1.x=xg;
	p1.y=yg;
	pathplanner(p1,p2);
	return(1);
}

/*.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=
		Routine		: get_centroid
		Function	: gets the centroid of an object from an array where
				  the points were previously calculated + stored.
.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=*/

void get_centroid(int obj,double *xc,double *yc)
{
		*xc=centroid[obj].xc;
		*yc=centroid[obj].yc;
}


/*.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=
		Routine		: removeloops
		Function:  remove any loops where the path might have looped
				  back and went in another direction to find its way
				  around an object.
.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=.=*/

void removeloops()
{
		pathpoints *tmp;
		coord inter1={0,0},inter2={0,0};
		int obj;

		tmp=curr=&gpath;
		tmp=tmp->next; if(tmp==NULL) return;
		tmp=tmp->next;
		while(1)
```

```
	{
		while(tmp!=NULL)
		{
			if(visible(curr->x,curr->y,tmp->x,tmp->y,&inter1,&inter2,&obj))
				shortenpath(curr,tmp);
			tmp=tmp->next;
		}
		tmp=curr=curr->next;
		tmp=tmp->next; if(tmp==NULL) break;
		tmp=tmp->next; if(tmp==NULL) break;
	}
}

/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine		: shortenpath
	Function	: remove all unwanted nodes from curr to tmp
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void shortenpath(pathpoints *curr,pathpoints *tmp)
{
	pathpoints *p1,*p2;

	p1=p2=curr->next;
	while(p2!=tmp)
	{
		p2=p2->next;
		free(p1);
		p1=p2;
	}
	curr->next=tmp;
}


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	delete_path
	Function :	deletes the path from screen and memory.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void delete_path(void)
{
	pathpoints *tmp1,*tmp2;

	if (PATH_PICKED)
	{
		PATH_PICKED = FALSE;
		tmp1=tmp2=gpath.next;
		while(tmp1!=NULL)
		{
			tmp2=tmp2->next;
			free(tmp1);
			tmp1=tmp2;
		}
		gpath.next=NULL;
		clearworkarea();
		display_all_objects();
	}
}
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	intersect_angle
	Function :	Calculates the acute angle between two lines.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
double intersect_angle(struct LINESTRUCT line1, struct LINESTRUCT line2)
{
	double m1, m2;
	double theta;

	m1 = slope(line1.x1, line1.y1, line1.x2, line1.y2);
	m2 = slope(line2.x1, line2.y1, line2.x2, line2.y2);
	if ( (m1 * m2) == -1.0)
		theta = 90.0;
	else
	{
		theta = atan( (m1 - m2) / (1 - m1*m2)) * R_TO_D;
		if ( theta < 0.0 )	/* Make sure the angle is positive */
			theta = - theta;
		if ( theta >= 90.0 )	/* Make sure the angle is the acute angle */
			theta = 180.0 - theta;
	}
	return( theta );
}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	slope
	Function :	Calculates the slope of the line formed by two points
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
double slope(double x1, double y1, double x2, double y2)
{
	if ( x2 == x1 )
		return(100000);	/* Simulate infinity ! */
	else	return( (y2 - y1)/(x2 - x1) );
}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	distance
	Function :	Calculates the distance between two points
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
double distance(double x1, double y1, double x2, double y2)
{
	return(sqrt( (x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1) ));
}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	addpath
	Function :	adds a new node onto the path
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void addpath(double x,double y)
{
	pathpoints *tmp;

	tmp=curr->next;
	curr->next=(struct pth *)malloc(sizeof(struct pth));
```

```
        curr->next->x=x;
        curr->next->y=y;
        curr->next->next=tmp;
}


/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :    growobjs
        Function :   calculates the extents along the objects for
                         use in planning the path.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void growobjs(void)
{
        int i;

        for (i=0;i<nextobj;i++)
        {
                if (gobjects[i].type == POLY)
                        calc_poly_extent(i);
                else if (gobjects[i].type == CIRCLE)
                        calc_circle_extent(i);
        }
}

/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       calc_circle_extent
        Function :      Calculates the extent points of a circle.
                        Eight points are used in this program.
                        Calculations in integers as points are in PC screen
                        coordinates.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void calc_circle_extent(int obj)
{
        double radius;
        /*  double C_R;*/

        radius = distance(gobjects[obj].points[0],gobjects[obj].points[1],
                        gobjects[obj].points[4],gobjects[obj].points[1]);

        /*C_R = (CORNER_R * 1.1) + (0.1 * radius);*/

        /* Point 1 : North */
        gobjects[obj].extent_points[0] = gobjects[obj].points[0];
        gobjects[obj].extent_points[1] = gobjects[obj].points[1] - radius - S_DIST;

        /* Point 2 : North-East */
        gobjects[obj].extent_points[2] = gobjects[obj].points[0] + (radius + S_DIST) * COS45;
        gobjects[obj].extent_points[3] = gobjects[obj].points[1] - (radius + S_DIST) * SIN45;

        /* Point 3 : East */
        gobjects[obj].extent_points[4] = gobjects[obj].points[0] + radius + S_DIST;
        gobjects[obj].extent_points[5] = gobjects[obj].points[1];

        /* Point 4 : South-East */
        gobjects[obj].extent_points[6] = gobjects[obj].points[0] + (radius + S_DIST) * COS45;
        gobjects[obj].extent_points[7] = gobjects[obj].points[1] + (radius + S_DIST) * SIN45;
```

```
        /* Point 5 : South */
        gobjects[obj].extent_points[8] = gobjects[obj].points[0];
        gobjects[obj].extent_points[9] = gobjects[obj].points[1] + radius + S_DIST;

        /* Point 6 : South-West */
        gobjects[obj].extent_points[10] = gobjects[obj].points[0] - (radius + S_DIST) * COS45;
        gobjects[obj].extent_points[11] = gobjects[obj].points[1] + (radius + S_DIST) * SIN45;

        /* Point 7 : West */
        gobjects[obj].extent_points[12] = gobjects[obj].points[0] - radius - S_DIST;
        gobjects[obj].extent_points[13] = gobjects[obj].points[1];

        /* Point 8 : North-West */
        gobjects[obj].extent_points[14] = gobjects[obj].points[0] - (radius + S_DIST) * COS45;
        gobjects[obj].extent_points[15] = gobjects[obj].points[1] - (radius + S_DIST) * SIN45;

        /* Point 9 : Center */
        gobjects[obj].extent_points[16] = gobjects[obj].extent_points[0];
        gobjects[obj].extent_points[17] = gobjects[obj].extent_points[1];

}

/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :       calc_poly_extent
        Function :      calculates the extent around a polygon.
                        Calculations in integers as points are in PC screen
                        coordinates.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void calc_poly_extent(int obj)
{
    int i,j;
    double x1,y1,x2,y2,prox,proy;
    double projx, projy;
    double interx,intery;
    double m_ratio,n_ratio,m,n;
    struct LINESTRUCT *lines;
    int PCx2,PCy2;

    lines = (struct LINESTRUCT *) malloc (sizeof(double)*4*(gobjects[obj].numpoints-1));

    for (i=0;i<gobjects[obj].numpoints - 1; i++)
    {
        x1 = gobjects[obj].points[i*2];
        x2 = gobjects[obj].points[i*2+2];
        y1 = gobjects[obj].points[i*2+1];
        y2 = gobjects[obj].points[i*2+3];

        m_ratio = distance(x1, y1, x2, y2) + EXT;
        n_ratio = EXT;
        m = m_ratio - EXT;
        n = n_ratio;

        prox = ( m_ratio * x2 - n_ratio * x1 ) / ( m_ratio - n_ratio );
        proy = ( m_ratio * y2 - n_ratio * y1 ) / ( m_ratio - n_ratio );
        projx = ( m_ratio * x1 - n_ratio * x2 ) / ( m_ratio - n_ratio );
        projy = ( m_ratio * y1 - n_ratio * y2 ) / ( m_ratio - n_ratio );
```

```
        if(pointinobjarr(gobjects[obj].points,gobjects[obj].numpoints,prox,proy))
        {
            prox = x2 - ( n * (x2 - x1) ) / m;
            proy = y2 - ( n * (y2 - y1) ) / m;
        }
        if(pointinobjarr(gobjects[obj].points,gobjects[obj].numpoints,projx,projy))
        {
            projx = x1 + ( n * (x2 - x1) ) / m;
            projy = y1 + ( n * (y2 - y1) ) / m;
        }

        if ( i+1 == gobjects[obj].numpoints -1 )
        {
            lines[0].x1 = prox;
            lines[0].y1 = proy;
        }
        else
        {
            lines[i+1].x1 = prox;
            lines[i+1].y1 = proy;
        }
        if ( i == 0)
        {
            lines[gobjects[obj].numpoints - 2].x2 = projx;
            lines[gobjects[obj].numpoints - 2].y2 = projy;
        }
        else
        {
            lines[i-1].x2 = projx;
            lines[i-1].y2 = projy;
        }
    }

    for (i=0;i<gobjects[obj].numpoints -1; i++)
    {
        if (i == gobjects[obj].numpoints - 2)
            j = 0;
        else  j = i+1;

        calc_line_interc(lines[i],lines[j],&interx,&intery);
        gobjects[obj].extent_points[i*2] = interx;
        gobjects[obj].extent_points[i*2+1] = intery;
    }
    gobjects[obj].extent_points[i*2] = gobjects[obj].extent_points[0];
    gobjects[obj].extent_points[i*2+1] = gobjects[obj].extent_points[1];

    for (i=0;i<gobjects[obj].numpoints -1; i++)
    {
        WORLDtoPC(gobjects[obj].extent_points[i*2],gobjects[obj].extent_points[i*2+1],
                    PCx2,&PCy2);
        hidemouse();
        setcolor(WHITE);
        circle(PCx2-wl,PCy2-wt,3);
        setcolor(YELLOW);
        showmouse();
    }
    free(lines);
}
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
          Routine :     calc_line_interc
          Function :    Calculates the intersection points
                            between two lines
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
int calc_line_interc(struct LINESTRUCT l1,struct LINESTRUCT l2, double  *x, double *y)
{
          double ax1, ay1, ax2, ay2;
          double bx1, by1, bx2, by2;
          double am,bm;

          ax1 = l1.x1;
          ay1 = l1.y1;
          ax2 = l1.x2;
          ay2 = l1.y2;

          bx1 = l2.x1;
          by1 = l2.y1;
          bx2 = l2.x2;
          by2 = l2.y2;



          if (ax2-ax1 == 0)
          {
                    if (bx2-bx1 == 0)
                    {
                              return(FALSE);
                    }
                    bm = (by2 - by1) /(bx2 - bx1);
                    *x = ax1;
                    *y = by2 + bm * (ax2 - bx2);
          }
          else
          {
                    am = (ay2 - ay1) /(ax2 - ax1);
                    if (bx2-bx1 == 0)
                    {
                              *x = bx1;
                              *y = ay2 + am * (bx2 - ax2);
                    }
                    else
                    {
                              bm = (by2 - by1) /(bx2 - bx1);
                              if (am == bm)
                              {
                                        return(FALSE);
                              }
                              *x = (- (bm * bx1) + by1 - ay1 + (am * ax1)) / (am-bm);
                              *y = ay1 + am * ( *x - ax1);
                    }
          }
          return(TRUE);
}
```

```
/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :    get_extent
        Function :   calculates the extent around a circle or polygon.
                     Calculation in doubles for world coordinates
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void get_extent(int numpoints,int type,double *WORLDpoly, double *EXTENTpoly)
{
        switch(type)
        {
                case CIRCLE: calc_circ_extent(WORLDpoly,EXTENTpoly);break;
                case POLY  : calc_pol_extent(numpoints,WORLDpoly,EXTENTpoly);break;
        }
}

/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        Routine :    calc_circ_extent
        Function :   calculates the extent around a circle.
                     Calculation in doubles for world coordinates
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/

void calc_circ_extent(double *points, double *EXTENTpoly)
{
        double radius;

        radius = distance(points[0],points[1],points[4],points[1]);

        /* Point 1 : North */
        EXTENTpoly[0] = points[0];
        EXTENTpoly[1] = points[1] - radius - S_DIST;

        /* Point 2 : North-East */
        EXTENTpoly[2] = points[0] + (radius + S_DIST) * COS45;
        EXTENTpoly[3] = points[1] - (radius + S_DIST) * SIN45;

        /* Point 3 : East */
        EXTENTpoly[4] = points[0] + radius + S_DIST;
        EXTENTpoly[5] = points[1];

        /* Point 4 : South-East */
        EXTENTpoly[6] = points[0] + (radius + S_DIST) * COS45;
        EXTENTpoly[7] = points[1] + (radius + S_DIST) * SIN45;

        /* Point 5 : South */
        EXTENTpoly[8] = points[0];
        EXTENTpoly[9] = points[1] + radius + S_DIST;

        /* Point 6 : South-West */
        EXTENTpoly[10] = points[0] - (radius + S_DIST) * COS45;
        EXTENTpoly[11] = points[1] + (radius + S_DIST) * SIN45;

        /* Point 7 : West */
        EXTENTpoly[12] = points[0] - radius - S_DIST;
        EXTENTpoly[13] = points[1];

        /* Point 8 : North-West */
        EXTENTpoly[14] = points[0] - (radius + S_DIST) * COS45;
```

```
        EXTENTpoly[15] = points[1] - (radius + S_DIST) * SIN45;

        /* Point 9 : Center */
        EXTENTpoly[16] = EXTENTpoly[0];
        EXTENTpoly[17] = EXTENTpoly[1];

}


/*-----------------------------------------------------------------------
        Routine :       calc_poly_extent
        Function :      calculates the extent around a polygon.
                        Calculation in doubles for world coordinates
-----------------------------------------------------------------------*/
void calc_pol_extent(int numpoints,double *points,double *EXTENTpoly)
{
        int i,j;
        double x1,y1,x2,y2,prox,proy;
        double projx, projy;
        double interx,intery;
        double m_ratio,n_ratio,m,n;
        struct LINESTRUCT *lines;
        int PCx2,PCy2;

        lines = (struct LINESTRUCT *) malloc (sizeof(double)*4*(numpoints-1));

        for (i=0;i<numpoints - 1; i++)
        {
                x1 = points[i*2];
                x2 = points[i*2+2];
                y1 = points[i*2+1];
                y2 = points[i*2+3];

                m_ratio = distance(x1, y1, x2, y2) + EXT;
                n_ratio = EXT;
                m = m_ratio - EXT;
                n = n_ratio;

                prox = ( m_ratio * x2 - n_ratio * x1 ) / ( m_ratio - n_ratio );
                proy = ( m_ratio * y2 - n_ratio * y1 ) / ( m_ratio - n_ratio );
                projx = ( m_ratio * x1 - n_ratio * x2 ) / ( m_ratio - n_ratio );
                projy = ( m_ratio * y1 - n_ratio * y2 ) / ( m_ratio - n_ratio );


                if(pointinobjarr(points,numpoints,prox,proy))
                {
                        prox = x2 - ( n * (x2 - x1) ) / m;
                        proy = y2 - ( n * (y2 - y1) ) / m;
                }
                 if(pointinobjarr(points,numpoints,projx,projy))
                {
                        projx = x1 + ( n * (x2 - x1) ) / m;
                        projy = y1 + ( n * (y2 - y1) ) / m;
                }

                if ( i+1 == numpoints -1 )
                {
```

```
                lines[0].x1 = prox;
                lines[0].y1 = proy;
        }
        else
        {
                lines[i+1].x1 = prox;
                lines[i+1].y1 = proy;
        }
         if ( i == 0)
        {
                lines[numpoints - 2].x2 = projx;
                lines[numpoints - 2].y2 = projy;
        }
        else
        {
                lines[i-1].x2 = projx;
                lines[i-1].y2 = projy;
        }

    }

    for (i=0;i<numpoints -1; i++)
    {
        if (i == numpoints - 2)
                j = 0;
        else    j = i+1;

        calc_line_interc(lines[i],lines[j],&interx,&intery);
        EXTENTpoly[i*2] = interx;
        EXTENTpoly[i*2+1] = intery;


    }
    EXTENTpoly[i*2] = EXTENTpoly[0];
    EXTENTpoly[i*2+1] = EXTENTpoly[1];

    for (i=0;i<numpoints -1; i++)
    {
        WORLDtoPC(EXTENTpoly[i*2],EXTENTpoly[i*2+1],&PCx2,&PCy2);
        hidemouse();
        setcolor(WHITE);
        circle(PCx2-wl,PCy2-wt,3);
        setcolor(YELLOW);
        showmouse();
    }
    free(lines);
}

/*-----------------------------------------------------------------
    Routine :   get_poly_bounds
    Function :  gets the top and bottom bounds of an object
-----------------------------------------------------------------*/
void getpolybounds(int numpoints, int *PCpoly, int *left, int *top,
                   int *right, int *bottom)
{
    int i;
    *left = getmaxx();
    *top = getmaxy();
```

```
	*right = 0;
	*bottom = 0;
	for (i=0;i<numpoints;i++)
	{
		if (PCpoly[i*2] < *left ) *left  = PCpoly[i*2];
		if (PCpoly[i*2] > *right ) *right  = PCpoly[i*2];
		if (PCpoly[i*2+1] < *top ) *top  = PCpoly[i*2+1];
		if (PCpoly[i*2+1] > *bottom ) *bottom  = PCpoly[i*2+1];
	}
}

/*-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
	Routine :	mallocerror
	Function :	exits program if not enough memory
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
void mallocerror(void)
{
	closegraph();
	printf("Not enough memory to run program\n");
	exit(1);
}
```