INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

NEURO-FUZZY BASED SYSTEM FOR MOBILE ROBOT NAVIGATION

by

Petru Rusu

A thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
(Electrical & Computer Engineering)

Faculty of Engineering
University of Ottawa

September, 2001

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank Dr. Emil Petriu and Dr. Dan Ionescu, my thesis supervisors, for their guidance and advice throughout my work. Their confidence in my capabilities as a student has contributed to the will, discipline and self-confidence that was necessary to achieve this goal. Their ideas, knowledge and judgement have been my strongest support.

I am also grateful to colleagues at work and at University of Ottawa, especially to Carol Gal, with whom I had many useful discussions, shaping my ideas.

Last but not least, my warmest thanks go to my wife Daniela and my children Anne-Marie and Christian, for their incredible support and encouragement, and especially for their patience during my work. Words are not enough here...

Abstract

The main topic of this thesis is the neuro-fuzzy control approach for mobile robot navigation in indoor environments. A system for directing the robot towards any target point has been developed and tested in simulated environments. The autonomous mobile robot uses only information obtained from its infrared and contact sensors in order to reach the target point and to avoid collisions. The control system is organized in a top-bottom hierarchy of various tasks to be executed by the robot, tasks which correspond to specific robot behaviors during his navigation. The task to be executed at every control cycle is determined according to the robot behavior required to be performed at that time. Each task is implemented through fuzzy (or neuro-fuzzy) techniques. When multiple low-level behaviors are required, a command fusion method is used to combine the output of several fuzzy sub-systems. A switching coordination method is used to select a behavior among higher level behaviors.

Chapter 1

Introduction

The development of techniques for autonomous navigation in real-world environments constitutes one of the major trends in the current research in robotics. This trend is motivated by the current gap between the available technology and the new application demands. On the one hand, current industrial robots lack flexibility and autonomy: typically, these robots perform pre-programmed sequences of operations in highly constrained environments, and are not able to operate in new environments or to face unexpected situations. On the other hand, there is a clear emerging market for truly autonomous robots. Possible applications include intelligent service robots for offices, hospitals, and factory floors; maintenance robots operating in hazardous or hardly accessible areas; domestic robots for cleaning or entertainment; semi-autonomous vehicles for help to disabled people; and so on.

An important problem in autonomous navigation is the need to cope with the large amount of uncertainty that is inherent of natural environments. Fuzzy logic has features that make it an adequate tool to address this problem.

This thesis proposes a neuro-fuzzy based system for controlling the navigation of a mobile robot. In the recent years fuzzy (or neuro-fuzzy) applications are widely

used in control [27, 8, 24]. In this thesis, the overall robot behavior is decomposed into a bottom-up hierarchy of increased complexity behaviors, from primitive ones to more complex ones. These behaviors are implemented by fuzzy rule-based models, and the capabilities of a neural network to learn some of these models are being used.

Fuzzy logic based controllers are expert control systems that smoothly interpolate between rules. Rules fire to continuous degrees and the multiple resultant actions are combined into an interpolated result. Processing of uncertain information and saving energy using common-sense rules and natural language statements are the basis for fuzzy logic control. The computational load of fuzzy inference systems is considerably lighter than other control systems.

The mobile robot navigation problem is presented in Chapter 2, together with the prototype of the mobile robot used for testing the control system. The sensory system proposed for the robot is detailed here also.

Chapter 3 answers the question why a neuro-fuzzy approach has been adopted for robot navigation and introduces the overall architecture of the neuro-fuzzy behavior control system. The decentralized control model is described, together with a brief introduction to the fuzzy logic control theory.

The control system is presented in greater details in the next chapters in a bottom-up manner: starting with the low-level (primitive) behavior implementations in Chapter 4, and continuing with higher level ones - Chapter 5.

The experimental results are discussed in Chapter 6, validating the model for controlling the robot navigation. Chapter 7 presents the conclusions of this thesis.

The specific contributions made in this thesis are in the area of fuzzy behavior control and robot navigation. For hierarchical behavior-based models, new primitive

behaviors are introduced. Their definitions are based on the requirement of a smooth trajectory for the robot. For these new primitive behaviors, the parameters of the fuzzy inference system have been identified using the learning capabilities of the neural networks. A new command fusion method specific to robot navigation problem is described in this thesis. This method combines multiple output fuzzy sets into a single output fuzzy set, therefore allowing the control system to take into account the recommendation from several primitive behaviors seen as expert systems. The whole system has been tested with several indoor environment maps of high level of complexity, similar to the one offered by real indoor environments, for example schools or office buildings.

Chapter 2

Mobile robot navigation

2.1 Problem definition

The basic problem of mobile robot navigation is to navigate from a start point to an endpoint without any collisions. This problem can be extended if one considers deadlines for reaching these endpoints, path exclusions for various reasons, or responsiveness to dynamic changes of the environment. Different aspects of the mobile robot navigation problem have been investigated, many of them with solutions in the fuzzy control field [36, 7, 40]. This thesis addresses the basic problem (navigation between two locations) in the context of a complex indoor environment which poses several challenges to any control system:

1. The need of an autonomous system which incorporates the whole logic for navigation, without using the processing power of a remote system.

2. Dealing with the real world in a dynamic way: the robot faces not only static obstacles - walls, fixed objects - but dynamic obstacles too - objects can move or environment could be changed by other agents. In such an environment, a static map cannot be used. Other self-localization techniques are investigated [10, 25].

3. Robots are equipped with range sensors and contact sensors which are well known for the concerns raised when considering their sensory data: their indication depends on the reflective properties of the environment, the surface illuminance and ambient temperature.

4. Any imperfections in the environment causing wheel slippage and errors in the dead-reckoning system impact the navigation.

5. The complexity of a indoor environment (partitioned in rooms, corridors, etc) leads to the need of more intelligent controllers capable of sensing the environment and reasoning about the environment in order to achieve their goal of reaching a target.

Existing solutions to mobile robot navigation problem can be classified [40] as *model- based solutions* (the robot control system uses a model of the environment to navigate towards the endpoint), *sensor-based solutions* (the robot control system uses only sensory data) and *hybrid solutions* - a combination of both approaches.

The *sensor-based* control system presented in this thesis tries to address some of the above mentioned challenges. The neuro-fuzzy approach does not require accurate sensory data and accommodates very well the uncertainty. The system relies only on input information from sensors, no environment map is required. And the hierarchical behavior architecture is well suited to deal with the complexity of any indoor environment.

2.2 Robot prototype

A popular design element for smaller robots is a circular body plan. This body shape for the robot was considered together with a classic differential steering system. For every control cycle, the system takes as input sensory data and offers as output the rotation angle to be executed by the robot. Therefore, the type of steering mechanism used by the robot is significant only for the purpose of achieving a certain smoothness degree during his move between endpoints.

In a differential steering system, two DC motors control independently two wheels which are mounted to a common axis. When both wheels turn at different, but fixed speeds, the robot steers a circular path. If the two wheels turn at the same speed but in opposite directions, the robot can execute a pivot.

A third wheel, usually a caster, is provided for support. Additional support wheels may be supplied for heavier robots. Smaller robots may use for support a skid or a simple dowel with a rounded end.

The control system has been tested in a simulated environment for a robot having the following characteristics: wheel base (distance between wheels): 70 cm; wheel radius: 7.5 cm; wheel thickness: 3 cm; robot speed: 0.4 m/s. Other speeds (up to 0.6 m/s) have been tested also.

2.3 Sensory system for navigation

The sensory system used in our case for the robot is based exclusively on infrared sensors. Infrared (IR) ranging eliminates some of the disadvantages sonar sensors have: multiple reflections, limited speed of firing, little energy returned to the transducer if sensor does not point normal to the target surface.

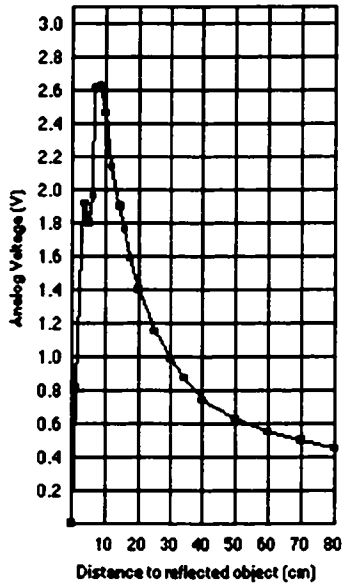

Figure 2.1: Range sensor - analog output voltage vs. distance to reflective objects

As an example of a low cost range sensor is Sharp GP2D12 Infrared Ranger [1] ($20 CDN) which has a detection range of 10 cm to 80 cm. The same range has been used for the inputs of the neuro-fuzzy system. The sensors report the distance as an analog voltage, with a characteristic as shown in Figure 2.1.

These sensors offer much better immunity to ambient lighting conditions because of a new method of ranging based on a triangle created between the emitter, the point of reflection and the detector. This method is almost immune to interference from ambient light and offers an excellent indifference to the color of object being detected. Detecting a black wall in full sunlight is possible.

As a contact sensor, another Sharp sensor, GP2D15, is a good choice. Its distance characteristic is shown in Figure 2.2. In order to use GP2D15 as a solid-state bumper (contact sensor), a pair of two detectors that cross over each other in front of the area of interest can be used. Their beams are roughly oval shaped and with the widest

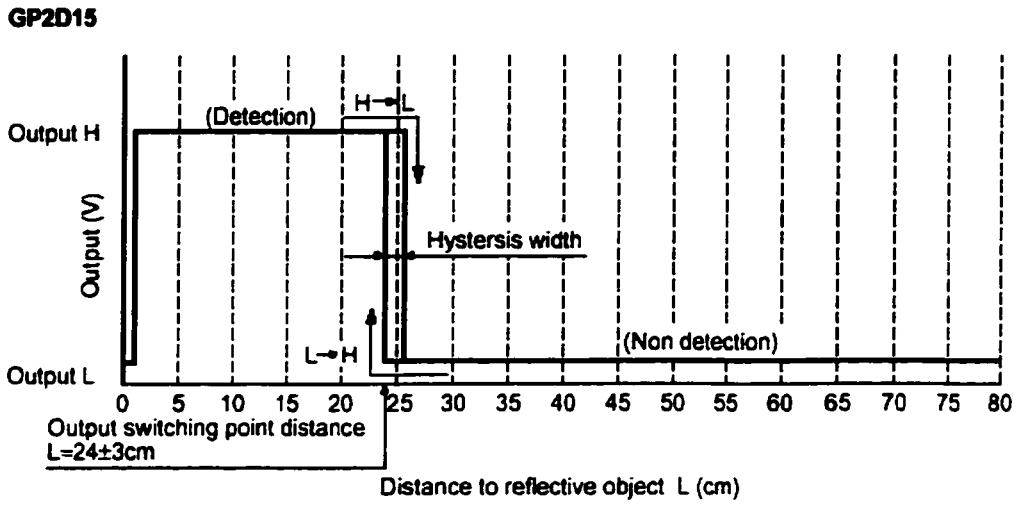

Figure 2.2: Contact sensor - distance characteristic

portion in the middle being about 16 cm wide.

On a real multi-sensor robot with overlapping beams, multi-sensor data fusion techniques are required and also sensor resolution impact on fuzzy controllers has to be taken into account [29]. Focusing more on the control algorithm, this thesis considers the information received from a sensor already pre-processed.

The simulated robot has been tested with various number of sensors, in an attempt to minimize this number. During experimental tests the following conclusions were summarized regarding the sensory system:

- Forward facing range sensors are required for collision avoidance.
- Range sensors with a 90° orientation relative to robot direction are required for a smooth contour-follow movement.

- Backward facing range sensors are required for turning corners when no other sensor has a valid indication.

- Contact sensors are required to compensate for the cases when range sensors fail to detect the obstacles, for example when the distance to an obstacle is below the minimum detection range, which is 10 cm for the Sharp sensor.

The sensory system of the robot is shown in Figure 2.3. It includes 9 range sensors and 4 contact sensors. In order to discover efficiently any potential collision, the forward facing sensors must cover a wide area, so they need different orientation angles relative to robot moving direction, and they need also the support of contact sensors. The size of the robot which has been simulated (70cm diameter) required three front sensors F1, F2, F3, parallel with the robot direction, and two oblique sensors (at 45°), O1 on the right side and O2 on the left side. These five sensors are considered forward facing range sensors. W1 and W2 have a 90° orientation relative to robot direction and they are named 'wall' sensors because their main goal is to help in following a contour, most often a wall. Sensors B1 and B2 are the back sensors or backward facing range sensors.

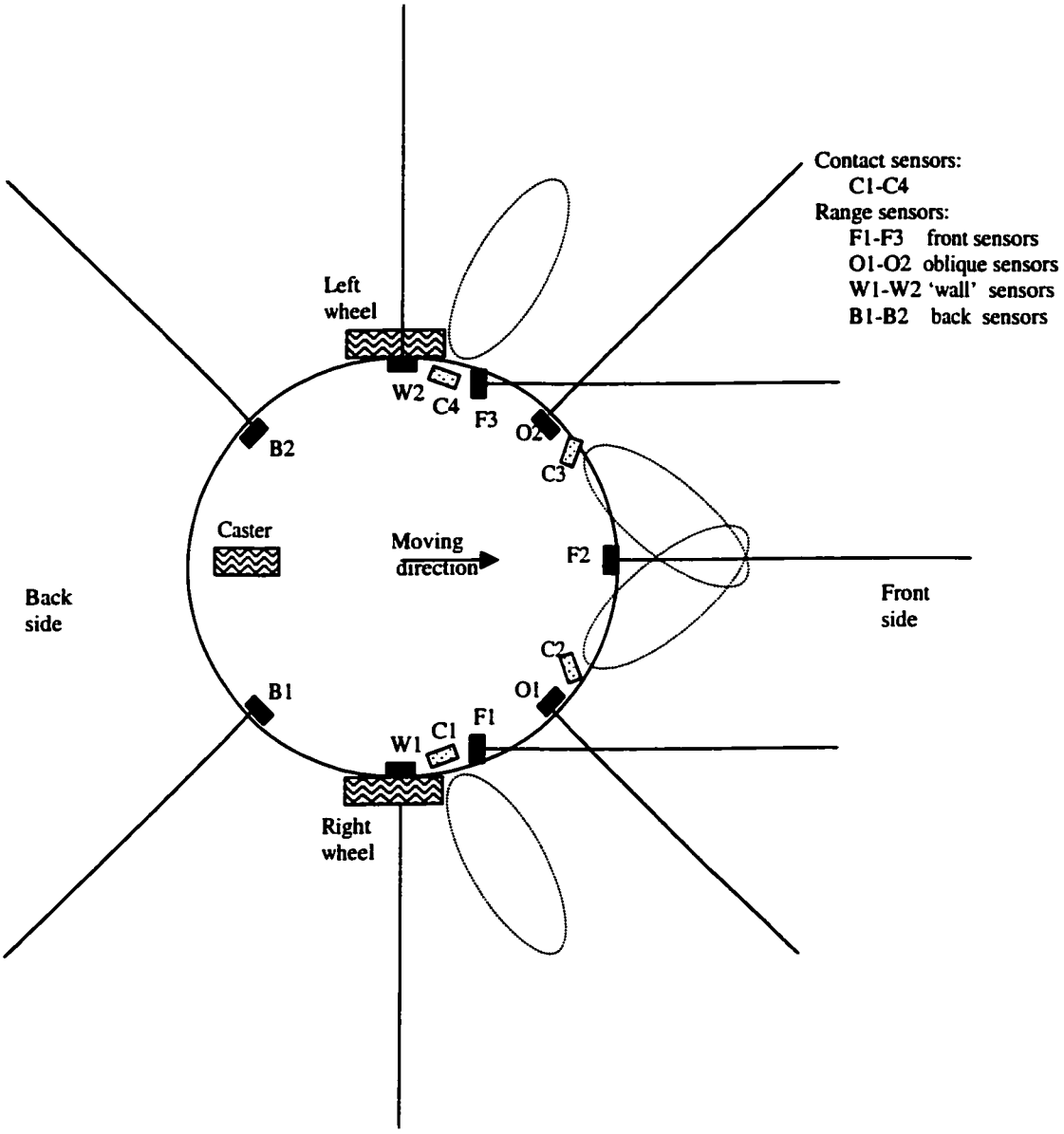

Figure 2.3: Sensory system of the mobile robot

Chapter 3

Neuro-fuzzy behavior control system

3.1 Why Neuro-Fuzzy for robot navigation

Robots are facing today very complex tasks to be executed, tasks which require that the uncertainty be accommodated by the control system of the robot. The complexity originates in the nature of the real-world, unstructured environments and in the large uncertainties that are inherent to these environments. Usually the prior information about the environment is incomplete, the maps may omit some details or become outdated, and, the real-world environment has unpredictable dynamics.

Fuzzy logic is particularly well suited for implementing navigation controllers, due to its capabilities of inference and approximate results under uncertainty. Proven advantages are robustness in the presence of system and external perturbations, ease of design and efficiency of knowledge representation. The conventional fuzzy logic controller (FLC) has been successfully used in a number of industrial plants and processes [16, 26, 9], and it is sometimes the more favorable controller even when classical controllers (PID and its variants) are applicable [12].

The few limitations imposed by fuzzy logic controllers can be easily overcome by

appropriate design. The potentially negative effect of large rule-bases on real-time performance is not anymore an issue in this thesis due to hierarchical architecture which divides the system in several smaller fuzzy subsystems. The enigma regarding FLC design in the absence of sufficient domain knowledge is resolved in this work by using neural networks and their capabilities for learning the parameters of the FLC. This is why the control system is called *adaptive fuzzy controller* or *neuro-fuzzy controller.*

A last issue to be considered when choosing a fuzzy logic controller versus a conventional controller is the stability of fuzzy systems. Fuzzy logic controllers can be seen as nonlinear controllers and it is difficult to obtain general results regarding their stability. However, this is not necessary a limitation. First, there is a considerable effort done in the direction of FLC stability analysis [17, 6, 39]. Second, several researchers (Mamdani, Jager) consider stability as a desirable, but optional step for acceptance of a control system [20, 13]. In his paper " *Twenty years of fuzzy control: experiences gained and lessons learnt*", Mamdani affirms clearly his opinions in favor of prototype testing as a necessary condition for control system acceptance. He considers stability analysis as not being a sufficient condition for this, and different ways have to be found to study stability. He was much criticized for these opinions, but they contain a lot of truth, building prototypes being a well tried and tested approach in the industry.

3.2 General architecture

The control system proposed for robot navigation has the purpose of moving the robot towards a target positioned at arbitrary relative coordinates. These coordinates can

be considered relative to the position of the robot and could be anywhere in the 2-dimensional space. The input variable of the system is the sensory data, filtered and with additional pre-processing which includes speed calculation towards an obstacle. The output variable of the system is the rotation angle to be executed by the robot. This angle, calculated at each control cycle, represents the command of the control system.

Currently, different definitions for the term *behavior* exist in the literature [22, 37, 2], all of them valid. The one which is closest to the concept used in this thesis is given by Matarić [22]:

> A *behavior* is a control law that satisfies a set of constraints to achieve and maintain a particular goal.

For reaching a XY location, the robot must perform tasks corresponding to behaviors of various complexity. The high-level behaviors are considered the more complex ones, which can be decomposed into lower-level, primitive behaviors, easier to be implemented. The behavioral architecture of a control system proposed in this thesis is presented in Figure 3.1. This architecture suites very well to a sensor-based model in which each primitive behavior is self-contained and reacts to valid indications from a certain type of sensors. Go-Tangent behavior reacts to forward facing sensors, Wall-Follow behavior to 'wall' sensors and Turn-Corner behavior to backward facing sensors.

A switching technique will be applied to chose between high-level behaviors, depending on the number of sensors offering valid input. For example, for no sensor indication Open-Space behavior will be selected, or, for sensor indications from all three sides (front, left and right side) a Dead-End behavior will be applied. If only

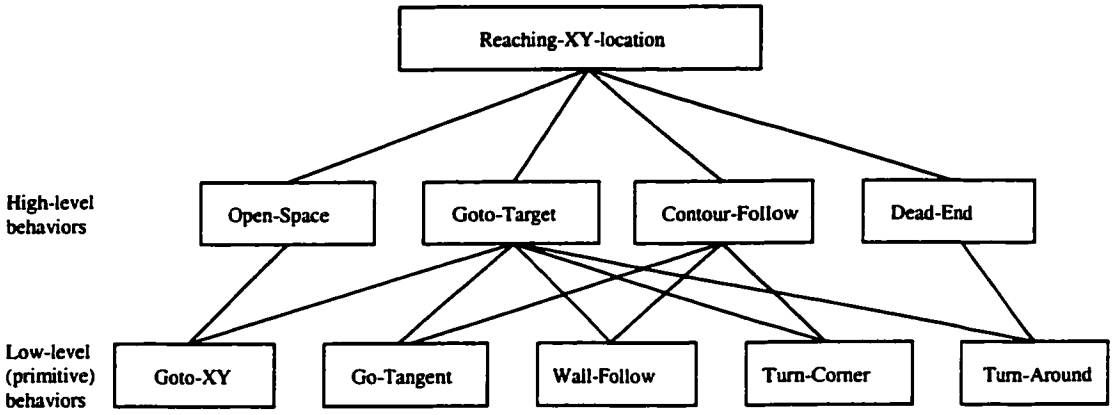

Figure 3.1: Behavioral architecture of the control system

sensor indications from one side are relevant, then Goto-Target behavior or Contour-Follow behavior will be selected. Choosing between these last two behaviors depends on the total rotation angle of the robot relative to the direction towards the target, so that over a certain threshold the robot is considered 'lost', and he is only allowed to follow a contour and not to go to a XY location, for the purpose of re-orienting the robot towards the target.

High-level behaviors are using a selected number of primitive behaviors. The lines in Figure 3.1 show which primitive behaviors could be applied within each high-level behavior. Fuzzy sub-systems are used to implement each primitive behavior. A neuro-fuzzy approach is used for the Go-Tangent primitive. The Turn-Around primitive is the only one which does not need a fuzzy implementation, the turn angle is hard coded to 180°. When multiple primitives must be applied, a command fusion technique is used to combine several fuzzy outputs into a single fuzzy output. The defuzzification process is applied only once, to this fuzzy output variable.

The idea of a fuzzy behavior hierarchy in control systems for robot navigation is well-known in the area of neuro-fuzzy robotics. E. Tunstel [38] and A. Saffiotti [33]

present similar concepts. As contributions to this area, in this thesis new primitive behaviors are defined (Go-Tangent and Turn-Corner), neural networks are used for training their fuzzy implementation, a new command fusion method between commands from several behaviors is proposed and the architecture is demonstrated in a very complex indoor environment.

Based on the above mentioned behavioral architecture, the neuro-fuzzy implementation is shown in Figure 3.2. Every control cycle (100 ms), the system re-calculates the turn angle to be executed by the robot, based on information from the sensory data and from its own dead-reckoning system.

The selected high-level behavior will fire only the appropriate primitive behaviors it relies on (see Figure 3.2). Because the primitive behaviors are implemented in Fuzzy Inference Systems (FIS), there is a reduction in the total number of rules to be consulted during the control cycle. The irrelevant behaviors - not belonging to the current selected high-level behavior - will not lengthen the total inference processing time.

The Very-Close FIS is not related to any of the robot behaviors. It is a characteristic of the sensory system and it does the opposite of all the other fuzzy inference systems do: it does NOT recommend a fuzzy turn angle, based on proximity (very close or not very close) to an obstacle. This fuzzy output is used only by the command fusion module to filter out eventual commands which might lead to collisions. As an example, the Goto-XY primitive could recommend a turn angle towards a target found beyond an obstacle.

The measure of the distance travelled since the previous time-step provided by the wheel encoders is commonly referred to as *dead-reckoning*. This is the single input for

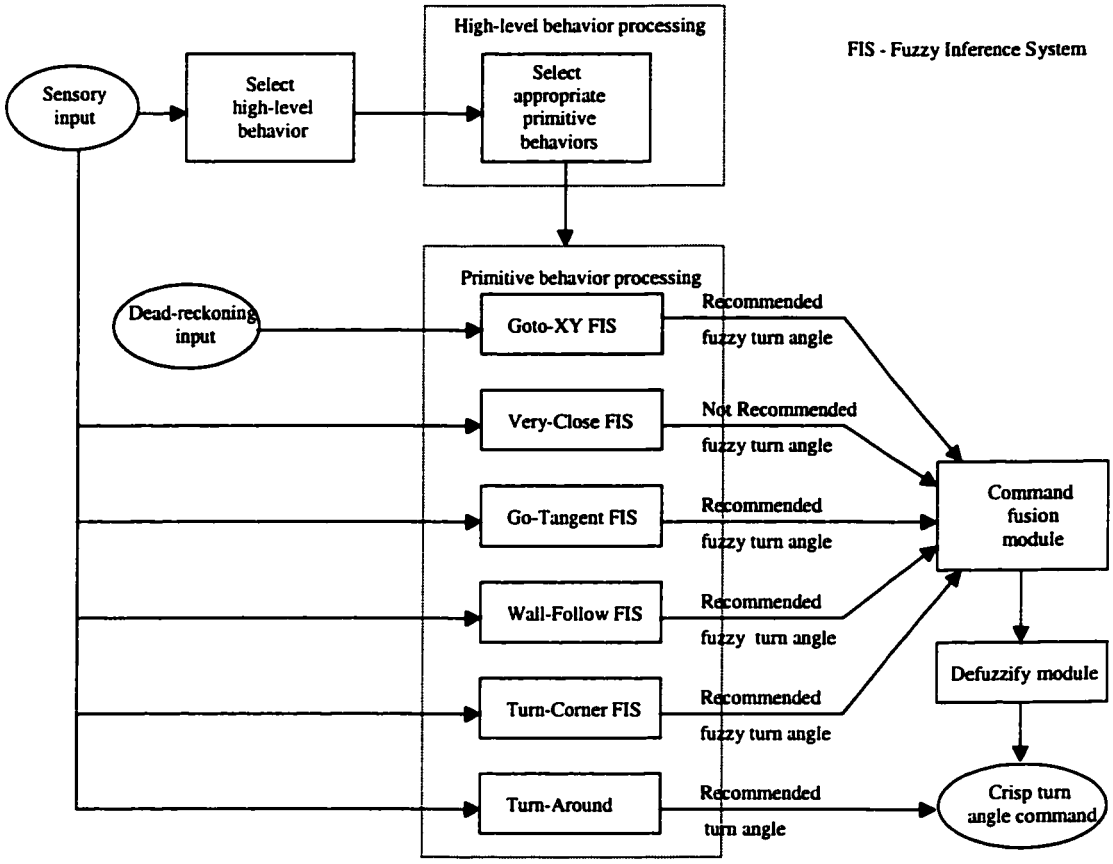

Figure 3.2: Neuro-fuzzy implementation of the control system

the Goto-XY primitive. Because dead-reckoning is a source of error and uncertainty (e.g. wheel slippage or blockage), the computed angle towards target is considered as the input into a fuzzy inference system. This FIS maps the output value to a fuzzy set for the turn angle.

As a result of the defuzzification process, a crisp output value is obtained. Usually the defuzzification is the next step after fuzzy inference. In the case of multiple behaviors fired simultaneously, the multiple outputs yielded cannot be aggregated immediately due to a potential conflict between different outputs. Using the same example, Goto-XY recommends the turn angle α while Very-Close does NOT recommend α. Therefore a command fusion module dealing with this problem is required. The defuzzification process is deferred until this module executes the appropriate operation between the output fuzzy sets, solving potential conflicts and combining the outputs of different fuzzy inference systems. It can be said that recommendations from different expert systems are taken into account.

For a larger number of fuzzy variables (our case), this decentralized behavior-based control architecture is preferred to a monolithic control architecture. It is well known that the number of rules to be consulted for a monolithic rule base grows exponentially with the number of variables [3], implying a heavier computational load for the system.

3.3 Fuzzy logic control theory

In the case of the system proposed in this thesis, most of the behavior implementations require fuzzy logic approaches. A short introduction to fuzzy logic theory will be presented here. The rest of the chapters rely mainly on these fuzzy logic concepts.

3.3.1 Fuzzy sets

The very basic notion of fuzzy logic is a *fuzzy set.* In mathematics, usually the notion of a *crisp set* is much more familiar: the membership $\mu_A(x)$ of x for a crisp set A, as subset of the universe X, is defined by:

$$\mu_A(x) = \begin{cases} 1 & \text{if } x \in A; \\ 0 & \text{if } x \notin A. \end{cases} \tag{3.3.1}$$

Therefore an element x is either member of the crisp set A or not.

In practice there are many situations where the truth is not so crisp for the membership of a value x. Some classical examples: temperature is hot or not, a price for a given item is expensive or not. A fuzzy set allows a more flexible classification than a crisp set.

A fuzzy set assigns a degree of membership between 0 and 1 to a crisp value x using a *membership function.* For a fuzzy set $\tilde{A}$, such a function is denoted by $\mu_{\tilde{A}}(x)$:

$$\mu_{\tilde{A}}(x) : X \rightarrow [0, 1] \tag{3.3.2}$$

The fuzzy set $\tilde{A}$ is a crisp set of ordered pairs of $x \in X$ and $\mu_{\tilde{A}}(x)$:

$$\tilde{A} = \{(x, \mu_{\tilde{A}}(x)) | x \in X\} \tag{3.3.3}$$

For a *linguistic variable* like `temperature`, examples of fuzzy sets are `hot`, `warm` and `cold` (*linguistic values*), and their associated membership functions. A crisp temperature of 40° Celsius can be considered using the membership functions as 60% hot, 40% warm and 0% cold (the sum of the percentages not being necessarily 100%)

The definition of a fuzzy set can be also seen in the literature as Lotfi Zadeh [41] defined it for the first time ($\sum$ having the meaning of a countable enumeration):

$$\tilde{A} = \sum \mu_A(x_i)/x_i \tag{3.3.4}$$

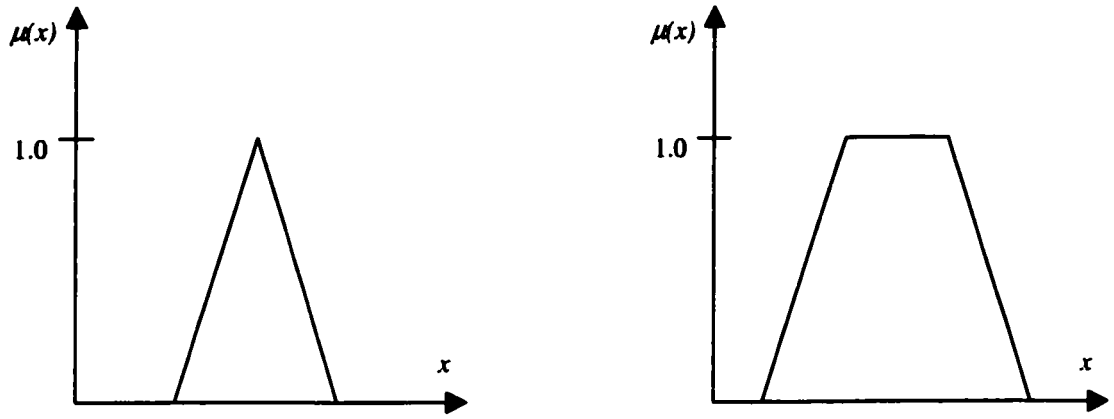

Figure 3.3: Typical membership functions used in fuzzy logic control

Typical membership functions used in fuzzy logic control have the triangular and trapezoidal shapes (Figure 3.3). Other forms could be the Gaussian distribution function, the generalized bell or the sigmoid curve. The choice of the triangular and trapezoidal functions is highly recommended for embedded real-time applications where these functions can be evaluated fast.

Fuzzy union, fuzzy intersection and fuzzy complement can be defined as operation on fuzzy sets. The extension of these operations from the classical sets is not uniquely defined. Usually T-norm operators for intersection and T-conorm (or S-norm) operators for union are defined. These operators have in common the property of monotonicity, commutativity and associativity, but have different properties of boundary:

$$T(0,0) = 0, \quad T(a,1) = T(1,a) = a \tag{3.3.5}$$

$$S(1,1) = 1, \quad S(a,0) = S(0,a) = a \tag{3.3.6}$$

Zadeh [41] proposed to use the following definitions (used also in this thesis) for intersection (3.3.7), union (3.3.8) and complement (3.3.9):

$$\mu_{\tilde{A}\cap\tilde{B}}(x) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x)), \tag{3.3.7}$$

$$\mu_{\tilde{A}\cup\tilde{B}}(x) = \max(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x)), \tag{3.3.8}$$

$$\mu_{\overline{A}}(x) = 1 - \mu_{\tilde{A}}(x), \tag{3.3.9}$$

where $\overline{A}$ is the fuzzy complement set.

3.3.2 Fuzzy reasoning process

A Fuzzy Logic Controller (FLC) implies three processing stages, as it can be seen in Figure 3.4.

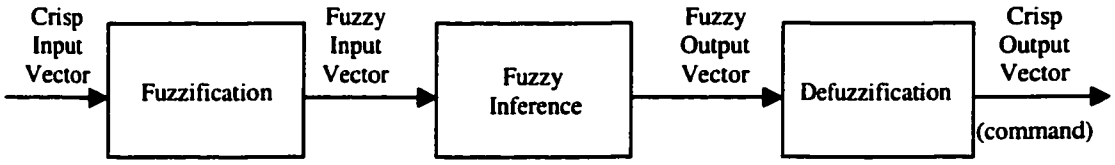

Figure 3.4: Processing stages of a fuzzy logic controller

The first stage is *fuzzification* which translates the input vector of given crisp values into a fuzzy input variable. This is done by evaluating the membership functions of each linguistic input variable.

The next stage is performed inside the *Fuzzy Inference System* (FIS) which relies on a set of if-then rules with linguistic variables as antecedents and consequents. The general form of such a rule is:

$$IF\ x\ is\ \tilde{A}_i\ THEN\ u\ is\ \tilde{B}_i \tag{3.3.10}$$

where x and u represent input and output linguistic variables in a rule-base of size N and $\tilde{A}_i$ and $\tilde{B}_i (i = 1, 2, ...N)$ are fuzzy sets representing linguistic values of x and u.

The inference process implies a rule firing strength evaluation, a implication function calculation and an aggregation process. We will examine each of them.

Let us consider a rule with m propositions in the antecedent, connected with fuzzy operators. A *rule firing strength* is calculated as:

$$\rho_i = \min_j \mu_{\tilde{A}_{ij}}(x_j) \quad j = 1, 2, ..., m \tag{3.3.11}$$

in case of the fuzzy intersection operator from (3.3.7), or:

$$\rho_i = \max_j \mu_{\tilde{A}_{ij}}(x_j) \quad j = 1, 2, ..., m \tag{3.3.12}$$

in case of the fuzzy union operator from (3.3.8).

The control algorithm is given by the set of fuzzy rules. Each rule is represented by an *implication function* with the rule firing strength as input and a fuzzy set for the command u as output.

There are many possible implication functions, but one of the most used in fuzzy logic control (and also used in this thesis) is the Mamdani-type implication function *min* [19], which can be written as:

$$\tilde{u}_i(u) = \min(\rho_i, \tilde{B}_i) \tag{3.3.13}$$

The case of a single proposition in the consequent was considered because rules with multiple propositions in the consequent are assumed to be separable into distinct rules.

The *aggregation* is the process by which the fuzzy sets from different rules are combined into a single output fuzzy set, $\tilde{U} \in U$. Fuzzy operators like *max* can be

used for this:

$$\tilde{U} = \max_i(\tilde{u}_i(u)) \tag{3.3.14}$$

When equations (3.3.14) and (3.3.13) are used for aggregation and implication respectively, the inference method is known as the *max-min* method. This method is well suited for inference systems in fuzzy control due to its minimum computational overhead, therefore it has been used in this work. Other methods are also known: *max-prod* method, *sum-prod* method, where the first part denotes the aggregation operator used, and the second part the implication operator.

The FIS built on rules having the form described by equation (3.3.10) is called a Mamdani-type system. Another type of inference system used in this thesis is the Sugeno-type system. This is how a Sugeno fuzzy rule base of size N is defined, for two inputs and one output, without loss of generality:

$$IF\ x_1\ is\ \tilde{A}_{1,i}\ AND\ x_2\ is\ \tilde{A}_{2,i}\ THEN\ u_i\ is\ f_i(x_1, x_2) \quad i = 1, 2, ..., N \tag{3.3.15}$$

In this definition an *AND* notation was used as a fuzzy intersection.

A zero-order Sugeno fuzzy system has the consequent defined as a constant: $f_i(x_1, x_2) = const_i$. In this case, the system resembles with a Mamdani-type system, having the output membership functions as singleton spikes and the aggregation function including all the singletons.

A first-order Sugeno fuzzy system has the consequents as linear functions of input variables:

$$f_i(x_1, x_2) = p_i \cdot (x_1) + q_i \cdot (x_2) + r_i \tag{3.3.16}$$

In section 4.2 this type of inference system will be detailed.

The last stage (Figure 3.4) is the *defuzzification* process which translates the fuzzy output of a fuzzy controller into a crisp numerical representation. Defuzzification

is done using an averaging technique. Various methods are known: center-of-area, center-of-sums, mean-of-maxima [13]. In this thesis the center-of-area method is used. Considering the output fuzzy set $\tilde{U}$, and the discrete universe of possible crisp commands $U = \{u_1, u_2, ...u_M\}$, then the result of the defuzzification process is:

$$c = \frac{\sum_{k=1}^{M} u_k \cdot \mu_{\tilde{U}}(u_k)}{\sum_{k=1}^{M} \mu_{\tilde{U}}(u_k)} \tag{3.3.17}$$

where $\mu_{\tilde{U}}(u_k)$ is calculated as:

$$\mu_{\tilde{U}}(u_k) = \max_i \mu_{\tilde{u}_i}(u_k) \quad i = 1, 2, ..., N \tag{3.3.18}$$

It is well-known that defuzzification is computational intensive, especially for Mamdani-type systems which require integration across the two-dimensional function to find the centroid. The Matlab package used in this thesis employs a weighted average method of a few data points to obtain the centroid, so that the center-of-area method can be used without significant real-time penalty.

Chapter 4

Low-level behaviors

The low-level behaviors, or primitive behaviors, are implemented as fuzzy logic controllers. They have been designed using Matlab 6.0 which supports both types of Fuzzy Inference Systems described in §3.3, Mamdani-type FIS and Sugeno-type FIS. A description of each behavior follows, specifying the fuzzy inference type used by each of them.

4.1 Goto-XY behavior

This behavior allows the robot to move towards the target. Because the sensory system could indicate an obstacle between the robot position and the target point, the Goto-XY behavior is implemented with two Mamdani-type fuzzy inference systems:

1. Goto-XY FIS which does the conversion of the turn angle towards the target into a fuzzy set.

2. Very-Close FIS which calculates the turn angle not recommended due to potential collisions.

A T-norm operator (*min*) will be applied between the two output fuzzy sets to obtain the resultant fuzzy set. This operation is performed in the command fusion module, see section 5.5.

4.1.1 Goto-XY Fuzzy Inference System

This fuzzy inference system has one input, the orientation angle towards the target (Figure 4.1) and one output, the rotation angle as a fuzzy set (Figure 4.2). Both variables use triangular membership functions of the same shape, with a scope of 45 degrees.

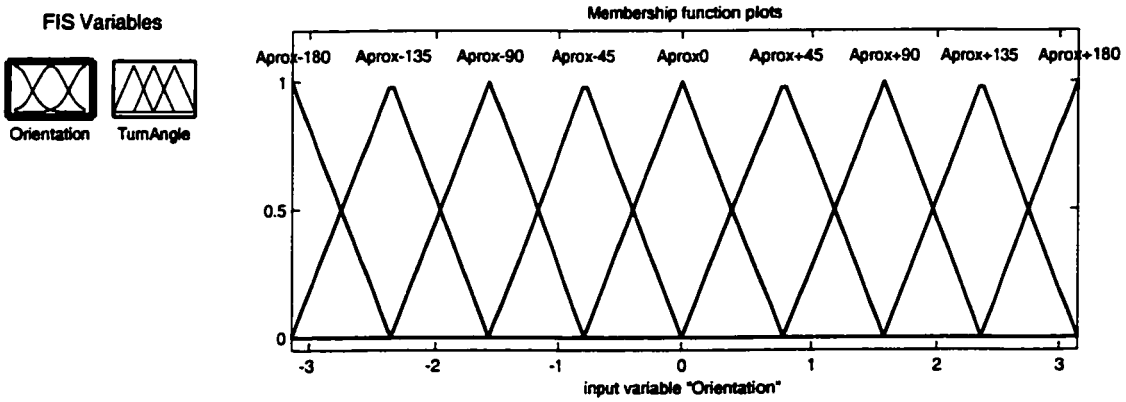

Figure 4.1: Goto-XY FIS : Input variable

Once the fuzzification of the crisp input (orientation angle) is finished, the fuzzy inference plays the role of identity function from algebra ($f(x) = x$), converting a fuzzy input set into an output fuzzy set of the same value. This can be seen from the fuzzy rules defined in Table 4.1.

There are some benefits of working with both fuzzy sets (input and output):

- the same implementation as for the other behaviors;

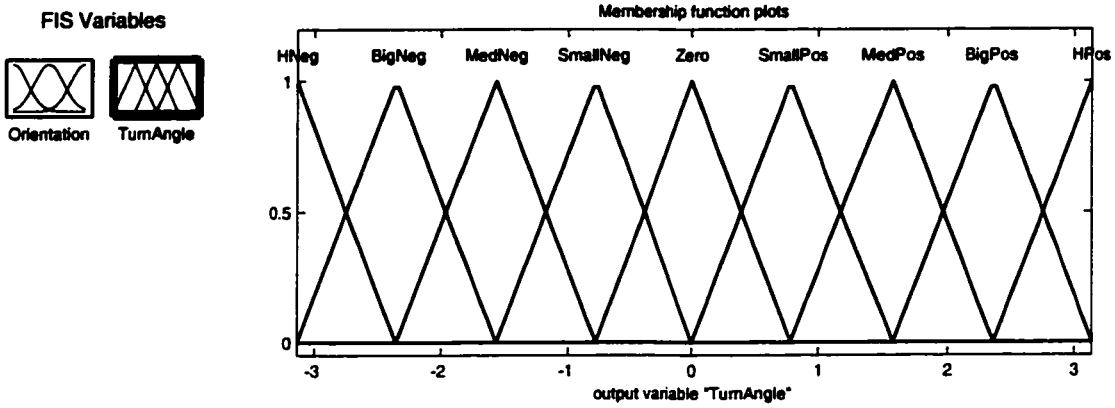

Figure 4.2: Goto-XY FIS : Output variable

No.	Rule definition
1	If (Orientation is Aprox-180) then (TurnAngle is HNeg)
2	If (Orientation is Aprox-135) then (TurnAngle is BigNeg)
3	If (Orientation is Aprox-90) then (TurnAngle is MedNeg)
4	If (Orientation is Aprox-45) then (TurnAngle is SmallNeg)
5	If (Orientation is Aprox0) then (TurnAngle is Zero)
6	If (Orientation is Aprox+45) then (TurnAngle is SmallPos)
7	If (Orientation is Aprox+90) then (TurnAngle is MedPos)
8	If (Orientation is Aprox+135) then (TurnAngle is BigPos)
9	If (Orientation is Aprox+180) then (TurnAngle is HPos)

Table 4.1: Goto-XY FIS : Fuzzy rules

- possible membership function adjustment for the the output in the testing phase;
- the computational time is the same whether fuzzy inference works with the `orientation` fuzzy set or `turnAngle` fuzzy set.

4.1.2 Very-Close Fuzzy Inference System

This is a 9-input/1-output fuzzy inference system. Each sensor range represents an input variable named after the sensor's position - see Figure 2.3. These variables

have associated only one membership function (**VeryClose**) which shows the degree to which a robot is close to an obstacle in the direction pointed by a given sensor. All membership functions are identically defined for every input, as seen in Figure 4.3 for sensor **Front2**.

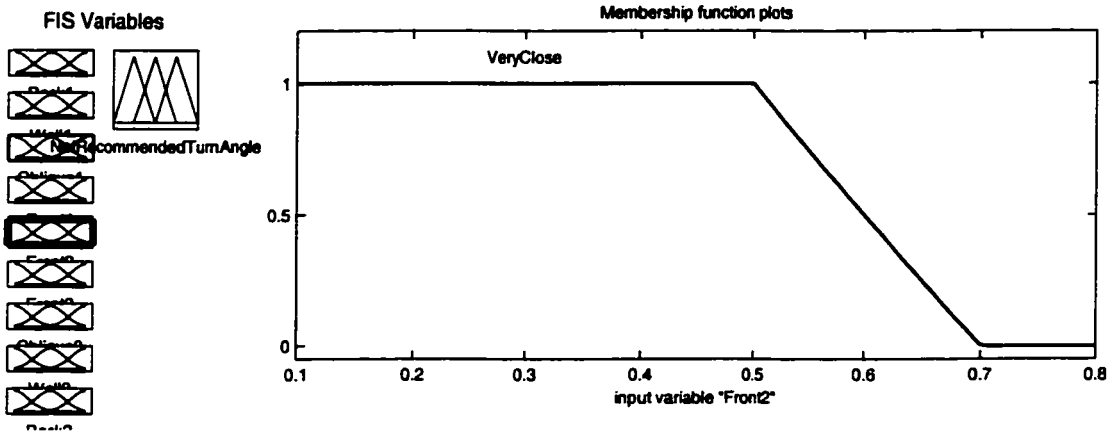

Figure 4.3: Very-Close FIS : Input variable

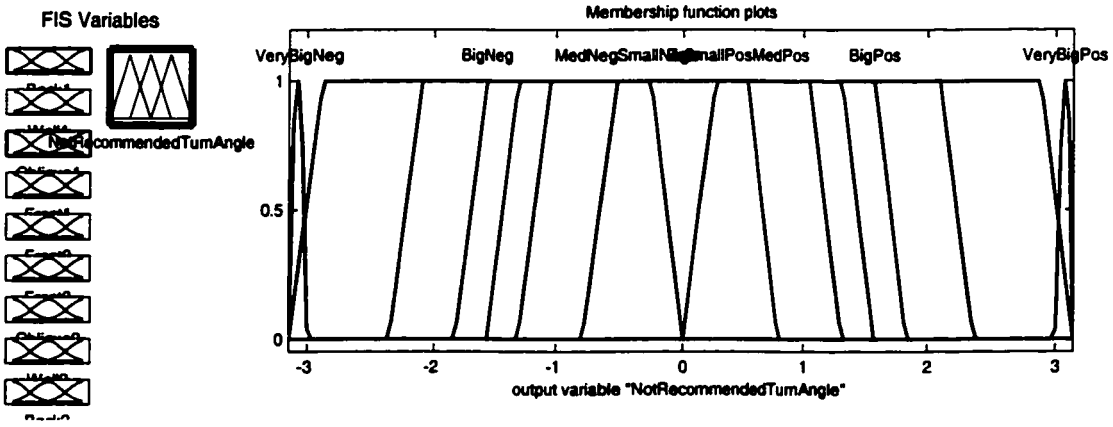

Figure 4.4: Very-Close FIS : Output variable

The output variable represents the 'Not Recommended Turn Angle' for the robot (Figure 4.4). This variable is described by 9 linguistic values or fuzzy sets, so that

each input is mapped to one single output fuzzy set using a fuzzy rule as shown in Table 4.2. The 9 membership functions of trapezoidal shape overlap significantly due to the fact that the scope of each fuzzy set covers 180 degrees. This value has been determined through experiments, but clearly it has a theoretical justification also: if there is an obstacle seen by a robot sensor in a certain direction, then there is a plane which separates the unsafe directions (leading to potential collisions) from the safe directions to be followed by the robot. This plane passes through the center of the robot, it is perpendicular to the direction pointed by the sensor, and it imposes the 180 degrees value as the scope of each membership function.

No.	Rule definition
1	If (Back1 is VeryClose) then (NotRecommendedTurnAngle is VeryBigNeg)
2	If (Wall1 is VeryClose) then (NotRecommendedTurnAngle is BigNeg)
3	If (Oblique1 is VeryClose) then (NotRecommendedTurnAngle is MedNeg)
4	If (Front1 is VeryClose) then (NotRecommendedTurnAngle is SmallNeg)
5	If (Front2 is VeryClose) then (NotRecommendedTurnAngle is Zero)
6	If (Front3 is VeryClose) then (NotRecommendedTurnAngle is SmallPos)
7	If (Oblique2 is VeryClose) then (NotRecommendedTurnAngle is MedPos)
8	If (Wall2 is VeryClose) then (NotRecommendedTurnAngle is BigPos)
9	If (Back2 is VeryClose) then (NotRecommendedTurnAngle is VeryBigPos)

Table 4.2: VeryClose FIS : Fuzzy rules

After the aggregation process, the resultant fuzzy set can be seen in Figure 4.5. This figure, captured with Matlab application, shows the result of the defuzzification process too, but this will not be used in our control system. Instead, the command fusion module will use the resultant fuzzy output as a direction not recommended, so it will apply a fuzzy complement operator (*not*) to this fuzzy set and then a *min* operator to the resultant fuzzy set and the Goto-XY output fuzzy set.

VeryClose FIS uses membership functions of trapezoidal shape for the purpose

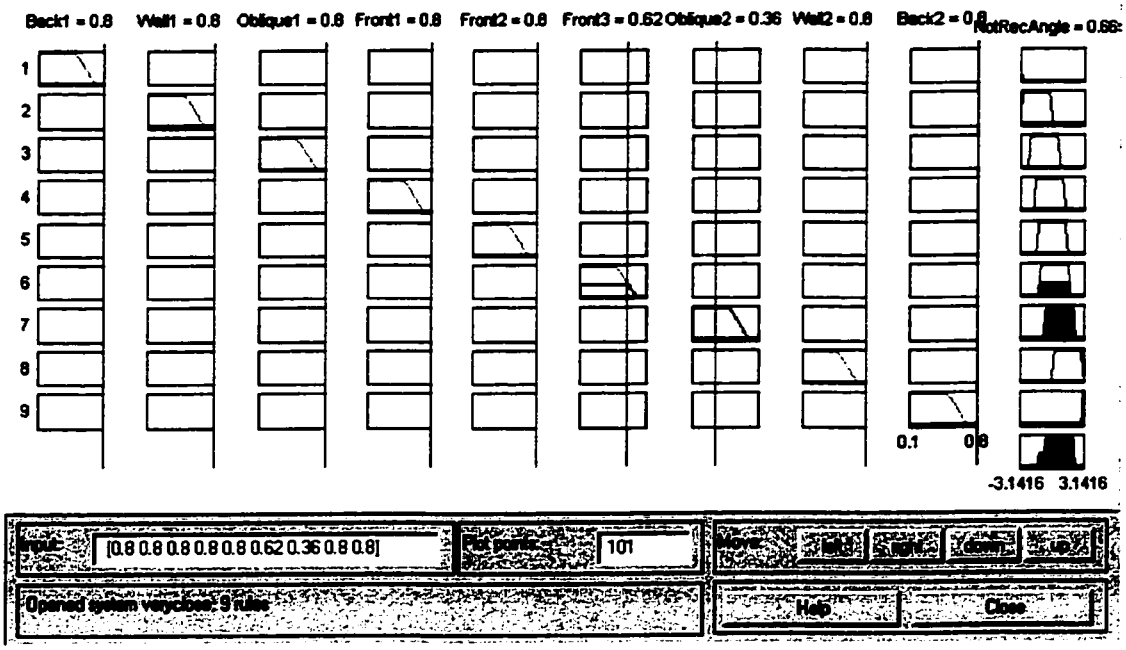

Figure 4.5: Very-Close FIS : Rule evaluation for a given input vector

of completely filtering out the Goto-XY fuzzy set in the vicinity of the direction towards the obstacle. This is achieved by a degree of membership equal with 1 for a continuous interval and not only in a single point as in the case of triangular membership functions.

4.2 Go-Tangent behavior

A key point in a successful robot navigation solution is the degree of smoothness in robot trajectory. Several researchers of the behavior-based robot navigation field [38, 40] were interested in solving the collision avoidance problem together with attaining a smooth trajectory of the robot when travelling between two endpoints. In this thesis both avoiding obstacles and obtaining a high degree of smoothness in robot trajectory are considered primary objectives. A new behavior is proposed here which, if correctly

implemented, may ease up the task of smooth navigation.

Go-Tangent behavior has the role of calculating a turn angle which places the robot on a trajectory parallel with the surface of the obstacle reflecting the sensor signal. Although the obstacle could have an irregular shape, its surface which reflected the sensor signal can be considered in the proximity of the reflection point as a vertical plane. The angle between this plane and the current robot direction is the angle required for steering so that the robot will enter in a trajectory parallel with this reflection plane. Because the reflection plane can be assumed tangent locally to the irregular surface of the obstacle, the behavior has been named 'Go-Tangent'.

There are two cases in which this angle can be computed:

1. There is sensory data from one single sensor. In this case, this sensor should be oblique relative to robot direction so that two valid indications (distances) obtained in two consecutive control cycles allow us to compute the speed towards the obstacle. Based on this speed and the current distance to the obstacle, a fuzzy inference system (Go-Tangent-Oblique-Sensor) is capable of calculating the required turn angle.

2. There is sensory data from two sensors. In this case, the sensors should be oriented parallel to the same direction, and their simultaneous indication is enough to calculate the turn angle. Another fuzzy inference engine (Go-Tangent-Front-Sensor) has been used for this calculation, using the range indication from sensors `Front1` and `Front2` (Figure 2.3) for the right side of the robot, and `Front2` and `Front3` for the left side.

Instead of using the fuzzy inference systems, an analytical model could have been established so that based on the two input values, the model calculates a crisp turn

angle. But several reasons led to a fuzzy approach versus an analytical model:

- first, the implementation of such an analytical model is more time consuming than the design of a fuzzy system using on-the-shelf tools;
- second, modelling the fuzzy parameters with neural networks and collecting the training data sets with a 'point-and-click' technique described in the experimental part of this work, provided very accurate fuzzy models;
- third, the imperfections of the sensory data are better accommodated in fuzzy systems than in analytical models: fuzzy models can be tuned-up until the output surface has a desired shape for the entire input space.

4.2.1 Neuro-adaptive learning algorithm

When there is limited knowledge regarding the desired behavior of a fuzzy system, as in the case of Go-Tangent primitive, the task of choosing the parameters of the membership functions is a difficult one. This is where the neuro-adaptive learning techniques fit very well and can provide a better model for the fuzzy inference system.

The parameters of the membership functions are chosen so as to tailor these functions to a given input/output data in order to account for the variations in the data values. The data used for modelling is also called the *training data set* or *learning data set.* The parameters for the membership functions are computed on the basis of a learning method applied on a training data set.

The learning method used in this thesis is ANFIS (Adaptive Neuro-Fuzzy Inference System), a method presented for the first time by Jang [14] and implemented in Matlab application.

This learning procedure is based on a hybrid technique: Least- Square Estimation (LSE) technique and backpropagation technique. A brief introduction to these two methods follow.

Least-square estimation method is applicable to a linear system $y(t)$ of the form:

$$y(t) = f_1(u(t))\theta_1 + f_2(u(t))\theta_2 + ...f_n(u(t))\theta_n \tag{4.2.1}$$

where $f_1, ...f_n$ are known functions of input vector $u(t)$ and $\theta_1, ..., \theta_n$ are parameters to be estimated. With the notations:

$$\varphi(t) = [f_1(u(t))\ f_2(u(t))\ ...\ f_n(u(t))]^T \tag{4.2.2}$$

$$\theta = [\theta_1\ \theta_2\ ...\ \theta_n]^T \tag{4.2.3}$$

the system is:

$$y(t) = \varphi^T(t) \cdot \theta \tag{4.2.4}$$

For a set of P measurements of the input/output data $\{(u(t), y(t))|t = 1, ..., P\}$, the estimation $\hat{\theta}$ is by definition:

$$\hat{\theta} = \arg\min_{\theta} V(\theta)\ ; \quad V(\theta) = \sum_{t=1}^{P}[y(t) - \varphi^T(t) \cdot \theta]^2 \tag{4.2.5}$$

Expanding $V(\theta)$ we obtain:

$$\begin{aligned} V(\theta) &= \sum_{t=1}^{P} y^2(t) - 2\left[\sum_{t=1}^{P} y(t)\varphi^T(t)\right]\theta + \sum_{t=1}^{P}[\varphi^T(t)\theta]^2 = \\ &= \sum_{t=1}^{P} y^2(t) - 2\,\theta^T\left[\sum_{t=1}^{P}\varphi(t)y(t)\right] + \theta^T\left[\sum_{t=1}^{P}\varphi(t)\varphi^T(t)\right]\theta \end{aligned}$$

$V(\theta)$ being a scalar function, its local minimum is reached whenever $\hat{\theta}$ satisfies the equation $\frac{\partial V(\theta)}{\partial \theta} = 0$, or:

$$-2\left[\sum_{t=1}^{P}\varphi(t)y(t)\right] + 2\left[\sum_{t=1}^{P}\varphi(t)\varphi^T(t)\right]\hat{\theta} = 0 \tag{4.2.6}$$

So the general form of the least-square estimator is:

$$\hat{\theta} = \left[\sum_{t=1}^{P} \varphi(t)\varphi^T(t)\right]^{-1} \left[\sum_{t=1}^{P} \varphi(t)y(t)\right] \tag{4.2.7}$$

Backpropagation learning method is a parameter identification procedure based on the simple steepest descent method in which the gradient vector is calculated in the direction opposite to the flow of the output of each node. The gradient vector is defined as the derivative of the error measure with respect to each parameter to be identified [15]. If we consider a neural network of K nodes with a single output node, this is how a generic parameter α can be updated after each epoch of a training set of P measurements:

$$\triangle\alpha = -\eta\frac{\partial E}{\partial \alpha} = -\eta\sum_{p=1}^{P}\frac{\partial E_p}{\partial \alpha} \tag{4.2.8}$$

where η is a factor called *learning rate* and E_p is the output error of the last node (Kth) for a measurement p. The derivative $\frac{\partial E_p}{\partial \alpha}$ is calculated based on the error signal of the node which depends on the parameter α. More details can be read in Appendix A.

Let us see how these two techniques can be used for modelling the fuzzy inference systems for Go-Tangent behavior. If we take into account the equation (3.3.16), a general fuzzy rule for a 2-input/1-output first-order Sugeno fuzzy model can be written as:

$$IF\ x\ is\ A_i\ AND\ y\ is\ B_j\ THEN\ F_k = p_k x + q_k y + r_k, \tag{4.2.9}$$

$$i = 1, ..., L; \quad j = 1, ..., M; \quad k = 1, ..., N; \quad N = L \times M$$

where x and y are the two inputs (linguistic variables), F_k is the output for the k-th rule, L is the size of the fuzzy set A, M is the size of the fuzzy set B, and N is the

size of the rule base. For the Go-Tangent-Oblique-Sensor FIS, x is the sensor range with the corresponding fuzzy set A as {**short**, **medium**, **long**}, y is the sensor speed towards the obstacle with the corresponding fuzzy set B as {**slow**, **medium**, **fast**}, F_k is the turn angle for the k-th rule, and $N = 9$, the size of the rule base. Based on equation (4.2.9), we will describe further only this fuzzy inference system, the other FIS used in this behavior (Go-Tangent-Front-Sensor) having a similar description, with the single difference that the inputs x and y are the two range values from the two sensors respectively.

During experimental tests, several types of membership functions for fuzzy sets A and B have been tried (see Chapter 6), with various sizes for L and M. The case of triangular membership functions and a size of 3 for the fuzzy set ($L = M = 3$) were identified as not only the simplest, but also the best suited to model the given training data set. With the notations from Figure 4.6, a general formula for such a function is given by:

$$\mu_{ij}(x_j) = \begin{cases} 1 - \frac{2|x_j - a_{ij}|}{b_{ij}} & , \text{ for } a_{ij} - \frac{b_{ij}}{2} < x_j < a_{ij} + \frac{b_{ij}}{2}; \\ 0 & , \text{ otherwise.} \end{cases} \tag{4.2.10}$$

where we took advantage of the fact that both fuzzy sets A and B have the same size ($L = M = 3$), so instead of having two functions $\mu_{A_i}(x)$ and $\mu_{B_i}(y)$, we considered one single definition $\mu_{ij}(x_j)$ where $i = 1, 2, 3$ and $j = 1$ (for x) or j = 2 (for y).

Figure 4.7 describes the structure of the neural network used for identifying the set of parameters $\{a_{ij}, b_{ij}\}$ for the equation (4.2.10) and the set of parameters $\{p_k, q_k, r_k\}$ for the equation (4.2.9). The former set is known as *premise parameters set* and

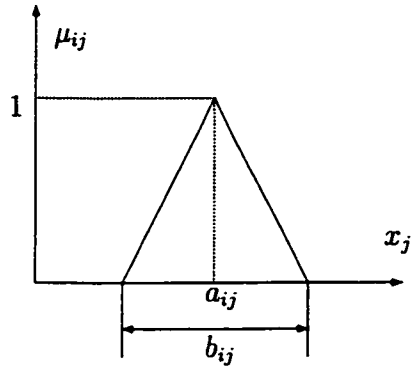

Figure 4.6: General form of MF for Go-Tangent fuzzy inference systems

the latter as *consequent parameters set.* In order to show the fact that fuzzy sets of input variable should not be necessarily of the same size, the initial distinct notations for the fuzzy sets (A and B) has been kept.

Squares were used to designate adaptive nodes which depend on the parameter set of the adaptive network. Circles were used to represent fixed nodes, or nodes which have a function independent of the parameter set.

The first layer of this neural network is composed of adaptive nodes represented by the membership functions associated with each linguistic value. These functions are described by equation (4.2.10).

The second layer implements the fuzzy rules. It includes only fixed nodes doing a product operation Π between the two inputs ($\mu(x)$ and $\mu(y)$) corresponding to the two propositions in the antecedent of each fuzzy rule. The output of each node is noted W_k.

The third layer is represented by another set of adaptive nodes, which include the output membership functions F_k. These nodes compute the product $W_k F_k$.

Two other layers of fixed nodes implement the weighted average procedure for

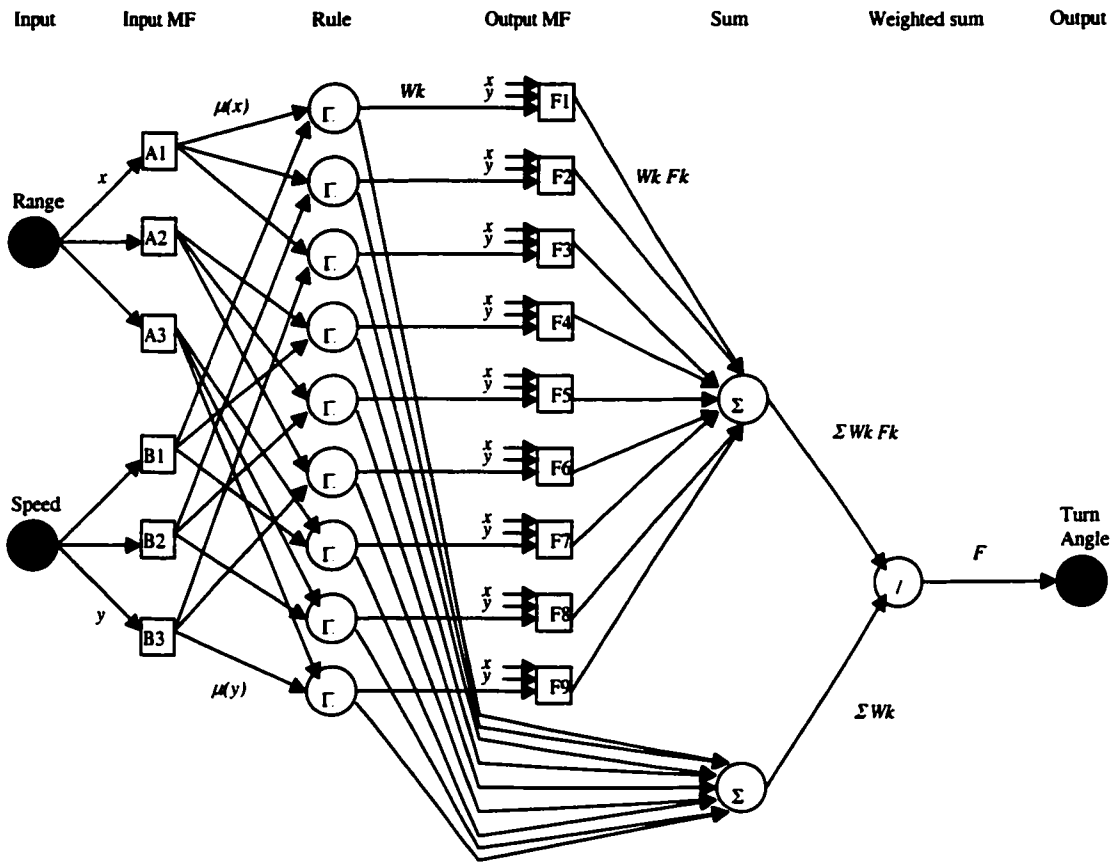

Figure 4.7: Neural network structure used for training goTangent FIS

calculating F, the output of the neural network. This output value can be expressed as a function of the inputs x_j:

$$\begin{aligned} F &= \frac{\sum_{k=1}^{N} W_k F_k}{\sum_{k=1}^{N} W_k} = \sum_{k=1}^{N} \frac{W_k}{\sum_{k=1}^{N} W_k}(p_k x + q_k y + r_k) = \\ &= \sum_{k=1}^{N} (\overline{W}_k x) p_k + \sum_{k=1}^{N} (\overline{W}_k y) q_k + \sum_{k=1}^{N} \overline{W}_k r_k \end{aligned} \tag{4.2.11}$$

where $\overline{W}_k = \frac{W_k}{\sum_{k=1}^{N} W_k}$. This equation is linear in p_k, q_k and r_k, and it is in the same form as equation (4.2.1) for $n = 3N$. So we have 27 consequent parameters $\{p_1, ..., p_9, q_1, ..., q_9, r_1, ...r_9\}$ which can be identified with the least square estimator

from equation (4.2.7) for a training data set $\{x, y, F\}$ of size P, and for premise parameters $\{a_{ij}, b_{ij}\}$ considered fixed.

The hybrid learning algorithm proposes the identification of the parameters in two steps: In the forward pass, the input membership functions are fixed and the consequent parameters (associated with the output membership functions) are determined with the least square estimation method. With these parameters, the FIS model delivers a certain calculated output $\hat{\alpha}$ for the turn angle. The difference between $\hat{\alpha}$ and the desired value α for the turn angle (Figure 4.8) represents the error signal. This error signal is propagated backward in a second pass when the premise parameters (associated with the input membership functions) can be calculated with a backpropagation technique.

4.2.2 Go-Tangent-Oblique-Sensor Fuzzy Inference System

The Go-Tangent-Oblique-Sensor fuzzy inference system is a Sugeno-type FIS which has two inputs, range and speed towards obstacle, and one output, the turn angle α to be executed by the robot (Figure 4.8).

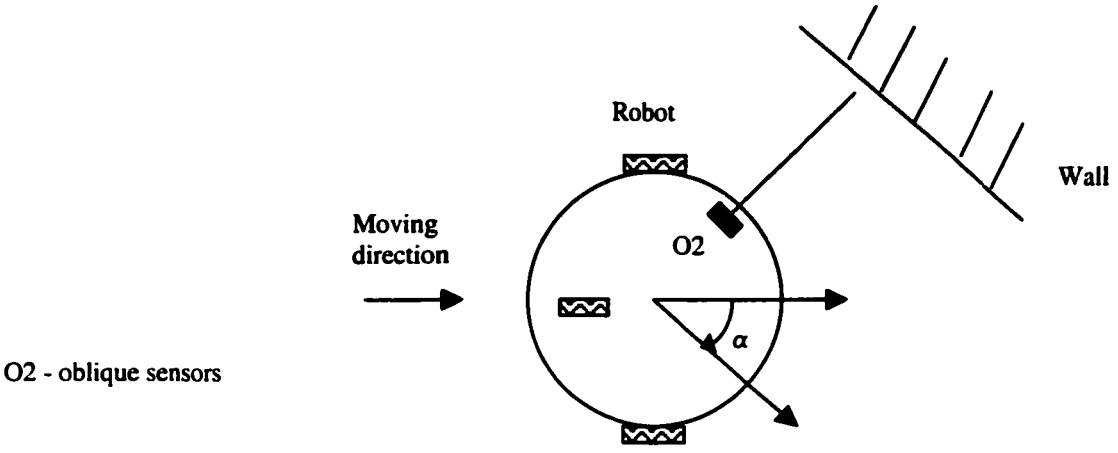

Figure 4.8: Sensors used by Go-Tangent-Oblique-Sensor FIS

The model for this FIS was built using the adaptive learning algorithm previously described. The training data set composed of 29 measurements of pairs (range, speed, desired turn angle) is shown in Figure 4.9, together with a checking data set used for validation. The checking data set is obtained from the training data set applying a 10% random noise.

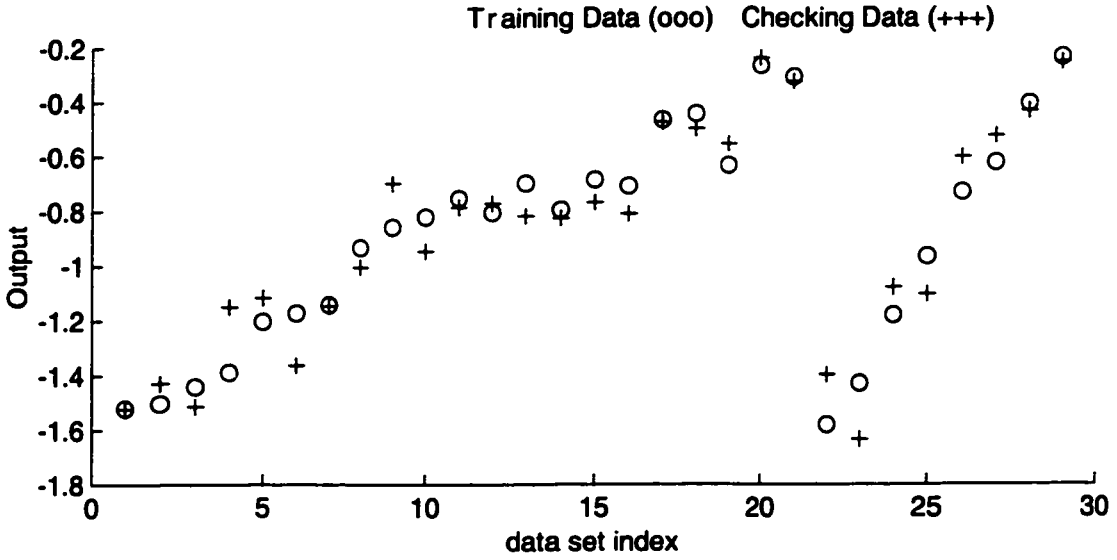

Figure 4.9: Output distribution in training Go-Tangent-Oblique-Sensor FIS

The basic idea behind using a checking data set for model validation is that after a certain point in training, the model begins overfitting the training data set. In principle, the model error for the checking data set tends to decrease as the training takes place up to the point that overfitting begins, and then the model error for the checking data suddenly increases. The results of the training process are presented in Figure 4.10. The training error shown on the vertical axis represents the root mean squared error of the training set at each epoch. A total of 25 epochs have been executed, but the training process converges sooner than the last epoch.

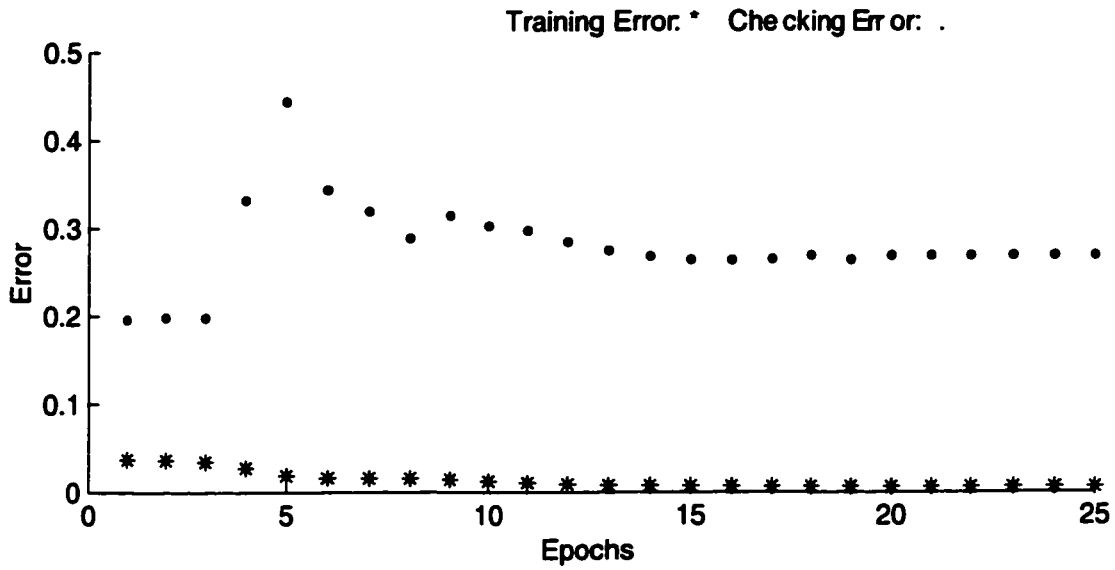

Figure 4.10: Error plot during the training process for Go-Tangent-Oblique-Sensor FIS

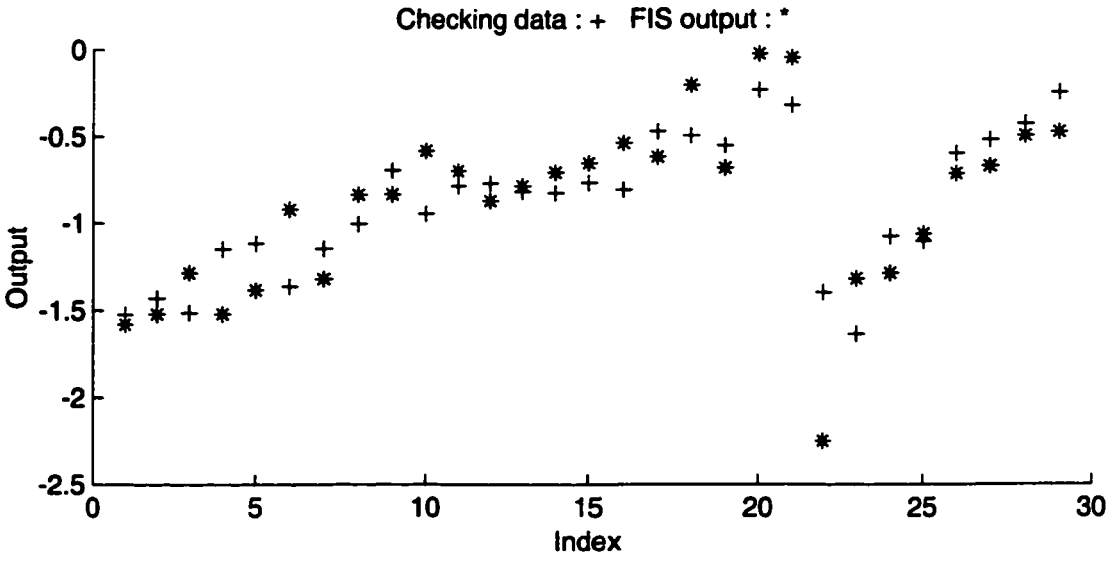

Figure 4.11: Testing Go-Tangent-Oblique-Sensor FIS against the checking data

In order to test the trained FIS, the checking data is applied to the model to see how well the model responds to this data. The results are shown in Figure 4.11.

One result of the training is represented by the set of premise parameters $\{(a_{ij}\ b_{ij}) \mid i = 1,2,3;\ j = 1,2\}$ required by the equation (4.2.10). This set describes the input membership functions for **Range** and **Speed**, as it can be seen in Figures 4.12 and 4.13.

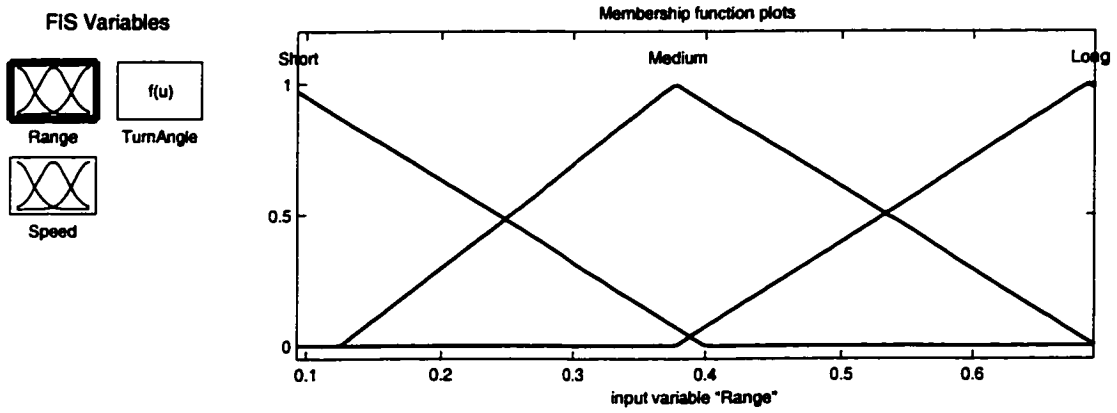

Figure 4.12: Go-Tangent-Oblique-Sensor FIS : First input variable

It was mentioned already that several shapes of membership functions were tried but these triangular shapes proved to be the best estimators of the training data set. Another reason for using them is given by the following: the use of linear membership functions (triangular or trapezoidal shape) will lead to linear characteristic of the fuzzy controller, the single nonlinearity being introduced only by the fuzzy rule set, which is in line with the basic idea of a fuzzy controller [13].

The learning algorithm is initialized with symmetric membership functions. The asymmetric shapes obtained (especially for the variable **Speed**) show clearly a tune-up process difficult to be reproduced manually, demonstrating the advantage of the adaptive learning technique.

The second result of training is represented by the set of consequent parameters

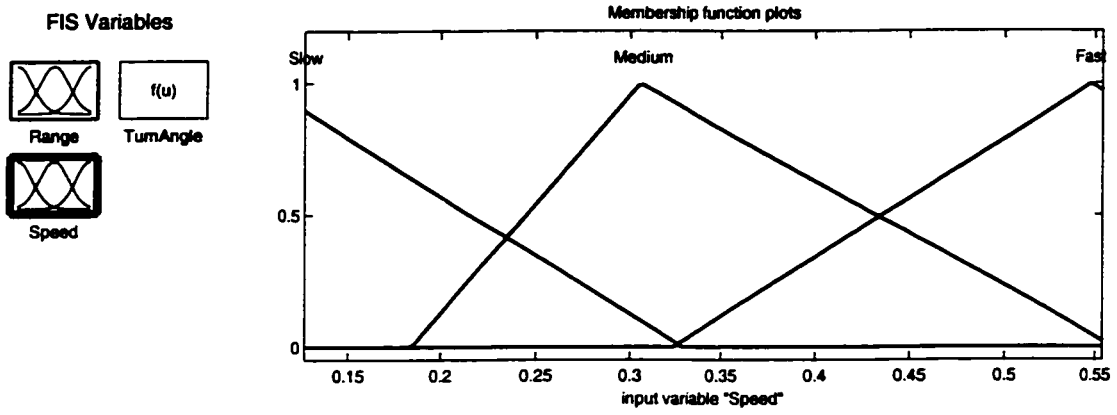

Figure 4.13: Go-Tangent-Oblique-Sensor FIS : Second input variable

$\{(p_k\ q_k\ r_k)|\ k = 1, 2, ..., 9\}$ required by the equation 4.2.9 for the definition of the output membership functions. With these parameters and with x as Range and y as Speed, the rule base for the fuzzy inference system looks like in Table 4.3.

No.	Rule definition
1	If (Range is Short and Speed is Slow) then (TurnAngle $= -3.253x - 8.858y + 1.561$)
2	If (Range is Short and Speed is Medium) then (TurnAngle $= 4.215x - 8.077y + 1.133$)
3	If (Range is Short and Speed is Fast) then (TurnAngle $= 0.006x - 6.715y + 2.008$)
4	If (Range is Medium and Speed is Slow) then (TurnAngle $= 2.967x - 9.659y + 3.276$)
5	If (Range is Medium and Speed is Medium) then (TurnAngle $= 2.241x - 8.763y + 0.933$)
6	If (Range is Medium and Speed is Fast) then (TurnAngle $= -1.113x - 7.021y + 2.762$)
7	If (Range is Long and Speed is Slow) then (TurnAngle $= 3.907x + 4.525y - 3.954$)
8	If (Range is Long and Speed is Medium) then (TurnAngle $= 1.249x + 3.560y - 2.943$)
9	If (Range is Long and Speed is Fast) then (TurnAngle $= -0.951x + 4.639y - 3.477$)

Table 4.3: Go-Tangent-Oblique-Sensor FIS : Fuzzy rules

The Input/Output space mapping for this FIS is presented in Figure 4.14, as captured with Matlab application. It can be seen that the Sugeno FIS model has a guaranteed continuity of the output surface, which varies smoothly between different inputs.

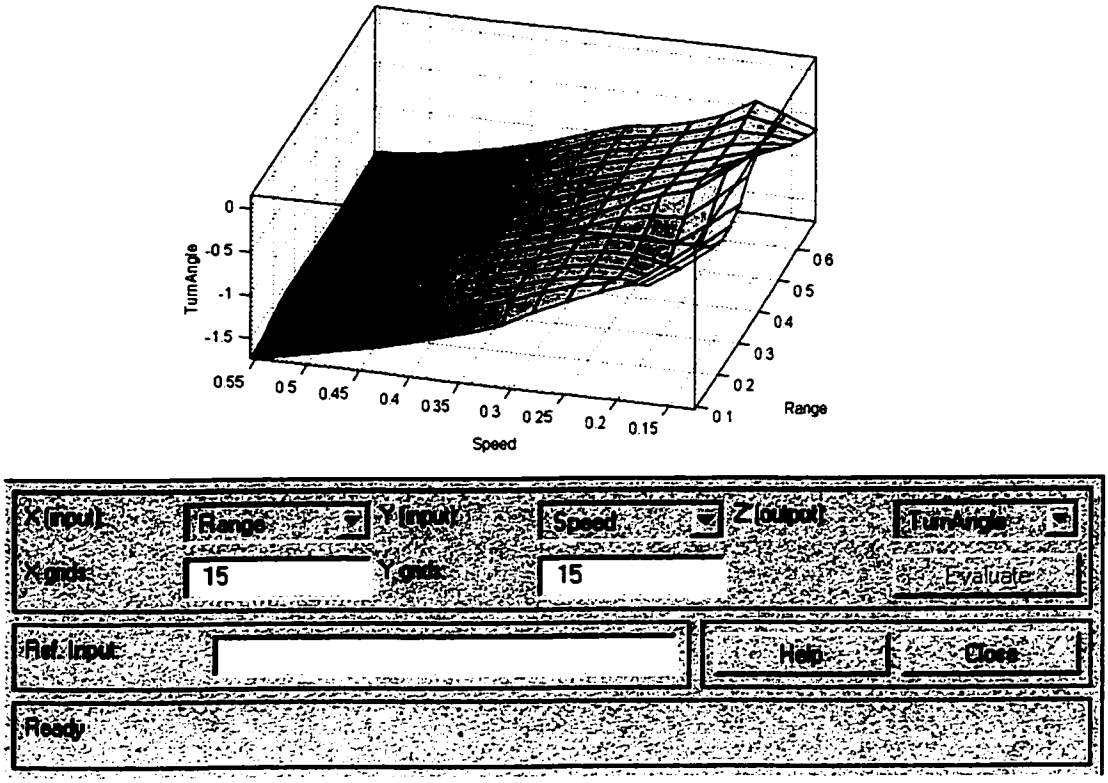

Figure 4.14: Go-Tangent-Oblique-Sensor FIS : Input/Output space mapping

4.2.3 Go-Tangent-Front-Sensor Fuzzy Inference System

The Go-Tangent-Front-Sensor fuzzy inference system is a Sugeno-type FIS built using a similar adaptive learning technique as the Go-Tangent-Oblique-Sensor system. This time the input variables are the two range values from the two sensors facing forward direction. These input variables are named **FrontRange** and **FrontLeftRange**, corresponding to sensors **Front2** and respectively **Front3**. Keeping the same notations as in Figure 2.3, these sensors are shown in Figure 4.15.

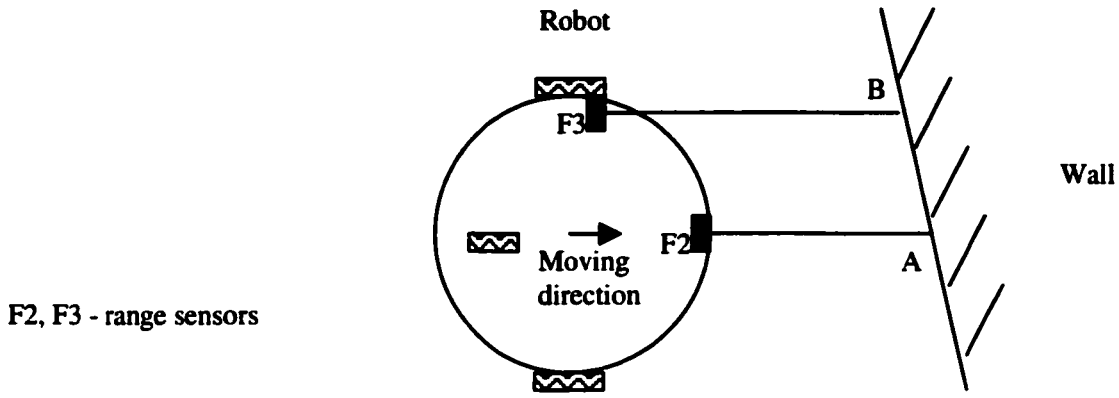

Figure 4.15: Sensors used by Go-Tangent-Front-Sensor FIS

The output variable is the same, the turn angle (`TurnAngle`) to be executed by the robot. The training data set and a corresponding checking data set is shown in Figure 4.16.

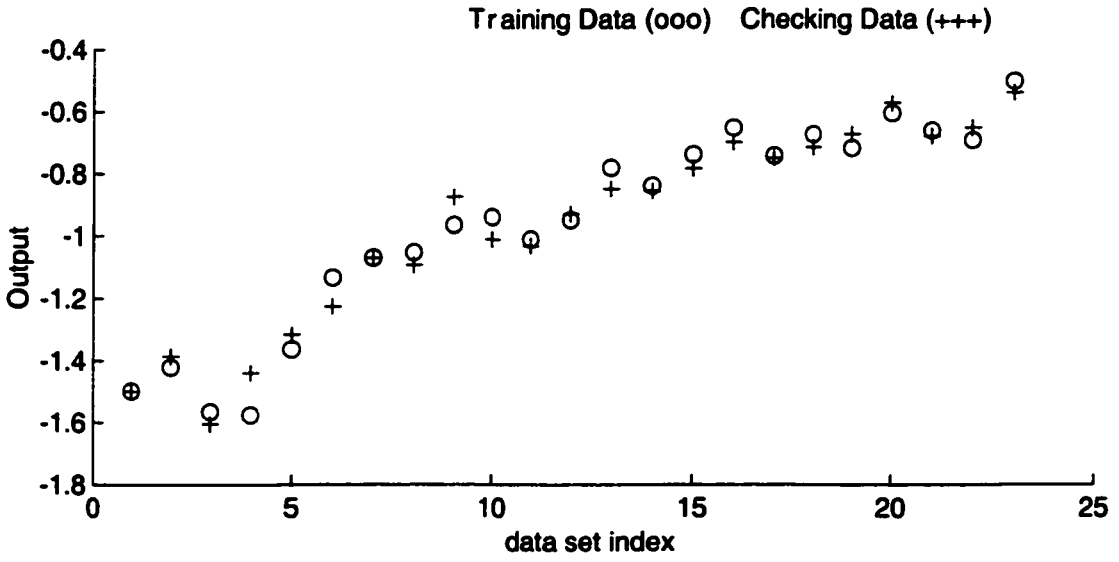

Figure 4.16: Output distribution in training Go-Tangent-Front-Sensor FIS

The training error obtained during each epoch of the learning process is plotted in Figure 4.17.

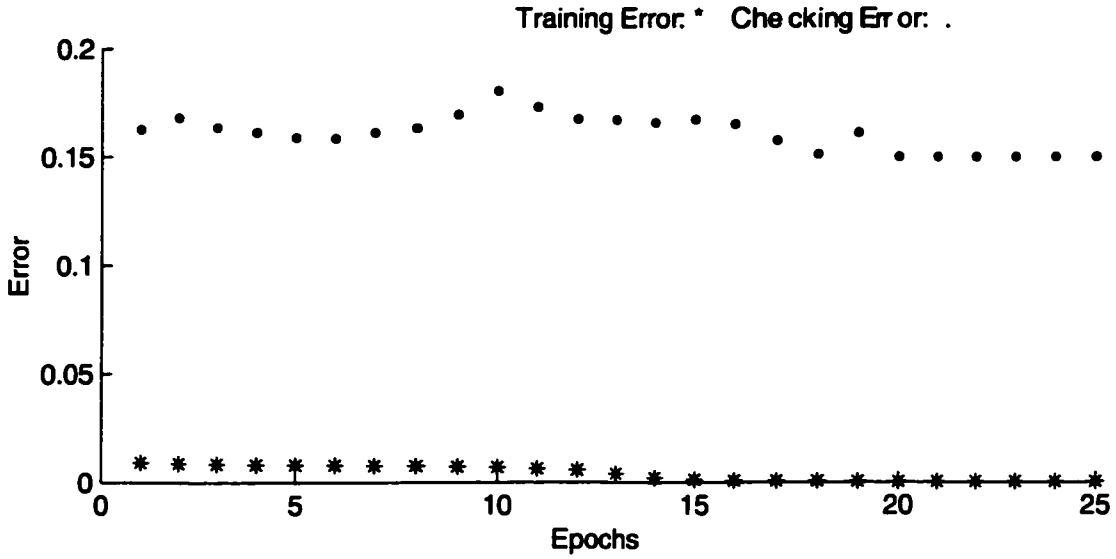

Figure 4.17: Error plot during the training process for Go-Tangent-Front-Sensor FIS

The same procedure in model validation has been used as in the previous subsection: the FIS model has been tried on the checking data set and results are displayed in Figure 4.18. It can be concluded that the model approximates well enough the checking data set, but there are still some points in which the difference is significant (approx. 20-30%). The explanation is found in the fact that not all input vectors of the input space have a valid mapping to an output value, due to physical constrains: there is a certain distance between the two sensors, so the four points represented by the two physical locations of the sensors and the two reflection points must form a trapezoid. In Figure 4.15 these points are F2, A, B and F3. As an example, an object cannot be placed very close to both sensors at 15cm if between the two sensors there is a distance of approx. 49cm.

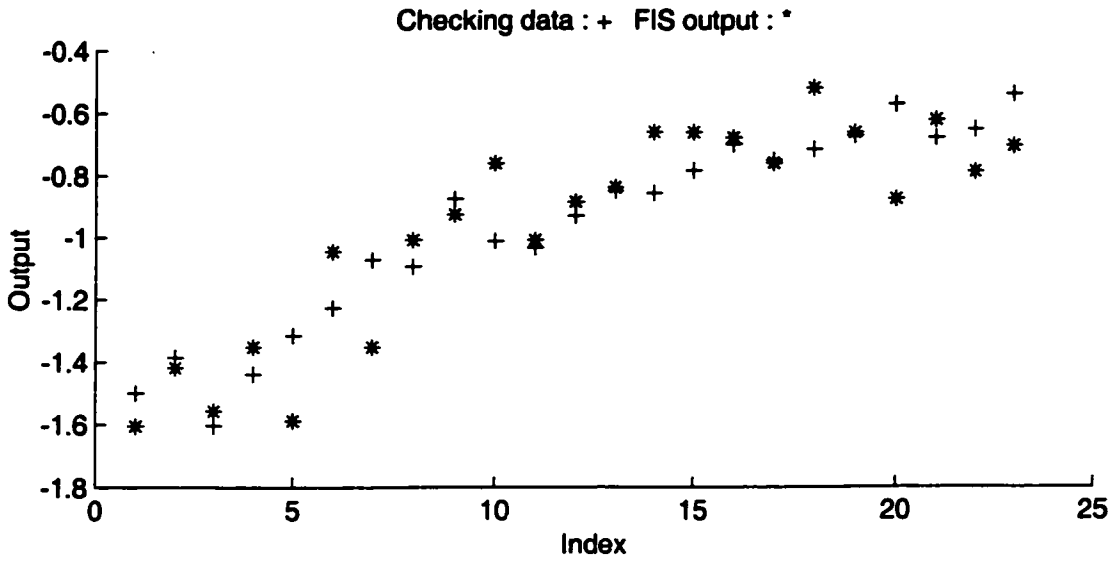

Figure 4.18: Testing Go-Tangent-Front-Sensor FIS against the checking data

For Go-Tangent-Front-Sensor FIS, the membership functions obtained after training are even more asymmetric than those obtained for Go-Tangent-Oblique-Sensor, as it can be seen in Figures 4.19 and 4.20 for input **FrontRange**, and respectively **FrontLeftRange**.

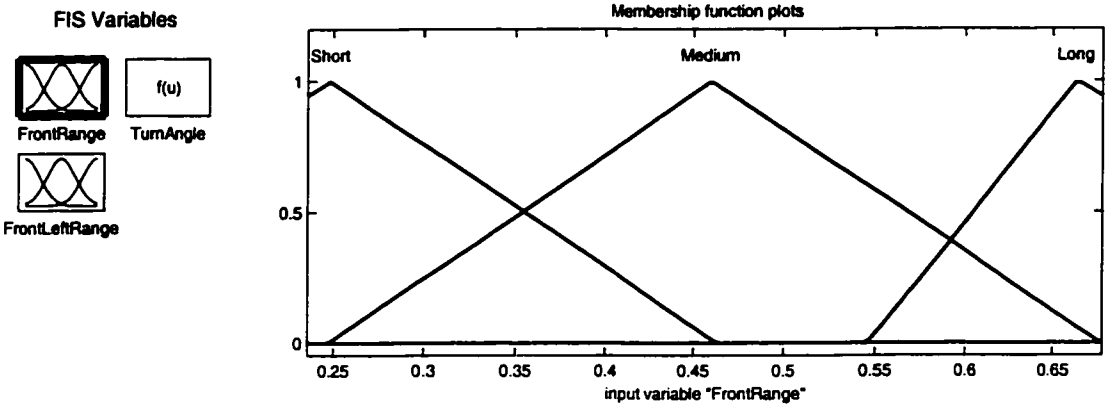

Figure 4.19: Go-Tangent-Front-Sensor FIS : First input variable

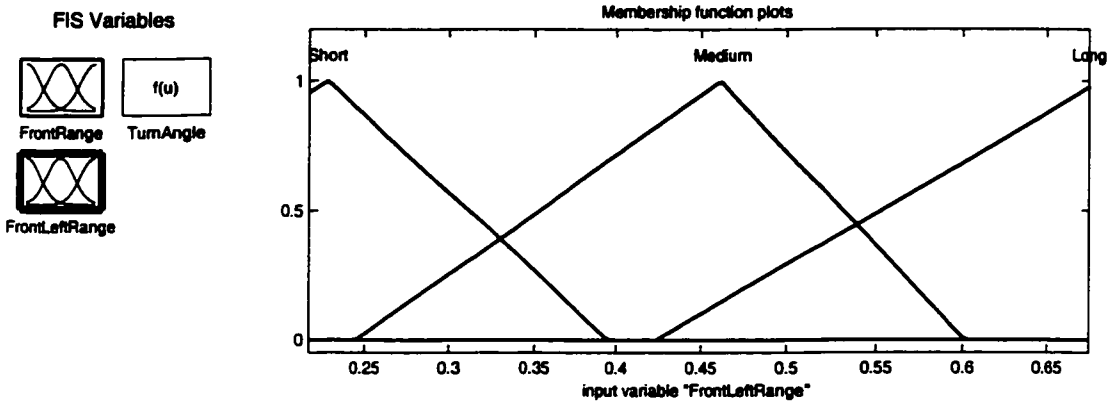

Figure 4.20: Go-Tangent-Front-Sensor FIS : Second input variable

No.	Rule definition
1	If (FrontRange is Short and FrontLeftRange is Short) then (TurnAngle $= 1.142x - 1.743y - 0.858$)
2	If (FrontRange is Short and FrontLeftRange is Medium) then (TurnAngle $= 2.400x - 3.710y - 0.536$)
3	If (FrontRange is Short and FrontLeftRange is Long) then (TurnAngle $= -2.666x + 2.356y - 2.177$)
4	If (FrontRange is Medium and FrontLeftRange is Short) then (TurnAngle $= 1.960x - 2.254y - 0.997$)
5	If (FrontRange is Medium and FrontLeftRange is Medium) then (TurnAngle $= -0.432x - 3.125y + 0.537$)
6	If (FrontRange is Medium and FrontLeftRange is Long) then (TurnAngle $= 2.131x - 3.970y + 0.180$)
7	If (FrontRange is Long and FrontLeftRange is Short) then (TurnAngle $= -1.698x - 4.967y - 0.854$)
8	If (FrontRange is Long and FrontLeftRange is Medium) then (TurnAngle $= -2.185x - 0.085y + 0.905$)
9	If (FrontRange is Long and FrontLeftRange is Long) then (TurnAngle $= -2.908x - 0.609y - 2.437$)

Table 4.4: Go-Tangent-Oblique-Sensor FIS : Fuzzy rules

The rule base for this fuzzy inference system is very similar to the one presented in Tabel 4.3. But this time the input variables are **FrontRange** and **FrontLeftRange** instead of **Range** and **Speed** respectively, as shown in Table 4.4.

As a result of the physical constraints already mentioned, the input/output space mapping (shown in Figure 4.21) has no interpretation in some areas, for example on the left side of the figure where FrontRange has very small values and FrontLeftRange very big values. For these values, the trapezoid condition mentioned above cannot be fulfilled. The same for the right side of the picture, where FrontRange has very big values and FrontLeftRange very small values.

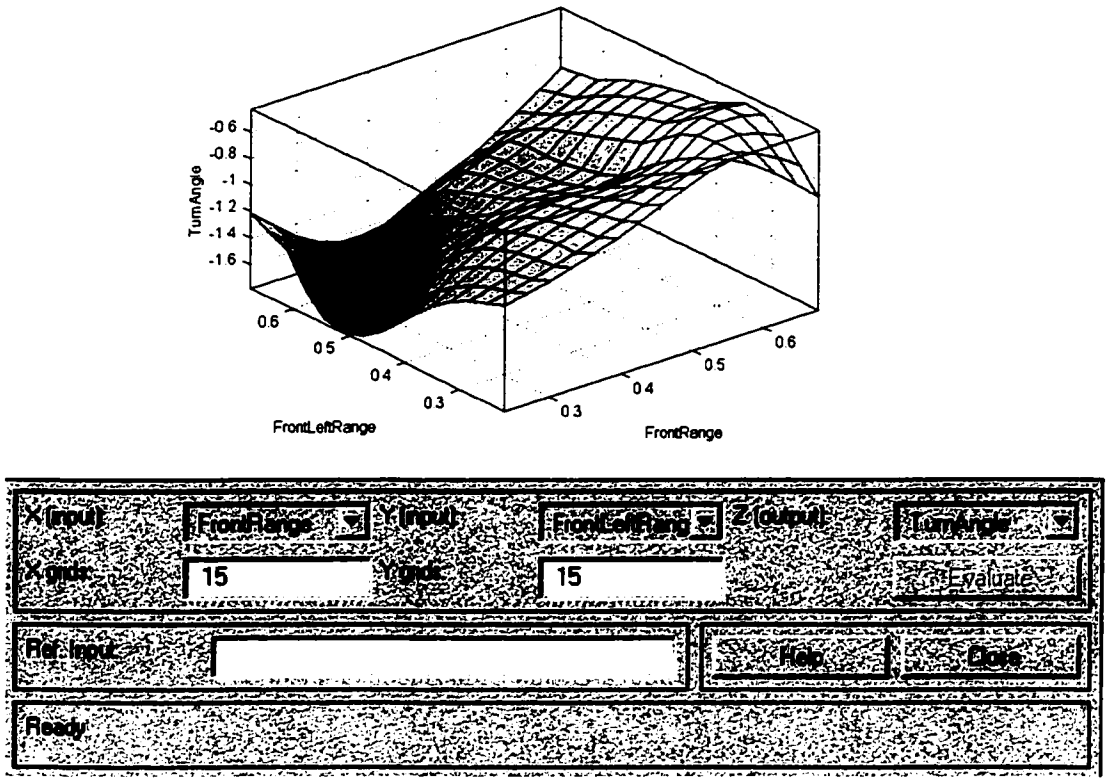

Figure 4.21: Go-Tangent-Front-Sensor FIS : Input/Output space mapping

4.3 Wall-Follow behavior

The Wall-Follow behavior allows the robot to move on a trajectory parallel with a wall. Many fuzzy implementations for this behavior exist, from simpler ones based on one input variable, the distance to the wall [11], to more complex ones, implementing a fuzzy-integration controller [23]. In this thesis the Wall-Follow behavior is implemented by a simple two input/one output fuzzy system, as in Figure 4.22.

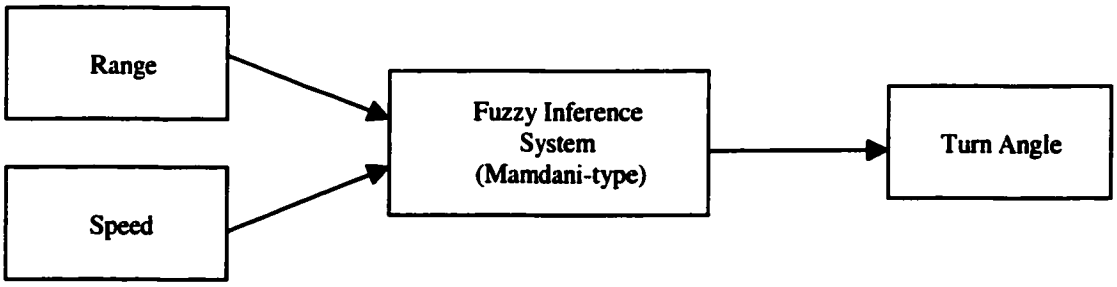

Figure 4.22: Wall-Follow FIS : block scheme

A Mamdani-type FIS is intuitive, it has widespread acceptance and it is well-suited to human input. Beside all these advantages, such a system brings the benefit for our system that it allows us to combine its output fuzzy set with other fuzzy sets.

Both inputs use only three triangular membership functions, defined as in Figures 4.23 and 4.24.

The two inputs represent the sensor range to the wall (**Range**) and the speed towards the wall (**Speed**) which can be positive, if the robot is getting closer to the wall, or negative if the robot is getting farther from the wall.

In defining the membership functions for **Range** variable, the membership function for the **Medium** linguistic value has been 'pulled' to the left, its maximum being for

Range = 0.3m instead of the mean value 0.4m. In this way, when the robot enters in a Wall-Follow behavior, the commands for the turn angle will be calculated so that its trajectory is located at a distance from the wall less than the mean value.

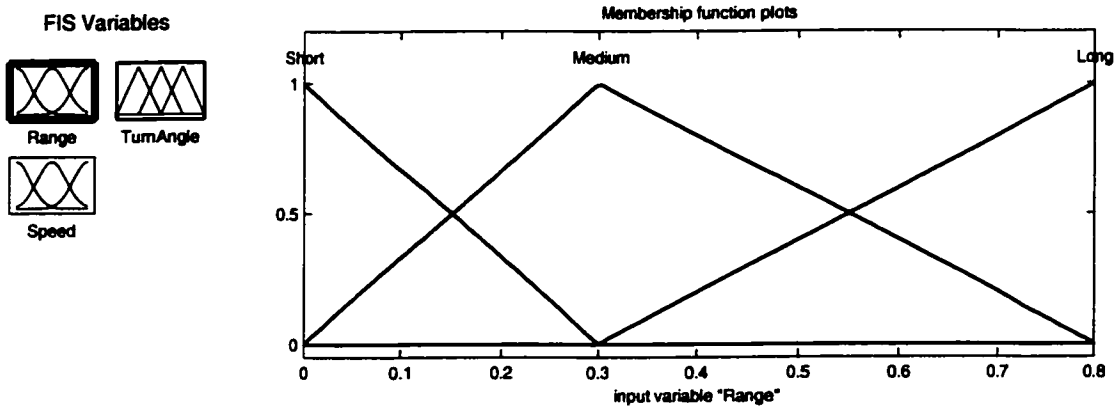

Figure 4.23: Wall-Follow FIS : First input variable

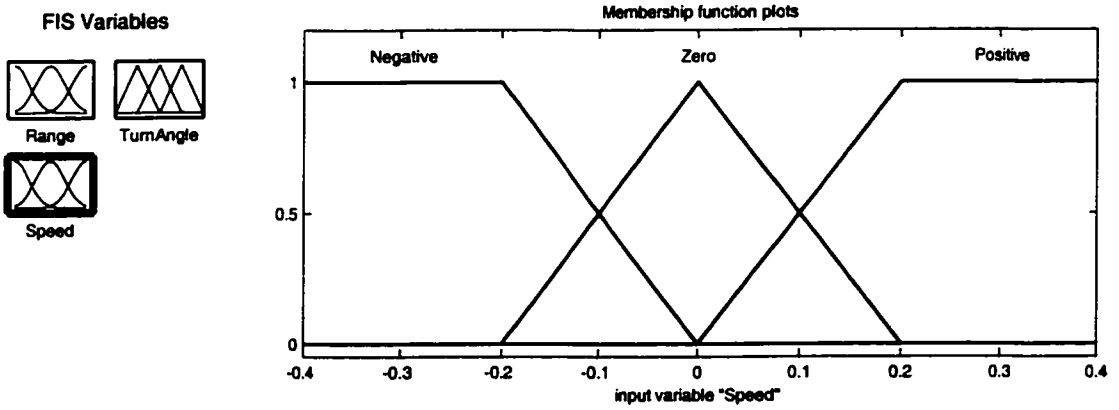

Figure 4.24: Wall-Follow FIS : Second input variable

The output variable represents the turn angle (**TurnAngle**) in radians to be executed by the robot. As it can be easily noticed in Figure 4.25, the union of the scopes (or domains) for all three membership functions is $[-\pi/3, \pi/3]$, far less than the whole universe of discourse which is $[-\pi, \pi]$. In a Wall-Follow behavior, a maximum turn

angle of $\pi/3$ is a common sense rule. Because other output fuzzy sets have a universe of discourse of $[-\pi, \pi]$, the same definition was imposed on this FIS, due to potential fuzzy operators applied to these fuzzy sets in the command fusion module.

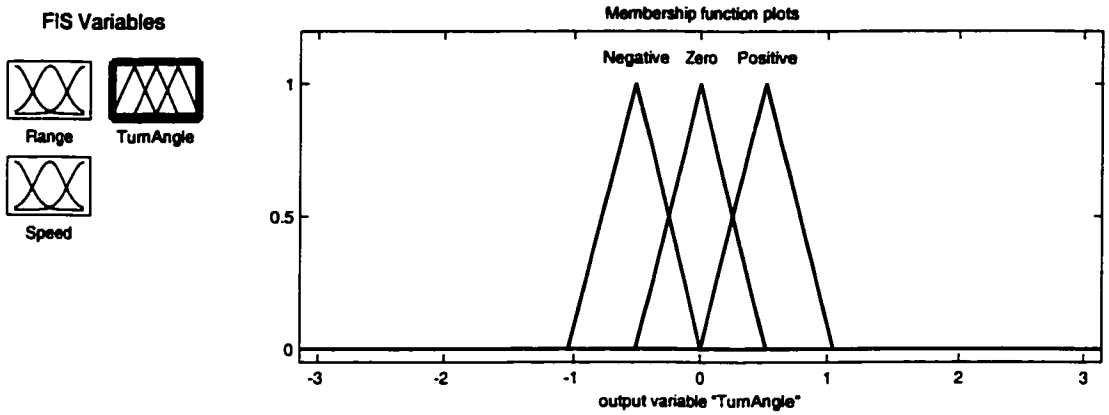

Figure 4.25: Wall-Follow FIS : Output variable

The fuzzy rules for the Wall-Follow FIS are captured in the Table 4.5. These rules are fairly intuitive, based on the idea of not letting the robot to get too close to the wall and also not allowing him to get too far.

No.	Rule definition
1	If (Range is Short and Speed is Negative) then (TurnAngle is Zero)
2	If (Range is Short and Speed is Zero) then (TurnAngle is Negative)
3	If (Range is Short and Speed is Positive) then (TurnAngle is Negative)
4	If (Range is Medium and Speed is Negative) then (TurnAngle is Positive)
5	If (Range is Medium and Speed is Zero) then (TurnAngle is Zero)
6	If (Range is Medium and Speed is Positive) then (TurnAngle is Negative)
7	If (Range is Long and Speed is Negative) then (TurnAngle is Positive)
8	If (Range is Long and Speed is Zero) then (TurnAngle is Positive)
9	If (Range is Long and Speed is Positive) then (TurnAngle is Zero)

Table 4.5: Wall-Follow FIS : Fuzzy rules

An example of rule evaluation for this rule base is given in Figure 4.26. For a range value of 0.202 m and a speed of 0.0836 m/s, four rules fire : 2, 3, 5 and 6. The

last column of diagrams show the result of the fuzzy implication process, represented by four significant fuzzy sets. A *max* operator between these fuzzy sets is shown in the bottom diagram of the last column, as a union of different trapezoidal shapes. If Wall-Follow behavior is applied directly, without being combined with another output fuzzy set, then a defuzzification process with a center-of-area method leads to a turn angle of -0.227 radians (or approx. -13 degrees).

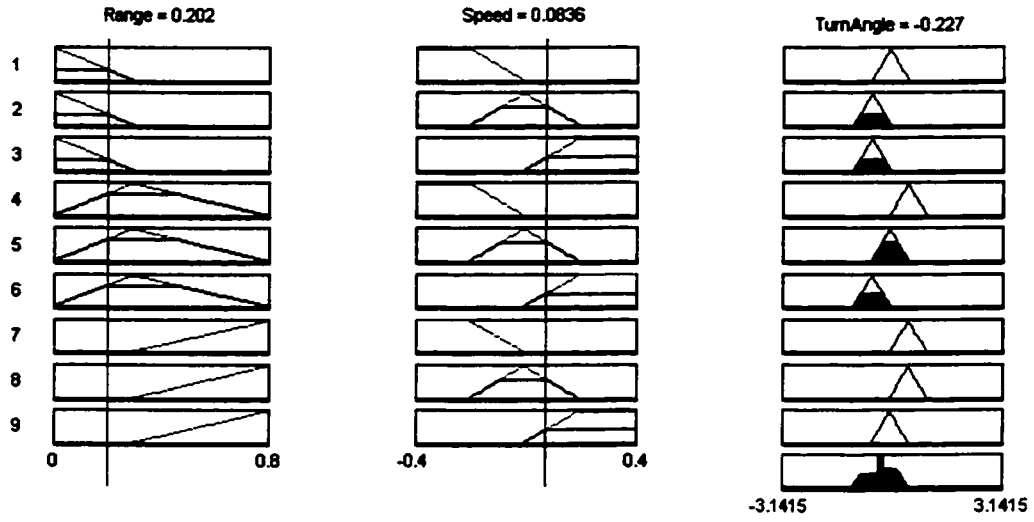

Figure 4.26: Wall-Follow FIS : Rule evaluation for a given input vector

The input/output mapping space for Wall-Follow FIS is presented in Figure 4.27. This mapping has a surface which guarantees a very smooth interpolation for all input vectors (**Range**, **Speed**) of the input space.

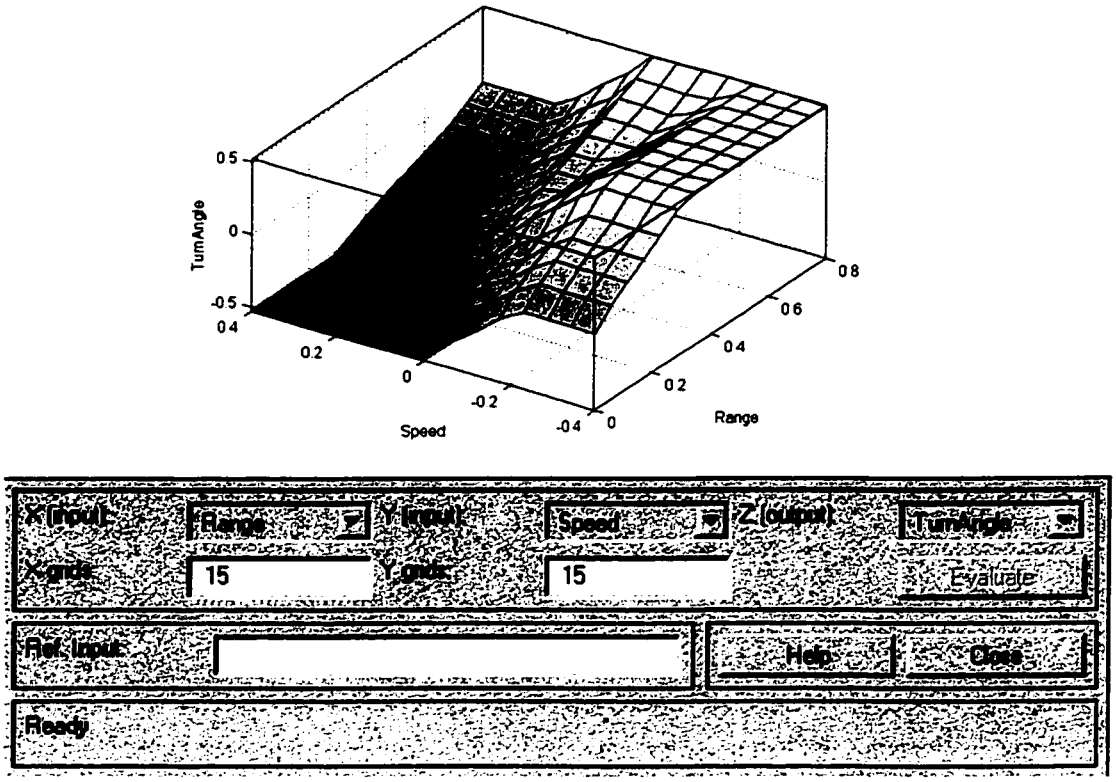

Figure 4.27: Wall-Follow FIS : Input/Output mapping space

4.4 Turn-Corner behavior

Turn-Corner behavior helps the robot in making a turn around a corner. Initially, a hard limit of 90 degrees has been used for performing this type of turn. But a fuzzy approach was needed due to different turn angles α required, as in Figure 4.28, cases b) and c). The robot trajectory is represented by a doted line and the current orientation of the robot by a continuous arrow. The turn angle α could be more or less than 90 degrees, depending of the direction of the current trajectory. Clearly the future trajectory is desired to be parallel with the wall, even if the current one is not. Therefore the angle α must be computed.

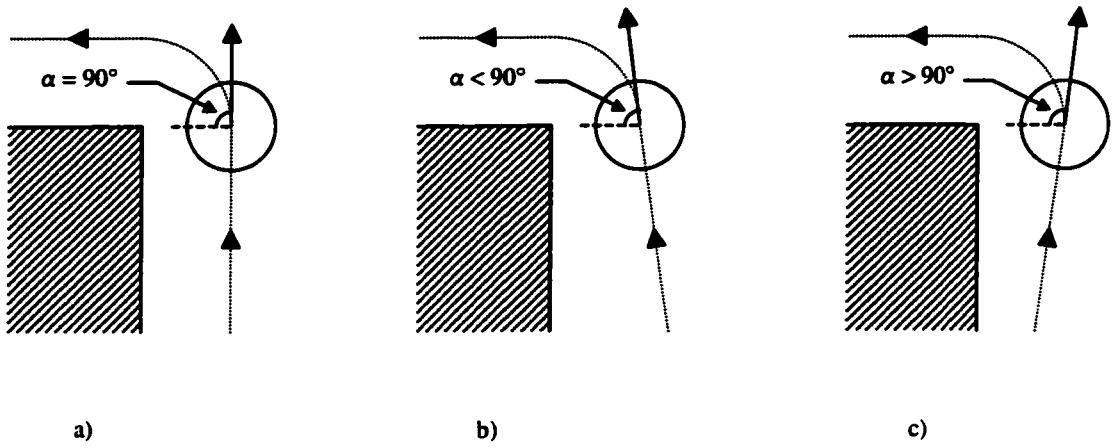

Figure 4.28: Different robot trajectories when turning a corner

There is no indication for the orientation of the current trajectory relative to the wall followed. But a good indication is the speed of the robot towards the wall, given by the backward sensor `Back1` (Figure 2.3). At the moment of executing a turn move, usually only the indication from this sensor is available. The fuzzy system for this behavior is a simple one, with only one input for the speed reported by `Back1` (`Speed`) and one output for the turn angle (`TurnAngle`). The membership functions defined for these variables are shown in Figures 4.29 and 4.30.

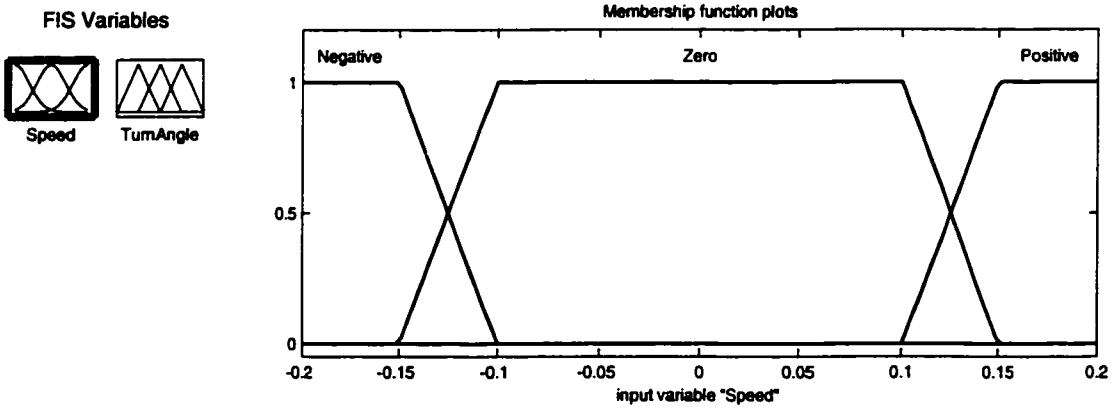

Figure 4.29: Turn-Corner FIS : Input variable

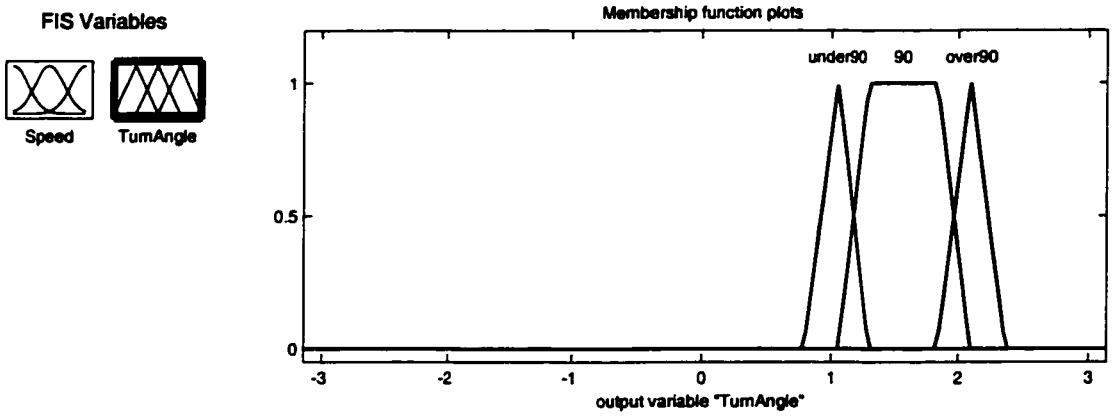

Figure 4.30: Turn-Corner FIS : Output variable

It can be observed that the universe of discourse for the output variable is the interval $[-\pi, \pi]$, for the same reason as in the case of output from Wall-Follow FIS: a potential fusion with another output fuzzy set.

A comment here has to be added regarding other corners than the 90° ones. Experimental tests on corners of 135° (which require a turn angle of approx. 45°) showed that other sensors like `Wall1` or even `Oblique1` give valid indications, so that `Back1` does not need to be taken into account.

The simple fuzzy rule base is summarized in Table 4.6, and the input/output space is presented in Figure 4.31.

No.	Rule definition
1	If (Speed is Negative) then (TurnAngle is over90°)
2	If (Speed is Zero) then (TurnAngle is 90°)
3	If (Speed is Positive) then (TurnAngle is under90°)

Table 4.6: Turn-Corner FIS : Fuzzy rules

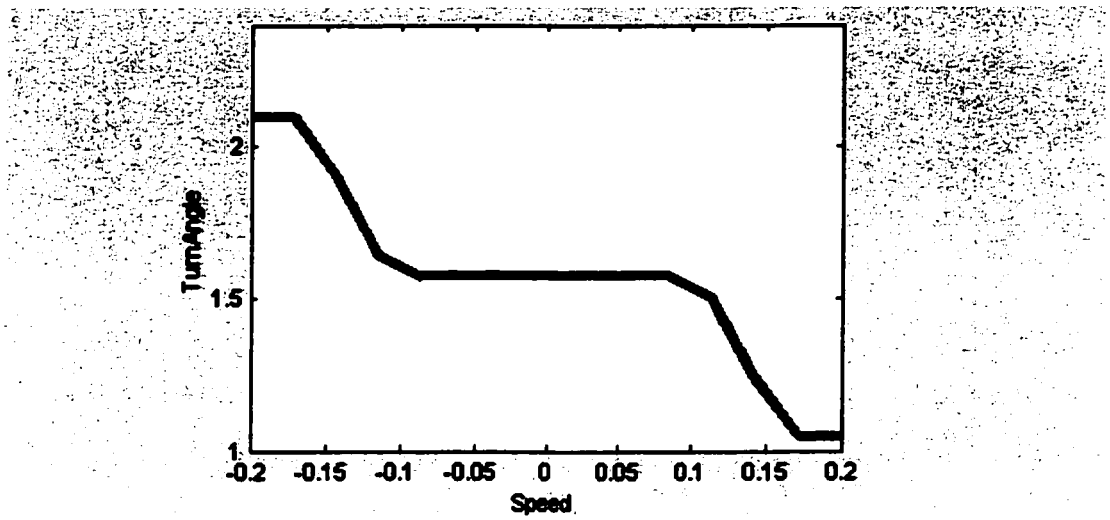

Figure 4.31: Turn-Corner FIS : Input/Output space mapping

4.5 Turn-Around behavior

Turn-Around behavior is the single behavior not requiring a fuzzy inference system. This behavior has a straight forward implementation: the turn angle to be executed by the robot is 180°.

The sign of this value is dictated by a common sense rule: if the robot has a positive total rotation angle relative to the initial direction towards the target, than the turn angle is $-180°$. If the the total rotation angle of the robot is negative, then the turn angle is $+180°$.

This primitive behavior is used by the Dead-End high level behavior when obviously there is no point to move forward, or by a Goto-Target behavior when it detects a certain threshold reached for the total rotation angle. Chapter 5 gives more details about this case.

Chapter 5

High level-behaviors

The high-level behaviors are complex tasks to be executed by the mobile robot. They are acting like decision systems which map the goal and sensory information to one or more primitive behaviors. Goal information is represented by the XY coordinates of both target and robot. These are used for moving the robot from its current location to the target point. Sensory information is represented by the current indications from all the sensors.

A simple switching technique is used to decide which high-level behavior applies. This switching technique depends on how many sensors have valid indications and also on the total rotation angle (β) of the robot relative to the initial direction towards target. If the robot executes at every control cycle i a turn angle (rotation) α_i, then the total rotation angle after N control cycles is defined as $\beta = \sum_{i=1}^{N} \alpha_i$.

The switching technique decides in the following way which high-level behavior is used:

1. for no valid indications, Open-Space behavior is applied;

2. for valid indications from front, left and right side, Dead-End behavior is used;

3. for all the other cases of valid indications:

   (a) for $|\beta| < \beta_{threshold}$, Goto-Target behavior is applied;

   (b) for $|\beta| \geq \beta_{threshold}$, Contour-Follow behavior is applied

Goto-Target behavior uses obviously the Goto-XY primitive. If the robot in his search for the target has rotated too much (β is over the threshold value $\beta_{threshold}$), than the robot is not allowed anymore to focus on the target. The Contour-Follow primitive is used instead, so that the robot follows back the contour which led him in this situation until his total rotation angle falls within limits.

The experimental tests showed that it is better to allow the value $\beta_{threshold}$ to be dynamically changed, each behavior setting his own threshold, and therefore defining for how long controls the robot. Goto-Target sets it at π, Contour-Follow sets it at $\pi/2$. This approach is similar to what other researchers named 'degree of importance' modified dynamically for each goal [34, 32].

Each high-level behavior is based on one or more low-level behaviors. When more than one low-level behavior are used, their output fuzzy sets is passed to the command fusion module for proper processing. In this work, only two behaviors are used simultaneously, but the principles of the command fusion remain the same for a larger number. A presentation of each high-level behavior follows.

5.1 Open-Space behavior

Open-Space behavior is based on a single primitive behavior, Goto-XY. There are no indications of any obstacle around, therefore the single task of the robot is to go to the target point.

A rotation angle is calculated through direct defuzzification of the output fuzzy set obtained from the primitive behavior. This angle might not be exactly the difference between the current orientation of the robot and the direction towards target. The reason is the defuzification process itself, but this causes a very small difference, at least for the current implementation chosen for the Goto-XY FIS. Subsequent control cycles correct eventually this difference.

Open-Space behavior resets the total rotation angle β mentioned at the beginning of this chapter. This is done at the end of the activity performed by this behavior which is a rotation move. At this time it can be assumed that the robot points to the target.

The accuracy of this behavior relies on the accuracy of the dead-reckoning system which offers information regarding the distance travelled since the last command issued. Using this information, the robot is able to locate itself, as discussed in Appendix B. Usually the dead-reckoning system is a source of errors, but for indoor environments it is acceptable especially if further improvements are made as discussed at the end of Chapter 7. In general, fuzzy system errors can be analyzed and corrected, several methods being avaialble [4, 21, 28].

5.2 Goto-Target behavior

Goto-Target behavior is the most complex behavior, having the goal to direct the robot to the target. It uses all the primitive behaviors defined in this work.

Goto-Target is called whenever there is a valid indication from at least one sensor. If more than one sensor indicate an obstacle, a priority scheme will be applied to decide which sensor is the most significant one. Then the adequate low-level behavior

for this type of sensor is used. The priority scheme follows a simple rule: the smaller the absolute value of the focus angle of the sensor, the higher the priority. By *focus angle* it is understood the angle between the forward axis of the robot and the axis of the sensor pointing to the obstacle.

So the sensors facing forward direction are the most important, due to the fact that they could signal eventual collisions. The sensors focusing backward are the least significant. In terms of notations established in §2.3, the priority for considering valid indications from sensors is the following (starting with the highest priority):

1. **Front sensors** F1, F2, F3 with an absolute value of the focus angle 0°, and **Oblique sensors** O1, O2,with an absolute value of the focus angle 45°.
2. **Wall sensors** W1, W2; absolute value of the focus angle: 90°.
3. **Back sensors** B1, B2; absolute value of the focus angle: 135°.

The **Front** and **Oblique sensors** involve the use of Go-Tangent primitive (Go-Tangent-Front-Sensor and Go-Tangent-Oblique-Sensor respectively). During experimental tests, it was concluded that these two types of sensors should have equal priorities. If there are valid range indications from both types of sensors, then both primitives will be applied and a maximum value for the turn angle between the two results will be used.

If there are no indications from Front or Oblique sensors, then the indications from the **Wall sensors** are taken into account. For these type of indications, Goto-XY and Wall-Follow primitives will be applied. When both primitives are used, the results from their fuzzy inference systems are combined in the command fusion module. In the same way as a threshold value $\beta_{threshold}$ has been introduced to select between

different high-level behaviors, a threshold is required here also to distinguish cases when only one low-level behavior (Wall-Follow) needs to be used:

- for $|\beta| < \beta^{w}_{threshold}$ Goto-XY and Wall-Follow are applied;
- for $|\beta| \geq \beta^{w}_{threshold}$ only Wall-Follow primitive is applied.

Combining the two primitives is required by the condition that the robot cannot follow the wall indefinitely, at a certain point he must be able to go away from the wall. But there are cases when only Wall-Follow is required. For example, when the target is beyond a room, then the robot has to enter the room to explore it. After that, he has to be able to exit the room, and this can be achieved only with a Wall-Follow behavior, without Goto-XY. The threshold value for Wall-Follow primitive ($\beta^{w}_{threshold}$) is currently set at $\pi/2$.

The exit point from this high-level behavior happens when $\beta > \beta_{threshold}$, case in which the Turn-Around primitive is used. As described at the beginning of this chapter, the switching technique will select Contour-Follow behavior from now on.

The last to be taken into account are the **Back sensors**, if there are no valid indications from the other sensors. In this case, Goto-XY and Turn-Corner primitives will be used. Like in the case of the Wall sensors, a similar total rotation angle threshold value $\beta^{t}_{threshold}$ (currently $\pi/2$) is used to determine when the Turn-Corner primitive is used alone.

5.3 Contour-Follow behavior

Contour-Follow behavior helps the robot to follow a path close to a certain contour. In an indoor environment, usually this is represented by the walls of room or of a

corridor.

This high-level behavior is implemented in a similar way as Goto-Target behavior, with the same priority scheme for the sensors. There are no threshold values for this behavior, and for each type of sensor one single type of low-level primitive being used:

- for **Front** and **Oblique sensors** the Go-Tangent primitive is applied;
- for **Wall sensors** the Wall-follow primitive is applied;
- for **Back sensors** the Turn-Corner primitive is applied.

In his attempt to reach the XY location of the target, the robot might actually move away from it, due to obstacles like corridors, walls, etc., which form a contour. If the total rotation angle of the robot becomes too big, then the robot should continue his search in an opposite direction, following the same contour. But this time he is not allowed to use the Goto-XY primitive until the total rotation angle β falls under a threshold $\beta_{threshold}$, as previously discussed. This is the Contour-Follow behavior which, from the implementation point of view, is a simplified Goto-Target behavior where Goto-XY primitive is missing.

One exit point from this behavior is when $\beta < \beta_{threshold}$. Another exit point is whenever the robot 'loses' the contact with the contour (all sensors offer no valid range indication), case in which the Open-Space behavior is applied.

5.4 Dead-End behavior

Dead-End behavior has the goal of backing off the robot from a dead-end situation. Such a situation is defined by all sensors on the left, front and right side providing valid range indications. These sensors are: Front, Oblique and Wall sensors.

The single alternative for the robot is to look for an exit in the opposite direction. Therefore this behavior applies only the Turn-Around primitive.

There are cases when the target is placed at the end of a corridor. In order to allow the robot to avoid a potential turn-around right in the vicinity of the target, this behavior is inhibited in the case of a distance to the target less than the distance given by the Front sensors.

5.5 Command fusion module

The command fusion module takes as inputs the output fuzzy sets from different Mamdani-type fuzzy inference systems and delivers as output a resultant fuzzy set to be defuzzified.

The fuzzy inference systems considered for this process are: Goto-XY, Very-Close, Wall-Follow and Turn-Corner FIS. Their output fuzzy set is combined using *not*, *min* and *max* fuzzy operators as in Figure 5.1.

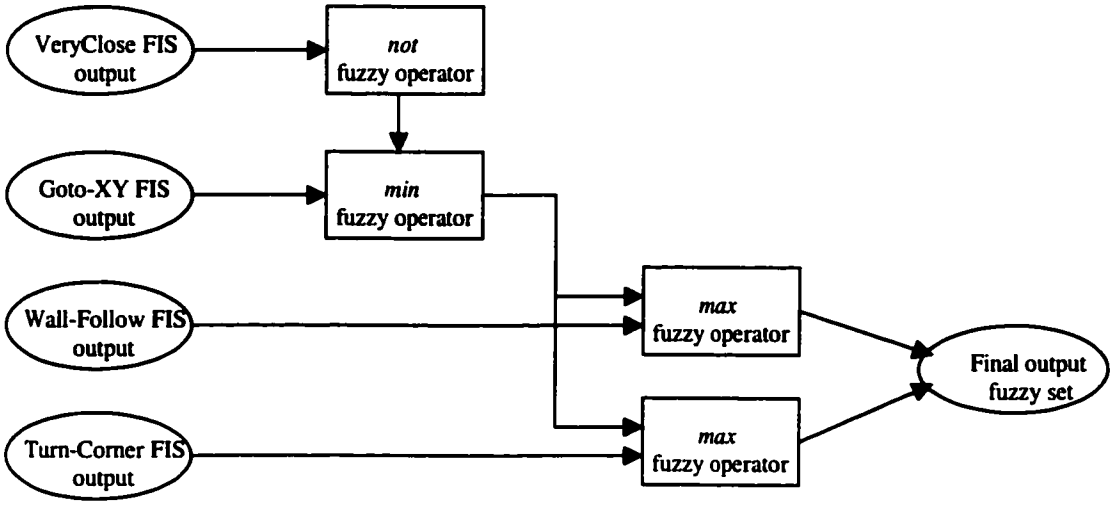

Figure 5.1: Fuzzy operations in the command fusion module

A *not* operator is applied against the *not recommended* turn angle obtained from Very-Close FIS, so that a *recommended* turn angle is obtained as a fuzzy set.

The *min* operator is used between this fuzzy set and the output from Goto-XY. This operator limits the directions suggested by Goto-XY (also a recommended turn angle) to a set which guarantees that the robot will not collide with any obstacle. For this purpose the membership functions in the Very-Close system have been designed with a large scope.

A *max* operator is used further to include other fuzzy sets representing recommended turn angle sets, as the output from the Wall-Follow FIS or from Turn-Corner FIS.

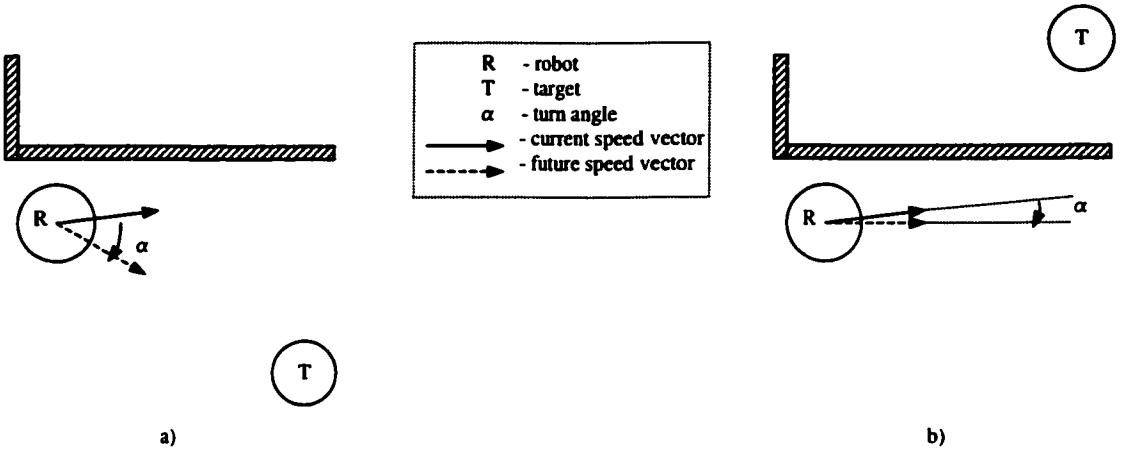

Figure 5.2: Target positioning relative to a followed wall

An example of how the command fusion module works is shown in Figure 5.3, where the three output fuzzy sets from Goto-XY, Very-Close and Wall-Follow FIS are combined with the respective fuzzy operators. The diagrams representing fuzzy sets are drawn for the case when the robot has to depart from the wall in order to reach a target, as presented in Figure 5.2a. In this case the robot and the target are

on the same side of the wall. The other case, when robot and target are on different sides of the wall (Figure 5.2b), is discussed later.

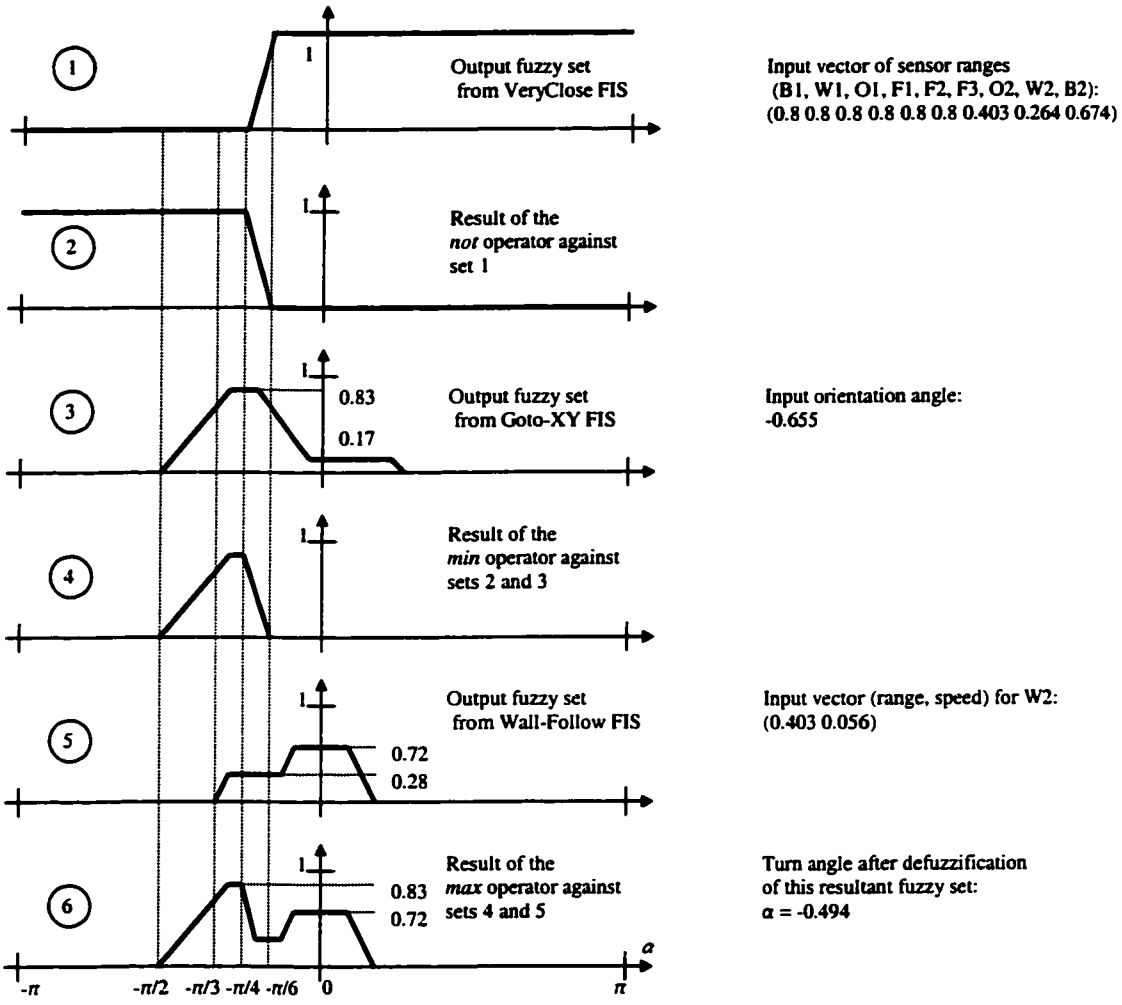

Figure 5.3: Fusion of fuzzy sets for robot and target on the same side of a wall

In Figure 5.2a, the followed wall is on the left side of the robot, so only the sensors placed on this side give valid indications: `Wall2`, `Oblique2` and `Back2`. These values applied to Very-Close FIS generate the fuzzy set no. 1. We number the fuzzy sets from 1, the top diagram, to 6, the bottom diagram - see Figure 5.3.

The *not* operator applied to fuzzy set 1 produces fuzzy set 2. From the orientation angle towards the target, a fuzzy set 3 can be obtained using Goto-XY FIS. Fuzzy set 4 is the resultant of the *min* operation between fuzzy set 2 and 3, cancelling the directions which might lead to collisions. Fuzzy set 5 is the result of Wall-Follow FIS. The *max* operation between fuzzy set 4 and 5 is the resultant fuzzy set which is passed to the defuzzification module. A center-of-area defuzzification method allows us to calculate the crisp value for the turn angle α.

In the resultant fuzzy set it can be seen that a significant contribution has the Goto-XY behavior. This is due to the fact that there is no obstacle towards the target, and the robot has to move away from the wall.

Figure 5.2b is an example of a case where there is an obstacle (a wall) between the robot and the target. The robot has to continue his path till the end of the wall. The diagrams are similar with the ones previously described, but this time the output of the Goto-XY FIS is shifted more to the right as it can be seen for fuzzy set 3 in Figure 5.4. A positive angle is required this time to direct the robot towards the target. In this case, the result of the *min* operation is a small fuzzy set 4. Sometimes this could be even a zero fuzzy set, which leads to Wall-Follow behavior as being the sole contributor to the resultant fuzzy set, as indicated by fuzzy set 5 and 6.

It can be noticed that in this second case the turn angle α is much smaller then in the first case, a characteristic of the Wall-Follow behavior which usually produces a small value for the turn angle.

The above mentioned cases are examples of how Goto-XY behavior interacts with Wall-Follow behavior. The command calculated by the system can be seen as a consensus of recommendations offered by multiple behaviors.

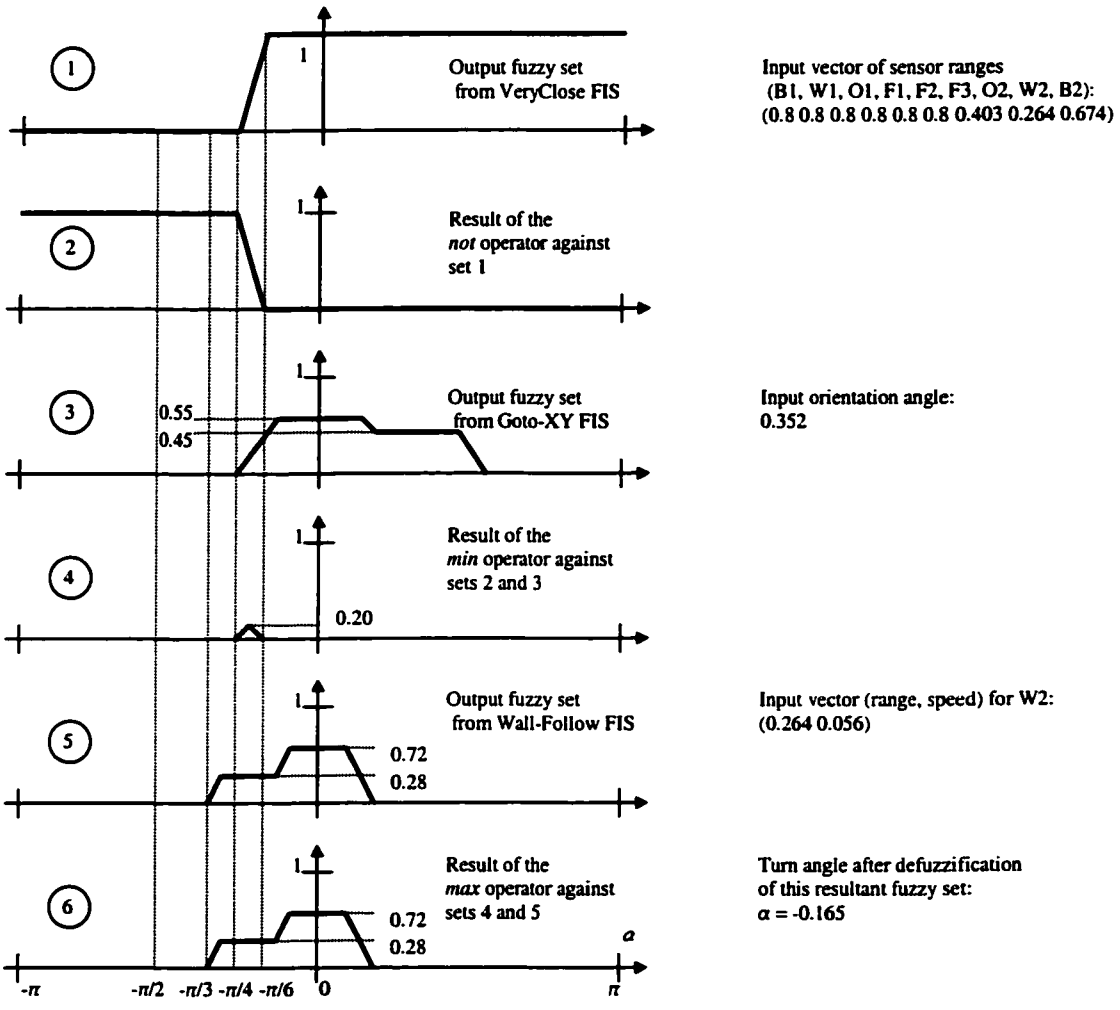

Figure 5.4: Fusion of fuzzy sets for robot and target on different sides of a wall

Chapter 6

Experimental results

6.1 Indoor environments

The neuro-fuzzy control system described in this thesis has been tested in simulated indoor environments. The main benefit of the simulation is the huge number of cases (start point, end point) which can be tested. In four different indoor environments, over 2,000 pairs of endpoints (robot start point, target point) have been tested. Another important benefit of simulation is that the tests do not require user intervention or presence, once the pairs of endpoints are specified by the user directly or through a data file. Clearly this level of testing is hard to be achieved in a real environment, and the importance of simulation is highlighted by other researchers too [31]. For validating a control model, the simulation becomes a necessary condition, but not a sufficient one. Testing in real indoor environments may be subject of future work.

The four indoor environments used for testing are shown in Figure 6.1. As endpoints every corner of every room and every endpoint of every corridor has been considered. For one indoor environment with n such endpoints, a number of $n(n-1)$ paths have been tested.

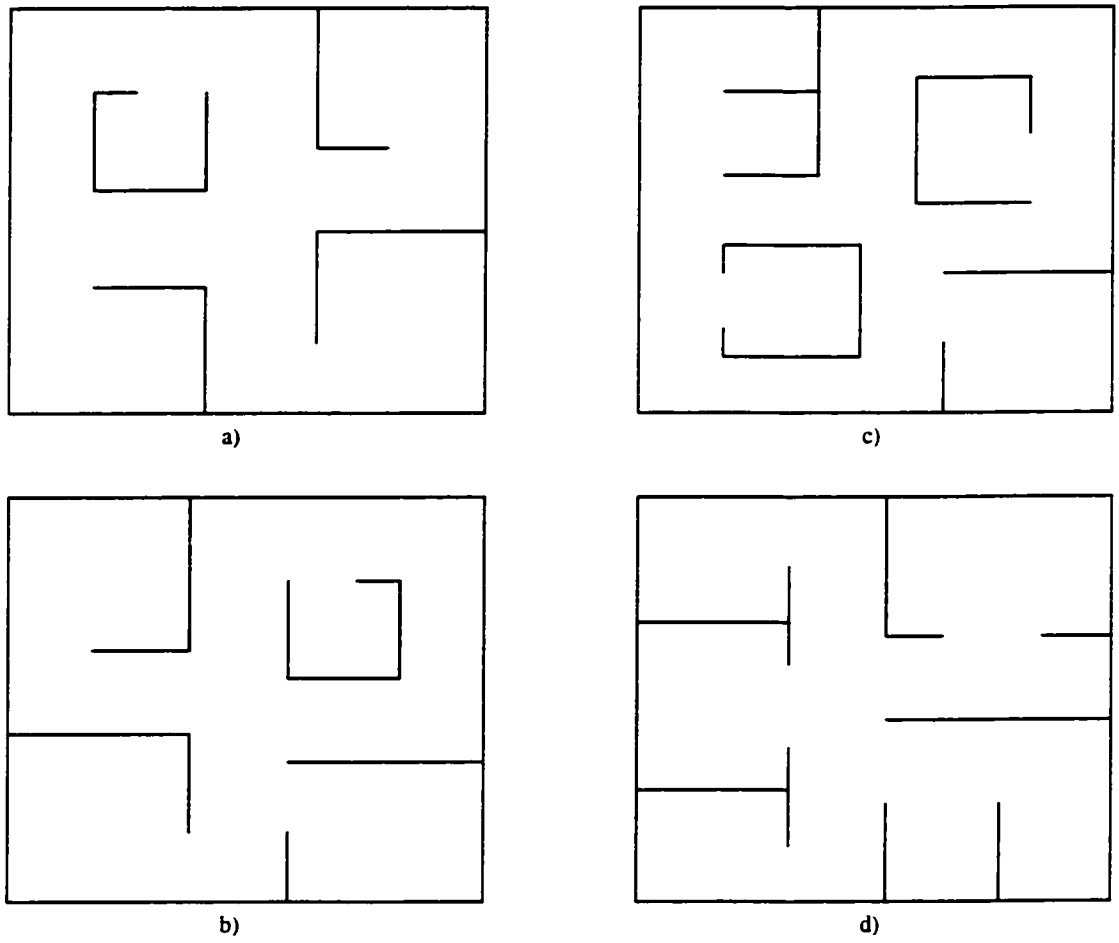

Figure 6.1: Samples of indoor environments used for testing

The simulation environment used in this work is based on a two-dimensional robot simulator written for Windows by G.W.Lucas [18], named Rossum's Playhouse(RP1). This is a tool specially designed for testing navigation algorithms and control logic for robots and it is based on a client-server architecture.

The server application was written in Java and it has been used with minor changes as provided by Rossum's Playhouse. It has the benefit of providing an existing graphical interface for the robot representation, and an excellent environment for testing

new indoor configurations. More details about the server application can be found in Appendix C.

The client application is completely new and it incorporates the neuro-fuzzy system implementation. The source code of the client application is included in Appendix D.

The representation of the robot in the simulated environment looks like in Figure 6.2, where the robot is shown on his way from a start point H ("home") to the target point T. One path between a robot start point and a target point is not necessarily the most optimal between these points, especially when the robot and target are placed initially in separate rooms. As described in Chapter 5, when the robot hits a wall, he will continue his search on his left or right side, depending on the angle between his moving direction and the wall. This might not be the shortest way to an exit from the room. Further, the total rotation angle of the robot could impose a change in the search direction.

For a given set of endpoints, the automated test monitored the time for each target search, which is the time elapsed between the departure of the robot from a start point H and his arrival at the target point T (Figure 6.2). A robot speed of 0.4 m/s has been used and sizes of indoor environments between 10x10 m and 13x13 m. If the robot travels between two opposite corners of such an indoor environment following a path parallel with two consecutive sides, roughly 60 seconds should be enough to reach the target point. But during a real target search, many turns/rotations are required and walls impose certain trajectories. All these increase significantly the time required for a target search. During experimental tests, it has been concluded that none of the target searches required more than 200 seconds to complete.

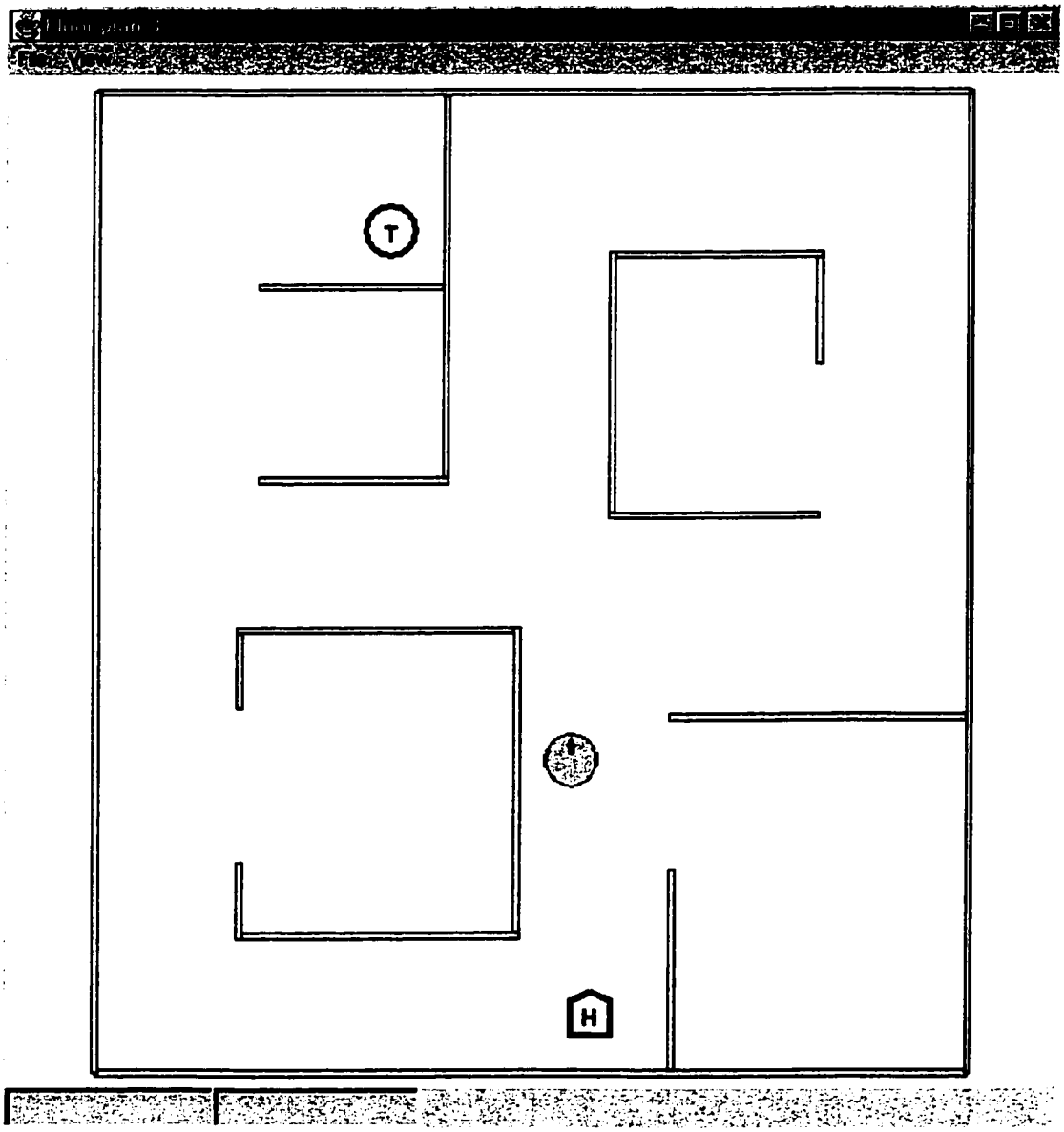

Figure 6.2: Simulated robot in an indoor environment

6.2 Client application tune-up

Three main objectives were targeted during the experimental tests:

- Reaching the target point within a reasonable period of time.
- Avoid collisions with obstacles.
- High degree of smoothness in robot trajectory.

If the first objective was easier to be achieved, avoiding collisions and obtaining a smoother path required a tune-up process of the client application. This was done in few areas described below:

1. Initially only a turn command has been considered. This command executes a turn around a fixed point situated outside the body of the robot. Basically both wheels move in the same direction but with different speeds, so that the robot executes a turn. Such a command could lead to positions in which collision was unavoidable, due to the fact that sensory system was disabled for the turn period.

   A rotation command was introduced for rotating the robot around its central axis. Both wheels move with the same speed, but in different directions so that the robot executes a pivot.

   Choosing between these two commands was a tune-up exercise. A distance of 0.4 m to the closest obstacle (half of the maximum sensor range) was identified to be the most suitable threshold value for selecting the appropriate command. If distance to obstacle is greater than the threshold value, a turn command

is used. If distance to obstacle is less than the threshold value, a rotation command is used.

2. Another threshold value was introduced for the turn angle calculated at the end of every control cycle. If this angle is below a certain limit, there is no point in executing the command. The robot follows the existing direction. Experimental tests indicated that a threshold of 0.05 radians provides a very smooth trajectory.

3. The robot speed used for testing was 0.4 m/s. Other speeds in the range $0.2-0.6$ m/s have been tried successfully. For speeds beyond 0.6 m/s, a tune-up of the adaptive fuzzy systems (§6.3) is required.

On a real robot, other conditions might impose future tune-up. For example: different sensor ranges or different processing power of the processor used in a real robot than the one used in simulation. A control cycle of 100 ms was used in the simulation, which gave enough time to the fuzzy systems to compute their outputs. On a real robot this might not be the case and the control cycle might need adjustment.

A note regarding the control cycle: every 100 ms, a timer allows the client application to request the sensor values, to perform the FIS calculation for the turn angle and to execute the appropriate command. Whenever a turn command or a rotation command is executed, next control cycle command is cancelled for the purpose of calculating the speed towards obstacle for every sensor. These speeds require two range indications after the turn/rotate command is completed. Therefore, there are cases when the control cycle action is delayed up to a 200 ms interval.

6.3 Training of the adaptive neuro-fuzzy inference systems

A separate application has been designed to allow data collection for training the neuro-fuzzy inference systems required by Go-Tangent behavior. This application allows robot re-positioning and robot rotation so that he can travel towards an obstacle at various angles. The obstacle can be extended at the limit to a wall, so we use our intuitive experience to stop the robot in time, avoiding a collision, then rotating him with the desired angle. All this data (range, speed, angle) is collected for training purposes.

Figure 6.3 shows a trajectory followed by the robot during data collection for training. The following steps are executed:

1. Position the robot in point A.
2. Rotate the robot towards a target point T.
3. Start the robot.
4. Stop the robot whenever user decides it is the turn time to avoid collision. Let S be this point.
5. Rotate the robot towards a point B, such that the new moving direction is parallel with the wall.
6. Start the robot. Training data (range, speed, turn angle β) is retained by the system and printed out. If the sample obtained is not acceptable, data can be discarded from the training set.

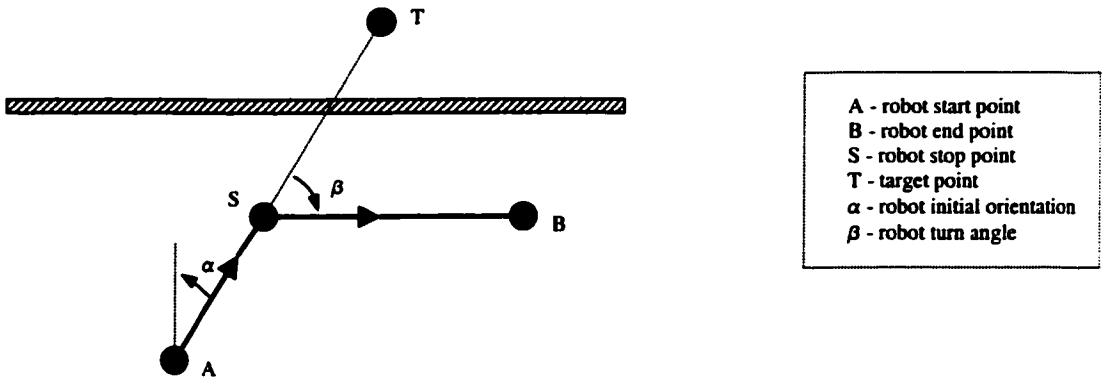

Figure 6.3: Sample of indoor environment used for training

The above steps must be repeated for every sampled data. The idea is to get sampled data for various angles α towards the target point T, and for each α to stop the robot at different distances from the wall, once the sensors of interest are changing their colors, meaning valid indications. By sensor of interest it is understood the sensors involved in the trained behavior. For example, for Go-Tangent-Oblique-Sensor FIS this is the `Oblique2` sensor which is situated on the left side of the robot.

It is worth mentioning here that the system is fully symmetric and therefore training the fuzzy inference systems for one side is enough. The same is true when defining any other FIS in this work, for example Wall-Follow or Turn-Corner FIS. When sensors from the other side offer valid indications, the same fuzzy inference systems are used and appropriate sign changes are applied to both input and output values. In this thesis all the fuzzy inference systems are defined for valid sensor indications from the left side.

For Go-Tangent-Oblique-Sensor FIS, 29 samples of data were used to generate the training data set, shown in Table 6.1.

No.	Range (m)	Speed (m/s)	Turn Angle (rad.)
1	6.9029e-01	5.3937e-01	-1.5210e+00
2	5.7449e-01	5.1754e-01	-1.4412e+00
4	5.9951e-01	4.6283e-01	-1.3895e+00
5	5.6220e-01	3.9988e-01	-1.2036e+00
6	5.8014e-01	3.9012e-01	-1.1734e+00
7	3.2199e-01	3.8423e-01	-1.1434e+00
8	4.8390e-01	3.1973e-01	-9.3385e-01
9	6.4523e-01	2.9948e-01	-8.5952e-01
10	6.4912e-01	2.7943e-01	-8.2223e-01
11	3.7148e-01	2.7559e-01	-7.5428e-01
12	2.2147e-01	2.9116e-01	-8.0548e-01
13	9.3093e-02	2.5427e-01	-6.9818e-01
14	4.1986e-01	2.8284e-01	-7.9632e-01
15	5.0205e-01	2.5306e-01	-6.8548e-01
16	3.6160e-01	2.7409e-01	-7.0831e-01
17	5.0634e-01	1.9029e-01	-4.6490e-01
18	3.2149e-01	1.6251e-01	-4.4406e-01
19	1.6094e-01	1.8182e-01	-6.3367e-01
20	5.1968e-01	1.2579e-01	-2.6597e-01
21	3.1647e-01	1.4681e-01	-3.0734e-01
22	3.6413e-01	5.5414e-01	-1.5813e+00
23	2.6482e-01	4.9247e-01	-1.4311e+00
24	1.4938e-01	3.8789e-01	-1.1805e+00
25	1.2594e-01	3.2409e-01	-9.6628e-01
26	1.4901e-01	2.7196e-01	-7.3217e-01
27	1.2975e-01	2.4647e-01	-6.2321e-01
28	1.1298e-01	1.8061e-01	-4.0587e-01
29	1.7789e-01	1.3148e-01	-2.3532e-01

Table 6.1: Training data set values for Go-Tangent-Oblique-Sensor FIS

The above mentioned steps were also applicable for training the Go-Tangent-Front-Sensor FIS. The single difference is that the data collected represents the front sensor range, front left sensor range and the turn angle respectively. The training data set used in this work is given in Table 6.2.

A last note regarding the checking data sets used as described in §4.2. They have been generated from the corresponding training data sets sets applying a random noise of 10%. Experimental results showed that a smaller or larger percentage does not influence the training. The absence of a checking data set has an influence on the

training error.

No.	Front range (m)	Left front range (m)	Turn Angle (rad.)
1	6.9029e-01	5.3937e-01	-1.5210e+00
1	3.7216e-01	5.5536e-01	-1.4979e+00
2	2.9512e-01	4.5134e-01	-1.4208e+00
3	3.1542e-01	6.4546e-01	-1.5647e+00
4	2.4937e-01	5.8327e-01	-1.5750e+00
5	5.3477e-01	6.7609e-01	-1.3640e+00
6	5.0357e-01	5.6020e-01	-1.1341e+00
7	5.5453e-01	6.0876e-01	-1.0705e+00
8	2.5319e-01	2.8950e-01	-1.0539e+00
9	3.8804e-01	4.0433e-01	-9.6659e-01
10	6.5299e-01	6.4964e-01	-9.4115e-01
11	2.3569e-01	2.4273e-01	-1.0116e+00
12	5.9425e-01	5.7324e-01	-9.5022e-01
13	6.5609e-01	5.6042e-01	-7.8240e-01
14	6.7724e-01	6.1324e-01	-8.4087e-01
15	6.2806e-01	5.0674e-01	-7.4021e-01
16	6.5124e-01	4.6984e-01	-6.5446e-01
17	4.6051e-01	3.1711e-01	-7.4432e-01
18	5.6680e-01	3.6556e-01	-6.7592e-01
19	4.3090e-01	2.4714e-01	-7.1899e-01
20	6.0279e-01	3.8393e-01	-6.0621e-01
21	4.2851e-01	2.1751e-01	-6.6304e-01
22	5.8974e-01	3.9481e-01	-6.9405e-01
23	6.1726e-01	2.7143e-01	-5.0550e-01

Table 6.2: Training data set values for Go-Tangent-Front-Sensor FIS

Chapter 7

Conclusions

This thesis presented a control system for a mobile robot based on a neuro-fuzzy approach. The goal is to obtain truly autonomous systems capable of navigating in indoor environments towards a target point. A hierarchy of robot behaviors has been defined to achieve this goal. The preceding chapters presented the low level (or primitive) behaviors and the high-level ones. Fuzzy inference systems were used for behavior implementation and, where the definition of such systems was not so obvious, neural networks were used for learning the parameters of the fuzzy systems. The input of this control system is represented by the sensory data. The output of the system is the turn angle to be executed by the robot.

The role of the control system is a complex one due to the fact that reaching the target implies multiple other goals achieved (avoid collisions, limited search time, smooth trajectory), conflict resolution and multiple interacting behaviors. The neuro-fuzzy based control system proposed in this work proved to be an effective control solution to these problems. The system has been tested in complex indoor environments, resembling with the real ones found in offices, schools, hospitals, etc.

This thesis brings the following contributions to the area of fuzzy behavior control

and robot navigation:

- New primitive behaviors are defined for a hierarchical architecture of fuzzy behaviors.
- The use of adaptive learning techniques for defining the fuzzy inference systems of the new primitive behaviors.
- A new command fusion method for combining different output fuzzy sets is introduced, method specific to mobile robot navigation problem.

The navigation results presented in this thesis were obtained through simulation experiments. Testing on a real robot is the final purpose of any robot simulation and may be the subject of future work. Also further research and experiments could be directed in these areas:

1. Preliminary experiments were done for applying fuzzy techniques to high-level behaviors, but none of them gave positive results. However, more work is required in this direction.
2. Introducing maps and self-localization on these maps may bring the potential investigation of optimal paths between two points.
3. Eliminating potential errors in the dead-reckoning system - for example, by adding landmarks in the indoor environment emitting visible light or infrared signals, so that the robot re-adjust his coordinates once such landmarks are detected.

Appendix A - Error signal for backpropagation method

The error signal for a node of a neural network is defined as [15]:

$$\epsilon_k = \frac{\partial E_p}{\partial x_k} \tag{8.1.1}$$

where E_p is the output error of the last node (Kth) for the pth measurement ($p = 1, ..., P$), and x_k is the output of the kth node. The output of a node (x_k) can be considered as a result of a function f_k which depends of a certain number of inputs M.

The output error is defined as:

$$E_p = (t_K - x_K)^2, \quad t_K = \text{target output}; \quad x_K = \text{measured output} \tag{8.1.2}$$

The error signal for this case is:

$$\epsilon_{Kp} = 2(x_K - t_K) \tag{8.1.3}$$

and can be computed for every measurement p of the training data set. This error signal for the node in the final layer is the starting point of calculating backward the error signals for nodes from previous layers, until we reach the node of interest k which depends on parameter α to be identified. This is how the error signal in a

node i of a layer l with $N(l)$ nodes can be expressed as a function of error signals in the next layer $l+1$:

$$\epsilon_{l,i} = \frac{\partial E_p}{\partial x_{l,i}} = \sum_{n=1}^{N(l+1)} \frac{\partial E_p}{\partial x_{l+1,n}} \frac{\partial f_{l+1,n}}{\partial x_{l,i}} = \\ = \sum_{n=1}^{N(l+1)} \epsilon_{l+1,n} \frac{\partial f_{l+1,n}}{\partial x_{l,i}} \tag{8.1.4}$$

The error signal calculation continues in the backward direction until the node of interest is reached and $\epsilon_{l,i}$ has been determined. The derivative of the error with respect to a parameter α is now easy to calculate:

$$\frac{\partial E_p}{\partial \alpha} = \frac{\partial E_p}{\partial x_{l,i}} \frac{\partial f_{l,i}}{\partial x_{l,i}} = \epsilon_{l,i} \frac{\partial f_{l,i}}{\partial x_{l,i}} \tag{8.1.5}$$

This flow of calculation has to be repeated for each measurement p. The gradient vector can be calculated now according to equation (4.2.8).

The algorithm described above has to be performed for each parameter, completing an epoch. Several epochs must be executed until the error $\sum_{p=1}^{P} E_p$ is minimized.

Appendix B - Server application for the simulator

Rossum's Playhouse robot simulator RP1 implements a client-server architecture [18] (http://rossum.sourceforge.net). Any robot simulation involves two executables, a client and a server. The server models robot bodies, tracks their movements, simulates physical interactions, and processes sensor activity. All robot control functions come from the RP1 clients, such as the neuro-fuzzy based control system described in this work.

The server application accepts client connections and receives from the client the instructions for moving within the simulated environment. The server provides the clients with information about sensor states, object collisions, and robot responses.

The client-server architecture of the simulator provides several benefits.

First, the two components could coexist in different environments. The server application has been written in Java but the client program is free of any dependency on Java, and it could be running even on another platform, as long as there is a network connection between the two processes.

Second, the server functionality and its GUI (Graphical User Interface) reduces considerably the design time of a simulation. The user can focus only on his robot control algorithm, exactly like on a real robot.

Third, the server application offers extensibility to a simulation environment due to its open architecture based on Java. Users can add layers in their applications which extend the capabilities of current simulation.

The server communicates with the client using the TCP/IP protocol. A configurable IP address and a port number are used for establishing the connection with the client. By default, the loopback address 127.0.0.1 is used and a port value of 7758, so that clients can look first on their own machines for a server. If client resides on another machine, this IP address has to be changed accordingly.

Once the connection established, the server accepts requests from clients and sends messages named events to clients. Examples of requests are: Motion request, Halt request, Placement request (specific to the simulation). Examples of events are: Motion event, Range Sensor event, Contact Sensor event.

The server has the capability of understanding floor plans described with a very simple grammar. Loading a file with the description of the indoor environment is done immediately after start-up. The floor plan may include basic elements like walls, target position and robot home position.

Appendix C - Using dead-reckoning for self-localization

Dead-reckoning is the mechanism of calculating the distance travelled since the previous command. This information is provided by the wheel encoders, as either a wheel turn angle in radians or a number of steps executed by the wheel, knowing that a wheel can execute a certain number of steps per revolution. The conversion between the two methods is straightforward, so we will consider the real-valued angular specification which allows for more accurate control.

Let be r_1 the turn angle of the left wheel and r_2 the turn angle of the right wheel, in radians, with positive values when robot moves forward, negative values otherwise. This dead-reckoning information is used to re-localize the robot in a two dimensional space. Advanced techniques exist today for eliminating dead-reckoning errors and for an efficient self-localization [5, 35, 30]. For simulated environments we assume there are no errors. The formulas for doing this self-localization are used in the implementation of the robot control system presented in Appendix D. These formulas are demonstrated here.

A system of coordinates is placed in the left bottom corner of the indoor environment. Let be (x, y, β) the current position of the robot, where x and y are the

coordinates relative to the origin $C(0,0)$, and β is the orientation of the forward direction of the robot, as it can be seen in Figure 8.1. The current center of the robot is in $C(x', y')$. We want to determine the new coordinates of the robot (x'', y'', β'') after executing a turn with an angle γ around a fixed point M' situated at a distance x_2 from the right wheel. The fact that the right wheel was chosen (or a negative turn angle γ) has no impact on the final results. The selection was done just for an easier graphical representation.

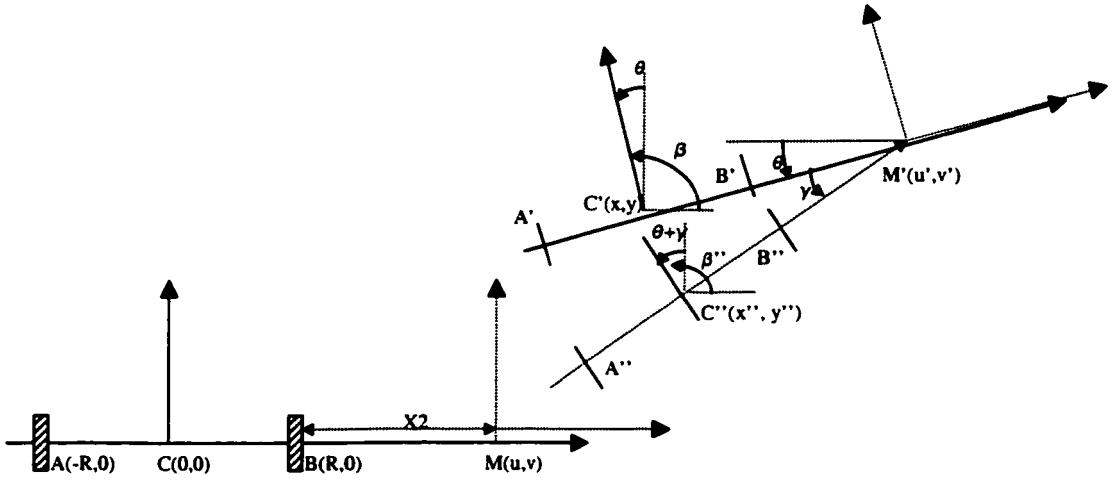

Figure 8.1: Robot wheels axis executing a turn with an angle γ

A direct observation is that $\beta'' = \beta + \gamma$ and also that $\theta = \beta - \frac{\pi}{2}$.

Distance x_2 can be calculated; if $x_1 = x_2 + 2R$ is the distance between the left wheel and the fixed point M, and W is the wheel radius, then:

$$\gamma = -\frac{r_1 W}{x_1} = -\frac{r_2 W}{x_2} \Rightarrow x_2 = \frac{2r_2 R}{r_1 - r_2} \tag{8.3.1}$$

and γ can be calculated as:

$$\gamma = -(r_1 - r_2)W/2R \tag{8.3.2}$$

Suppose the system of coordinates located in $C(0,0)$ is translated with (x,y) and rotated with θ, and no scaling is involved. Then a transformation matrix of the following form can be used to obtain the coordinates of a point $M'(u',v')$, transformed from a point $M(u,v)$:

$$\begin{bmatrix} cos\theta & -sin\theta & x \\ sin\theta & cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} \tag{8.3.3}$$

For $(u,v) = (x_2 + R, 0)$, we obtain:

$$\begin{aligned} u' &= x + (x_2 + R)cos\theta \\ v' &= y + (x_2 + R)sin\theta \end{aligned} \tag{8.3.4}$$

So u', v' are relative to the system originated in C (continuous arrows), as all the other points shown on Figure 8.1. If we consider point C relative to a system of coordinates originated in M (dotted arrows) which executes a translation $(u' - (x_2 + R), v')$ and a rotation $\theta + \gamma$ becoming a system originated in M' , then point $C(-(x_2 + R), 0)$ relative to M is transformed to $C''(u'', v'')$ relative to M also. The following transformation applies:

$$\begin{bmatrix} cos(\theta+\gamma) & -sin(\theta+\gamma) & u' - (x_2 + R) \\ sin(\theta+\gamma) & cos(\theta+\gamma) & v' \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -(x_2 + R) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} u'' \\ v'' \\ 1 \end{bmatrix} \tag{8.3.5}$$

u'', v'' can be expressed as a function of u', v' and ultimately as a function of x, y (8.3.4). To obtain these coordinates relative to point C, a last translation is required: $x'' = u'' + (x_2 + R)$ and $y'' = v''$. Applying all the substitutions for u'', v'', x_2 and θ,

the coordinates for point C'' are obtained:

$$\begin{aligned} x'' &= x + \frac{r_1 + r_2}{r_1 - r_2} \cdot R \cdot (sin\beta - sin(\beta + \gamma)) \\ y'' &= y + \frac{r_1 + r_2}{r_1 - r_2} \cdot R \cdot (cos(\beta + \gamma) - cos\beta) \end{aligned} \tag{8.3.6}$$

Appendix D - Source code for the control system

The neuro-fuzzy based control system has been implemented in C as the client application of the simulator. The source code follows in the next pages.

```
#include "robot.h"
#include "fis.h"
#include <math.h>

/* debug flag activations */
#if 0
#define RANGE_SENSOR_DEBUG
#define CALC_ROBPOS_DEBUG
#define RUN_STATE_DEBUG
#endif
#if 1
#define ROBOT_CONTROL_CYCLE_DEBUG
#define MOTION_HALTED_DEBUG
#define CONTACT_SENSOR_DEBUG
#define TURNCMD_DEBUG
#define ROTATECMD_DEBUG
#define GOSTRAIGHT_DEBUG
#define STARTROBOT_DEBUG
#define GOTO_TARGET_DEBUG
#define CONTOUR_FOLLOW_DEBUG
#define OPENSPACE_DEBUG
#define DEADEND_DEBUG
#define SELECT_HIGHLEVEL_BHV_DEBUG
#endif

/* global variables */
double r_speed = R_SPEED;
long useRotate = FALSE;
long rotateAtStartup = TRUE;
int  collisionFlag = FALSE;

/* vars used for regression test */
long regressionTest = REG_NOT_ACTIVE;
long contacts = 0;
long collisions = 0;
long contactSensorsCount = 0;
long contactSensorsEvents = 0;
double xStartColTest[MAX_COL_TST];
double yStartColTest[MAX_COL_TST];
double xEndColTest[MAX_COL_TST];
double yEndColTest[MAX_COL_TST];
double xCrtColTest[MAX_COL_TST];
double yCrtColTest[MAX_COL_TST];
long   pathColTest[MAX_COL_TST];

double xTest[MAX_TST];
double yTest[MAX_TST];
double xStartTest[MAX_TST*MAX_TST];
double yStartTest[MAX_TST*MAX_TST];
double xEndTest[MAX_TST*MAX_TST];
double yEndTest[MAX_TST*MAX_TST];
long   resultTest[MAX_TST*MAX_TST];
long   testPoints = 0;
long   testPaths = 0;
long   crtTestPath = 0;
long   startTestingTime = 0;
long   initRegFile = 0;
long   regressionStarted = FALSE;
long   startTimeForThisTestPath;



/* globals and defs used calling FIS API - Fuzzy Inference System */
#define  MF_POINT_N 101
FIS     *fis2;
double **fisMatrix2;
char    *fis_file2 = "veryclose.fis";
FIS     *fis3;
double **fisMatrix3;
char    *fis_file3 = "goTleftcheck.fis";
FIS     *fis4;
double **fisMatrix4;
```

```
char    *fis_file4 = "wallfollowleft.fis";
FIS     *fis5;
double **fisMatrix5;
char    *fis_file5 = "towardstarget.fis";
FIS     *fis6;
double **fisMatrix6;
char    *fis_file6 = "turncornerleft.fis";
FIS     *fis7;
double **fisMatrix7;
char    *fis_file7 = "goTfrontleftcheck.fis";


/* local used typedefs */
typedef struct {
   RsClient *rsClient;
   int       requestCount;
}UserData;


/* variables used by the debugging system for tracing the code */
int codepath = 0;

char *sensorName[10] = {
  "RightBack",
  "RightWheel",
  "Right",
  "Front",
  "Left",
  "LeftWheel",
  "LeftBack",
  "LeftFwd",
  "RightFwd",
  "NoSensor" };

char *defaultfnName = "No func";
char *wallFollowName = "wallFollow";
char *wallFollow2Name = "wallFollow2";
char *gotoxyName = "gotoxy";
char *goTangentName = "goT";
char *wallFollowAndGotoxyName = "wallFollowAndGotoxy";
char *turnCornerName = "turnCorner";
char *turnAroundName = "turnAround";


/*****************************************************************************
;
;   turnAroundPrimitive  - Calculate turn angle for the turnAround primitive
;
;   This function calculates the turn angle (precisely rotation angle)
;   for the turnAround primitive behavior. This primitive behavior is
;   the single one which does not need a fuzzy implementation, the angle is
;   M_PI (positive or negative, depending on the totalRotationAngle of the
;   robot)
;
;   RETURNS
;     angle = the turn angle
;
;   INTERFACE NOTES
;     It will always request a rotation instead of a turn (setting the
;     closePtr flag).
;
*****************************************************************************/
double
turnAroundPrimitive(
                    RsClient *rsClient,    /* pointer to client structure */
                    long *closePtr         /* flag if rotation is required */
                   ) {

  double turnAngle;


  /* turnAround imposes a rotation of 180 deg. in the apropriate direction;
```

```
    * a change of state follows if requested (only in goto_target behavior)
    */

   if(rsClient->totalRotationAngle > 0.0) {
     turnAngle = -M_PI;
   } else {
     turnAngle = M_PI;
   }
   *closePtr = TRUE;
   rsClient->behavior = B_TURNAROUND;

   return turnAngle;
}


/*****************************************************************************
;
;   gotoxyPrimitive  - Calculate turn angle for the gotoxy primitive
;
;   This function calculates the turn angle  for the gotoxy primitive,
;   based on the fuzzy system with one input (calculated angle towards target)
;   and one output (turn angle).
;
;   RETURNS
;     angle = the turn angle
;
;   IMPLEMENTATION
;     The advantage of having this primitive implemented as a fuzzy system
;     is the fact that it can be in this way combined with other fuzzy
;     primitives (e.g. wallFollow) and also does not rely on input 'angle'
;     as being very accurate (on a real robot the dead-reckoning is never
;     perfect)
;
*****************************************************************************/
double
gotoxyPrimitive(
                RsClient *rsClient,  /* pointer to client structure */
                double angle         /* robot rotation angle towards target */
               ) {

    fis5->input[0]->value = angle;
    fisEvaluate(fis5, MF_POINT_N, DEFUZZIFY);
    codepath += 10;
    rsClient->behavior = B_GOTOXY;

    return fis5->output[0]->value;
}

/*****************************************************************************
;
;   wallFollowPrimitive  - Calculate turn angle for the wallFollow primitive
;
;   This function calculates the turn angle  for the wallfollow
;   primitive, based on the fuzzy system with two inputs (distance to the
;   wall and speed towards the wall) and one output (turn angle).
;
;   RETURNS
;     angle = the turn angle
;
;   IMPLEMENTATION
;     Te primitive can be used for both wheel sensors and backward sensors.
;     The FIS file has been defined for wheel sensors. In order to use it for
;     backward sensors too, a scale factor of sqrt(2) is applied to the
;     input, considering the 45 deg. orientation of backward sensor relative
;     to the wheel sensor
;
*****************************************************************************/
double
wallFollowPrimitive(
                    long sensorIndex,  /* sensor which gives the fuzzy input */
                    RsClient *rsClient /* pointer to the client structure */
                   ) {
```

```
  long rightIndex, leftIndex;
  double scaleFactor = 1.0;
  double rangeInput;
  double speedInput;
  double turnAngle;

  if (isSideSensor(sensorIndex) == FALSE &&
      isWheelSensor(sensorIndex) == FALSE &&
      isBackwardSensor(sensorIndex) == FALSE) {
    printf("WALLFOLLOW INDEX = %d\n", sensorIndex);
  }

  /* determine if it's a backward sensor indication or not;
   * we call this function only for wheel sensors, but just in case for later
   */
  if (isBackwardSensor(sensorIndex) == TRUE) {
    rightIndex = RIGHT_BACKWARD_SENSOR_INDEX;
    leftIndex = LEFT_BACKWARD_SENSOR_INDEX;
    scaleFactor = 1.41;  /* aprox sqrt(2) - backward sensors at 45 deg. */
    /* (there is only one FIS file defined only for wheel sensors) */
  } else {
    rightIndex = RIGHT_WHEEL_SENSOR_INDEX;
    leftIndex = LEFT_WHEEL_SENSOR_INDEX;
  }

  if(rsClient->reverseSignForRightSide == TRUE) {
    rangeInput = rsClient->rangeSensor[rightIndex].range;
    speedInput = rsClient->rangeSensor[rightIndex].sSpeed;
    codepath += 40;
  } else {
    rangeInput = rsClient->rangeSensor[leftIndex].range;
    speedInput = rsClient->rangeSensor[leftIndex].sSpeed;
    codepath += 41;
  }

  rangeInput = rangeInput / scaleFactor;

  rangeInput = min(max(rangeInput, WALLFOLLOW_MINRANGE_FIS_VAL),
                   WALLFOLLOW_MAXRANGE_FIS_VAL);
  speedInput = min(max(speedInput, WALLFOLLOW_MINSPEED_FIS_VAL),
                   WALLFOLLOW_MAXSPEED_FIS_VAL);

  fis4->input[0]->value = rangeInput;
  fis4->input[1]->value = speedInput;
  fisEvaluate(fis4, MF_POINT_N, DEFUZZIFY);
  turnAngle = fis4->output[0]->value;
  if(rsClient->reverseSignForRightSide == TRUE) {
    turnAngle = -turnAngle;
  }
  rsClient->behavior = B_WALLFOLLOW;

  return turnAngle;
}

/*****************************************************************************
;
;   turnCornerPrimitive  - Calculate turn angle for the turnCorner primitive
;
;   This function calculates the turn angle  for the turnCorner
;   primitive, based on the fuzzy system with one input (sensor speed)
;   and one output (turn angle). This primitive is used only for backward
;   sensors
;
;   RETURNS
;     angle = the turn angle
;
;
*****************************************************************************/
double
turnCornerPrimitive(
                   long sensorIndex,  /* sensor which gives the fuzzy input */
```

```
                    RsClient *rsClient /* pointer to the client structure */
                   ) {

  long rightIndex, leftIndex;
  double speedInput;
  double turnAngle;

  if (isBackwardSensor(sensorIndex) == FALSE) {
    printf("TURNCORNER INDEX = %d\n", sensorIndex);
  }

  rightIndex = RIGHT_BACKWARD_SENSOR_INDEX;
  leftIndex = LEFT_BACKWARD_SENSOR_INDEX;

  if(rsClient->reverseSignForRightSide == TRUE) {
    speedInput = rsClient->rangeSensor[rightIndex].sSpeed;
    codepath += 60;
  } else {
    speedInput = rsClient->rangeSensor[leftIndex].sSpeed;
    codepath += 61;
  }

  speedInput = min(max(speedInput, TURNCORNER_MINSPEED_FIS_VAL),
                   TURNCORNER_MAXSPEED_FIS_VAL);

  fis6->input[0]->value = speedInput;
  fisEvaluate(fis6, MF_POINT_N, DEFUZZIFY);
  turnAngle = fis6->output[0]->value;
  if(rsClient->reverseSignForRightSide == TRUE) {
    turnAngle = -turnAngle;
  }
  rsClient->turnCornerVal = turnAngle;
  rsClient->behavior = B_TURNCORNER;

  return turnAngle;
}

/*****************************************************************************
;
;   wallFollowAndGotoxyPrimitive  - Calculate turn angle for the
;                                   wallFollowAndGotoxy primitive
;
;   This function calculates the turn angle  for the wallFollowAndGotoxy
;   primitive, implementing the command fusion between two primitives:
;   wallFollow and gotoxy.
;
;   RETURNS
;     angle = the turn angle
;
*****************************************************************************/
double
wallFollowAndGotoxyPrimitives(
                    long sensorIndex,  /* sensor which gives the fuzzy input */
                    RsClient *rsClient,/* pointer to the client structure */
                    double angle       /* robot rotation angle towards target*/
                   ) {

  long rightIndex, leftIndex;
  double scaleFactor;
  double rangeInput;
  double speedInput;
  double turnAngle;

  if (isSideSensor(sensorIndex) == FALSE &&
      isWheelSensor(sensorIndex) == FALSE &&
      isBackwardSensor(sensorIndex) == FALSE) {
    printf("WALLFOLLOWANDGOTOXY INDEX = %d\n", sensorIndex);
  }

  if (isBackwardSensor(sensorIndex) == TRUE) {
    rightIndex = RIGHT_BACKWARD_SENSOR_INDEX;
    leftIndex = LEFT_BACKWARD_SENSOR_INDEX;
```

```
    scaleFactor = 1.41;  /* aprox sqrt(2) - backward sensors at 45 deg. */
    /* (one FIS file defined only for wheel sensors) */
  } else {
    rightIndex = RIGHT_WHEEL_SENSOR_INDEX;
    leftIndex = LEFT_WHEEL_SENSOR_INDEX;
    scaleFactor = 1.0;
  }

  if(rsClient->reverseSignForRightSide == TRUE) {
    rangeInput = rsClient->rangeSensor[rightIndex].range;
    speedInput = rsClient->rangeSensor[rightIndex].sSpeed;
    codepath += 30;
  } else {
    rangeInput = rsClient->rangeSensor[leftIndex].range;
    speedInput = rsClient->rangeSensor[leftIndex].sSpeed;
    codepath += 31;
  }

  rangeInput = rangeInput / scaleFactor;

  rangeInput = min(max(rangeInput, WALLFOLLOW_MINRANGE_FIS_VAL),
                   WALLFOLLOW_MAXRANGE_FIS_VAL);
  speedInput = min(max(speedInput, WALLFOLLOW_MINSPEED_FIS_VAL),
                   WALLFOLLOW_MAXSPEED_FIS_VAL);

  fis4->input[0]->value = rangeInput;
  fis4->input[1]->value = speedInput;
  fisEvaluate(fis4, MF_POINT_N, DONT_DEFUZZIFY);
  if(rsClient->reverseSignForRightSide == TRUE) {
    fisReverseSign(fis4, MF_POINT_N);
  }

  fis5->input[0]->value = angle;
  fisEvaluate(fis5, MF_POINT_N, DONT_DEFUZZIFY);

  fis2->input[0]->value = min(max(rsClient->rangeSensor[0].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);
  /* range-135 - minus sensors are on the right side of the robot,
   * while plus sensors on the left side (trigonometric convention)
   */
  fis2->input[1]->value = min(max(rsClient->rangeSensor[1].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range-90 */
  fis2->input[2]->value = min(max(rsClient->rangeSensor[2].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range-45 */
  fis2->input[3]->value = min(max(rsClient->rangeSensor[8].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range-22 */
  fis2->input[4]->value = min(max(rsClient->rangeSensor[3].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range0 */
  fis2->input[5]->value = min(max(rsClient->rangeSensor[7].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range+22 */
  fis2->input[6]->value = min(max(rsClient->rangeSensor[4].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range+45 */
  fis2->input[7]->value = min(max(rsClient->rangeSensor[5].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /* range+90 */
  fis2->input[8]->value = min(max(rsClient->rangeSensor[6].range,
     VERYCLOSE_MINRANGE_FIS_VAL) , VERYCLOSE_MAXRANGE_FIS_VAL);  /*range+135 */

  fisEvaluate(fis2, MF_POINT_N, DONT_DEFUZZIFY);

  fisIntersect2(fis4, fis2, fis5, MF_POINT_N);

  turnAngle = fis4->output[0]->value;

  rsClient->behavior = B_WALLFOLLOWANDGOTOXY;

  return turnAngle;
}


/*****************************************************************************
;
;   goTangentPrimitive  - Calculate turn angle for the goTangent primitive
```

```
;
;   This function calculates the turn angle  for the goTangent
;   primitive, implementing the command fusion between front sensor goTangent
;   behavior and side sensor goTangent behavior.
;
;   RETURNS
;     angle = the turn angle
;
;   IMPLEMENTATION
;     A maximum between the two commands (given by the front sensor and
;     side sensor) is considered as turn angle
;
*****************************************************************************/
double
goTangentPrimitive(
                   long mainSensor,    /* front sensor with a valid range */
                   long sideSensor,    /* side sensor with a valid range */
                   RsClient *rsClient  /* pointer to the client structure */
                  ) {

  long rightIndex, leftIndex;
  double scaleFactor;
  double rangeInput;
  double range2Input;
  double speedInput;
  double turnAngle1 = 0.0;
  double turnAngle2 = 0.0;
  int    frontCount = 0;

  if (isTowardsFrontSensor(mainSensor) == FALSE) {
    printf("GOTANGENT mainSensor Idx=%d\n", mainSensor);
  }

  /* FIRST, check if a front sensor needs goTangent activation */
  if (isSideSensor(mainSensor) == FALSE) {

    /* this is a front sensor indicating goTangent behavior */
    rangeInput = rsClient->rangeSensor[FRONT_SENSOR_INDEX].range;

    /* the fuzzy primitive for front sensor detection takes as input
     * 2 range values, from 2 sensors: the front one and the side one
     * (which focuses also straight ahead)
     */
    if (rsClient->reverseSignForRightSide == TRUE) {
      range2Input = rsClient->rangeSensor[RIGHT_FORWARD_SENSOR_INDEX].range;
      codepath += 1;
    } else {
      range2Input = rsClient->rangeSensor[LEFT_FORWARD_SENSOR_INDEX].range;
      codepath += 2;
    }


    /* bring the measured values into the range used for
     * training (taken from Matlab)
     */
    rangeInput = min(max(rangeInput, GOTANGENTFRONT_RANGEFRONT_MINTRAINED_VALUE),
                     GOTANGENTFRONT_RANGEFRONT_MAXTRAINED_VALUE);
    range2Input = min(max(range2Input, GOTANGENTFRONT_RANGELEFT_MINTRAINED_VALUE),
                      GOTANGENTFRONT_RANGELEFT_MAXTRAINED_VALUE);

    fis7->input[0]->value = rangeInput;
    fis7->input[1]->value = range2Input;
    fisEvaluate(fis7, MF_POINT_N, DEFUZZIFY);
    turnAngle1 = fis7->output[0]->value;
    /* I never got this from Matlab. Although the network is trained in the
     * proper range of values and only with negative values for the output,
     * still Matlab 'View rules' can yield positive results for certain
     * input values at the top range (close to INF_RANGE). Bug in Matlab ?
     * Till we get the answer, reverse eventually the sign
     */
    if (turnAngle1 > 0.0) {
      turnAngle1 = -turnAngle1;
```

```
    }

    if (rsClient->rangeSensor[FRONT_SENSOR_INDEX].range < INF_RANGE) {
      frontCount++;
    }
    if (rsClient->rangeSensor[LEFT_FORWARD_SENSOR_INDEX].range < INF_RANGE) {
      frontCount++;
    }
    if(rsClient->rangeSensor[RIGHT_FORWARD_SENSOR_INDEX].range < INF_RANGE) {
      frontCount++;
    }
    /* Matlab FIS file has been trained only for both indications being valid;
     * otherwise, turn hard M_PI/2
     */
    if(frontCount <= 1) {
      turnAngle1 = -M_PI/2.0;
    }

    if(rsClient->reverseSignForRightSide == TRUE) {
      turnAngle1 = -turnAngle1;
    }
  }


  /* SECOND, check if a side sensor needs goTangent activation */

  if (isSideSensor(sideSensor) == TRUE) {

    /* this is a side sensor indicating go_Tangent behavior */

    if (rsClient->reverseSignForRightSide == TRUE) {
      rangeInput = rsClient->rangeSensor[RIGHT_SENSOR_INDEX].range;
      speedInput = rsClient->rangeSensor[RIGHT_SENSOR_INDEX].sSpeed;
      codepath += 10;
    } else {
      rangeInput = rsClient->rangeSensor[LEFT_SENSOR_INDEX].range;
      speedInput = rsClient->rangeSensor[LEFT_SENSOR_INDEX].sSpeed;
      codepath += 20;
    }

    /* bring the measured values into the range used for
     * training (taken from Matlab)
     */
    rangeInput = min(max(rangeInput, GOTANGENT_RANGE_MINTRAINED_VALUE),
                     GOTANGENT_RANGE_MAXTRAINED_VALUE);
    speedInput = min(max(speedInput, GOTANGENT_SPEED_MINTRAINED_VALUE),
                     GOTANGENT_SPEED_MAXTRAINED_VALUE);

    fis3->input[0]->value = rangeInput;
    fis3->input[1]->value = speedInput;
    fisEvaluate(fis3, MF_POINT_N, DEFUZZIFY);
    turnAngle2 = fis3->output[0]->value;
    if (turnAngle2 > 0.0) {
      turnAngle2 = -turnAngle2;
    }
    if(rsClient->reverseSignForRightSide == TRUE) {
      turnAngle2 = -turnAngle2;
    }
    /* In order to limit the number of inputs in the training process,
     * goT has been splitted into training for front  sensors and side sensors;
     * here is a quick fix to solve a simple case (caught only by a FIS for
     * all 5 inputs): don't steer too much if front sensors do not indicate
     * anything !
     */
    if (rsClient->rangeSensor[FRONT_SENSOR_INDEX].range == INF_RANGE &&
        rsClient->rangeSensor[LEFT_FORWARD_SENSOR_INDEX].range == INF_RANGE &&
        rsClient->rangeSensor[RIGHT_FORWARD_SENSOR_INDEX].range == INF_RANGE) {
      turnAngle2 = turnAngle2 / 2.0;
    }
  }

  rsClient->behavior = B_GOTANGENT;
```

```
  /* if side sensor has a valid indication and front sensor was hard limited,
   * then take into account only side sensor, otherwise take the max indication
   * between the side and the front sensor
   */
  if (turnAngle2 != 0.0 && frontCount <= 1) {
    return turnAngle2;
  } else {
    if(fabs(turnAngle1) > fabs(turnAngle2)) {
      return turnAngle1;
    } else {
      return turnAngle2;
    }
  }
}

/*****************************************************************************
;
;   openSpace  - Implement the Open-Space behavior
;
;   This function implements the openSpace behavior of the robot. Currently
;   only one primitive behavior is used: gotoxy.
;
;   RETURNS
;     angle = the turn angle
;
;
*****************************************************************************/
double
openSpace(
          RsClient *rsClient,     /* pointer to the client structure */
          double angle,           /* robot rotation angle towards target */
          long crtTime            /* current time (in usec) */
         ) {

  double turnAngle;

  /* On the simulator, the walls are very thin and a contour_follow behavior
   * could loose the track (wall sensor does not sense the thin wall) when
   * a turnCorner is executed. Only for this case, help that behavior.
   * On the real target this thing should never happen.
   * Just in case it happens, leave the code here.
   */
  if (rsClient->behavior == B_TURNCORNER && rsClient->bState != GOTO_TARGET) {
      turnAngle = rsClient->turnCornerVal;

#ifdef OPENSPACE_DEBUG
      printf("TURNCORNER in an openSpace case: angle=%f, rotAngle=%f, TURNANGLE=%f\n", angle, rs
Client->totalRotationAngle, turnAngle);
#endif
      return turnAngle;
  }


  turnAngle = gotoxyPrimitive(rsClient, angle);

  /* if too big the angle, leave something for the next cycle; better
   * to turn the robot in a couple of steps instead of one big chunk
   */
  if (turnAngle > MAX_TURN_ALLOWED) {
    turnAngle = MAX_TURN_ALLOWED;
  } else if (turnAngle < -MAX_TURN_ALLOWED) {
    turnAngle = -MAX_TURN_ALLOWED;
  }

  rsClient->prevIndmaxSide = -1;

  /* openSpace should reset totalRotationAngle
   * initialize it so that we will have 0 in this field after turnCmd,
   * which calls updateRotAngle() doing totalRotationAngle+=turnAngle;
   * (turnAngle-angle) term is due to truncation at M_PI/2 (aproximating
   * the result of gotoxyPrimitive with 'angle')
```

```
    */
    rsClient->totalRotationAngle = -turnAngle + (turnAngle - angle);


#ifdef OPENSPACE_DEBUG
  printf("OPENSPACE crtTime=%d, turnAngle=%f, angle=%f tCornerVal=%f tArnd=%d\n", crtTime, turnA
ngle, angle, rsClient->turnCornerVal, rsClient->turnAround);
  dbgPrintSensors(rsClient);
#endif

  return turnAngle;
}


/*****************************************************************************
;
;   deadend  - Implement the Dead-End behavior
;
;   This function implements the Dead-End behavior of the robot. Currently
;   only one primitive behavior is used: turnAround.
;
;   RETURNS
;     angle = the turn angle
;
*****************************************************************************/
double
deadend(
         RsClient *rsClient,    /* pointer to the client structure */
         long *closePtr,        /* flag if rotation is required */
         long crtTime           /* current time (in usec) */
        ) {

  double turnAngle;

  turnAngle = turnAroundPrimitive(rsClient, closePtr);

#ifdef DEADEND_DEBUG
  printf("DEADEND crtTime=%d, turnAngle=%f\n", crtTime, turnAngle);
  dbgPrintSensors(rsClient);
#endif

  return turnAngle;
}

/*****************************************************************************
;
;   goto_target  - Implement the Goto-Target behavior
;
;   This function implements the Goto-Target behavior of the robot. Currently
;   all primitive behaviors are used.
;
;   RETURNS
;     angle = the turn angle
;     *dtoPtr = updated with the shortest distance to the obstacle allowed
;               for the robot when he executes a rotation
;     *closePtr = flag set to TRUE if a turn around a fixed point situated
;                 outside the robot's body (turnCmd) is required; it is set
;                 FALSE if a rotation around its axis is required (rotateCmd)
;
;   IMPLEMENTATION
;     The function calls the appropriate primitive behavior depending on the
;     most important sensor and the total rotation angle of the robot.
;
*****************************************************************************/
double
goto_target(
            RsClient *rsClient,     /* pointer to the client structure */
            double angle,           /* robot rotation angle towards target */
            long mainSensor,        /* sensor index with highest priority */
            long sideSensor,        /* side sensor with shortest dist to obst.*/
            long crtTime,           /* current time (in usec) */
            double *dtoPtr,         /* distance to obstacle */
```

```
            long *closePtr          /* to be set to TRUE if obst.is too close */
           ) {

  double turnAngle = 0.0;
  double totalRotAngle;
  char *dbgFnName = defaultfnName;

  codepath = 1000;

  totalRotAngle = fabs(rsClient->totalRotationAngle);
  if (isTowardsFrontSensor(mainSensor)) {
    /* sideSensor may be -1 here, indicating no range from the side sensor */
    turnAngle = goTangentPrimitive(mainSensor, sideSensor, rsClient);
    dbgFnName = goTangentName;
  } else if (isWheelSensor(mainSensor)) {
    if (totalRotAngle < rsClient->threshold_fusion ||
          (rsClient->totalRotationAngle * (mainSensor - 3)) > 0) {
      /* the last condition in the above OR is to determine if robot
       * turns around a closed polygon from outside or inside: from outside,
       * it is still allowed to follow the wall; from inside, it is time to
       * search an exit in the opposite direction;
       * mainSensor could be either 1 (right wheel) or 5 (left wheel).
       */
       turnAngle = wallFollowAndGotoxyPrimitives(mainSensor, rsClient, angle);
       dbgFnName = wallFollowAndGotoxyName;
    } else if (totalRotAngle < rsClient->threshold_high) {
       turnAngle = wallFollowPrimitive(mainSensor, rsClient);
       dbgFnName = wallFollow2Name;
    } else {
      turnAngle = turnAroundPrimitive(rsClient, closePtr);
      dbgFnName = turnAroundName;
    }
  } else if (isBackwardSensor(mainSensor)) {
    if (totalRotAngle < rsClient->threshold_fusion) {
      turnAngle = wallFollowAndGotoxyPrimitives(mainSensor, rsClient, angle);
      dbgFnName = wallFollowAndGotoxyName;
      if (turnAngle > (M_PI/2.0)) {
        turnAngle = (M_PI/2.0);
      } else if (turnAngle < -(M_PI/2.0)) {
        turnAngle = -(M_PI/2.0);
      }
      *dtoPtr = rsClient->rangeSensor[mainSensor].range / 2.0;
    } else {
      turnAngle = turnCornerPrimitive(mainSensor, rsClient);
      dbgFnName = turnCornerName;
      *dtoPtr = rsClient->rangeSensor[mainSensor].range / 2.0;
    }
  } else {
    printf("GOTO_TARGET TURNANGLE IS 0.0\n");
  }



#ifdef GOTO_TARGET_DEBUG
  /* just to avoid a dummy pointer in the debug message... */
  if (sideSensor == -1) {
    sideSensor = 9;
  }
  printf("GOTO_TARGET crtTime=%d, sensor=%s, sensSide=%s, TURNANGLE=%f, angle =%f, bSt=%d p=%d,
total=%f, %s, cols=%d, contSens=%d, testPaths=%d, crtTestPath=%d, minutes=%d tArnd=%d\n", crtTim
e, sensorName[mainSensor], sensorName[sideSensor], turnAngle, angle, rsClient->bState, codepath,
 rsClient->totalRotationAngle, dbgFnName, collisions, contactSensorsCount, testPaths, crtTestPat
h, (crtTime - startTestingTime)/60000, rsClient->turnAround);
  if (contacts > 0) {
    printf("Last Contact path=%d\n",pathColTest[contacts - 1]);
  }
  dbgPrintSensors(rsClient);

#endif

  return turnAngle;
}
```

```
/*****************************************************************************
;
;   contour_follow  - Implement the Contour-Follow behavior
;
;   This function implements the Contour-Follow behavior of the robot.
;   Currently only goTangent, wallFollow and turnCorner primitives are used.
;
;   RETURNS
;     angle = the turn angle
;     *dtoPtr = updated with the shortest distance to the obstacle allowed
;               for the robot when he executes a rotation
;     *closePtr = flag set to TRUE if a turn around a fixed point situated
;                 outside the robot's body (turnCmd) is required; it is set
;                 FALSE if a rotation around its axis is required (rotateCmd)
;
;   IMPLEMENTATION
;     The function calls the appropriate primitive behavior depending on the
;     most important sensor and the total rotation angle of the robot.
;     This behavior is not supposed to use gotoxy primitive;
;
*****************************************************************************/
double
contour_follow(
            RsClient *rsClient,     /* pointer to the client structure */
            long mainSensor,        /* sensor index with highest priority */
            long sideSensor,        /* side sensor with shortest dist to obst.*/
            long crtTime,           /* current time (in usec) */
            double *dtoPtr,         /* distance to obstacle */
            long *closePtr          /* to be set to TRUE if obst.is too close */
              ) {

  double turnAngle = 0.0;
  char *dbgFnName = defaultfnName;

  codepath = 2000;

  if (isTowardsFrontSensor(mainSensor)) {
    /* sideSensor may be -1 here, indicating no range from the side sensor */
    turnAngle = goTangentPrimitive(mainSensor, sideSensor, rsClient);
    dbgFnName = goTangentName;
  } else if (isWheelSensor(mainSensor)) {
     turnAngle = wallFollowPrimitive(mainSensor, rsClient);
     dbgFnName = wallFollow2Name;
  } else if (isBackwardSensor(mainSensor)) {
    turnAngle = turnCornerPrimitive(mainSensor, rsClient);
    dbgFnName = turnCornerName;
    /* next value is the radius of the arc for turn cmd */
    *dtoPtr = rsClient->rangeSensor[mainSensor].range / 2.0;
  } else {
    printf("CONTOUR_FOLLOW TURNANGLE IS 0.0\n");
  }


#ifdef CONTOUR_FOLLOW_DEBUG
  /* just to avoid a dummy pointer in the debug message... */
  if (sideSensor == -1) {
    sideSensor = MAX_RANGESENSORS;
  }
  printf("CONTOUR_FOLLOW crtTime=%d, sensor=%s, sensSide=%s, TURNANGLE=%f, bSt=%d p=%d, total=%f
, %s, cols=%d, contSens=%d, testPaths=%d, crtTestPath=%d, minutes=%d\n", crtTime, sensorName[mai
nSensor], sensorName[sideSensor], turnAngle, rsClient->bState, codepath, rsClient->totalRotation
Angle, dbgFnName, collisions, contactSensorsCount, testPaths, crtTestPath, (crtTime - startTesti
ngTime)/60000);
  if (contacts > 0) {
    printf("Last Contact path=%d\n",pathColTest[contacts - 1]);
  }
  dbgPrintSensors(rsClient);

#endif

  return turnAngle;
```

```
}


/*****************************************************************************
;
;   selectHighLevelBehavior  - Implement the switching technique between
;                              high-level behaviors
;
;   This function sets the behavioral state of the robot and select - the
;   the appropriate high-level behavior according to this state
;
;   RETURNS
;     angle = the turn angle
;     *dtoPtr = updated with the shortest distance to the obstacle allowed
;               for the robot when he executes a rotation
;     *closePtr = flag set to TRUE if a turn around a fixed point situated
;                 outside the robot's body (turnCmd) is required; it is set
;                 FALSE if a rotation around its axis is required (rotateCmd)
;
;   IMPLEMENTATION
;     The function calls the appropriate primitive behavior depending on the
;     most important sensor and the total rotation angle of the robot.
;     This behavior is not supposed to use gotoxy primitive;
;
*****************************************************************************/
double
selectHighLevelBehavior(
            double distance,        /* distance to the target */
            double angle,           /* robot rotation angle towards target */
            RsClient *rsClient,     /* pointer to the client structure */
            long crtTime,           /* current time (in usec) */
            double *dtoPtr,         /* distance to obstacle */
            long *closePtr          /* to be set to TRUE if obst.is too close */
                      ) {

  long i, indmax;
  double minRange;
  double turnAngle;
  long   mainSensor = -1;
  long   sideSensor = -1;

  *closePtr = FALSE;
  *dtoPtr = 0.0;

  filterSensors(rsClient);

  /* see if there is any valid sensor indication; calculate also minRange */
  indmax = -1;
  minRange = rsClient->rangeSensor[0].range;

  for (i = 0; i < MAX_RANGESENSORS; i++) {
    /* calculate the minRange */
    if ( rsClient->rangeSensor[i].range < minRange) {
      minRange = rsClient->rangeSensor[i].range;
    }

    /*  skip invalid ranges */
    if (rsClient->rangeSensor[i].sSpeed == 0.0 &&
        rsClient->rangeSensor[i].range == INF_RANGE) {
      continue;
    }
    indmax = i;
  }

#ifdef SELECT_HIGHLEVEL_BHV_DEBUG
  printf("HLEVEL: bState=%d, tRotAng=%f, c_bhv=%d\n", rsClient->bState, rsClient->totalRotationA
ngle, rsClient->change_bhv);
#endif

  /* if contour_follow behavior state (left/right), see if rotation in the
   * other direction (right/left) was enough to switch back to
   * goto_target state
```

```
   */
  if (rsClient->bState == CONTOUR_FOLLOW_LEFT) {
    if (rsClient->change_bhv == FALSE) {
      if (rsClient->totalRotationAngle < -rsClient->threshold_low) {
        rsClient->change_bhv = TRUE;
      }
    } else {
      if (rsClient->totalRotationAngle > -rsClient->threshold_low) {
        rsClient->bState = GOTO_TARGET;
      }
    }
  } else if (rsClient->bState == CONTOUR_FOLLOW_RIGHT) {
    if (rsClient->change_bhv == FALSE) {
      if (rsClient->totalRotationAngle > rsClient->threshold_low) {
        rsClient->change_bhv = TRUE;
      }
    } else {
      if (rsClient->totalRotationAngle < rsClient->threshold_low) {
        rsClient->bState = GOTO_TARGET;
      }
    }
  }


  /**** check for openSpace behavior ****/

  if (indmax == -1 || distance < minRange || distance < 3*MAX_DISTANCE_ERR_ALLOWED) {
    /* if all sensors are INF-RANGE or if distance to Target is small enough */
    turnAngle = openSpace(rsClient, angle, crtTime);

  /**** check for dead-end beahaviour ****/
  } else if (rsClient->rangeSensor[LEFT_WHEEL_SENSOR_INDEX].range < INF_RANGE &&
      rsClient->rangeSensor[RIGHT_WHEEL_SENSOR_INDEX].range < INF_RANGE &&
      (rsClient->rangeSensor[LEFT_SENSOR_INDEX].range < INF_RANGE &&
       rsClient->rangeSensor[RIGHT_SENSOR_INDEX].range < INF_RANGE &&
       rsClient->rangeSensor[FRONT_SENSOR_INDEX].range < INF_RANGE) &&
      distance > rsClient->rangeSensor[FRONT_SENSOR_INDEX].range) {
    /* last '>' : if distance to Target is small enough,we don't call deadend */
    turnAngle = deadend(rsClient, closePtr, crtTime);

  /**** check for goto_target behavior ****/
  } else if (rsClient->bState == GOTO_TARGET) {

  /*  at least one sensor is indicating something; goto_target state */
    selectMostImpSensor(rsClient, &mainSensor, &sideSensor, angle);
    turnAngle = goto_target(rsClient, angle, mainSensor, sideSensor,
                            crtTime, dtoPtr, closePtr);
    selectRotationIfTooClose(rsClient, mainSensor, sideSensor, closePtr);

  /**** contour_follow behavior ****/
  } else {

  /*  at least one sensor is indicating something; contour_follow state */
    selectMostImpSensor(rsClient, &mainSensor, &sideSensor, angle);
    turnAngle = contour_follow(rsClient, mainSensor, sideSensor,
                               crtTime, dtoPtr, closePtr);
    selectRotationIfTooClose(rsClient, mainSensor, sideSensor, closePtr);

  }


  /* do updates for the next control cycle: */
  /*  - change state if necessary */
  if (rsClient->behavior == B_TURNAROUND) {
    if(rsClient->totalRotationAngle > 0.0) {
      rsClient->bState = CONTOUR_FOLLOW_LEFT;
    } else {
      rsClient->bState = CONTOUR_FOLLOW_RIGHT;
    }
    rsClient->change_bhv = FALSE;
  }
```

```
  /* restore the sensor values - some values may have been filtered out */
  restoreFilteredSensors(rsClient);

#ifdef SELECT_HIGHLEVEL_BHV_DEBUG
  dbgPrintSensors(rsClient);
#endif

  return turnAngle;
}

/*****************************************************************************
;
;   robotControlCycle - Perform a control cycle activity of the robot
;
;   This function executes the activities required at every control cycle.
;   Mainly this consists of the turn angle calculation to be executed by
;   the robot.
;
;   IMPLEMENTATION
;     This function will execute either a turnCmd (turn around a fixed
;     point sitated outside the robot's body) or a rotateCmd (rotation
;     around robot's central axis).
;     This function should be called every POLL_RATE_MILLIS msec (100msec)
;
*****************************************************************************/
void
robotControlCycle(
            RsClient *rsClient,     /* pointer to the client structure */
            long crtTime            /* current time (in usec) */
                ) {

  double angle;
  double turnAngle;
  double dleft, dright;
  long   targetToLeft = FALSE;
  double dx, dy, distance;
  double dto;
  long   closeTarget;

  calculateRobotPositionInRunState(rsClient, crtTime);

  angle = calculateRotationAngle(rsClient->orientation_in_runstate,
          rsClient->xCurrent_in_runstate, rsClient->yCurrent_in_runstate,
          rsClient->xTarget, rsClient->yTarget);
  dx = rsClient->xCurrent_in_runstate - rsClient->xTarget;
  dy = rsClient->yCurrent_in_runstate - rsClient->yTarget;
  distance  = sqrt(dx*dx + dy*dy);

  if (fabs(distance) <= MAX_DISTANCE_ERR_ALLOWED &&
      fabs(angle) <= MAX_ANGLE_ERR_ALLOWED) {
    printf("TARGET FOUND !!\n");
    rsClient->disableSensor = TRUE;
    rsClient->targetFound = TRUE;
    RsSendMotionHaltedRequest(rsClient);
    return;
  }

#ifdef ROBOT_CONTROL_CYCLE_DEBUG
  printf("robotControlCycle crtTime=%d angle=%f dist=%f prevDist=%f navState=%d\n", crtTime, ang
le, distance, rsClient->prevDistance, rsClient->navState);
#endif

  if (rsClient->navState != NAV_STATE_STOP) {

    turnAngle = selectHighLevelBehavior(distance, angle, rsClient, crtTime, &dto, &closeTarget);

  } else {

    /* robot is at start-up; check front sensors before we 'release' it */

    if (rsClient->rangeSensor[FRONT_SENSOR_INDEX].range == INF_RANGE &&
        rsClient->rangeSensor[LEFT_FORWARD_SENSOR_INDEX].range == INF_RANGE &&
```

```
        rsClient->rangeSensor[RIGHT_FORWARD_SENSOR_INDEX].range == INF_RANGE) {
      /* free way; go ahead */
      goStraightCmd(rsClient, r_speed,  (distance/r_speed)*1000.0, crtTime);
      turnAngle = 0.0;
    } else {
      /* obstacle ahead; don't 'release' the robot yet, rotate it a bit */
      dto = 0.0;
      closeTarget = TRUE;
      turnAngle = rsClient->firstTimeRotAngle;
      if(turnAngle == 0.0) {
        if (rsClient->rangeSensor[LEFT_FORWARD_SENSOR_INDEX].range <
            rsClient->rangeSensor[RIGHT_FORWARD_SENSOR_INDEX].range) {
          turnAngle = -SMALL_ANGLE;
        } else {
          turnAngle = SMALL_ANGLE;
        }
      } else if (turnAngle < 0.0) {
        turnAngle = -SMALL_ANGLE;
      } else {
        turnAngle = SMALL_ANGLE;
      }
      rsClient->firstTimeRotAngle += turnAngle;
    }
  }

  rsClient->prevDistance = distance;

  /* execute the rotation given by turnAngle, if it is big enough */

  if (fabs(turnAngle) >= MAX_ANGLE_ERR_ALLOWED) {
    updateRobotPosition(rsClient);
    /* disable the sensor(s) until the turn cmd is over (hopefully we calculate
     * well the turn angle, so we dont colide...)
     */
    if (useRotate == TRUE || closeTarget == TRUE) {
      rotateCmd(rsClient, turnAngle, crtTime);
    } else {
      turnCmd(rsClient, turnAngle, turnAngle < 0.0 ? TRUE : FALSE , dto, crtTime);
    }
    rsClient->disableSensor = TRUE;
  }
}

/*****************************************************************************
;
;   placementEventHandler - Event handler for EVT_PLACEMENT
;
;   This function performs the activities required when a placement
;   event is reported as successful by the Server.
;   Various re-initializations in the rsClient structure are done, preparing
;   the robot for a new path to the target
;
*****************************************************************************/
void
placementEventHandler(
              RsPlacementEvent *e, /* pointer to event structure */
              void *userData       /* pointer to a user data which retains */
                                   /*   the pointer to the client structure */
                    ){

  UserData  *ud = userData;
  RsClient  *rsClient = ud->rsClient;
  int i;

  rsClient->xCurrent = rsClient->prev_x = e->x;
  rsClient->yCurrent = rsClient->prev_y = e->y;
  rsClient->orientation = rsClient->prev_orientation = e->orientation;

  for (i = 0; i < MAX_RANGESENSORS; i++) {
    rsClient->rangeSensor[i].requested = FALSE;
  }
  for (i = 0; i < MAX_CONTACTSENSORS; i++) {
```

```
    rsClient->contactSensor[i].disable = FALSE;
  }
  rsClient->minMeasurementsRangeSensor = 0;
  rsClient->lastCalculationTime = 0;

  rsClient->prevIndmaxSide = -1;
  rsClient->lastContactSensorOnLeftSide = FALSE;
  rsClient->turnAround = TRUE;
  rsClient->turnCornerVal = 0.0;

  rsClient->navState = NAV_STATE_STOP;
  rsClient->bState = GOTO_TARGET;
  rsClient->firstTimeRotAngle = 0.0;
  rsClient->prevDistance = INIT_DIST_TO_TARGET;
  if (regressionStarted == TRUE) {
    regressionTest = REG_DELAY_NEW_PATH;
  }
}

/*****************************************************************************
;
;   motionHaltedEventHandler - Event handler for EVT_MOTION_HALTED
;
;   This function performs the activities required when a motion halted
;   event is reported by the Server.
;   This function is called for every motion which stops, so it has a lot of
;   work to do to distinguish between various cases: robot movements,
;   buttons pressed from the user which might halt the robot, or regression
;   tests which has to jump to a new path test once a target search is over
;
*****************************************************************************/
void
motionHaltedEventHandler(
              RsMotionHaltedEvent *e, /* pointer to event structure */
              void *userData          /* pointer to user data which retains */
                                      /*the pointer to the client structure */
                        ){
  UserData  *ud = userData;
  RsClient  *rsClient = ud->rsClient;
  double     angle;
  double     distance, dx, dy;
  RsTimeoutRequest r;
  int        i;
  int        blocked = FALSE;

 if (rsClient->rLeftStarted == rsClient->rRightStarted &&
       rsClient->rLeftStarted > 0.0) {
   /* it was a goStraightCmd;check if robot is blocked(when overlaps sim.map)*/
    if (rsClient->prev_orientation == rsClient->orientation &&
        rsClient->prev_x == rsClient->xCurrent &&
        rsClient->prev_y == rsClient->yCurrent) {
      blocked = TRUE;
    }
 }

#ifdef MOTION_HALTED_DEBUG
  printf("motionHalted: simTime=%d, xCrt=%f, yCrt=%f, ornt=%f, xEv=%f, yEv=%f, orEv=%f, xT=%f, y
T=%f\n",
   e->simTime, rsClient->xCurrent, rsClient->yCurrent,
   rsClient->orientation, e->x, e->y, e->orientation, rsClient->xTarget,
   rsClient->yTarget);
#endif

if (e->causeOfHalt == RS_HALTED_ON_COLLISION) {
  correctTotalRotAngleIfRotNotCompleted(rsClient, e->simTime);
}

#if NO_SIMULATOR
  /* in real world, we don't have Server appl; rely on our own calculation */
  calculateRobotPosition(rsClient, e->simTime);
#else
  /* use info from Server appl. */
```

```
  rsClient->prev_orientation = rsClient->orientation;
  rsClient->prev_x = rsClient->xCurrent;
  rsClient->prev_y = rsClient->yCurrent;
  rsClient->orientation = e->orientation;
  rsClient->xCurrent = e->x;
  rsClient->yCurrent = e->y;
#endif

  if(rsClient->toBePlaced == TRUE) {
    rsClient->toBePlaced = FALSE;
    RsSendPlacementRequest(rsClient, " ", rsClient->rot_x, rsClient->rot_y);

  } else if(rsClient->toBeRotated != 0) {
    if (--rsClient->toBeRotated) {
      angle = calculateRotationAngle(rsClient->orientation, rsClient->xCurrent,
                                     rsClient->yCurrent, rsClient->rot_x, rsClient->rot_y);
      rotateCmd(rsClient, angle, e->simTime);
    }
  } else if (rsClient->toBeStopped == TRUE) {
    rsClient->toBeStopped = FALSE;
  } else {
    if (e->causeOfHalt == RS_HALTED_ON_COLLISION) {
      if (regressionTest == REG_IN_PROGRESS) {
        if (contacts < MAX_COL_TST) {
          xStartColTest[contacts] = xStartTest[crtTestPath];
          yStartColTest[contacts] = yStartTest[crtTestPath];
          xEndColTest[contacts] = xEndTest[crtTestPath];
          yEndColTest[contacts] = yEndTest[crtTestPath];
          xCrtColTest[contacts] = e->x;
          yCrtColTest[contacts] = e->y;
          pathColTest[contacts] = crtTestPath;
          contacts++;
        }
        if (collisionFlag == TRUE) {
          collisions++;
#ifdef MOTION_HALTED_DEBUG
          printf("***** COLLISION ****\n");
#endif
        } else {
          contactSensorsCount++;
        }
      }
      /* back-off */
      goStraightCmd(rsClient, -r_speed,  (0.6/r_speed)*1000.0, e->simTime);
      rsClient->disableSensor = TRUE;
    } else {
      dx = rsClient->xCurrent - rsClient->xTarget;
      dy = rsClient->yCurrent - rsClient->yTarget;
      distance  = sqrt(dx*dx + dy*dy);
      if (fabs(distance) <= MAX_DISTANCE_ERR_ALLOWED) {
        printf("TARGET FOUND !!!!\n");
        rsClient->disableSensor = TRUE;
        rsClient->targetFound = TRUE;
        rsClient->navState = NAV_STATE_STOP;
        if (regressionTest == REG_IN_PROGRESS) {
          regressionNextTest(rsClient, e->simTime);
        }
      } else {
        /* check first to see if previous move wasn't a back-off;
         * if yes, rotate a little bit (15 deg.)
         */
        if (rsClient->rLeftStarted == rsClient->rRightStarted &&
            rsClient->rLeftStarted < 0.0) {
          for (i = 0 ; i < MAX_CONTACTSENSORS; i++) {
            rsClient->contactSensor[i].disable = FALSE;
          }
          rotateCmd(rsClient, rsClient->lastContactSensorOnLeftSide == TRUE ?
                    -SMALL_ANGLE : SMALL_ANGLE, e->simTime);
          rsClient->disableSensor = TRUE;
          rsClient->blockingTime = 4;
        } else if (rsClient->navState != NAV_STATE_STOP) {
          /* not reached the target yet, go there */
```

```
          goStraightCmd(rsClient, blocked == FALSE ? r_speed : -r_speed,
                        (distance/r_speed)*1000.0, e->simTime);
          rsClient->disableSensor = FALSE;
          if(rsClient->blockingTime == 0) {
            rsClient->block_controlcycle = 2;
          } else {
            rsClient->block_controlcycle = rsClient->blockingTime;
            rsClient->blockingTime = 0;
          }
        } else if (rsClient->targetFound == FALSE) {
          rsClient->disableSensor = FALSE;
        } else if (rsClient->targetFound == TRUE) {
          if (regressionTest == REG_IN_PROGRESS) {
            regressionNextTest(rsClient, e->simTime);
          }
        }
      }
    }
  }
}

/*****************************************************************************
;
;   contactSensorEventHandler - Event handler for EVT_CONTACT_SENSOR
;
;   This function performs the activities required when a contact sensor
;   event is reported by the Server.
;   This function will force the robot to back-off if a contact sensor
;   is on.
;
*****************************************************************************/

void
contactSensorEventHandler(
              RsContactSensorEvent *e, /* pointer to event structure */
              void *userData           /* pointer to user data which retains*/
                                       /*the pointer to the client structure*/
                        ){
   UserData  *ud = userData;
   RsClient  *rsClient = ud->rsClient;
   double     backoff_speed;
   int        i;

   contactSensorsEvents++;

   for (i = 0; i < MAX_CONTACTSENSORS; i++) {
     if (rsClient->contactSensor[i].sensorId == e->sensorId) {
       break;
     }
   }

#ifdef CONTACT_SENSOR_DEBUG
   printf("CONTACT SENSOR Id=%d, status=%d, crtTime=%d, disableStatus=%d\n", e->sensorId, e->sta
tus, e->simTime, rsClient->contactSensor[i].disable);
#endif

   if (e->status == 0 ||
       i >= MAX_CONTACTSENSORS ||
       rsClient->navState == NAV_STATE_STOP ||
       rsClient->contactSensor[i].disable == TRUE) {
     return;
   }

   rsClient->contactSensor[i].disable = TRUE;

   correctTotalRotAngleIfRotNotCompleted(rsClient, e->simTime);

   if (e->sensorId <= RIGHT_COLLISION_SENSOR_ID) {
     /* the simulator server does not make any distinction between
      * a collision and a contact sensor; we do this here, for counting
      * purposes; anyway a different visual effect is displayed for each
      * event category: collision events (which exist only in the simulator
```

```
      * world) and contact sensor events
      */
     collisionFlag = TRUE;
   } else {
     collisionFlag = FALSE;
   }

   if (e->sensorId == LEFT_CONTACT_SENSOR_ID ||
       e->sensorId == RIGHT_CONTACT_SENSOR_ID ||
       /* on a real robot, remove next two lines */
       e->sensorId == LEFT_COLLISION_SENSOR_ID ||
       e->sensorId == RIGHT_COLLISION_SENSOR_ID) {
     /* this is a front contact sensor; back-off. */
     goStraightCmd(rsClient, -r_speed,  (0.6/r_speed)*1000.0, e->simTime);
     rsClient->disableSensor = TRUE;
     if (e->sensorId == LEFT_CONTACT_SENSOR_ID ||
         e->sensorId == LEFT_COLLISION_SENSOR_ID) {
       rsClient->lastContactSensorOnLeftSide = TRUE;
     } else {
       rsClient->lastContactSensorOnLeftSide = FALSE;
     }
   }
}

/*****************************************************************************
;
;   rangeSensorEventHandler - Event handler for EVT_RANGE_SENSOR
;
;   This function performs the activities required when a range sensor
;   event is reported by the Server.
;   The speed of the robot towards the detected obstacle is calculated,
;   if a previous measurement is available. If not, speed is considered 0.
;   If range is not valid, it is considered INF_RANGE;
;   At the end of this function, if all the requested sensors reported their
;   indication (i.e. this is the last reporting sensor), then the
;   robot control cycle is executed
;
*****************************************************************************/
void
rangeSensorEventHandler(
              RsRangeSensorEvent *e, /* pointer to event structure */
              void *userData         /* pointer to user data which retains */
                                     /* the pointer to the client structure */
                        ){
  UserData        *ud = userData;
  RsClient        *rsClient = ud->rsClient;
  int              ind;
  rangeSensorType *ptr;
  static int       dbg_cnt = 0;

  if (rsClient->disableSensor == TRUE) {
    return;
  }


  for(ind = 0; ind < MAX_RANGESENSORS; ind++) {
    if (e->sensorId == rsClient->rangeSensor[ind].sensorId) {

      ptr = &rsClient->rangeSensor[ind];

      if (ptr->requested == FALSE) {
        break;
      }
      ptr->requested = FALSE;

      ptr->prevTime = ptr->time;
      ptr->prevRange =  ptr->range;
      ptr->time = e->simTime;
      ptr->range = e->range;
      if (e->status == FALSE) {
        ptr->range = INF_RANGE;
      }
```

```
        if (ptr->prevRange == INF_RANGE || ptr->range == INF_RANGE) {
          ptr->sSpeed = 0.0;
        } else {
          ptr->sSpeed = (ptr->prevRange - ptr->range) /
            ((ptr->time - ptr->prevTime)/1000.0);
        }

#ifdef RANGE_SENSOR_DEBUG
   printf("rangeS: simTime=%d, cnt=%d, sensorIdx=%d, range=%f, status=%d, prevRange=%f, crtRange
=%f, speed=%f, reqCounter=%d, block_cc=%d\n", e->simTime, dbg_cnt++, ind, e->range, e->status, p
tr->prevRange, ptr->range, ptr->sSpeed, rsClient->rangeSensorRequestCounter, rsClient->block_con
trolcycle);

#endif

        rsClient->rangeSensorRequestCounter--;
        if(rsClient->rangeSensorRequestCounter == 0) {
          if (--rsClient->block_controlcycle == 0) {
            rsClient->block_controlcycle = 1;
            robotControlCycle(rsClient, e->simTime);
          }
        }

        break;
      }
    }
}

/*****************************************************************************
;
;   timeoutEventHandler - Event handler for EVT_TIMEOUT
;
;   This function performs the activities required when a timeout
;   event is reported by the Server.
;
;   All the sensors are requested to report their indication. This function
;   re-schedules itself after another POLL_RATE_MILLIS.
;
*****************************************************************************/
void
timeoutEventHandler(
              RsTimeoutEvent *e,    /* pointer to event structure */
              void *userData        /* pointer to a user data which retains */
                                    /*   the pointer to the client structure */
                   ) {
  UserData         *ud = userData;
  RsTimeoutRequest  r;
  RsClient         *rsClient = ud->rsClient;
  int               i;


  r.timeoutIndex = ++(rsClient->timeoutIndex);
  r.timeMillisec = POLL_RATE_MILLIS;
  RsSendTimeoutRequest(rsClient, &r);

  if (rsClient->disableSensor != TRUE) {
    for(i  = 0; i < MAX_RANGESENSORS; i++) {
      RsSendSensorRequest(rsClient, rsClient->rangeSensor[i].sensorId);
      rsClient->rangeSensor[i].requested = TRUE;
    }
    rsClient->rangeSensorRequestCounter = MAX_RANGESENSORS;
  }

  if (regressionStarted == TRUE && regressionTest == REG_DELAY_NEW_PATH) {
    regressionTest = REG_IN_PROGRESS;
    startRobot(rsClient, rsClient->xTarget, rsClient->yTarget,e->simTime);
  }
}

/*****************************************************************************
;
;   mouseClickEventHandler - Event handler for EVT_MOUSE_CLICK
```

```
;
;   This function performs the activities required when a mouse click
;   event is reported by the Server.
;
*****************************************************************************/
void
mouseClickEventHandler(
              RsMouseClickEvent *e, /* pointer to event structure */
              void *userData        /* pointer to a user data which retains */
                                    /*  the pointer to the client structure */
                    ) {
   UserData        *ud = userData;
   RsMotionRequest  r;
   RsClient        *rsClient = ud->rsClient;
   double           angle;
   FILE            *fp;
   long             ind;

   if (e->clickCount != MOUSE_SINGLE_CLICK) {
     fprintf(stderr, "No doubleclick; current mouse buttons supported:\n left button:      start
/stop the Robot\n right button:     rotate the Robot\n CTRL-left button: re-position the Robot\n
 ALT-left button:  re-position the Target\n");
     if(rsClient->navState != NAV_STATE_STOP) {
       RsSendMotionHaltedRequest(rsClient);
     }
     return;
   }

   switch(e->button) {
   case START_STOP_BUTTON:
     if(rsClient->navState == NAV_STATE_STOP) {
       if(regressionTest == REG_COLLECT_DATA) {
         regressionStartTest(rsClient, e->simTime);
         return;
       }
       if (rsClient->prevDistance == INIT_DIST_TO_TARGET) {
         startRobot(rsClient, rsClient->xTarget, rsClient->yTarget,e->simTime);
       } else {
         goStraightCmd(rsClient, r_speed,  (0.6/r_speed)*1000.0, e->simTime);
         rsClient->disableSensor = FALSE;
         rsClient->block_controlcycle = 2;
         rsClient->blockingTime = 0;
       }
     } else {
       rsClient->toBeStopped = TRUE;
       RsSendMotionHaltedRequest(rsClient);
     }
     break;
   case ROTATE_ROBOT_BUTTON:
     if(rsClient->navState != NAV_STATE_STOP) {
       rsClient->rot_x = e->x;
       rsClient->rot_y = e->y;
       rsClient->toBeRotated = 2;
       RsSendMotionHaltedRequest(rsClient);
     } else {
#ifdef DEBUG
       fprintf(stderr, "Rotate cmd to: y=%f, x=%f, crt orientation=%f\n", e->y, e->x, rsClient->
orientation);
#endif
       rsClient->toBeRotated = 1;
       angle = calculateRotationAngle(rsClient->orientation,
                    rsClient->xCurrent, rsClient->yCurrent, e->x, e->y);
       rotateCmd(rsClient, angle, e->simTime);
     }
     break;
   case REPOS_ROBOT_BUTTON:
     if(rsClient->navState != NAV_STATE_STOP) {
       rsClient->rot_x = e->x;
       rsClient->rot_y = e->y;
       rsClient->toBePlaced = TRUE;
       RsSendMotionHaltedRequest(rsClient);
     } else {
```

```
        RsSendPlacementRequest(rsClient, " ", e->x, e->y);
      }
      break;
    case REPOS_TARGET_BUTTON:
      if(rsClient->navState != NAV_STATE_STOP) {
        RsSendMotionHaltedRequest(rsClient);
      }
      if (regressionTest == REG_IN_PROGRESS) {
        dbgRegressionInfo();
        break;
      }
      rsClient->xTarget = e->x;
      rsClient->yTarget = e->y;
      if (regressionTest == REG_COLLECT_DATA) {
        if (testPoints < MAX_TST) {
          xTest[testPoints] = e->x;
          yTest[testPoints++] = e->y;
        } else {
          printf("Too many test points !\n");
        }
      }
      break;
    }
}

/*****************************************************************************
;
;   calculateRobotPosition - calc.  x, y and theta(orientation) in a STOP state
;
;   RETURNS
;     status = OK - calculation succesful;
;            = ERROR - data base of the robot corrupted - error.

;   IMPLEMENTATION
;     When robot is stopped, this function can be called to determine the new
;     coordinates of the robot. These are calculated as a function of how
;     much the robot travelled since last stop time (dead-reckoning)
;
*****************************************************************************/
int
calculateRobotPosition(
            RsClient *rsClient,     /* pointer to the client structure */
            long crtTime            /* current time (in usec) */
                    ) {
  double r1, r2;
  double alpha, beta, gamma;
  double new_x, new_y;
  double distance;

  long deltaT;

  if(rsClient->durationMillisStarted == 0) {
    return ERROR;
  }

  deltaT = crtTime -  rsClient->timeMotionStarted;

  r1 = rsClient->rLeftStarted;
  r2 = rsClient->rRightStarted;
  deltaT = rsClient->durationMillisStarted;

  if (fabs(r1 - r2) < 1.0e-06) {
    /* this was a goStraight command */

    distance = rsClient->speed * deltaT / 1000.0;

    new_x = rsClient->xCurrent + cos(rsClient->orientation) * distance;
    new_y = rsClient->yCurrent + sin(rsClient->orientation) * distance;

    rsClient->prev_orientation = rsClient->orientation;
#ifdef CALC_ROBPOS_DEBUG
    fprintf(stderr, "RobPos after goStraightCmd: r1=%f, r2=%f, deltaT=%d, durMillisStarted=%d, d
```

```
istance=%f, prev_x=%f, prev_y=%f, x=%f, y=%f, simTime=%d\n",
             r1, r2, deltaT, rsClient->durationMillisStarted, distance,
             rsClient->prev_x, rsClient->prev_y, new_x, new_y, crtTime);
#endif

  } else {
    /* this was a rotation/turn command */

    /* rotation angle since last halt */
    gamma = (r2 - r1) * WHEEL_RADIUS / WHEEL_BASE;
    /* previous orientation */
    beta = rsClient->orientation;
    /* current orientation */
    alpha = beta + gamma;

    distance = ((r1 + r2)/(r1 - r2))*(WHEEL_BASE/2.0);
    new_x = rsClient->xCurrent + distance * (sin(beta) - sin(alpha));
    new_y = rsClient->yCurrent + distance * (cos(alpha) - cos(beta));
#ifdef CALC_ROBPOS_DEBUG
    fprintf(stderr, "RobPos after rotateCmd: r1=%f, r2=%f, deltaT=%d, durMillisStarted=%d, gamma
=%f, beta=%f, alpha=%f, distance=%f, prev_x=%f, prev_y=%f, x=%f, y=%f, simTime=%d\n",
            r1, r2, deltaT, rsClient->durationMillisStarted, gamma, beta,
            alpha, distance, rsClient->prev_x, rsClient->prev_y, new_x, new_y, crtTime);
#endif


    rsClient->prev_orientation = rsClient->orientation;
    rsClient->orientation = alpha;
  }

  rsClient->prev_x = rsClient->xCurrent;
  rsClient->prev_y = rsClient->yCurrent;

  rsClient->xCurrent = new_x;
  rsClient->yCurrent = new_y;

  return OK;
}

/*****************************************************************************
;
;   calculateRobotPositionInRunState - calc.  x, y and theta in a RUN state
;
;   RETURNS
;     status = OK - calculation succesful;
;            = ERROR - data base of the robot corrupted - error.

;   IMPLEMENTATION
;     When robot is running, this function can be called to determine the new
;     coordinates of the robot. These are calculated as a function of how
;     much the robot travelled since last stop time (dead-reckoning) and
;     the current time supposing the robot stops
;
*****************************************************************************/
int
calculateRobotPositionInRunState(
            RsClient *rsClient,    /* pointer to the client structure */
            long crtTime           /* current time (in usec) */
                                ) {
  double r1, r2;
  double alpha, beta, gamma;
  double new_x, new_y;
  double distance;

  long deltaT;

  alpha = rsClient->orientation;

  if(rsClient->durationMillisStarted == 0) {
    return ERROR;
  }
```

```
  deltaT = crtTime -  rsClient->timeMotionStarted;

  r1 =  rsClient->rLeftStarted * (double)deltaT /
              (double)(rsClient->durationMillisStarted);
  r2 =  rsClient->rRightStarted * (double)deltaT /
              (double)(rsClient->durationMillisStarted);

  if (fabs(r1 - r2) < 1.0e-06) {
    /* this was a goStraight command */

    distance = rsClient->speed * deltaT / 1000.0;

    new_x = rsClient->xCurrent + cos(rsClient->orientation) * distance;
    new_y = rsClient->yCurrent + sin(rsClient->orientation) * distance;

#ifdef RUN_STATE_DEBUG
    fprintf(stderr, "RobPos after goStraightCmd: r1=%f, r2=%f, deltaT=%d, durMillisStarted=%d, d
istance=%f, prev_x=%f, prev_y=%f, x=%f, y=%f, simTime=%d\n",
            r1, r2, deltaT, rsClient->durationMillisStarted, distance,
            rsClient->prev_x, rsClient->prev_y, new_x, new_y, crtTime);
#endif

  } else {
    /* this was a rotation/turn command */

    /* rotation angle since last halt */
    gamma = (r2 - r1) * WHEEL_RADIUS / WHEEL_BASE;
    /* previous orientation */
    beta = rsClient->orientation;
    /* current orientation */
    alpha = beta + gamma;

    distance = ((r1 + r2)/(r1 - r2))*(WHEEL_BASE/2.0);
    new_x = rsClient->xCurrent + distance * (sin(beta) - sin(alpha));
    new_y = rsClient->yCurrent + distance * (cos(alpha) - cos(beta));
#ifdef RUN_STATE_DEBUG
    fprintf(stderr, "RobPos after rotateCmd: r1=%f, r2=%f, deltaT=%d, durMillisStarted=%d, gamma
=%f, beta=%f, alpha=%f, distance=%f, prev_x=%f, prev_y=%f, x=%f, y=%f, simTime=%d\n",
            r1, r2, deltaT, rsClient->durationMillisStarted, gamma, beta,
            alpha, distance, rsClient->prev_x, rsClient->prev_y, new_x, new_y, crtTime);
#endif


  }

  rsClient->orientation_in_runstate = alpha;
  rsClient->xCurrent_in_runstate = new_x;
  rsClient->yCurrent_in_runstate = new_y;

  return OK;
}

int
updateRobotPosition(RsClient *rsClient) {
  rsClient->prev_orientation = rsClient->orientation;
  rsClient->prev_x = rsClient->xCurrent;
  rsClient->prev_y = rsClient->yCurrent;

  rsClient->orientation = rsClient->orientation_in_runstate;
  rsClient->xCurrent = rsClient->xCurrent_in_runstate;
  rsClient->yCurrent = rsClient->yCurrent_in_runstate;

  /* For each command sent to the Server app., there is a difference btw
   * the movement interval (usually a straightCmd) considered by the Server
   * and the one considered by the Client, due to the time required by
   * the client-server communication (thanks, Windows(tm) !);
   * this time difference (milliseconds) leads
   * to difference of centimeters in the position considered by the Server
   * and by the Client. Overall, this difference builds up and causes
   * serious errors; so for the time being, we will sync with the Server
   * by halting the robot and read server's coordinates
   */
```

```
  return 0;
}

/*****************************************************************************
;
;   calculateRotationAngle - calculate rotation angle towards the target
;
;   Given the robot coordinates and target coordinates, this function
;   calculates the required rotation angle for the robot to point towards
;   the target
;
;   RETURNS
;     angle = rotation angle
;
*****************************************************************************/
double
calculateRotationAngle(
                       double alpha,     /* orientation of the robot */
                       double xInit,     /* x robot */
                       double yInit,     /* y robot */
                       double xFin,      /* x target */
                       double yFin       /* y target */
                      ) {

  long   n;
  double beta;
  double gamma;
  double deltaX;
  double deltaY;
  double rotAngle;

  /* adjust alpha ( the orientation in [0, 2*M_PI]) */
  n = (int)(alpha / (2.0 * M_PI));
  beta = alpha - (n * (2.0 * M_PI));

  /* if negative, bring it to a positive value */
  if (beta < 0.0) {
    beta += 2.0 * M_PI;
  }

  deltaX = xFin - xInit;
  deltaY = yFin - yInit;

  if (fabs(deltaX) < 1.0e-06) {
    if(deltaY > 0.0) {
      rotAngle = 0.0;
    } else {
      rotAngle = M_PI;
    }
  } else {
    gamma = atan(deltaY / deltaX);
    if (deltaX < 0.0) {  /* for second and third cadran, adjust with M_PI */
      gamma += M_PI;
    }
    if (gamma < 0.0) {   /* for fourth cadran, bring it to a positive value */
      gamma += 2.0 * M_PI;
    }
    /* at this point, gamma is in [0, 2*M_PI] */

    rotAngle = gamma - beta;

    /* take the shortest path to (xFin, yFin) */
    if (rotAngle > M_PI) {
      rotAngle -= 2.0 * M_PI;
    } else if (rotAngle < -(M_PI)) {
      rotAngle += 2.0 * M_PI;
    }
  }

  return rotAngle;
}
```

```
/*****************************************************************************
;
;   updateTotalRotationAngle - update the total rotation angle of the robot
;                              with the given value
;
*****************************************************************************/
void
updateTotalRotationAngle(
            RsClient *rsClient,    /* pointer to the client structure */
            double    angle        /* value to update total rot. angle with */
                      ) {
  double val;
  int flag = FALSE;

  val = rsClient->totalRotationAngle + angle;

  /* bring the value in (-2*M_PI, 2*M_PI) */
  if(val > 0.0) {
    while (val >= 2.0*M_PI) {
      val -= 2.0*M_PI;
      flag = TRUE;
    }
  } else {
    while (val <= -2.0*M_PI) {
      val += 2.0*M_PI;
      flag = TRUE;
    }
  }

  rsClient->totalRotationAngle = val;
}

/*****************************************************************************
;
;   correctTotalRotAngleIfRotNotCompleted - when rotation is not completed,
;       just do a correction of the total rotation angle
;
*****************************************************************************/
void
correctTotalRotAngleIfRotNotCompleted(
            RsClient *rsClient,    /* pointer to the client structure */
            long crtTime           /* current time (in usec) */
                                   ) {

  double r1, r2;
  double gamma;
  long deltaT;

  r1 = rsClient->rLeftStarted;
  r2 = rsClient->rRightStarted;
  deltaT = crtTime - rsClient->timeMotionStarted;

  /* rotation angle since last halt if rotation had been completed */
  gamma = (r2 - r1) * WHEEL_RADIUS / WHEEL_BASE;

  /* actual rotation missed */
  gamma = -gamma * ((double)(rsClient->durationMillisStarted - deltaT)) /
          (double)(rsClient->durationMillisStarted);

  rsClient->totalRotationAngle += gamma;

}

/*****************************************************************************
;
;   turnCmd - Execute a turn command
;
;   This function passes to the Server the command for turning the robot
;   around a fixed point situated outside the robot's body.
;   Usually this is a rotation around a fixed wheel.
;
```

```
;   RETURN
;     status = OK - if command has been succesfully sent to the Server
;              ERROR otherwise
;
*****************************************************************************/
int
turnCmd(
        RsClient *rsClient,     /* pointer to the client structure */
        double angle,           /* turn angle */
        int leftWheel,          /* TRUE if turn around left wheel,
                                /*      FALSE otherwise */
        double distance,        /* distance from the inner wheel to the fixed
                                /*    point (the center of rotation) */
        long crtTime            /* current time (in usec) */
       ) {
  RsMotionRequest r;
  double origAngle = angle;


#ifdef TURNCMD_DEBUG
  printf("turnCmd: time=%d, angle=%f, wheel=%d, x=%f, y=%f, orientation=%f, xTarget=%f, yTarget=
%f\n", crtTime, angle, leftWheel, rsClient->xCurrent, rsClient->yCurrent, rsClient->orientation,
 rsClient->xTarget, rsClient->yTarget);
#endif

  if (fabs(angle) < MAX_ANGLE_ERR_ALLOWED) {
    printf("turnCmd ERROR !!!  angle=%f\n",angle);
    return ERROR;
  }
  if (leftWheel == TRUE) {
    r.rLeft = (-angle) * (WHEEL_BASE + distance) / WHEEL_RADIUS;
    r.rRight = (-angle) * (distance) / WHEEL_RADIUS;
  } else {
    r.rRight = angle * (WHEEL_BASE + distance) / WHEEL_RADIUS;
    r.rLeft = angle * (distance) / WHEEL_RADIUS;
  }

  r.useStepMethod = 0;
  r.nStepLeft     = 0;
  r.nStepRight    = 0;

  if (angle < 0.0) {
    angle = -angle;
  }
  r.durationMillis = (long)((angle * (WHEEL_BASE) / r_speed) * 1000);

  RsSendMotionRequest(rsClient, &r, crtTime);
  rsClient->navState = NAV_STATE_TURN;

  updateTotalRotationAngle(rsClient, origAngle);

  return OK;
}

/*****************************************************************************
;
;   turnCmd - Execute a rotate command
;
;   This function passes to the Server the command for rotating the robot
;   around his central axis.
;
;   RETURN
;     status = OK - if command has been succesfully sent to the Server
;              ERROR otherwise
;
*****************************************************************************/
int
rotateCmd(
          RsClient *rsClient,     /* pointer to the client structure */
          double angle,           /* rotation angle */
          long crtTime            /* current time (in usec) */
         ) {
```

```
  RsMotionRequest r;
  double          origAngle = angle;

#ifdef ROTATECMD_DEBUG
  printf("rotateCmd: time=%d, angle=%f, navState=%d\n", crtTime, angle, rsClient->navState);
#endif

  if (angle > -1.0e-6 && angle < 1.0e-6) {
    printf("ANGLE ALMOST ZERO !\n");
    return ERROR;
  }
  r.rLeft = (-angle) * (WHEEL_BASE/2) / WHEEL_RADIUS;
  r.rRight = angle * (WHEEL_BASE/2) / WHEEL_RADIUS;

  r.useStepMethod = 0;
  r.nStepLeft     = 0;
  r.nStepRight    = 0;

  if (angle < 0.0) {
    angle = -angle;
  }
  r.durationMillis = (long)((angle * (WHEEL_BASE/2) / r_speed) * 1000);

  RsSendMotionRequest(rsClient, &r, crtTime);

  updateTotalRotationAngle(rsClient, origAngle);

  return OK;

}

/*****************************************************************************
;
;   startRobot - Start the search of the target by orienting him towards
;   the target; for this purpose, a rotate cmd is issued
;
*****************************************************************************/
void
startRobot(
            RsClient *rsClient,     /* pointer to the client structure */
            double x,               /* x target */
            double y,               /* y target */
            long crtTime            /* current time (in usec) */
          ) {
  double angle;
  double distance, dx, dy;

  rsClient->targetFound = FALSE;

  angle = calculateRotationAngle(rsClient->orientation, rsClient->xCurrent,
             rsClient->yCurrent, x, y);

#ifdef STARTROBOT_DEBUG
  printf("startRobot   angle=%f, or=%f, xCrt=%f, yCrt=%f, x=%f, y=%f\n", angle, rsClient->orient
ation, rsClient->xCurrent, rsClient->yCurrent, x, y);
#endif

  if (fabs(angle) < MAX_ANGLE_ERR_ALLOWED || rotateAtStartup == FALSE) {
    dx = rsClient->xCurrent - x;
    dy = rsClient->yCurrent - y;
    distance  = sqrt(dx*dx + dy*dy);
    goStraightCmd(rsClient, r_speed,  (distance/r_speed)*1000.0, crtTime);
    rsClient->disableSensor = FALSE;
    rsClient->block_controlcycle = 2;
    rsClient->blockingTime = 0;
  } else {
    rotateCmd(rsClient, angle, crtTime);
    rsClient->disableSensor = TRUE;
    rsClient->totalRotationAngle = 0.0;
  }
}
```

```
/*****************************************************************************
;
;   goStraightCmd - Execute a go straight command
;
*****************************************************************************/
void
goStraightCmd(RsClient *rsClient, double speed, long timeMillis, long crtTime) {
  RsMotionRequest r;

#ifdef GOSTRAIGHT_DEBUG
  printf("goStraightCmd: time=%d, speed=%f, tMillis=%d\n", crtTime, speed, timeMillis);
#endif

  if (fabs(speed) < 1.0e-06) {
    speed = r_speed;
  }

  if (timeMillis == 0) {
    timeMillis = MAX_TIME_MILLIS;
  }

  r.rLeft = r.rRight = speed * ((double)timeMillis / 1000.0) / WHEEL_RADIUS;

  r.useStepMethod = 0;
  r.nStepLeft     = 0;
  r.nStepRight    = 0;

  r.durationMillis = timeMillis;

  RsSendMotionRequest(rsClient, &r, crtTime);
  rsClient->speed = speed;
  rsClient->navState = NAV_STATE_RUN;
}

/*****************************************************************************
;
;   mainHelp - help function for the robot program
;
*****************************************************************************/
void
mainHelp() {
    printf("Usage:\n");
    printf("       robot -s speed<CR>\n");
    printf("         Robot goes to the positioned target. Options:\n");
    printf("          -r rotation: 0 - turn; 1 - rotate. Default: 0\n");
    printf("           (turn around a fixed wheel; rotate around its center)\n");
    printf("          -s speed : robot speed (in m/s). Default: 0.4\n");
    printf("             for test purpose, speed=10.0 will inhibit\n");
    printf("             robot rotation at start-up\n");
    printf("\n");
    printf("       Mouse button commands:\n");
    printf("            left button:       start/stop the Robot\n");
    printf("            right button:      rotate the Robot\n");
    printf("            CTRL-left button:  re-position the Robot\n");
    printf("            ALT-left button:   re-position the Target\n\n");

}

/*****************************************************************************
;
;
;   main function of the robot program
;
;
*****************************************************************************/
int main(int argc, char *argv[]){

  RsProperties *rsProperties;
  RsClient     *rsClient;
  int           status;
  int           ind;
  UserData     *ud = calloc(1, sizeof(UserData));
```

```
FILE        *fp;
int          ind_speed;
int          ind_rot;


ind_speed = -1;
ind_rot = -1;

if(argc == 1) {
  printf("       Mouse button commands:\n");
  printf("            left button:       start/stop the Robot\n");
  printf("            right button:      rotate the Robot\n");
  printf("            CTRL-left button:  re-position the Robot\n");
  printf("            ALT-left button:   re-position the Target\n\n");
  ind_speed = 0;
  ind_rot = 0;
} else if (argc == 3) {
  if (strcmp(argv[1], "-s") == 0) {
    ind_speed = 2;
    ind_rot = 0;
  } else if (strcmp(argv[1], "-r") == 0) {
    ind_speed = 0;
    ind_rot = 2;
  }
} else if (argc == 5) {
  if (strcmp(argv[1], "-s") == 0 && strcmp(argv[3], "-r") == 0) {
    ind_speed = 2;
    ind_rot = 4;
  } else if (strcmp(argv[3], "-s") == 0 && strcmp(argv[1], "-r") == 0) {
    ind_speed = 4;
    ind_rot = 2;
  }
}

if (ind_speed > 0) {
  r_speed = atof(argv[ind_speed]);
  if (r_speed <= 0.0) {
    ind_speed = -1;
  } else if (r_speed == 10.0) {
    /* for testing purposes, avoid rotation at startup if required */
    r_speed = R_SPEED;
    rotateAtStartup = FALSE;
  } else if (r_speed == 20.0) {
    /* regression testing: allow the user the select n points and
     * force the robot to go automatically btw all points
     */
    r_speed = R_SPEED;
    regressionTest = REG_COLLECT_DATA;
  } else if (r_speed == 21.0) {
    /* regression testing: pick-up the n points for paths definition
     * from a previously regression test, so that the test is
     * identically reproduced
     */
    r_speed = R_SPEED;
    regressionTest = REG_COLLECT_DATA;
    initRegFile = 1;
  } else if (r_speed > 1000.0) {
    /* regression testing: pick-up the n points for paths definition
     * from a previously regression test, so that the test is
     * identically reproduced and start with a given path id
     */
    initRegFile = (int)r_speed - 1000;
    r_speed = R_SPEED;
    regressionTest = REG_COLLECT_DATA;
  }
}

if(ind_rot > 0) {
    if(strlen(argv[ind_rot]) == 1) {
      if(*(argv[ind_rot]) == '0') {
        useRotate = FALSE;
      } else if (*(argv[ind_rot]) == '1') {
```

```
          useRotate = TRUE;
        } else {
          ind_rot = -1;
        }
      } else {
        ind_rot = -1;
      }
  }

  if (ind_speed < 0 || ind_rot < 0) {
    mainHelp();
    exit(0);
  }


  RsInitDataStream();

  rsProperties = RsLoadProperties(NULL);

  rsClient = RsCreateClient(rsProperties);
  if(!rsClient)
     exit(-1);
  ud->rsClient = rsClient;

  /* other initializations */

  /* initialize the client side of the robot */
  RsClientInit(rsClient);

  /* initialize the FIS for 'veryclose' primitive */
  fisInit(&fis2, fisMatrix2, fis_file2, MF_POINT_N);

  /* initialize the FIS for 'gotangent' primitive */
  fisInit(&fis3, fisMatrix3, fis_file3, MF_POINT_N);

  /* initialize the FIS for 'wallfollow' primitive */
  fisInit(&fis4, fisMatrix4, fis_file4, MF_POINT_N);

  /* initialize the FIS for 'gotoxy' primitive */
  fisInit(&fis5, fisMatrix5, fis_file5, MF_POINT_N);

  /* initialize the FIS for 'turnCorner' primitive */
  fisInit(&fis6, fisMatrix6, fis_file6, MF_POINT_N);

  /* initialize the FIS for 'gotangent - frontsensor' primitive */
  fisInit(&fis7, fisMatrix7, fis_file7, MF_POINT_N);

  RsSendBody(rsClient, "body.dat");
  RsSendPlacementRequest(rsClient, "home", 0.0, 0.0);

  RsAddPlacementEventHandler(   rsClient, placementEventHandler,    ud);
  RsAddMotionHaltedEventHandler(rsClient, motionHaltedEventHandler, ud);
  for (ind = 0; ind < MAX_RANGESENSORS; ind++) {
    RsAddRangeSensorEventHandler(rsClient, rangeSensorEventHandler, ud,
       rsClient->rangeSensor[ind].sensorId);
  }
  for (ind = 0; ind < MAX_CONTACTSENSORS; ind++) {
    RsAddContactSensorEventHandler(rsClient, contactSensorEventHandler, ud,
       rsClient->contactSensor[ind].sensorId);
  }
  RsAddTimeoutEventHandler(rsClient, timeoutEventHandler, ud);
  RsAddMouseClickEventHandler(rsClient, mouseClickEventHandler, ud);
  status = RsClientMainLoop(rsClient);
  exit(0);
}
```

Bibliography

[1] ***, *GP2D12 and GP2D15 Technical Data Sheet*, Sharp Corporation, 2001.

[2] D.S. Blank and J.O. Ross, *Learning in a fuzzy logic robot controller*, Proceedings of 14th National Conference on Artificial Intelligence, pages 778-779, Rhode Island, 1997.

[3] A. Bonarini, *Learning dynamic behaviors from easy missions*, Proceedings of Information Process and Management of Uncertainty, pages 1223-1228, Granada, Spain, 1996.

[4] B.S. Butkiewicz, *Steady-state error of a system with fuzzy controller*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.28, no. 6, pages 855-860, 1998.

[5] H. Chung, L. Ojeda, and J. Borenstein, *Accurate mobile robot dead-reckoning with a precision-calibrated fiber-optic gyroscope*, IEEE Transactions on Robotics and Automation, vol 17, pages 80-84, 2001.

[6] F. Cuesta, F. Gordillo, J. Aracil, and A. Ollero, *Stability analysis of nonlinear multivariable Takagi-Sugeno Fuzzy Control Systems*, IEEE Transactions on Fuzzy Systems, Vol.7, no. 5, pages 508-520, 1999.

[7] H. Danny, T. Lippincott, M. Jamshidi, and E. Tunstel, *Autonomous navigation using an adaptive hierarchy of multiple fuzzy behaviors*, Proceedings of Intl. Symposium on Computational Intelligence in Robotics and Automation, pages 276-281, Monterey CA, 1997.

[8] M. Figueiredo and F. Gomide, *Design of fuzzy systems using neurofuzzy networks*, IEEE Transactions on Neural Networks, Vol.10, no. 4, pages 815-827, 1999.

[9] C.W. Frey and H.B. Kuntze, *A neuro-fuzzy supervisory control system for industrial batch processes*, IEEE Transactions on Fuzzy Systems, Vol.9, no. 4, pages 570-577, 2001.

[10] J Gasós and A. Martín, *Mobile robot localization using fuzzy maps*, Springer Verlag - Lecture Notes in Artificial Intelligence, A Ralescu & T. Martin (Eds.), 1996.

[11] Jelena Godjevac, *Comparative study of fuzzy control, neural-network control and neuro-fuzzy control*, École Polytechnique fédérale de Lausanne, Technical Report no. 103, 1995.

[12] B. Hu, G.K.I. Mann, and R.G. Gosine, *New methodology for analytical and optimal design of fuzzy pid controllers*, IEEE Transactions on Fuzzy Systems, Vol.7, no. 5, pages 521-539, 1999.

[13] R. Jager, *Fuzzy logic in control*, Technische Universiteit Delft, Ph.D Thesis, 1995.

[14] J.-S.R. Jang, *ANFIS: Adaptive-Network-based Fuzzy Inference Systems*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.23, No. 3, pages 665-685, 1993.

[15] J.-S.R. Jang and C.-T. Sun, *Neuro-fuzzy and soft computing: A computational approach to learning and machine intelligence*, Prentice Hall, 1997.

[16] R. Kovacevic and Y.M. Zhang, *On-line measurement of weld fusion state using weld pool image and neurofuzzy model*, Proc. of the IEEE International Symposium on Intelligent Control, Dearborn, MI pages 307-312, 1996.

[17] J.-C. Lo and Y.-M. Chen, *Stability issues on Takagi-Sugeno fuzzy model - parametric approach*, IEEE Transactions on Fuzzy Systems, Vol.7, no. 5, pages 597-607, 1999.

[18] George W. Lucas, *Rossum's Playhouse RP1 - User's Documentation Guide 0.46*, http://rossum.sourceforge.net, 2000.

[19] E. H. Mamdani and S. Assilian, *An experiment in linguistic synthesis with a fuzzy logic controller*, International Journal of Man-Machine Studies, Vol. 7, No. 1, pages 1-13, 1975.

[20] E.H. Mamdani, *Twenty years of fuzzy control: experiences gained and lessons learnt*, IEEE Transactions on Fuzzy Systems, pages 339-344, 1993.

[21] J. Mao, *Reduction of the quantization error in fuzzy logic controllers by dithering*, University of Ottawa, M.A.Sc. Thesis, Ottawa, Ontario, 1998.

[22] M. J. Matarić, *Behavior-based control: Examples from navigation, learning and group behavior*, Journal of Experimental and Theoretical Artificial Intelligence, Special Issue: Software Architecture for Physical Agents, 9, pages 46-54, 1997.

[23] Q. Meng, D. Liu, and M. Zhang, *Wall-following by an Autonomously Guided Vehicle (AGV) using a new fuzzy-Integration controller*, Robotica, vol 17, pages 79-86, 1999.

[24] K.C. Ng and M.M. Trivedi, *A neuro-fuzzy controller for mobile robot navigation and multirobot convoying*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.28, no. 6, pages 829-840, 1998.

[25] C.F. Olson, *Probabilistic self-localization for mobile robots*, IEEE Transactions on Robotics and Automation, Vol.16, no. 1, pages 55-66, 2000.

[26] E. Petriu, A. Cornell, and G. Earthley, *Fuzzy control for mobile robot backing-up*, Proc. Third Int. Conf. Technical Informatics, CONTI'98, Vol.1, pages 211-217, Timisoara, Romania, 1998.

[27] E. Petriu and J. Mao, *Fuzzy sensing and control for a truck*, Proc. VIMS-2000, IEEE Workshop on Virtual and Intelligent Measurement Systems, pages 27-32, Annapolis, MD, 2000.

[28] E. Petriu, K.M Moulton, and A. Cornell, *A fuzzy error correction system*, Proc. IMTC/98, IEEE Instrum. Meas. Technol. Conf., pages 733-738, Venice, Italy, 1999.

[29] E. Petriu, P. Wide, and G. Eatherley, *Sensor resolution effect on fuzzy logic controller behavior*, Proc. IMTC/1998, IEEE Instrumentation and Measurement Technology Conference, pages 848-852, St. Paul, MN, 1998.

[30] I.M. Rekleitis, G. Dudek, and E. Milios, *Multi-robot exploration of an unknown environment, efficiently reducing the odometry error*, Proceedings of 15th International Joint Conference on Artificial Intelligence, pages 1340-1352, Japan, 1997.

[31] N. Roy and R. Sim, *Autonomous exploration: An integrated system approach*, Proc. of 14th National Conference on Artificial Intelligence, pages 780-781, Rhode Island, 1997.

[32] E.H. Ruspini, *Fuzzy logic-based planning and reactive control of autonomous mobile robots*, IEEE Intl. Conference on Fuzzy Systems, pages 1071-1076, Japan, 1995.

[33] A. Saffiotti, *The uses of fuzzy logic in autonomous robot navigation: a catalogue raisonné*, Soft Computing 1(4) pages 180-197, Springer-Verlag, 1997.

[34] A. Saffiotti, E.H. Ruspini, and K. Konolige, *Blending reactivity and goal-directedness in a fuzzy controller*, Proc. 3rd IEEE Intl. Conference on Fuzzy Systems, pages 134-139, Orlando, FL, 1994.

[35] A. Saffiotti and L.P. Wesley, *Perception-based self localization using fuzzy locations (Reasoning with uncertainty in Robotics), pages 368-385*, Springer, Berlin, 1996.

[36] Z. Sakr, E. Petriu, and P. Wide, *Fuzzy controller for a hexaped robot*, Proc. VIMS'99, IEEE Workshop on Virtual and Intelligent Measurement Systems, pages 42-46, Venice, Italy, 1999.

[37] T. L. Skillman and K. Hopping, *Dynamic composition and execution of behaviors in a hierarchical control system*, Mobile robots IV, pages 349-360, 1989.

[38] E. Tunstel, *Behavior hierarchy for autonomous mobile robots: Fuzzy-behavior modulation and evolution*, Intl. Journal of Intelligent Automation and Soft Computing, Special Issue: Autonomous Control Engineering, Vol. 3, No.1, pages 37-50, 1997.

[39] H. Yamamoto and T. Furuhashi, *A new sufficient condition for stable fuzzy control system and its design method*, IEEE Transactions on Fuzzy Systems, Vol.9, no. 4, pages 554-569, 2001.

[40] J. Yen and N. Pfluger, *A fuzzy logic based extension to Payton and Rosenblatt's command fusion method for mobile robot navigation*, IEEE Transactions on Systems, Man, and Cybernetics Vol 25, No. 6, pages 971-978, 1995.

[41] L. A. Zadeh, *Fuzzy sets*, Information and Control, Vol. 8, pages 338-353, 1965.