

Faculdade de Engenharia da Universidade do Porto



**Neuro-Fuzzy Software for Intelligent Control
and Education**

Erik Joaquin Moreira Pegoraro

Dissertation submitted to obtain the degree of
Master of Science in Eletrotechnical and Computer Engineering
Major in Automation

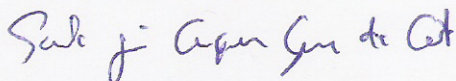
Supervisor: Prof. Armando Jorge Sousa, PhD

June 2009

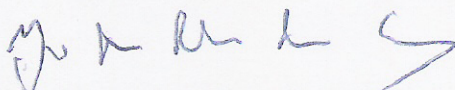
A Dissertação intitulada
“NEURO-FUZZY SOFTWARE FOR INTELLIGENT CONTROL AND EDUCATION”

foi aprovada em provas realizadas em 17/Julho/2009

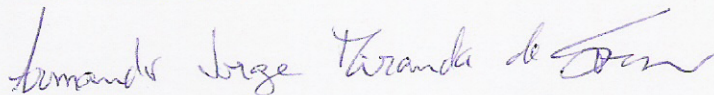
o júri



Presidente Professor Doutor Paulo José Cerqueira Gomes da Costa
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da
Faculdade de Engenharia da Universidade do Porto

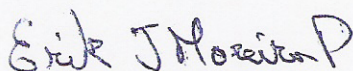


Professor Doutor José Nuno Panelas Nunes Lau
Professor Auxiliar do Departamento de Engenharia Electrónica, Telecomunicações e
Informática da Universidade de Aveiro



Professor Doutor Armando Jorge Miranda de Sousa
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da
Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da
sua exclusiva autoria e foi escrita sem qualquer apoio externo não
explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros
extractos tomados de ou inspirados em trabalhos de outros autores, e
demais referências bibliográficas usadas, são correctamente citados.



Autor - ERIK JOAQUIN MOREIRA PEGORARO

To Ilan and my dear Ana Cristina

Resumo

A teoria dos conjuntos difusos originou-se no ano de 1965 após a publicação da obra de Lotfi Zadeh. Vinte anos depois a sua aplicação noutras áreas, especialmente na engenharia do controlo tem vindo a sofrer um marcado crescimento. No entanto uma parte da comunidade mantém o cepticismo do potencial da aplicação da lógica difusa na resolução de problemas de engenharia.

Alguns autores ressaltam que isto deve-se fundamentalmente à ausência de software livre. O presente trabalho pretende contrariar esta tendência e promover as disciplinas da lógica difusa e do soft computing, bem como dotar a comunidade educativa de uma poderosa ferramenta de auxílio para o ensino dos conceitos da lógica difusa.

O FEUP Fuzzy tool é uma ferramenta educativa originalmente criada num ambiente de programação baseado no Object Pascal, o Delphi 6 Personal Edition, sendo posteriormente portada para o ambiente de desenvolvimento Lazarus que tem como vantagens principais uma fácil utilização, programação visual e orientação a eventos. A livreria foi construída inicialmente para permitir o controlo de processos não lineares com os recursos computacionais disponíveis.

A implementação e aplicação de conceitos avançados como as arquitecturas neuro-difusas são o objectivo principal desta dissertação. O documento apresenta uma descrição detalhada do marco teórico e da implementação destes conceitos, bem como uma explicação dos ambientes de simulação desenvolvidos para validação e teste.

O suporte para a concepção de controladores difusos para sistemas embebidos também é de particular importância e será também o objecto de discussão, junto com a descrição da importação da livreria do FEUP Fuzzy Tool para sistemas baseados em microcontroladores.

Abstract

The theory of fuzzy sets traces its origins back to 1965 after its introduction by Lotfi A. Zadeh. Twenty years later suffers its greatest breakthrough in the control community and after the years its application has been continuously increasing in many other areas. However, there is still some part of the community that remains doubtful about fuzzy logic potential. Some author point out that this is due the lack of availability of free tool. The present work intends to counteract this trend and also to promote a higher level of education by enlarging the functionality of an already existing educational toolkit, the FEUP Fuzzy Tool and deliver a state-of-the-art software.

The FEUP Fuzzy Tool was created in the Faculty of Engineering of the University of Oporto, Portugal in order to easily allow the control of nonlinear plants with an standard computer. It was programmed under an Object Pascal based platform known as Delphi 6 Personal Edition, and it was ported later to Lazarus IDE which enjoys properties like ease of use, visual, event driven and fast.

The implementation of advanced concepts from the fields of neuro-fuzzy and soft computing is of primary concern. The document presents a detailed description about theoretical framework and the implementation of these concepts and provides also an explanation of the simulation environments that were used as validation tests.

Also, the importance of the realization of fuzzy controllers for embedded systems is highlighted and discussed through the extension of the library to microcontroller-based environments.

Acknowledgements

First of all, I want to show my gratitude to God for blessing the path which made me reach this moment, a faith that cleared every obstacle that ever appears and for the strength passed on when everything else seems hopeless.

To my parents and family for the educations they imparted, for raising me the man I am and for all the valuable lesson that taught me tenacity and persistency among other things.

To my beloved Ana for all our sacrifices, her patience, understanding and invaluable support throughout this venture. Without her everything would have been in vain.

To my little brother, in whose eyes I found a reflection of perfection that makes me want to break the human barrier that keeps me from that image.

A special thanks to my supervisor, Prof. Armando Sousa for the shared wisdom and guidance offered through this endeavor, with whom I found inspiration and deep admiration.

And finally, but not least, to my forever friend and colleague Ana Mora whose friendship and unconditional support is invaluable.

Index

Resumo	iv
Abstract	vi
Acknowledgements	viii
Index	x
Figure List	xiv
Table List	xvii
Acronyms and Symbols	xix
Chapter 1	1
Introduction	1
1.1 - Motivation	1
1.1.1 - From an industrial point of view	1
1.1.2 - From an educational point of view	3
1.2 - The Feup Fuzzy Tool: Starting point	4
1.3 - Contributions	5
1.4 - Document Structure	6
1.5 - Summary	6
Chapter 2	7
Fuzzy Logic Theory	7
2.1 - Classical sets	7
2.1.1 - Classical set operations	8
2.1.2 - Properties	8
2.2 - Fuzzy Sets	9
2.2.1 - Basic definitions	10
2.2.2 - Operations and properties	12
2.3 - Relations	13
2.3.1 - Classical relations	13
2.3.2 - Fuzzy relations	15
2.4 - Approximate Reasoning	15
2.4.1 - Linguistic variables	15
2.4.2 - Linguistic hedges	16
2.4.3 - The if-then statements	17
2.4.4 - Rule-based system	18
2.5 - Summary	19

Chapter 3	21
State of the Art - Market Analysis.....	21
3.1 - Programming libraries.....	21
3.2 - Application Tools	22
3.3 - Comparative Requirement Analysis	24
3.3.1 - Core Fuzzy Engine	24
3.3.2 - Learning.....	24
3.3.3 - Portability	24
3.4 - Summary	25
Chapter 4	27
Fuzzy Control and FEUP Fuzzy Tool.....	27
4.1 - Fuzzy Controller Structure	27
4.1.1 - Fuzzification module.....	27
4.1.2 - Knowledge base.....	27
4.1.3 - Inference Engine and Defuzzification module	28
4.2 - The Feup Fuzzy Tool	29
4.2.1 - Why FPC/Lazarus as a programming tool	29
4.2.2 - The Fuzzy core engine.....	29
4.2.3 - Fuzzy sets.....	30
4.2.4 - Fuzzy variable.....	31
4.2.5 - Fuzzification.....	32
4.2.6 - Fuzzy Rules	32
4.2.7 - The TFFSystem and the inference engine	33
4.2.8 - Defuzzification methods.....	35
4.3 - A new inference model	35
4.4 - The Graphical User Interface (GUI)	38
4.4.1 - Rule Scope	38
4.4.2 - Rule Editor.....	39
4.4.3 - The control surface	40
4.4.4 - Fuzzy Associative Memory (FAM).....	41
4.5 - Summary	43
Chapter 5	45
Producing Embedded Controllers	45
5.1 - Fuzzy logic hardware.....	45
5.2 - Microcontroller-based implementation.....	46
5.2.1 - Aitken-Neville interpolation formula and the dual interpolation.....	48
5.2.2 - Fuzzy controller coding.....	49
5.3 - The C code Editor.....	53
5.4 - Validation Test	57
5.5 - Summary	59
Chapter 6	61
Self-Learning fuzzy architectures	61
6.1 - The gradient steepest descent method	62
6.2 - Artificial neural networks	63
6.3 - The Perceptron neuron model	65
6.4 - Backpropagation Multilayer Perceptron.....	67
6.5 - Multilayer perceptron data structure.....	68
6.5.1 - Requirement analysis	69
6.5.2 - Model implementation.....	71
6.6 - Model validation: the XOR problem.....	77
6.7 - Neuro-Fuzzy architectures	78
6.7.1 - Introduction	78
6.7.2 - Adaptive Neural-Based Fuzzy Inference System - ANFIS	80

6.7.3 - Hybrid learning rule: combining the steepest descent method and the Least-Squares estimation.....	82
6.7.4 - ANFIS data structure.....	84
6.7.5 - The ANFIS graphical user interface.....	87
6.7.6 - Model Validation: Approximating the two-input sinc function.....	88
6.8 - Summary	90
Chapter 7	91
Results - Simulated Problems.....	91
7.1 - Fuzzy control of an autonomous mobile robot.....	92
7.2 - Neuro-Fuzzy control of a nonlinear plant.....	95
7.2.1 - Inverse Learning	95
7.3 - Summary	99
Chapter 8	101
Conclusion and Future work	101
8.1 - Future Trends	103
References.....	105

Figure List

Figure 1.1 Structural view of the initial version of the FEUP Fuzzy Tool Software.	4
Figure 2.1 Plots of various membership functions: Trapezoidal (upper left), Triangular (upper right), bell-shaped (lower left) and Gaussian (lower right).	10
Figure 2.2 An arbitrary fuzzy set where the concepts of core, boundary and support are shown.	11
Figure 2.3 Exemplification of why the excluded middle axiom and the law of contradiction fails for fuzzy sets. Adapted from [10].	13
Figure 2.4 A Sagittal diagram of a classical relation.....	14
Figure 4.1 Fuzzy Controller Structure.....	28
Figure 4.2 UML Class Diagram of the FEUP Fuzzy Tool	30
Figure 4.3 Layered composition of a MIMO fuzzy controller. Figure adapted from [31]	33
Figure 4.4 Graphical Inference of a two rule fuzzy system of the Sugeno model. Adapted from [10].	36
Figure 4.5 The GUI Main Form.....	38
Figure 4.6 A picture of the Rule Scope form.....	39
Figure 4.7 The Rule Editor form	40
Figure 4.8 The Control surface 3D plotter. It shows the transfer function of a Sugeno controller.....	41
Figure 4.9 Table representation of the rule base. This representation is also called fuzzy associative memory.....	42
Figure 5.1 Illustration of the sampling of a control surface.	47
Figure 5.2 A simplified one-dimensional representation of a possible nonlinear control law.	51
Figure 5.3 Snapshot of the C code Editor.....	53
Figure 5.4 Use-Case UML diagram. It depicts the behavior of the system for the C code generation functionality.	54

Figure 5.5 User Interface for parameter selection to automate the process of programming code generation.	55
Figure 5.6 Control Surfaces created through fuzzy inference (left) and by samples and dual interpolation (right).	58
Figure 6.1 Illustration of the concepts of the gradient steepest descent method.	62
Figure 6.2 The feedforward and recurrent ANN architectures.	63
Figure 6.3 Signal backpropagation through a feedforward ANN.....	65
Figure 6.4 The perceptron model.	65
Figure 6.5 Graphical interpretation about the classification process of two classes in a plane	66
Figure 6.6 Structural view of the layer model. Adapted from [44].	71
Figure 6.7 UML class diagram of the neuro-fuzzy structures inside the FEUP Fuzzy Tool.....	72
Figure 6.8 Training error evolution for the MLP to approximate the XOR function.	78
Figure 6.9 The ANFIS structure for a dual-input-single-output Sugeno Model.	80
Figure 6.10 An ideal fuzzy set represented by a piecewise linear function defined by parameters p and q among others. For that particular x_1 input the membership function can be seen as a line segment.	86
Figure 6.11 The ANFIS GUI.	87
Figure 6.12 Initial and post-training distribution of fuzzy sets for the sinc function approximation	88
Figure 6.13 Training log for the sinc function approximation	89
Figure 6.14 3D plot of the sinc function approximated by the ANFIS in various training epochs	89
Figure 7.1 The wall following problem description and input denotations.	93
Figure 7.2 Fuzzy sets configuration for the autonomous robot fuzzy controller	94
Figure 7.3 Simulation environment for the autonomous robot control.	95
Figure 7.4 Training log (a), pre- and post-training configuration of fuzzy sets (b) for the inverse learning problem.	97
Figure 7.5 Transfer characteristic of the neuro-fuzzy controller.	98
Figure 7.6 Simulation environment nonlinear System + ANFIS controller with a $\cos(t)$ reference.	98
Figure 7.7 Zoom of the plot in Figure 7.6 to visualize the systems as a pure one step delay system.	99

Table List

Table 2.1 Properties of classical set operations	8
Table 5.1 Comparative results of the PI fuzzy controller when implemented on an MCU and on a standard PC.	58
Table 6.1 Truth table of the XOR operation for three inputs and the ANN classification.....	77
Table 7.1 Fuzzy rule table for the autonomous robot fuzzy controller	94

Acronyms and Symbols

Acronyms List

2D	<i>Two-Dimensional</i>
3D	<i>Three Dimensional</i>
A/D	<i>Analog-to-Digital</i>
ANFIS	<i>Adaptive Network-base Fuzzy Inference System</i>
ANN	<i>Artificial Neural Network</i>
API	<i>Application Programming Interface</i>
ARIC	<i>Approximate Reasoning-based Intelligent Control</i>
ASCII	<i>American Standard Code for Information Interchange</i>
CANFIS	<i>Coactive ANFIS</i>
CASE	<i>Computer Aided Software Engineering</i>
CMOS	<i>Complementary Metal Oxide Semiconductors</i>
COG	<i>Center of Gravity</i>
DISO	<i>Dual-Input-Single-Output</i>
FEUP	<i>Faculdade de Engenharia da Universidade do Porto</i>
FLL	<i>Free Fuzzy Logic Library</i>
FIDE	<i>Fuzzy Inference Development Enviroment</i>
FPL	<i>Fuzzy Programming Language</i>
GCC	<i>GNU Compiler Collection</i>
GNU	<i>GNU's not UNIX</i>
GUI	<i>Graphical User Interface</i>
IDE	<i>Integrated Development Environment</i>
LFE	<i>Learning From Example</i>
LSE	<i>Least-Squares Estimate</i>
MCU	<i>Micro-Controller Unit</i>
MIMO	<i>Multi-Input-Multi-Output</i>
MLP	<i>Multi Layer Perceptron</i>
MOM	<i>Middle of Maxima</i>

NEFCON	<i>Neuro-Fuzzy Control</i>
NEFPROX	<i>Neuro-Fuzzy Function Approximator</i>
NNDFR	<i>Neural Network Driven Fuzzy Reasoning</i>
R&D	<i>Research and Development</i>
RAM	<i>Random Access Memory</i>
RBF	<i>Radial Basis Function</i>
RLSE	<i>Recursive Least-Squares Estimate</i>
RMSE	<i>Root Mean Squared Error</i>
ROM	<i>Read-Only Memory</i>
SISO	<i>Single-Input-Single-Output</i>
SOM	<i>Self-Organizing Maps</i>
USART	<i>Synchronous/Asynchronous Serial Peripherals</i>

Symbols List

$\mu_A(x)$	<i>Membership function of fuzzy set A</i>
U	<i>Universe set</i>
\emptyset	<i>Empty Set</i>
θ	<i>Vector parameter</i>
Δ	<i>Incremental</i>
$x_{l,i}$	<i>Output of the Neuron l from layer l</i>
ϵ_i	<i>Error signal for the network's node i</i>
w_{ij}	<i>Weight connection from node j to node i</i>
η	<i>Learning rate</i>
α	<i>Momentum Constant</i>

Chapter 1

Introduction

This dissertation arises as a sequence of a previous work devoted on the development of a software package that integrates the concepts of fuzzy set theory. Thus, FEUP Fuzzy Tool was created at the Faculty of Engineering of the University of Oporto, FEUP, as an educational tool that allow students to easily control non-linear Multiple Input Multiple Output MIMO systems under soft real time constraints, using the computational resources available. It was originally conceived under the Delphi 6 personal Edition IDE and presented to the scientific community with the publication of a scientific paper [1]. As fuzzy logic is now an integral part of the educational curricula in electrical and computer engineering, this toolbox has been used over the last years for introducing the concepts of fuzzy set theory and has had a tremendous impact on the courses dynamic, promoting motivation and enthusiasm for the soft computing sciences.

Throughout the years the prospective around the toolbox has been increasing dramatically, and so much was expected to be added to the FEUP Fuzzy Tool, in order to provide a better tool and to become more than a mere educational software. Today many of those things has been accomplished, thus delivering a powerful software with very special and unique features that puts the FEUP Fuzzy Tool on a very competitive position with respects to other cutting-edge fuzzy logic software.

1.1 - Motivation

1.1.1 - From an industrial point of view

The fuzzy control technology has emerged as one of the most effective nonlinear control technologies used in industrial applications. Everything began in 1970s with the pioneering work of E.H. Mamdani and the first successful application of fuzzy logic for steam engine control. Until the 80's fuzzy logic was a discipline only used by a restricted part of the

research community. It had some success in areas like operation research and soft sciences, whereas engineering and computing areas faced tremendous obstacles by the lack of enthusiasm from the industrial side in undertake larger projects using fuzzy logic. R&D investments of larger industrial groups of Northern Europe were devoted to smaller project instead of breakthrough projects seeking for new core of technology. Fuzzy control was faced with skepticism. The uncertain nature upon which fuzzy logic is build, led to believe that the application of this discipline to solve industrial problems seems to be an utopia.

Fifteen years after L.A. Zadeh first introduced this theory [2], fuzzy logic reach a clear level of acceptance in Japan and broad range of application started to emerge. They claimed fuzzy control to be the future universal tool of control engineering. Successfully stories of its implementation in home appliances, white good and home electronics began to attract the attention of other countries. By that time the fuzzy wave had reached Europe, and thus, the 90's marks the beginning of the "Fuzzy Boom". Recognized as a serious discipline, its application has dramatically increased ever since, thanks to the joint efforts of university and industry research teams, although many of the skepticism still remains by those who proclaim that everything that can be done using fuzzy logic, can be done as well with classical tools.

After the initial success of fuzzy controllers there has been no major success of fuzzy approaches in applications and fuzzy methods are still shunned by industry. It has been pointed out that one of the major challenges of fuzzy logic was the lack of tools that enable serious development [3]. There is a strong believe that fuzzy technology will not be picked up by industry unless useful software is widely available [4]. Also, a highly evolved infrastructure is indispensable for the effective development of fuzzy products to provide information, design tools and platforms for the realization of controllers [5].

Considering the range of applications of fuzzy logic in the industry, its benefits falls into one of the following categories:

- **Implements expert knowledge for a higher degree of automation:** for processes that requires human intervention in order to function properly. The knowledge of this operator cannot be easily translated into the formal mathematical framework upon which the control theory is built, instead lends itself to be defined by a set of cause-reaction statements, or rules. Fuzzy logic offers a method for representing and implementing this expert's knowledge.
- **Robust nonlinear control:** traditional controllers are only efficient over a restricted part of the process dynamics. As will be covered later chapters, fuzzy logic offers simple yet robust solutions that cope with nonlinear issues characteristic of the vast majority of industrial processes, acting as a simple knowledge-based controller or by complementing the traditional control approaches.

- **Reduction of development and maintenance time:** fuzzy logic provides two level of abstraction. The first, as mentioned before, enables to condense the characteristic inherent to the process as a set of qualitative rules. It facilitates the communication between groups of people involved on the development phase of the project, offering an informal and very spontaneous way of expression, the natural language. On the other hand, fuzzy operations are compiled into very elemental numerical objects and algorithms, such as look-up tables, interpolation, comparators etc. The existence of this compiled level for fast, real-time applications, as well as embedding fuzzy control essentially on the numerical environment of classical control.
- **Marketing and Patents:** in some business areas, the patent situation is so hard that coming up with a new solution with the standard technology is almost impossible without violating, or at least interfering, with the competitor's patent. In some cases fuzzy logic offers means to find qualitative equivalent solution by-passing the existing patents. Another reason is that product that uses fuzzy logic became synonym of user friendly or high quality, especially in the realm of home appliances after all the successfully stories that emerge from Japan.

1.1.2 - From an educational point of view

Education is an overall framework where teaching and dissemination plays two different roles for common purposes. While teaching can be considered as educating individuals, dissemination is, at a certain level, educating "The Society". Education of fuzzy logic theory, or in a more general framework soft computing, is itself an objective, but at the same time is a way to promote science and innovation in an ever changing society.

As explained in [6] four major activities related to scientific advancements and its effects on the society can be defined. Those four components are teaching, research, transfer and dissemination. Any institution makes its own personal use of these components, thus defining its own profiles. Universities are centered on teaching and research, whereas R&D centers have their interest on research and transfer, and very few pay attention on the matter of dissemination. As mentioned before, it's believed that one of the major obstacles that the emancipation of soft computing faces is the lack of available tools, especially free software tools. As the society is brought in deeper contact with the technology, the more it will get familiar with the potential benefits of its uses.

Nowadays fuzzy logic is an essential discipline in various engineering and science courses. Introducing the mathematical and theoretical foundations of fuzzy logic is typically the first step in teaching fuzzy logic. After learning the basics, students are required to work through various problems. Having students use fuzzy logic to solve a real world problem can be a good way to have students extend and reinforce their classroom knowledge. This step can be difficult for students and educators alike. There are numerous issues that can arise when

attempting this final step. Students may not have the necessary programming background that is required for the creation and coding of a fuzzy logic application. So providing the essential tools for accomplishing this task is crucial in developing the professional skills aimed.

Simulators and software packages can be effectively used as teaching aids that are able to significantly enhance the quality of knowledge acquisition when compared with conventional teaching means. Modern simulation is combined with processes visualization, model animation and multimedia technologies. Visualization should reflect, as close as possible, the true nature of the phenomena or behavior of the system under study. Visualized modeling results and especially animated models are advantageous as they facilitate deeper insight into the essence of the investigated phenomena or a problem and allow for profound understanding of the core of a complex mechanism or a system that is studied via simulation.

1.2 - The Feup Fuzzy Tool: Starting point

The main objective of this work is to empower the FEUP Fuzzy Tool with a new set of functionalities in order to deliver much powerful and competitive software. The following discussion will take into account the version of the FEUP Fuzzy Tool prior to this dissertation, and will briefly describe its structure. A hierarchical structure of the system prior to this dissertation can be seen at Figure 1.1.

This version is build over a set of libraries which delivers a programming Application Program Interface (API) with data structures that assembles the fuzzy inference system, and graphic functions.

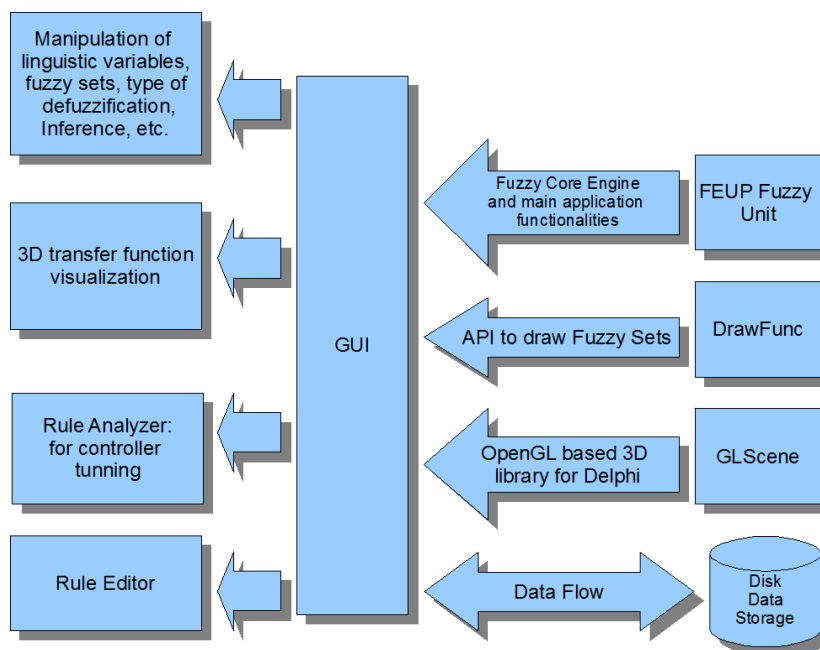


Figure 1.1 Structural view of the initial version of the FEUP Fuzzy Tool Software.

These libraries, also integrate a user friendly Graphical User Interface (GUI), from which the manipulation of the main fuzzy logic's entities and other functionalities can easily be accessed.

The core fuzzy engine of the FEUP Fuzzy Tool provides the central concepts like fuzzy sets, linguistic variables, fuzzy rules, fuzzy relations, linguistic hedges and is based on the Mamdani inference model with two defuzzification methods: the centroid and the MOM. This core was designed for speed which means that takes implementation details of fuzzy sets and operators into account as will be explained in latter chapters. Also, all the information about the fuzzy system under design can be accessed or stored on the disk from both API and GUI as they both offer means to accomplish these actions.

Finally, there are also a couple of libraries dedicated to implement and support graphical issues (i.e., graphical representation of membership functions or the fuzzy system's transfer characteristic, etc.) for 2D and 3D plotting purposes.

1.3 - Contributions

This dissertation provides the following relevant contribution:

- The participation in the CISTI 2009 - *4º Conferência Ibérica de Sistemas e Tecnologias da Informação*, with the contribution of a scientific paper titled "FEUP Fuzzy Tool - Renewed and Extended for Intelligent control and Education" [7].
- A powerful and more complete API to the FEUP Fuzzy Tool. With a new fuzzy inference system, namely the Takagi-Sugeno inference model.
- Implementation of the fuzzy associative memory interface for rule base manipulation.
- Added functionality of automatic generation of compatible programming code for embedded systems.
- Introduction of the neural network paradigm to the toolbox, specially the Multilayer Perceptron (MLP) also known as Backpropagation Network.
- Implementation of self-learning fuzzy architectures, that derives from the junction of the fuzzy logic paradigm with the neural networks. Namely the implementation of the ANFIS structure.
- Demonstration and validation of the concepts described previously.
- Improved Graphical User Interface GUI, with new interfaces for new functionalities.
- Illustrative examples of controlling an autonomous robotic system and a nonlinear plant control.

1.4 - Document Structure

The document is organized in eight chapters; this is the first one which gives an introduction to the context of this work. The second chapter introduces the basic concepts around fuzzy logic theory. The state of the art analysis is carried out at chapter three.

Chapter four describes the theory of fuzzy control and its implementation with the FEUP Fuzzy tool, and together with chapter five the implemented solution for fuzzy embedded controllers is described. Chapter six introduces the implementation of a new paradigm into the toolbox, namely the neural networks and neuro-fuzzy architectures. Chapter seven exposes the application of the toolbox functionalities and discusses some results through some computer simulated examples.

Finally at chapter eight the conclusion and future work recommendations are presented.

1.5 - Summary

This chapter introduced the objectives and motivation behind this work. It started highlighting with some historical aspects of the evolution of fuzzy logic, and explained the major concerns faced by the industry and society about fuzzy logic. Several conclusions can be inferred from this: first of all it is extremely helpful to have a relevant number of software tool that assist experimental system evaluation, implementation and tuning. The availability of free software is also crucial as it can be accessed with very few, if any, restrictions. The impact of this in education has proven to be critical.

Finally, the contribution of this work has also been exposed, in which has been emphasized the most important and relevant aspects of this work.

Chapter 2

Fuzzy Logic Theory

Frequently, in our interaction with the real world, we encounter classes of object with undefined, or at least partially defined, criteria of membership. An example of such ambiguity arises if we are asked to consider the relation of an integer, say 10, with the class of number much greater than the unity. It's clear that the greater the magnitude, the greater the membership degree to this class.

These characteristics cannot be simply described with the classical set theory, as it lacks the flexibility or a formal framework to describe these phenomena. Although classical sets are suitable for various applications and have proven to be an important tool for mathematics and computer science, they do not reflect the nature of human concepts and thoughts, which tend to be abstract and imprecise.

The notion of a fuzzy set provides the basis of a more general framework which completes the framework used in the case of ordinary sets and, provides a much wider scope of applicability, particularly in the fields of pattern recognition and information processing. Essentially, such a framework provides a natural way of dealing with problems in which the source of imprecision is the absence of sharply defined criteria of class membership rather than the presence of random variables.

This chapter introduces the basic definitions, notation, and operations for fuzzy sets, fuzzy relation and approximate reasoning. Its aim is to provide a brief introduction to the basic concepts central to the study of fuzzy logic theory. More detailed information can be found in [8], [9], [10] and [11].

2.1 - Classical sets

A classical set is a collection of objects of any kind. The concept of a set has become one of the most fundamental notions of mathematics. Let A be a set, " $x \in A$ " means that x is an

element of the set A and " $x \notin A$ " means that x does not belong to the set A . A classical set may be countable or uncountable. The manner which the objects are specified can be either by listing up the individual elements of the set or by a statement that describes common characteristics of all the objects, that is if $P(x)$ is a predicate stating that every x has the property P , the set can be specified by the notation $\{x | P(x)\}$.

Two sets are of great importance, namely the universe U containing all the elements of the universe of discourse, and the empty set \emptyset containing no elements at all. They complement each other.

When the sets A and B are such that $A=B$, then A has the same elements that B . This defines the equality among sets. If all elements of A are also elements of B , that is if A is subset of B , it is denoted by $A \subseteq B$. Furthermore, if B contains all the elements of A but also has more elements, then A is a proper set of B and it is denoted by $A \subset B$.

Definition 2.1 $\mu_A: X \rightarrow [0,1]$ is a characteristic function of the set A if and only if for all x

$$\mu_A(x) = \begin{cases} 1 & \text{when } x \in A \\ 0 & \text{when } x \notin A \end{cases} \quad (2.1)$$

2.1.1 - Classical set operations

The following list represents the most common sets operation:

- Complement of A , $A' = \{x | x \notin A\}$.
- Intersection of A and B , $A \cap B = \{x | x \in A \wedge x \in B\}$.
- Union of A and B , $A \cup B = \{x | x \in A \vee x \in B\}$.
- Difference of A and B , $A - B = \{x | x \in A \wedge x \notin B\}$.
- Power Set of A , $P(A) = \{X | X \subseteq A\}$.
- Cartesian product of A and B , $A \times B = \{(x, y) | x \in A \wedge y \in B\}$.
- Power n of A , $A^n = A \times \dots \times A$.

The concepts of intersection, union and the Cartesian product can be extended to n sets:

$$\bigcup_{i=1}^n A_i = A_1 \cup \dots \cup A_n \quad (2.2)$$

$$\bigcap_{i=1}^n A_i = A_1 \cap \dots \cap A_n \quad (2.3)$$

$$\times_{i=1}^n A_i = A_1 \times \dots \times A_n = \{(x_1, \dots, x_n) | x_1 \in A_1, \dots, x_n \in A_n\} \quad (2.4)$$

2.1.2 - Properties

The most important properties of classical set operation are summarized in

Table 2.1 Properties of classical set operations

Involution	$(A')' = A$
Commutativity	$A \cup B = B \cup A,$ $A \cap B = B \cap A$

Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Distributivity	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
Idempotence	$A \cup A = A$ $A \cap A = A$
Absorption	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$
Absorption of complement	$A \cup (A' \cap B) = A \cup B$ $A \cap (A' \cup B) = A \cap B$
Absorption of U and \emptyset	$A \cup U = U$ $A \cap \emptyset = \emptyset$
Identity	$A \cap U = A$ $A \cup \emptyset = A$
Law of contradiction	$A \cap A' = \emptyset$
Law of excluded middle	$A \cup A' = U$
De Morgan's Law	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$

2.2 - Fuzzy Sets

In classical sets the transition for an element in the universe between membership and non-membership in a given set is abrupt and well-defined, in other words it is said to be “crisp”. For an element in a universe that contains fuzzy sets, this transition can be gradual. This transition among various degrees of membership can be thought of as conforming to the fact that the boundaries of the fuzzy sets are vague and ambiguous, describing uncertainty on the information they convey.

Definition 2.2 The membership function of a fuzzy set F is a function $\mu_F: U \rightarrow [0,1]$, so that every element u from U has a membership degree $\mu_F(u) \in [0,1]$. F is completely specified by the set of tuples

$$F = \{(u, \mu_F(u)) | u \in U\}. \quad (2.5)$$

In fuzzy sets theory, the range of possible quantitative values considered for fuzzy set members is called universe of discourse. Universe of discourse can be continuous or discrete. Discrete universe of discourse is normally bounded and contains a finite number of elements. A fuzzy set with discrete universe of discourse is called a discrete fuzzy set.

Another type of notation introduced by Zadeh to describe any countable or discrete universe is

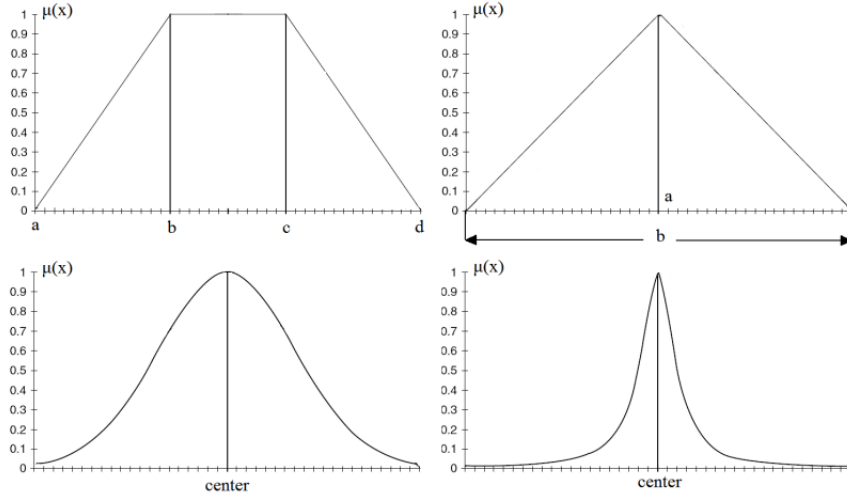


Figure 2.1 Plots of various membership functions: Trapezoidal (upper left), Triangular (upper right), bell-shaped (lower left) and Gaussian (lower right).

$$F = \sum_{u \in U} \mu_F(u)/u, \quad (2.6)$$

but when U is uncountable or continuous is denoted by

$$F = \int_U \mu_F(u)/u. \quad (2.7)$$

Notice that on the previous notation neither the integral (Summation) nor the division operator corresponds to the classical notation for these operations; they represent a collection of associated elements, namely membership degree and the respective set object. Membership functions can attain different forms. However, triangular, trapezoidal, Gaussian, and bell-shaped forms, shown in Figure 2.1, are used more than others. Their mathematical expressions are the following

$$\mu_{\text{triangle}}(x) = \begin{cases} 1 - \frac{2 \cdot |x-a|}{b}, & \text{if } 2|x-a| \leq b \\ 0, & \text{otherwise} \end{cases} \quad (2.8)$$

$$\mu_{\text{trapezoidal}}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x < b \\ 1, & b \leq x < c \\ \frac{d-x}{d-c}, & c \leq x < d \\ 0, & x \geq d \end{cases} \quad (2.9)$$

$$\mu_{\text{bell-shaped}}(x) = \frac{1}{1+(x-\text{center})^2} \quad (2.10)$$

$$\mu_{\text{gaussian}}(x) = e^{-(x-\text{center})^2/\sigma} \quad (2.11)$$

2.2.1 - Basic definitions

All the information contained in a fuzzy set is described by its membership function; some basic definitions about its properties are given next. For purposes of simplicity, the functions shown in the following figures will all be continuous, but the terms apply equally for both discrete and continuous fuzzy sets. Figure 2.2 will be helpful in this description.

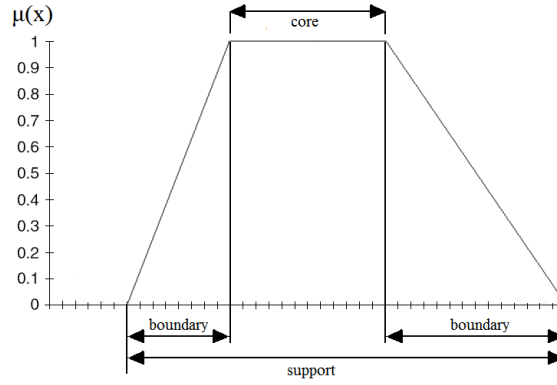


Figure 2.2 An arbitrary fuzzy set where the concepts of core, boundary and support are shown.

The core of a membership function for some fuzzy set A is defined as that region of the universe that is characterized by complete and full membership in the set A . That is, the core comprises those elements x of the universe such that $\mu_A(x) = 1$.

The support of a membership function for some fuzzy set A is defined as that region of the universe that is characterized by nonzero membership in the set A . That is, the support comprises those elements x of the universe such that $\mu_A(x) > 0$.

The boundaries of a membership function for some fuzzy set A , are defined as that region of the universe containing elements that have a nonzero membership but not complete membership. That is, the boundaries comprise those elements x of the universe such that $0 < \mu_A(x) < 1$. These elements of the universe are those with some degree of fuzziness, or only partial membership in the fuzzy set A . Figure 2.2 illustrates the regions in the universe comprising the core, support, and boundaries of a typical fuzzy set.

A normal fuzzy set is one whose membership function has at least one element x in the universe whose membership value is unity. For fuzzy sets where one and only one element has a membership equal to one, this element is typically referred to as the prototype of the set, or the prototypical element.

The notion of convexity can readily be extended to fuzzy sets in such a way as to preserve many of the properties which it has in the context of ordinary sets. In what follows, it is assumed for concreteness that the universe U is a real Euclidean space E^n .

Definition 2.3 A fuzzy set A is *convex* if and only if the crisp sets defined by $\Gamma_\alpha = \{x | \mu_A(x) \geq \alpha\}$ are convex for every α . Each of the Γ_α sets are called α -cuts of the fuzzy set A . The definition of convexity can also be alternatively given by

$$\mu_A(\lambda \cdot x + (1 - \lambda) \cdot y) \geq \min\{\mu_A(x), \mu_A(y)\}, \quad \forall x, y \in U \wedge \lambda \in [0, 1] \quad (2.12)$$

A convex fuzzy set is described by a membership function whose membership values are strictly monotonically increasing, or whose membership values are strictly monotonically decreasing, or even those whose membership values are strictly monotonically increasing then strictly monotonically decreasing with increasing values for elements in the universe.

If A is a convex single-point normal fuzzy set defined on the real line, is often termed a fuzzy number. A fuzzy set whose support is a single element in U with $\mu_A(x) = 1$ is called fuzzy Singleton.

The crossover points of a membership function are defined as the elements in the universe for which a particular fuzzy set A has values equal to 0.5, i.e., for which $\mu_A(x) = 0.5$.

2.2.2 - Operations and properties

The notions of equality, inclusion, and intersection of fuzzy sets are immediately derived from the classical set theory.

Two fuzzy sets are said to be equal ($A=B$) if and only if

$$\forall x \in U: \mu_A(x) = \mu_B(x). \quad (2.13)$$

The fuzzy set A is subset of B ($A \subseteq B$) if and only if

$$\forall x \in U: \mu_A(x) \leq \mu_B(x). \quad (2.14)$$

In classical set theory the union, intersection and complement of sets are simple operations that are unambiguously defined. This follows from the fact that the logical operators *and*, *or* and *not* have a well defined semantics based on propositional logic. On the other hand, on fuzzy set theory this semantic offer a non-unanimous interpretation. Zadeh [2] proposed the following operators:

$$\forall x \in U: \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x)), \quad (2.15)$$

$$\forall x \in U: \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)), \quad (2.16)$$

$$\forall x \in U: \mu_{A'}(x) = 1 - \mu_A(x). \quad (2.17)$$

These are the most common operations that extend the basic operations to fuzzy sets although other operators are possible as well, as long as they satisfy some properties. More generally, Triangular norms (T-norms, S-norms) are used to represent intersection, union and complement operators.

Definition 2.4 A Triangular norm or T-norm $\hat{\star}$ denotes a class of binary operators that can represent the intersection operation. These operators satisfy the following criteria:

- $a \hat{\star} b = b \hat{\star} a$;
- $(a \hat{\star} b) \hat{\star} c = a \hat{\star} (b \hat{\star} c)$;
- $a \leq c$ and $b \leq d \implies a \hat{\star} b \leq c \hat{\star} d$.
- $a \hat{\star} 1 = a$.

Definition 2.5 A triangular co-norm or S-norm $\check{\star}$ denotes a class of binary operators that can represent the union operation. It satisfies the first three criteria of the T-norms, but also $a \check{\star} 0 = a$.

Definition 2.6 The complement operation c should satisfy at least:

- $c(0) = 1$;
- $a < b$ implies $c(a) > c(b)$;
- $c(c(a)) = a$;

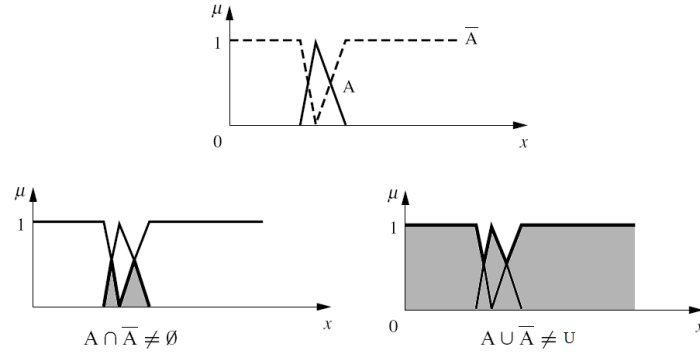


Figure 2.3 Exemplification of why the excluded middle axiom and the law of contradiction fails for fuzzy sets. Adapted from [10].

This enables to define a more general class of intersection, union and complement operations:

$$\forall x \in U: \mu_{A \cap B}(x) = \mu_A(x) \hat{\star} \mu_B(x), \quad (2.18)$$

$$\forall x \in U: \mu_{A \cup B}(x) = \mu_A(x) \star \mu_B(x), \quad (2.19)$$

$$\forall x \in U: \mu_{A'}(x) = c(\mu_A(x)). \quad (2.20)$$

Also the properties from Table 2.1 also hold for fuzzy sets with the exception of the excluded middle axiom and the law of contradiction. The reason behind this is that given the overlap between the boundaries of a fuzzy set and its complements, together with the definition of the union and intersection operators, it can easily be seen that $A \cup A' \neq U$ and $A \cap A' \neq \emptyset$ as shown in Figure 2.3.

2.3 - Relations

2.3.1 - Classical relations

A relation can be considered a set of tuples, where a tuple is an ordered pair. A binary tuple is denoted as (x,y) , and a general n -ary tuple is (x_1, \dots, x_n) .

It's obvious that the Cartesian product mentioned before is an n -ary relation in its more general form. Just as classical sets relations can be described by a characteristic function.

Definition 2.7 $\mu_R: X_1 \times \dots \times X_n \rightarrow [0,1]$ is a characteristic function if and only if for all x_1, \dots, x_n ,

$$\mu_R(x_1 \dots x_n) = \begin{cases} 1 & \text{when } (x_1 \dots x_n) \in R, \\ 0 & \text{when } (x_1 \dots x_n) \notin R. \end{cases} \quad (2.21)$$

When the universes, or sets, are finite the relation can be conveniently represented by a matrix, called a relation matrix. An r -ary relation can be represented by an r -dimensional relation matrix. The relation represented by the Sagittal diagram of Figure 2.4 is the same that the relation represented by the matrix

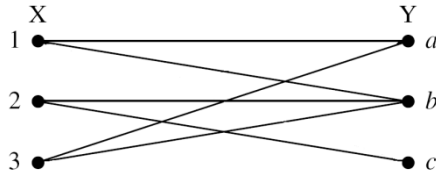


Figure 2.4 A Sagittal diagram of a classical relation

$$R = \begin{matrix} & a & b & c \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \end{matrix}.$$

Define R and S as two separate relations on the Cartesian universe $X \times Y$, and define the null relation and the complete relation as the relation matrices O and E , respectively. An example of a 3×3 form of the O and E matrices is given here:

$$O = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The following operations for the two crisp relations (R, S) can now be defined.

- Union $R \cup S \rightarrow \mu_{R \cup S}(x, y) = \mu_{S \cup R}(x, y) = \max(\mu_R(x, y), \mu_S(x, y))$.
- Intersection $R \cap S \rightarrow \mu_{R \cap S}(x, y) = \mu_{S \cap R}(x, y) = \min(\mu_R(x, y), \mu_S(x, y))$.
- Complement $\bar{R} \rightarrow \mu_{\bar{R}}(x, y) = 1 - \mu_R(x, y)$.

The properties of commutativity, associativity, distributivity, involution, and idempotency all hold for crisp relations just as they do for classical set operations. Moreover, De Morgan's principles and the excluded middle axioms also hold for classical relations just as they do for classical sets. The null relation, O , and the complete relation, E , are analogous to the null set, \emptyset , and the whole set, U , respectively, in the set-theoretic case.

Let R be a relation that relates, or maps, elements from universe X to universe Y , and let S be a relation that relates, or maps, elements from universe Y to universe Z . A useful operation that provides means to find a relation T , which relates the same elements in universe X that R contains to the same elements in universe Z that S contains, is known as composition.

There are two common forms of the composition operation; one is called the max-min composition and the other the max-product composition. The max-min composition is defined by the set-theoretic and membership function-theoretic expressions

$$T = R \circ S = \mu_T(x, z) = \max_{y \in Y} [\min(\mu_R(x, y), \mu_S(y, z))]. \quad (2.22)$$

The max-product (sometimes called max-dot) composition is defined by the expression:

$$T = R \cdot S = \mu_T(x, z) = \max_{y \in Y} [\mu_R(x, y) \cdot \mu_S(y, z)]. \quad (2.23)$$

2.3.2 - Fuzzy relations

The former section stated that a relation can be considered a set of tuples. The same way a fuzzy relation is a fuzzy set of tuples, i.e., each tuple has a membership degree. More formally,

Definition 2.8 Let U and V be uncountable or continuous universes, and $\mu_R: U \times V \rightarrow [0,1]$, then

$$R = \int_{U \times V} \mu_R(u, v)/(u, v). \quad (2.24)$$

is a binary fuzzy relation on $U \times V$. If U and V are discrete universes then

$$R = \sum_{U \times V} \mu_R(u, v)/(u, v). \quad (2.25)$$

In the same way as in classical relations, it is possible to define n -ary fuzzy relations as a fuzzy set of n -tuples.

$$\mu_R(x_1 \dots x_n)/(x_1 \dots x_n). \quad (2.26)$$

Because fuzzy relations in general are fuzzy sets, we can define the Cartesian product to be a relation between two or more fuzzy sets. Let A be a fuzzy set on universe X and B be a fuzzy set on universe Y ; then the Cartesian product between fuzzy sets A and B will result in a fuzzy relation $R = A \times B$ with membership function

$$\mu_R(x, y) = \min(\mu_A(x), \mu_B(y)). \quad (2.27)$$

All the operations defined for classical relations can be extrapolated to the fuzzy relation based on these definitions. Just as for crisp relations, the properties of commutativity, associativity, distributivity, involution, and idempotency all hold for fuzzy relations. Moreover, De Morgan's principles hold for fuzzy relations just as they do for crisp relations, but as happens with fuzzy sets neither the law of contradiction nor the excluded middle axiom properties holds for fuzzy relations.

2.4 - Approximate Reasoning

Approximate reasoning is the best known form of fuzzy logic and whose form of inference is in sharp contrast to that used in classical logic. Inference in approximate reasoning is computation with fuzzy sets that represents the meaning of a certain proposition. Here the concepts of linguistic variable, fuzzy proposition and fuzzy rule will be introduced as they are the backbone of fuzzy logic.

2.4.1 - Linguistic variables

By a linguistic variable is understood a variable whose values are words or sentences in a natural or artificial languages. If we think of *temperature* as a linguistic variables a set of possible values are hot, warm, cool, cold, etc. In our everyday activities we use this kind of term which embeds the same information as their numerical counterpart. Thus, the term temperature may attain two different values: numerical (35 °C) and linguistic (warm). The linguistic variables may assume different linguistic values over a specified universe of

discourse. This means that linguistic values defined by an appropriate semantic rule represent nothing but informative attributes about the physical values in a part of a specified universe of discourse.

A linguistic variable can be noted in this way:

$$\langle X, T, \chi, M_x \rangle. \quad (2.28)$$

Where X denotes the symbolic name of the linguistic variable, T is the set of linguistic values that X can take; x represents the universe of discourse or physical domain over which the linguistic variable takes is quantitative crisp values and M_x is a semantic function which gives meaning, or interpretation, of a linguistic value in terms of the quantitative elements of x , i.e., it maps a linguistic term from T , to the correspondent fuzzy set.

Definition 2.9 Let $x \in X$ be a linguistic variable and $T_i(x)$ be a fuzzy set associated with a linguistic value T_i . Then the structure

$$x \text{ is } T_i, \quad (2.29)$$

represent what is called a fuzzy proposition, and is the atomic primitive that expresses knowledge in approximate reasoning.

The meaning of an atomic expression is defined by the corresponding fuzzy set, or membership function which defines it, giving the corresponding degree that a particular value from the universe of discourse satisfy this statement. This process is known as *fuzzification*.

2.4.2 - Linguistic hedges

In linguistics, fundamental atomic terms are often modified with adjectives or adverbs like very, low, slight, more or less, fairly, slightly, almost, barely, mostly, roughly, approximately, and so many more that it would be difficult to list them all. They are called modifiers or linguistic hedges, that is, the singular meaning of an atomic term is modified, or hedged, from its original interpretation. Using fuzzy sets as the calculus of interpretation, these linguistic hedges have the effect of modifying the membership function for a basic atomic term. As an example, let us look at the basic linguistic atom, a fuzzy proposition p , and subject it to some hedges. Define the meaning of p as $\rho = \int_{y \in Y} \mu_p(y)/y$, then

$$\text{very } p = \rho^2 = \int_{y \in Y} \mu_p(y)^2 / y, \quad (2.30)$$

$$\text{plus } p = \rho^{1.25}, \quad (2.31)$$

$$\text{slightly } p = \sqrt{\rho} = \int_{y \in Y} \sqrt{\mu_p(y)} / y, \quad (2.32)$$

$$\text{minus } p = \rho^{0.25}. \quad (2.33)$$

The expressions shown in equations (2.30) and (2.31) are linguistic hedges known as concentrations. Concentrations tend to concentrate the elements of a fuzzy set by reducing the degree of membership of all elements that are only “partly” in the set. Alternatively, the expressions given in equations (2.32) and (2.33) are linguistic hedges known as dilations, or dilutions. Dilations stretch or dilate a fuzzy set by increasing the membership of elements that are “partly” in the set.

Definition 2.10 Let p and q be two fuzzy propositions, p : “ X is A ” and q : “ Y is B ” where A and B are fuzzy sets defined on the universes of discourse X and Y . The meaning of the compound proposition “ X is A and Y is B ” is given by the relation

$$R \subseteq X \times Y = \int_{(x,y) \in X \times Y} \min(\mu_A(x), \mu_B(y)) / (x, y). \quad (2.34)$$

Definition 2.11 Let p and q be two fuzzy propositions, p : “ X is A ” and q : “ Y is B ” where A and B are the aforementioned fuzzy sets defined on the universes of discourse X and Y . The meaning of the compound proposition “ X is A or Y is B ” is given by the relation

$$R \subseteq X \times Y = \int_{(x,y) \in X \times Y} \max(\mu_A(x), \mu_B(y)) / (x, y). \quad (2.34)$$

If A and B are defined on the same universes the meaning of the previous compound propositions, for “ X is A and Y is B ” and “ X is A or Y is B ” will be given by the intersection and union respectively. The meaning of a negated proposition “ X is not A ” is given by the complement fuzzy set of A .

2.4.3 - The if-then statements

A fuzzy conditional production rule is symbolically expressed as if <fuzzy proposition> then <fuzzy proposition>, where <fuzzy proposition> is either an atomic or compound proposition.

The if-then statement is nothing more than an implication from a mathematical point of view. From classical logic we know that this implication can be described, in a more general framework, as a relation. Let the implication operation involves two different universes of discourse; P is a proposition described by set A , which is defined on universe X , and Q is a proposition described by set B , which is defined on universe Y . Then the implication $(P \rightarrow Q)$ can be represented in set-theoretic terms by the relation R , where R is defined by

$$R = (A \times B) \cup (A' \times Y). \quad (2.35)$$

In classical logic it is useful to consider compound propositions that are always true, irrespective of the truth values of the individual simple propositions. Classical logical compound propositions with this property are called tautologies. Tautologies are useful for deductive reasoning, for proving theorems, and for making deductive inferences. One tautology, known as modus ponens deduction, is a very common inference scheme used in forward-chaining rule-based expert systems. It is an operation whose task is to find the truth value of a consequent in a production rule, given the truth value of the antecedent in the rule. Modus ponens deduction concludes that, given two propositions, P and $(P \rightarrow Q)$, both of which are true, then the truth of the simple proposition Q is automatically inferred. That is,

$$(A \wedge (A \rightarrow B)) \rightarrow B \quad (2.36)$$

A typical if-then rule is used to determine whether an antecedent (cause or action) infers a consequent (effect or reaction). Suppose we have a rule of the form IF A , THEN B , where A is a set defined on universe X and B is a set defined on universe Y . As discussed before, this rule can be translated into a relation between sets A and B .

Now suppose a new antecedent, say A_1 is known. We can use modus ponens deduction, to infer a new consequent, say B_1 . Since “A implies B” is defined on the Cartesian space $X \times Y$, B_1 can be found through the following set-theoretic formulation

$$B_1 = A_1 \circ R = A_1 \circ (A \times B) \cup (A' \times Y). \quad (2.37)$$

The symbol \circ denotes the composition operation. The veracity of B_1 , is automatically inferred by the veracity of A_1 . This kind of deductive inference scheme can be extended with the fuzzy sets theory concepts, establishing theoretical foundation for reasoning about imprecise propositions; such reasoning has been referred to as approximate reasoning. Approximate reasoning is analogous to classical logic for reasoning with precise propositions, and hence is an extension of classical propositional calculus that deals with partial truths.

In classical binary logic this inverse does exist; that is, crisp modus ponens would give $B = A \circ R = A_1 \circ R = B_1$, where the sets A and B are crisp, and the relation R is also crisp. In the case of approximate reasoning, the fuzzy inference is not precise but rather is approximate. However, the inference does represent an approximate linguistic characteristic of the relation between two universes of discourse, X and Y.

There are many other definitions for the implication operator in the literature [8], [10] and [11]. One of the most commonly used in practice is known as correlation-minimum or Mamdani implication after the British Prof. Mamdani’s work in the area of system control [12]. The formulation for the implication is also equivalent to the fuzzy cross product of fuzzy sets A and B, i.e., $R=A \times B$, thus the meaning of IF A, THEN B is given by

$$\mu_R(x, y) = \min[\mu_A(x), \mu_B(y)] \quad (2.38)$$

2.4.4 - Rule-based system

In the field of artificial intelligence (machine intelligence) there are various ways to represent knowledge. Perhaps the most common way to represent human knowledge is to form it into natural language expressions of the type IF premise (antecedent), THEN conclusion (consequent). The form in this expression is commonly referred to as the IF-THEN rule-based form; this form generally is referred to as the deductive form. It typically expresses an inference such that if we know a fact (premise, hypothesis, antecedent), then we can infer, or derive, another fact called a conclusion (consequent). Most rule-based systems involve more than one rule. The process of obtaining the overall consequent (conclusion) from the individual consequents contributed by each rule in the rule-base is known as aggregation of rules. As the result of each rule is a fuzzy set, the aggregation of the rule base can be carried out by the intersection and union operator, depending on whether we are dealing with a conjunctive or disjunctive system of rules respectively.

In general, there are two principal ways of computing the contribution of each activated rule: by using either an individual rule based or a composition based inference engine. The first step of individual rule-based inference, which is predominantly used in fuzzy controller design, is that for each activated rule we first calculate the membership function of the rule

antecedent, and then calculation of the influence on the consequent part of the rule is made. When this procedure is carried out for all activated fuzzy rules, the aggregation operation concludes individual rule-based inference with one output fuzzy set, which may be used thereafter for the computation of crisp output value.

The composition based inference uses membership functions of all the set of rules for the calculation of one compositional output membership function

$$\mu_{compositional}(x, y) = \bigcup_i \mu_{R_i}(x, y). \quad (2.39)$$

Once a compositional membership function is calculated, equation (2.37) must be used to infer the output fuzzy set.

2.5 - Summary

This chapter focused on the basic concepts of fuzzy sets and their algebra. After the brief introduction of the basic definitions, properties, and operations of crisp sets and fuzzy sets, it also introduces those concepts for classical and fuzzy relations. The idea of composition was given, which is critical notion for approximate reasoning. Because they are used throughout the book, it was necessary to introduce basic definitions of terms such as linguistic variables, fuzzy propositions, relations, implications, and rule-based systems.

The basic rule of inference in traditional two-valued logic is modus ponens, according to which we can infer the truth of a proposition B from the truth of A and the implication $(A \rightarrow B)$. This concept is extended to the field of fuzzy set and together with the definition of linguistic variable, forms the theoretical foundation for reasoning about imprecise propositions. Finally, the rule-based system appears as the main structure to embed knowledge, and as a whole dictates the behavior of the system subjected to the interaction with its environment. The individual rule based and the composition-based inference engine are two different ways to derive the meaning to a set of fuzzy rules.

Chapter 3

State of the Art - Market Analysis

In this chapter it'll be presented some of the more important and currently used fuzzy logic software. Most of the software was created by academic groups with licenses that ranges from GNU open source software to packages with undisclosed source code that are only free for research and education.

The available software packages can be categorized into programming libraries that provides an API for specific applications where fuzzy logic is needed, and application tools.

3.1 - Programming libraries

This type of fuzzy software is ideal for both research and applications development, because it could be conveniently extended and used for different purposes, but most of them have limited use as an educational tool. Unfortunately there are not much of these tools.

The Free Fuzzy Logic Library FFL [13], is an open source fuzzy logic class library and API that is optimized for speed critical applications, and has proven to be successful in areas such as video games. FFL is able to load files that adhere to the IEC 61131-7 standard for programmable logic controllers, which section seven deals with fuzzy logic programming. It has been written in C++ under the object oriented programming paradigm. The library is free and the source code is available, but has not been updated since 2003. It defines the concepts of fuzzy sets, linguistic variables, rules through a well defined class hierarchy, descending from an unique ancestor, the FFLBase class. It offers two methods of defuzzification: the COG (Center of Gravity) and MOM (Mean of Maxima). As said before, the library is intended to support software development and by itself it has limited use in terms of educational purposes.

Togai InfraLogic, Inc. [14] intends to provide support for the development of fuzzy controllers with a number of tools like the Fuzzy Programming Language FPL, the Fuzzy C-Compiler and the TILShell. The TILShell is a Windows-based software development tool

providing users with a way to design, debug and test fuzzy logic expert systems, including embedded control systems. Is intended to be graphical object-based "point and click" CASE (Computer Aided Software Engineering) tool. It uses the FPL, an object-oriented fuzzy logic development language, to define the fuzzy systems, and the Fuzzy C-Compiler to translate the system into C code. Another tool created by Togai InfraLogic, Inc. is FuzzyCLIPS [15]. It was developed under a Small Business Innovation Research contract with NASA's Johnson Space Center to add fuzzy reasoning capability to an already existing product named CLIPS. These products are all portable to other ANSI-C platforms.

Another interesting set of libraries with some interesting features can be found under the name of StarFLIP++ [16], although this project hasn't been update since 1997. It was written in C++ by a group of academics from the Institute of Information Systems of Vienna University of Technology. The source code and documentation for the libraries are online. One of its best features is that it offers support for membership function adjustment through a neural networks library extension.

Finally, the FuzzyJ Toolkit [17] from the National Research Council of Canada's Institute for Information Technology is a set of Java classes that provide the capability for handling fuzzy concepts and reasoning. It's free and provides a rich API with 12 types of fuzzy sets, 3 types of fuzzy rules, fuzzy values, fuzzy variables and linguistic modifiers that can be used standalone to create fuzzy systems and do reasoning. Unlike the other software in this category, this one is appealing as an educational resource, as it comes with some interesting examples that show some insight into the fuzzy logic capabilities.

All of the above mentioned tools implement the core of fuzzy logic theory, some of them more complete in some aspects than others, but all of them lack the implementation of advance concepts like the implementation of self-learning or self-organizing fuzzy archetypes that are critical to many applications. They perform well when basic fuzzy logic algorithms are considered, but problems may arise when more complex or specific control structures are needed. The lack of a Graphical User Interface (GUI) of some libraries represents an obstacle when it comes to design and tune a fuzzy controller.

3.2 - Application Tools

The vast majority of fuzzy software comes in the shape of tools and demonstrators. This type of software is typically of limited use and not fit for large scale business applications. Exception to this affirmation are the MATLAB Fuzzy Logic Toolbox [18], [19] FuzzyTECH [20]. These are one of the most advanced software available with a very rich set of functionalities.

The Fuzzy Logic Toolbox product extends the MATLAB technical computing environment with tools for designing systems based on fuzzy logic. The GUI helps through the steps of fuzzy inference system design. Functions are provided for many common fuzzy logic methods, including fuzzy clustering and adaptive neurofuzzy learning, enabling functionalities rarely

seem in other software of this type. This learning capability is provided by the implementation of the Adaptive Neural-based Fuzzy Inference System (ANFIS) integrated on the toolbox by its creator Roger Jang. It supports the inclusion of the fuzzy logic blocks in its simulating environment Simulink, as an integrated part of the simulation. Also has the ability to generate embeddable C code or stand-alone executable fuzzy inference engines. Unfortunately it suffers some calamities, the great computational burden place on the computer and its speed are not of its best qualities.

The FuzzyTECH is a complete software which editions are specialized for technical applications. FuzzyTECH has a rich graphical user interface with 2D and 3D plotter that enables to analyze the transfer characteristics of a fuzzy logic system or its components in any possible way. Any modification of the system is immediately reflected in all analyzers. Also supports neuro-fuzzy learning and generation of portable C code, assembly source code for microcontrollers, or function blocks for programmable logic controllers. FuzzyTECH is not suited for educational purposes. Nonetheless, it supports portability to other platforms such as MATLAB's Simulink.

Other interesting fuzzy software are the Scilab Fuzzy Logic Toolbox [21], [22] and the Wolfram Mathematica Fuzzy Logic Toolbox [23]. They are well suited around the basic concepts of fuzzy logic, specially the Mathematica Fuzzy Logic Toolbox, which makes them good for teaching and at some degree for researching, but not for industrial needs.

There is also a quite advanced free tool, Xfuzzy [24], [25]. It XFuzzy is a development environment for fuzzy inference systems. It consists of number of tools for inferencing, simulation, learning, graphical representation, editing and program synthesis. The tools are based on a common specification language XFL3. Xfuzzy has been written in Java and is free software under the GNU Public License GPL. It is probably the most advanced free fuzzy software package that is available today .XFuzzy can generate source code in C, C++ and Java, it is a very appealing feature. The last update is from 2003 and it is not obvious from the team's web site, if the development still continues. Because it is already quite an extensive toolkit, has been written in Java and is published under the GPL it could serve as the foundation of a larger community process. However, the source code was not available at the download site.

Finally, there is one other tool called FIDE (Fuzzy Inference Development Enviroment) [26] whose purpose is to easy and supports all phases of application development: concept, design, tuning, simulation and implementation. Again its most advanced functionality is the automatic code generation, where FIDE outputs C and assembly code for a variety of microcontrollers.

3.3 - Comparative Requirement Analysis

This section intends to establish a comparative analysis of the FEUP Fuzzy Tool and emphasize the functional aspects that need to be implemented to make it an state-of-the-art software.

3.3.1 - Core Fuzzy Engine

Little improvements can be made in terms of this matter. Although the addition of the Takagi-Sugeno type of inference will play a central role in terms of learning capabilities. It is necessary to consider that many fuzzy applications will not be fully knowledge-based, but will have to be derived from data at least to some extent. That means is crucial to implement learning algorithms.

3.3.2 - Learning

This is one of the most important features that are actually needed on the FEUP Fuzzy Tool and very few tools have integrated this capability. One reason is that it will make it comparable with the top-rated fuzzy logic software, but mainly because it's actually needed in most of real world applications to deliver a superior degree of control. Different learning algorithms have different requirements on the representation of fuzzy systems. Many learning approaches manipulate parameters of membership functions and generate or change fuzzy rules and therefore need access to the implementation details of the fuzzy core engine. Learning in fuzzy system is still an active research area and we can expect that many researchers want to contribute with new algorithms. They should not need to change the core in order to integrate a new learning procedure. It would be more efficient if a clean core API would be available that enables fast and convenient implementation of learning algorithms. There lies the necessity of implementation for the Sugeno model as it was conceived in an effort to model a fuzzy system from acquired data. The Mamdani model is best suited for the integration of structured knowledge, and although there are various procedures to deliver a self-organizing or self-learning fuzzy architecture, some introduced by Mamdani himself, a richer API will emerge from the integration of the Sugeno type of inference. The reason behind this will be cleared latter on.

3.3.3 - Portability

In the interest of developing real-time autonomous systems a number of approaches have been focusing in the use of microcontrollers and integrated circuits alike. In order for an embedded system to achieve a greater level of intelligence it must handle uncertainty. FEUP Fuzzy Tool is designed to run on a general purpose personal computer under the most common platforms namely, Mac OS, MS Windows and GNU/Linux. Such implementation is suitable for a number of applications but is not very feasible for embedded systems. The

ability to export to an embedded system the fuzzy controller has become a greater need and most of the aforementioned software tools already support this.

3.4 - Summary

This chapter explores the current state of the art of the most important fuzzy logic software available and classifies it in two categories: programming libraries and application tools. The major part of the existing free fuzzy logic software haven't reach yet the needed level of maturity, exploiting only fragments of fuzzy logic theory or lacking functionality for real problems applications. This is also true for some of the commercial tools available. Another disadvantage of the commercially available tool is their license acquisition cost, making even the most advanced ones look less appealing when compared with the free software even if provides lesser functionality.

A conjoined effort of the academic circles and industry is needed in order to deliver a more mature and powerful second generation of fuzzy logic free software. The FEUP Fuzzy Tool project fits perfectly in this profile.

Chapter 4

Fuzzy Control and FEUP Fuzzy Tool

This chapter focuses on describing the fundamental issues of the implementation of fuzzy controllers inside the FEUP Fuzzy Tool. Instead of introducing this concept directly, first the classical structure of a fuzzy controller is described.

4.1 - Fuzzy Controller Structure

The basic structure of the fuzzy controller is shown in Figure 4.1 and consists of the following components.

4.1.1 - Fuzzification module

The fuzzification module has two functions: the first one performs a scale transformation or input normalization, which maps the physical values of the current process state variables into a normalized universe of discourse, i.e., a universe in which the domain values ranges between $[-1,1]$. This is not a mandatory function although there is evidence about the enormous advantages it brings when implemented as will be discussed later on this chapter. The second one performs the fuzzification, which converts the crisp image of the current state process into its fuzzy representation, in order to make it compatible with the fuzzy set representation of the process state variable in the rule antecedent.

4.1.2 - Knowledge base

This module together with the inference engine is the core of the fuzzy controller. It is composed of the data base and rule base. The basic function of the data base is to convey the necessary information for the proper functioning of the rule base, fuzzification and defuzzification module, which includes: the fuzzy sets that represent the meaning of linguistic values of the process variables and control output, and also the scaling factors for normalization and denormalization.

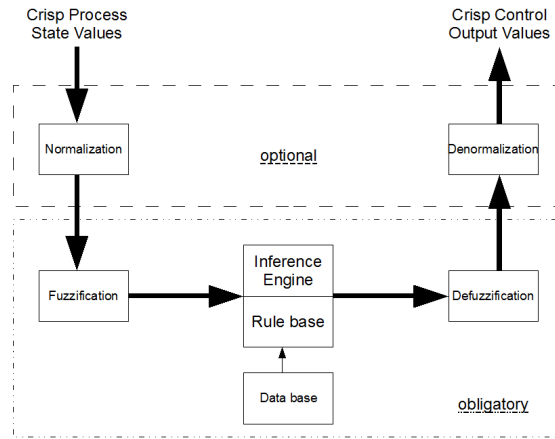


Figure 4.1 Fuzzy Controller Structure.

The rule base provide means to represent the structured knowledge which resembles the control policy of an experienced process operator or control engineer in the form of a set of fuzzy if-then statements or fuzzy rules.

4.1.3 - Inference Engine and Defuzzification module

The inference engine realizes the way the controller performs the deductive inference described in chapter 2. As mentioned before there are multiple ways to derive the meaning of a fuzzy rule that affect its outcome. Also the meaning of a set of rule can be computed from an individual or composite mode. The output of the fuzzification module is used by the inference engine to obtain the respective overall output fuzzy set for the control variables.

The functions of the defuzzification module are as follows: it performs the so called defuzzification that consists of finding an appropriate crisp value that describes the control output values. Finally, the denormalization function takes place, which maps the normalized control output into the respective physical domain. Like the normalization function, this is optional too.

There are numerous methods to perform the defuzzification on the literature. The ones that will be described here are the centroid and the middle-of-maxima, MOM, because these are actually implemented by the FEUP Fuzzy Tool. More information about this topic can be found in [8], [10] and [11].

The centroid procedure (also called center of area, center of gravity) is the most prevalent and physically appealing of all the defuzzification methods. It is given by the algebraic expression

$$u^* = \frac{\int_U \mu_U(u) \cdot u du}{\int_U \mu_U(u) du} . \quad (4.1)$$

The MOM method takes the average of the local maxima of the fuzzy sets. Formally

$$u^* = \frac{\inf_{u \in U} \{u \in U | \mu_U(u)=1\} + \sup_{u \in U} \{u \in U | \mu_U(u)=1\}}{2} . \quad (4.2)$$

4.2 - The Feup Fuzzy Tool

This section is devoted to briefly look at the implementation of the FEUP Fuzzy Tool and describe some added functionality in terms of fuzzy inference engine and rule base manipulation through the GUI.

4.2.1 - Why FPC/Lazarus as a programming tool

Lazarus [27] is an open-source, visual IDE (Integrated Development Environment) which provides a high degree of compatibility for Delphi and is available on a variety of platforms, including Windows, Mac OS X, and Linux. It is developed for and supported by the Free Pascal compiler which runs on many operating systems as well. It is designed to use and compile Object Pascal source code, which is an object-oriented superset of the Pascal programming language. Thus, the presented work kept the tool previously used.

Lazarus and Free Pascal philosophy aim to be write once, compile anywhere. As the same compiler is available for all of the above operating systems, there is no need for re-coding to produce identical products for different platforms, except when operating system-dependent features are used. As a visual, general-purpose programming tool, it supports the development of a wide variety of programs, including console applications, dynamically loadable libraries and GUI applications.

Lazarus is event-driven programmable, from which the flow of the program is controlled by handling events or messages from other programs or threads. This feature makes it perfect for automation purposes and soft real time control.

Lazarus supports components add-ons, one of special interest is the GLScene [28] component that allows easy access to Open GL features for ease of programming 3D graphics. The 3D plotter implemented on the FEUP Fuzzy Tool is based on the API provided by this library.

4.2.2 - The Fuzzy core engine

The UML class diagram of Figure 4.2 shows the architecture and data structures that implement the FEUP Fuzzy tool core fuzzy engine. Some of the classes were already implemented and some others were resulted from this work, each one of them will be described. This scheme binds together definitions such as fuzzy sets, linguistic variable, fuzzy rules, etc. in a hierarchical fashion through the concept of class. The following discussion will explain in some detail the relation between the classes and how they implement the concept of fuzzy controller. Pseudo code will be used as a convention to specify portions of code along the document when necessary.

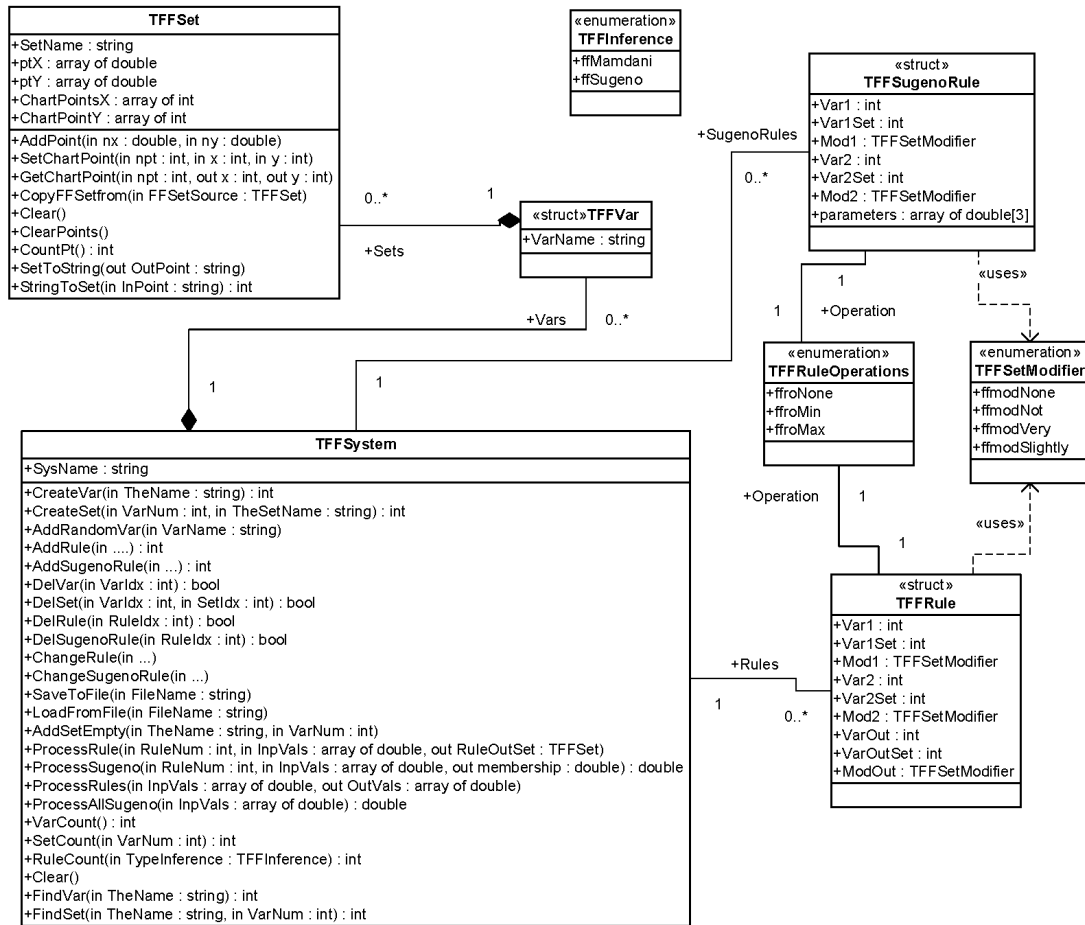


Figure 4.2 UML Class Diagram of the FEUP Fuzzy Tool

4.2.3 - Fuzzy sets

The class that represents a fuzzy set is the `TFFSet` that appear on the class diagram. Here a fuzzy set is defined as a piecewise linear function, coded as a set of data pairs named `ptX` and `ptY` implemented through dynamic arrays (dynamically sizable arrays) that represents samples values of the universe of discourse and its membership function respectively, i.e., $\mu(\text{ptX}) = \text{ptY}$. These two arrays are assumed to be incrementally ordered, and they serve as parameters to define the line segments that compose the membership functions. This implementation is optimized for speed computation when compared with other parameterized exponential-based functions. On [4] is stated that computing the membership degree with an exponential function takes about ten times longer than computing the membership degree for trapezoidal or triangular membership function. While the latter requires at most four comparisons, two subtractions and a division the former requires a call to the implementation of the exponential function.

Each fuzzy set has a linguistic label, coded through the `SetName` property of the class. The membership values for in-between samples are obtained by linear interpolation.

The methods of major relevance are the `AddPoint`, the `CopyFFsetfrom`, and the `CountPt` methods, whose action are quite intuitive to figure it out by their names. They offer means to manipulate and define the fuzzy set structure. The other ones are less interesting and their functionality is limited for graphical user interface purposes.

Linguistic hedges are implemented through some function provided by the `FeupFuzzyUnit` pascal unit of the FEUP Fuzzy Tool code that manipulates the specified set. The implemented linguistic hedges are listed in the enumeration type `TFFSetModifier` from the class diagram of Figure 4.2.

There are four types of defined modifiers as stated from the class diagram although the value `ffmodNone` only serve to indicate that the fuzzy set remain unchanged when it comes to evaluate fuzzy sets with linguistic hedges. The others modifiers act through the functions

- `FFNot (MyFFSet: TFFSet; FFSetOut: TFFSet);` for the “not” linguistic hedge.
- `FFVery(MyFFSet: TFFSet; FFSetOut: TFFSet);` for the “very” linguistic hedge.
- `FFLow (MyFFSet: TFFSet; FFSetOut: TFFSet);` for the “slightly” linguistic hedge.

These functions perform similar operations to those expressed in equations (2.30) to (2.33), with the exception of `FFNot()` function that performs the complement of the fuzzy set, equation (2.17). They receive as parameter two fuzzy sets. The first one is the set submitted to be modified, and the second parameter is the output fuzzy set resulting from the operation.

4.2.4 - Fuzzy variable

Fuzzy variables are implemented by the `TFFVar` data structure. It's composed also by a dynamic array of `TFFSet` class that represents all the fuzzy set that defines its linguistic values. As each linguistic value is described by a linguistic label, the set name can be viewed itself as the linguistic value.

An array is a group of elements that are accessed by indexing and sequentially organized in memory each set is uniquely determined by its array index within the fuzzy variable. Bearing this in mind, the set of linguistic values that the variable can take is embedded in the fuzzy set array. Also the universe of discourse can be obtained from this array. Evaluating the first element of the `ptX` field of each fuzzy set and taking the minimum of all this values, determine the lower limit of the universe of discourse. Likewise, taking the maximum of the last element of this field array for each fuzzy set define the upper bound. This is possible because of the restriction on each `ptX` array to be incrementally ordered.

The semantic function translate a linguistic value into the respective fuzzy set is implicitly defined with this implementation, because, as said before, the linguistic value and

the respective fuzzy set are in one to one correspondence through the `TFFSet` class' field `SetName`.

The last of the elements that are left to define for a fuzzy variable is its name that is coded through the `TFFVar` structure's field `VarName`.

4.2.5 - Fuzzification

The fuzzification module is implemented by the following function:

```
function FFFSetEvaluate (CrispInputX: double; MySet: TFFSet)...
return double;
```

This function accepts as parameters the crisp input value and the fuzzy set which defines the degree of membership. Given the piecewise definition of fuzzy sets two tasks are needed to compute the fuzzification operation. The first task consist on finding the `ptX` array index "i" such that `ptX[i]` is smaller than the crisp input value and `ptX[i+1]` is bigger. Then through the equation that defines the line segment parameterized by the pair of samples (`ptX[i]`, `ptY[i]`) and (`ptX[i+1]`, `ptY[i+1]`), the membership value for the crisp input value is computed and returned to the function caller.

4.2.6 - Fuzzy Rules

The next basic structure that needs to be defined is the fuzzy rule. As shown in the class diagram, this entity is encoded by a structured type called `TFFRule` and whose fields retain the necessary information. Each rule is restricted to the following scheme

```
IF <variable1> IS <set modifier> <fuzzy set> <LOGIC operator> <variable2> IS <set modifier> <fuzzy set>
THEN <variable3> IS <set modifier> <fuzzy set>.
```

They have at most two propositions (2 inputs) and one output assignment. For each variable it's also required a fuzzy set and a linguistic modifier. The logic operation done in the aggregation of the fuzzy antecedent has also to be defined. The implemented operators are the fuzzy intersection from equation (2.15) and the fuzzy union from equation (2.16).

One issue that may arise is the thought of this structure being a limitation for its application to a class of more complex system with multiple input/output. Contrary to this belief this structure does not pose any limitation. Rarely fuzzy controllers with more than two or three inputs are used. One reason lies in the fact that human perception is limited. In an everyday decision process we usually do not take into account more than two or three propositions, very rarely four, and almost never five or more at the same time. Since the main task of a fuzzy controller is the interpretation of heuristic knowledge provided by the operator, two or three inputs are usually enough to summarize the operator's comprehension. Another reason is that is that for a given distribution of fuzzy sets, the number of fuzzy control rules increases geometrically with the number of inputs. The geometrical progression of the number of rules becomes an obstacle for practical applications of multi-input fuzzy controllers [9].

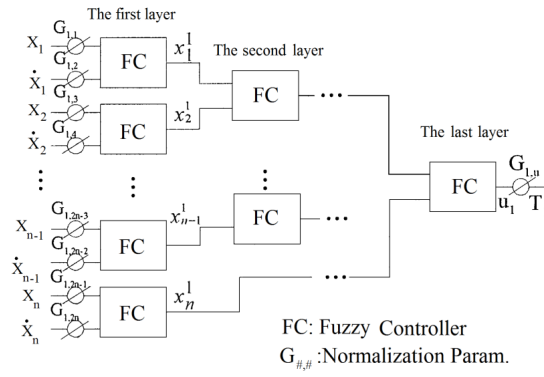


Figure 4.3 Layered composition of a MIMO fuzzy controller. Figure adapted from [31]

Fuzzy logic controllers for MIMO systems are realized in practice through the combination of elementary dual-input-single-output controller whose structure is similar to the aforementioned fuzzy rule structure. The difference between MIMO systems control and single-input single-output (SISO) systems control is based on the compensation of the process interaction among each degree of freedom. It is obvious that the difficulty of MIMO systems control is how to overcome the coupling effects among each degree of freedom. To obtain good performance, coupling effect cannot be neglected. Reference [31] proposes a detailed analysis, and design guidelines to the conception of this type of controller, including stability issues. Also [32] follows a similar procedure to conceive a MIMO controller. The design procedure of the fuzzy control strategy is used to control each degree of freedom of this MIMO system individually. Then, an appropriate coupling fuzzy controller is designed to compensate for the coupling effects of system dynamics among each degree of freedom. The essence of this discussion is resumed on Figure 4.3 that shows the common structure for a MIMO controller.

The reference to each of these entities is made through integer values. The reason behind this is that inside a variable, a fuzzy set is uniquely determined by the index which represents its position at the fuzzy set array. Furthermore, the organization of a fuzzy variable inside the controller follows the same principle, thus every variable is also determined by a unique integer. More on this subject will be discussed next. The set modifier and the logic operator that binds the antecedent fuzzy proposition are identified by their respective data types.

4.2.7 - The TFFSystem and the inference engine

The previous classes and data types serve as a blueprint to the implementation of the fuzzy controller. This is assembled by the `TFFSystem` class that integrates all the previous entities to provide the control action for which is intended. As mentioned before, a dynamic array of fuzzy variables is included to identify the process state variables and control variables. This keeps an organized reference where each variable is identified by its position on the array, although it is not mandatory to know this index, as this is retrieved by class'

method `FindVar()`, by specifying the name by which the variable is known. In case no variable under the given name is found, the method returns -1. This notifies the inexistence of such variable as the indexes of a dynamic array in Pascal has zero as a lower bound. In the same faction, the index of a fuzzy set within the variable's set array can be obtained through the method `FindSet()`, although the integer value that represent the variable must be known prior to the method invocation as is passed as a parameter. The methods included in this class linked to the user interface were based in the counting of the variables and rules and the saving and loading from file of the entire system.

The creation and deletion of variables and fuzzy sets (that belongs to particular variable as specified by the composite association on the class diagram) is supported by a set of methods as well. These methods can be easily identified by its definition from the operations that comprise the `TFFSystem` class, as they are quite intuitive. The controller's rule base is conceived also through dynamic arrays of the `TFFRule` type as it appears to be the most natural structure. Furthermore, there are implemented methods that aid and support rule base manipulation (modification, creation, deletion of fuzzy rules etc.).

Variable and fuzzy set deletion must be carefully implemented. Note that fuzzy set and variables can change (even emptied) but correct deletion means that rules must change to be kept up to date. Variable or fuzzy set deletion is not very frequent and almost unnecessary because generally the used resources are modest.

Rule processing is always done by the definition, by calculating the inference that includes fuzzifying, calculating propositions, using aggregation and inference, accumulating results and defuzzifying output variables. The inference engine is conceive as an individual rule based inference engine with the Mamdani's type of implication, and has the two following procedures that implement this functionality.

- `procedure ProcessRule(RuleNum: integer; InpVals: array of double; RuleOutSet : TFFSet);`
- `procedure ProcessRules(InpVals : array of double; OutVals: array of double);`

The first method computes the inference result of a single rule. It receives as a parameter an array that contains the current value of all the process state variables and is equally organized as the `TFFVar` array from the `TFFSystem` class. That is, the index "i" represents the same variable on both arrays. Also an integer value is received by parameter and represent the index of the rule we which to evaluate from the rule base. These two parameters contain all the information needed in order to process the rule inference. The resulting fuzzy set is provided by the third parameter as it is assumed to be passed by reference.

The second method simply uses the previous method to compute the result of every available rule at the rule table and then aggregate each resulting fuzzy set. Note that the

rule table may contain different input and output variables at different rules. This follows from the previous discussion of creating a MIMO controller as a collection of DISO fuzzy controllers. Bearing this in mind, the aggregation process may deliver various fuzzy sets as output that result from a partial set of rules that applies only to a portion to the MIMO controller. These sets are defuzzified (see the next section) and the result of this process is placed on the output array (second parameter of the method) at the position corresponding to the appropriate control variable. Once again, this array is equally organized as the `TFFVar` array.

4.2.8 - Defuzzification methods

FEUP Fuzzy Tool offer two methods for defuzzification: the centroid and MOM as was mentioned before. They are realized through the following functions

- `function FFDefuzCentroid(FFSet: TFFSet)... return double;`
- `function FFDefuzMOM(FFSet:TFFSet)... return double;`

that basically implements the computation of equations (4.1) and (4.2) respectively.

4.3 - A new inference model

The Sugeno fuzzy model (also known as the TSK fuzzy model) was proposed by Takagi, Sugeno, and Kang [33] in an effort to develop a systematic approach to generating fuzzy rules from a given input-output data set. A typical fuzzy rule in a Sugeno fuzzy model has the form

$$\text{if } x \text{ is } A \text{ and } y \text{ is } B \text{ then } z = f(x, y),$$

where A and B are fuzzy sets in the antecedent, while $z = f(x, y)$ is a crisp function in the consequent. Usually $f(x, y)$ is a polynomial in the input variables x and y , but it can be any function as long as it can appropriately describe the output of the model within the fuzzy region specified by the antecedent of the rule. When $f(x, y)$ is a first-order polynomial, the resulting fuzzy inference system is called a first-order Sugeno fuzzy model, which was originally proposed in [33].

The fuzzy reasoning procedure for a first-order Sugeno fuzzy model is shown at Figure 4.4. Since each rule has a crisp output, the overall output is obtained via weighted average, thus avoiding the time-consuming process of defuzzification required in a Mamdani model.

Unlike the Mamdani fuzzy model, the Sugeno fuzzy model cannot follow the compositional rule of inference strictly in its fuzzy reasoning mechanism.

Without the time-consuming and mathematically intractable defuzzification operation, the Sugeno fuzzy model is by far the most popular candidate for sample data- based fuzzy modeling. Since FEUP FUZZY TOOL has no such capacity to adapt or model a system through data acquired, it seems thoughtful to introduce this inference model that will expand the capacities of the software and will be even more useful with the implementation of fully compatible self-learning fuzzy archetypes like the ANFIS network.

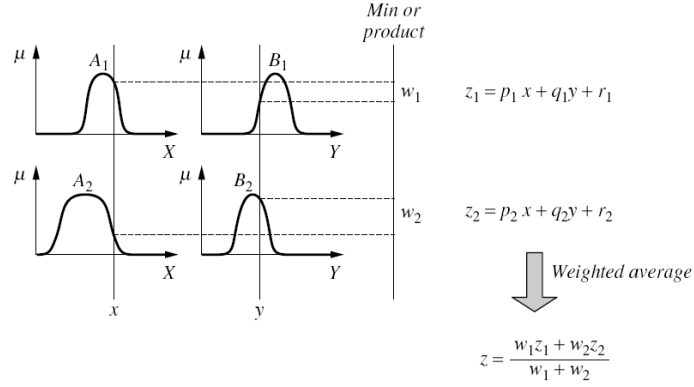


Figure 4.4 Graphical Inference of a two rule fuzzy system of the Sugeno model. Adapted from [10].

The requirements to implement this type of inference are the following. First of all, an appropriate data structure to represent the type of rule for this type of inference becomes a priority since the implemented `TFFRule` structure does not provide support. The data structure `TFFSugeno` rule will fulfill this need and its definition is shown also on the class diagram of Figure 4.2. Similar to the `TFFRule` the antecedent of the rule is represented in the same fashion, with at most two variables (without loss of generality) each one with an associated fuzzy set that can be affected by a linguistic modifier. The rule structure resembles the first-order Sugeno fuzzy model:

IF <variable1> IS <set modifier> <fuzzy set> <OPERATOR> <variable2> IS <set modifier> <fuzzy set>
 THEN <variable1><parameter[0]> + <variable2><parameter[1]> + <parameter[3]>

A fixed size array is used to store the consequent parameters. The first two elements represent the first order coefficient of the polynomial function, and the last one is the zeroth order coefficient. With the definition of this structure, the only thing that is left to implement is the inference engine. Inspired on the already implemented Mamdani's inference engine two functions will perform this functionality.

The evaluation of a single rule is carried out by the procedure `ProcessSugeno` from the `TFFSystem` class. The function first performs the fuzzification of the crisp input value corresponding to the value given by the `InpVals` array. The index of this position is specified by the integer stored on the Sugeno rule structure which represents the first variable on the rule antecedent. Then the value is affected appropriately by the corresponding set modifier operation. The same process is carried out for the second variable.

The logic operator with aggregate both result and deliver the membership value of the antecedent that will propagate through the rule consequent. Two things should be noted. The first is that many of the tuning algorithms are derivative-based optimization methods, and in order to be able to use them the logic operators must be derivable with respect to all its parameters. T-norms and S-norms operators others than those previously used maximum and minimum operators must be used to satisfy these requirements. Therefore the algebraic product and sum, defined by

$$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x) \quad (4.3)$$

$$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x) \quad (4.4)$$

are used for the intersection and union operations respectively.

The other detail that must be considered is that the computed membership value of the antecedent will be used again to compute the weighted average, so it must be preserved. The third function parameter named membership is used for this purpose and is passed inside the body of the function by reference. To finalize the computation of the first order polynomial function that describes the rule consequent take place and it's affected by the propagated membership degree of the antecedent part of the rule. This value is returned to the function caller.

To evaluate the rule base entirely the function `ProcessAllSugeno`. The function perform the following computations:

```
function ProcessAllSugeno(InpVals : array of real)
var
  i: integer;
  total, membership: float;
begin
  total:=0;
  result:=0;
  if RuleCount(ffSugeno)>0 then begin
    for i:=low(SugenoRules) to High(SugenoRules) do begin
      result:=result + ProcessSugeno(i,InpVals, membership);
      total:= total + membership;
    end;
    result:=result/total;
  end;
end;
```

It will simply go over the entire Sugeno's rule base and evaluate every rule, accumulation the pre-computed individual results to perform the weighted average and finally return the overall result.

The `LoadFromFile` and `SaveToFile` methods of the `TFFSystem` class suffer minor modifications due to this addition because they must offer support for loading and saving of Sugeno based controllers. This functions read, as well as write, from the (.ffsys) file the parameters of the Sugeno's rule base, that is the parameters that define rule by rule as they do it for the Mamdani's rule base. The .ffsys file is essentially an ASCII file where the most important parameters of variables, sets and rules are stored in order to support the controller design through GUI. The file is constructed in a human understandable format. The loading function simply parses this file and fetches the most essential features of system parameters.

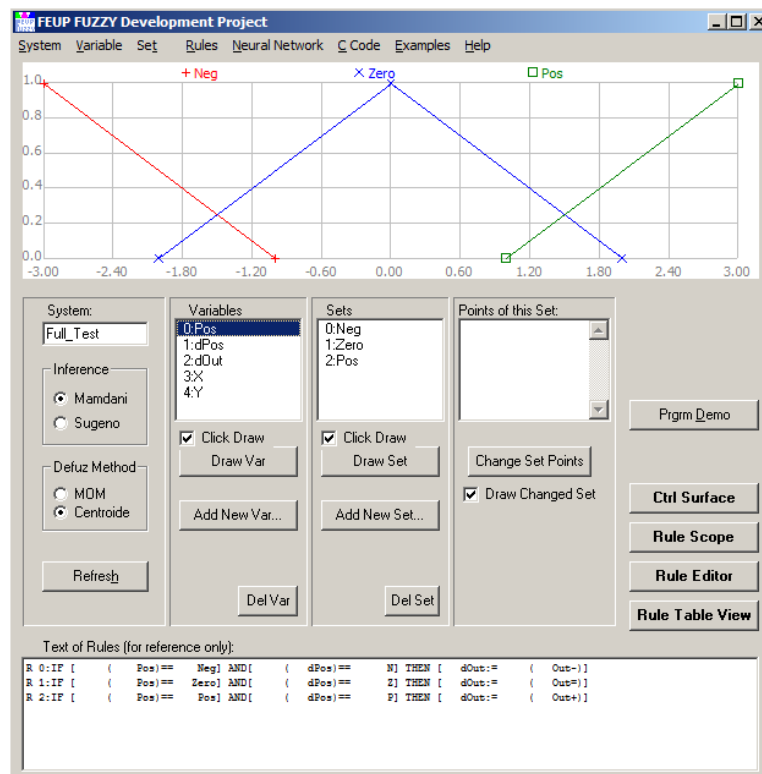


Figure 4.5 The GUI Main Form

4.4 - The Graphical User Interface (GUI)

The Graphical User Interface (GUI) is intended to provide assisted construction for the fuzzy system with sets, variables and rules. Each structure can be loaded and saved separately as well as the whole system can be loaded/saved all at once both by use of program calls, also accessible by GUI commands/menus. A view of the implemented GUI may be seen at Figure 4.5.

Fuzzy sets and variables can be drawn on the screen and edited by simply dragging set points. Changed points are reflected in the Memo Box that lists the fuzzy points, these points can also be hand edited.

The Main windows suffer few modifications with the addition of the Sugeno model of inference. The user must have the ability to choose and explore both types of inference models and be able to modify their respective rule bases as desired. This choice is made on the first panel, at the radio box titled "Inference", see Figure 4.5.

4.4.1 - Rule Scope

The Rule Scope is a feature which provides users a display of the fuzzy inference process for a selected rule for debugging and educational use. As seen at Figure 4.6 there are five small plotters on the Rule Scope's form.

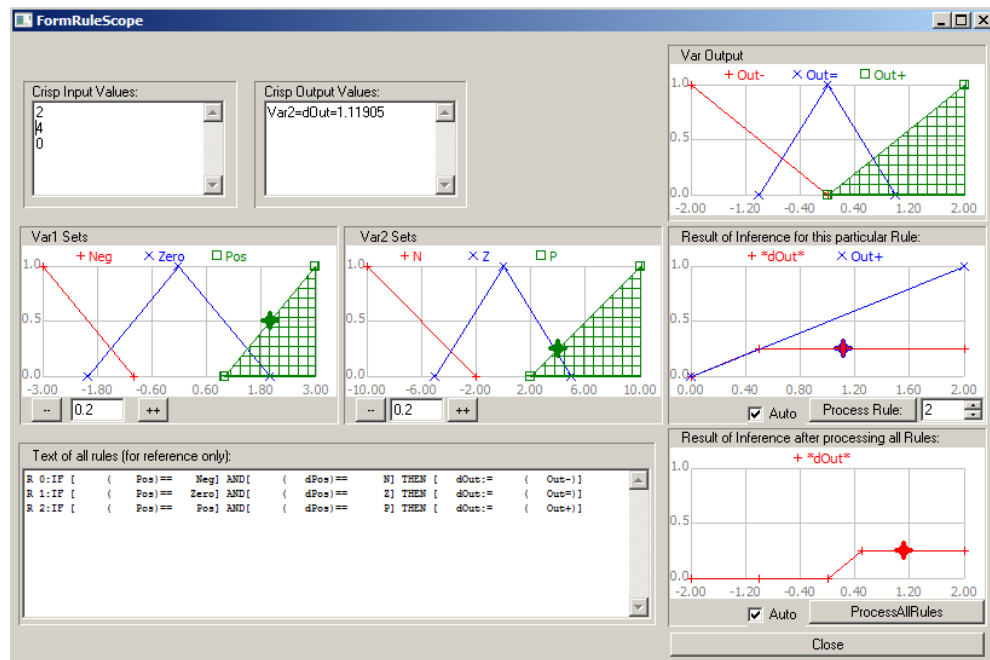


Figure 4.6 A picture of the Rule Scope form

The first two plots on the three plots row are the two input variables of the selected rule. Within these plots, the respective membership function is shaded and representing the fuzzification result is a bold cross on the set. On the third plot one may see the set representing the Implication result for that Rule and above it the output variable's plot. Last plot represents the resultant Set of Accumulation. Rule Scope is also equipped with buttons and text input fields which may change rule's inputs values, and text output fields displaying defuzzification results. On this form, the user is also able to modify Sets points 'online', by clicking and dragging them on each plot.

This form is only available for Mamdani's based inference. The reason behind this is that the main purpose is to analyze and fine tune the Mamdani fuzzy controller by modifying set's definition etc. On the other hand the Sugeno model is best suited to model system through data and there are a series of optimization algorithms that can be used to optimize this controller, (see Chapter 6). To analyze the outcome of a Sugeno controller the 3D plotter or control surface form will suffice.

4.4.2 - Rule Editor

The rule editor offers a generic way to define the rule base, see Figure 4.7. Originally, only support manipulation of the Mamdani controller's rule base, but now is perfectly generic and support both types of controller. It has graphic components whose function is to perform the selection of variables, sets, operators, and modifiers. It also contains an explicit text box to observe the current state of the rule base and modify any existing rule. The left-hand side is used to specify the rule antecedent. The upper right corner is used to consequents for the Sugeno's type of rule and this section is disabled when the selected controller is of Mamdani's

The Rule GUI window is divided into several sections. The top section contains two rule editors, Sugeno Fuzzy Model and Mamdani Fuzzy Model. The Sugeno Fuzzy Model section has a 'Then:' field with a formula $var1 * p + var2 * q + r$ and three input fields. The Mamdani Fuzzy Model section has a 'Then:' field with a formula $var1 * p + var2 * q + r$ and three input fields. Below these are two more rule editors, Var1 and Var2, each with a 'Modifier' and 'Set' dropdown menu and a small plot showing membership functions. The bottom section contains a list of rules, with two rules visible: Rule 0 and Rule 1. Rule 0 is: IF [(Pos) == Neg] AND [(dPos) == N] THEN [dOut := (Out-)]. Rule 1 is: IF [(Pos) == Zero] AND [(dPos) == Z] THEN [dOut := (Out=)].

Figure 4.7 The Rule Editor form

type. Likewise the panel below offer means to support the consequent edition for a Mamdani's model based fuzzy system. It also offers support for the correct manipulation of the rule system, inhibiting incorrect rule production or not allowed user actions, easing the learning process on the software utilization.

4.4.3 - The control surface

The control surface is able to plot the transfer function of the controller. To easily render and manipulate (rotate, zoom, etc) the 3D surface, the GLScene [28] graphic add-on for Lazarus is used. Many fuzzy control systems require more than two inputs or more than one output (generic MIMO system). Accordingly, the Surface Control Form in Delphi's Application is equipped with drop box lists and text input fields which allow users select any two inputs and any one output for plotting as well as defining values to fixed inputs. Users are also able to define the two inputs' plotting limits, surface resolution. These parameter influence surface calculation time. Surface calculation is done by a GLScene callback that uses the TFFSystem methods to compute the system response for each input combination. Drawing functions are handled by the GLScene wrapper over the Open GL routines that interface the graphic card directly (if the card has 3D). This results in very fast drawings and manipulations of the control surface. If any changes on the controller parameters are made (for example on set support, type of membership function etc.), they are automatically reflected on the 3D plot. Minor changes were made in order to be able to plot the correct transfer function for a specific rule base according to the user selection on the inference style.

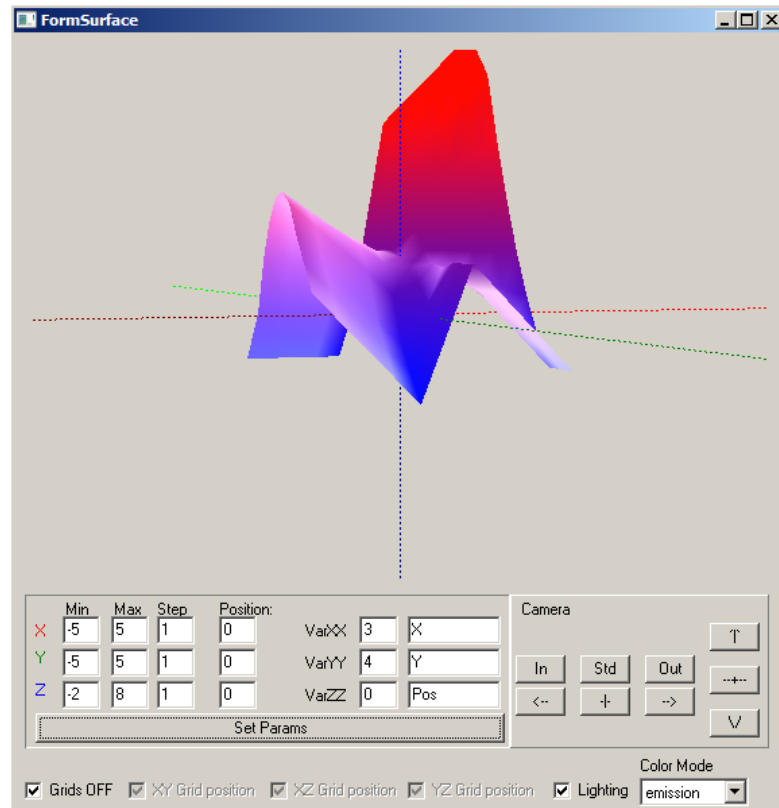


Figure 4.8 The Control surface 3D plotter. It shows the transfer function of a Sugeno controller.

Figure 4.8 shows an example of the surface generation for the Sugeno inference model defined by the rules

$$\left\{ \begin{array}{l} \text{If } X \text{ is small and } Y \text{ is small then } z = -x + y + 1, \\ \text{If } X \text{ is small and } Y \text{ is large then } z = -y + 3, \\ \text{If } X \text{ is large and } Y \text{ is small then } z = -x + 3, \\ \text{If } X \text{ is large and } Y \text{ is large then } z = x + y + 2. \end{array} \right.$$

The surface is complex, but it is still obvious that the surface is comprised of four planes, each of which is specified by the output function of each of the four rules. Figure 4.8 shows that there is a smooth transition between the four output planes.

4.4.4 - Fuzzy Associative Memory (FAM)

Also known as fuzzy rule table format, this functionality was added to the GUI and provides a particular way to interact and manipulate the Mamdani rule base suitable for Dual-Input-Single-Output (DISO) controllers, which are the most frequently used [9]. The aspect of the fuzzy rule table can be seen at Figure 4.9. Every rule in the fuzzy rule table is represented by an output fuzzy set engaged in the THEN part of the rule. The rule position within the fuzzy rule table is determined by coordinates of inputs fuzzy sets engaged in the IF part of the rule. This format provides a straight insight to the rule table and intrinsically has the advantage to create a rule base that satisfy the property of consistency, that is, automatically eliminates the creation of contradictory rules.

e/de	NL	NM	NS	Z	PS	PM	PB
NL	NL	NL	NL	NL	NM	NS	Z
NM	NL	NL	NL	NM	NS	Z	PS
NS	NL	NL	NM	NS	Z	PS	PM
Z	NL	NM	NS	Z	PS	PM	PL
PS	NM	NS	Z	PS	PM	PL	PL
PM	NS	Z	PS	PM	PL	PL	PL
PB	Z	PS	PM	PL	PL	PL	PL

Fuzzy Relation Operator

☒ AND

☐ OR

Clear all

Close

Figure 4.9 Table representation of the rule base. This representation is also called fuzzy associative memory.

The continuity and completeness properties are only satisfied under certain conditions, see [8] and [9]. The type of fuzzy aggregation operator that links both propositions in the antecedent part of the fuzzy rule can be selected as the AND/OR operators on the radio box titled “Fuzzy relation operator”.

When accessing this form and prior to the OnShow() form event, the user is asked to specify the intended variables, inputs and output respectively, and the kind of logic operator that links the antecedent proposition in order to populate the rule table appropriately, if any rule exists for the given variable combination. Therefore, the table structure is specified, as well as constructed, at run time. At this particular time, rules that match the specified variables and operator are presented to the user in this table format. After this, the user has control over the table and can use it to interact with the rule base. In order to manipulate rules, a drop-box is placed over the selected cell as the user makes his choice. The content is the collection of fuzzy sets defined of the defined output variable. At the end of this process, previous rules that are already part of the rule base are modified or deleted in accordance of what have been introduced on the fuzzy rule table, and new rules are created for those cells that do not have prior correspondence on the rule base.

The fuzzy rule table may be viewed as a partitioned phase plane formed by the two plant control variables. The controller will be composed of a set of control rules with two antecedents and one consequent. The antecedents of each fuzzy rule describe a fuzzy region in the state-space. By so doing one effectively quantizes an otherwise continuous state-space so that it is covered by a finite number of such regions (and consequently fuzzy rules).

There is a specific fuzzy action associated with each such fuzzy rule (fuzzy region). During the decision making process (control process) each point in the state-space is differently

affected by the actions associated with all the fuzzy regions in whose footprint the point falls. The fuzzy aggregation rule and the defuzzification method then yield a specific action for that point. As the point moves, the action also changes smoothly. This means that an effectively quantized representation of the state-space nevertheless yields a smooth action surface over the state-space. As a result of this mapping every trajectory in the phase plane has a matching control action. A fuzzy rule table viewed as a phase plane is frequently used for heuristic assessment of closed-loop system stability, as it offers an elegant way to investigate the influence of individual control rules (their THEN parts) on the shape of phase trajectories [9].

In effect given only the actions associated with the centre point of overlapping rectangular regions of the state-space, fuzzy logic algorithm provides the action for any individual point in the state-space. Clearly all the other values are interpolated from the available finite sets of values by a method which depends upon the choice of the mathematical expression used for the various fuzzy logic connectives (AND, OR, ELSE etc.) as well as the choice of the defuzzification method. The result is that the control can be specified compactly with as few rules as possible.

4.5 - Summary

This chapter reviews the basic fuzzy controller architecture, which is composed by a fuzzification module, a knowledge base, an inference engine and a defuzzification module. The function that performs each module was briefly described. Following this discussion a description of the implemented fuzzy core engine on the FEUP FUZZY TOOL was covered, highlighting the fundamental data structures, their mutual relation, functions and methods implemented, and how with these structures the concept of fuzzy controller was attained. The software is programmed under the object oriented paradigm thus the concepts of class, data abstraction, encapsulation, modularity, polymorphism, and inheritance are of particular interest. A brief discussion about how a MIMO controller is assembled through a layered composition of simpler fuzzy controllers was also of particular interest as there lays the foundation for the choice of implementation of the rule base on the FEUP Fuzzy Tool the way it was conceived. A new model of inference was added to the FEUP Fuzzy Tool, the Takagi-Sugeno inference model. The details of its implementation were presented.

Finally, an overview of some parts of the GUI was covered along the chapter, explaining they usage and what are they for. The rest of the features on both API and GUI will be covered later on the following chapters.

Chapter 5

Producing Embedded Controllers

This section looks at the implementation of fuzzy controllers for embedded systems. It is one of the principal and most important added functionalities to the FEUP Fuzzy Tool.

5.1 - Fuzzy logic hardware

The implementation of analog and digital embedded fuzzy controllers is in contrast with each other. As a result, there is a variety of analog, digital, and hybrid approaches for fuzzy controller architectures at the designer's disposal for solving both simple and very complex control problems.

Some controller structures employ analog circuitry only for input and output conversion, while fuzzy processing is done digitally with general-purpose microprocessors or application-specific integrated circuits. Other structures, such as the complementary metal oxide semiconductors (CMOS) analog fuzzy controller structure developed by Prof. Yamakawa exploits the functional capabilities of the MOS transistor to implement the fuzzy operators with very simple circuitry [34].

As might be expected, hardware implementations are generally orders of magnitude faster than software implementations. The reason for this is that fuzzy software controllers process information in a sequential manner, whereas fuzzy hardware processing is done in parallel. Hardware implementations can accomplish higher speed processing, are more compact, and are generally less expensive. Analog fuzzy controllers also have drawback. They are characterized by limited precision and susceptibility to the drifting characteristics of its constituents at one hand, and weak structural flexibility on the other. The former can be overcome at some level by a careful circuitry design, but larger flexibility and programmability can be achieved only with a fully digital implementation.

One approach to realizing embedded fuzzy control is through the use of conventional microprocessors. There are a number of tools that are able to create microcontrollers

compatible code, as was mentioned in Chapter 3. The Motorola Corporation, for instance, has developed software for fuzzy logic inference processing, and a development tool called Knowledge Base Generator (KBG). KBG translates rules and membership functions from natural language into data structures that can be understood by Motorola's conventional microcontroller units (MCU).

There are also fuzzy logic dedicated processors [35]. The architecture and functionality of fuzzy processors are very similar to those of standard microprocessors. The only difference is in enhancement of the processor core with additional digital circuits and additional instructions for execution of main fuzzy controller operations.

The significant increase in operating speed and ability to process a large number of inputs, outputs, and fuzzy control rules can be accomplished by programming fuzzy controller operations in programmable logic arrays (PLA) and field programmable gate arrays (FPGA).

5.2 - Microcontroller-based implementation

As mentioned before, most of the embedded applications are developed using a software emulation of fuzzy algorithms running on standard microcontrollers. The management of the typical Fuzzy Logic data structure is crucial, when using conventional microcontrollers, i.e. designed to implement non fuzzy algorithms. Furthermore, when running on a standard microcontroller, the fuzzy algorithm needs a big data structure to be stored on RAM (a large memory size involves the use of a medium or high cost microcontroller) and important computational effort that leads to a lengthy operation.

For a standard 8- or 16-bit microcomputer-based digital DISO fuzzy controller that it is necessary to implement performs the following basic operations:

1. Analog-to-digital (A/D) conversion of controller inputs.
2. The input signal fuzzification.
3. Scanning of active fuzzy control rules.
4. Fuzzy inference.
5. Defuzzification, that is, computation of the crisp controller output from the resulting output fuzzy set.
6. Digital-to-analog (D/A) conversion of the controller output.

The critical aspects of lies between from function 2 to 6 on the previous list. Normally the implementation of fuzzy logic demands the application of vector and matrixes that represent discrete fuzzy sets relations (see Chapter 2). This scenario is best suited for the implementation of the compositional rule of inference which a pre-computed matrix that represents the meaning of the whole rule base and the inference is made through the implementation of equation (2.37). The problem with this solution lies with the computational complexity (in both processing time and memory requirements) which increases in a proportional manner with the system dimension.

This section describes a solution to this issue and provides a methodology to implement embedded fuzzy controllers on standard microcontrollers.

High level language programming of popular 8- and 16-bit microcontrollers has become a common practice. Though advantages of high level language programming are unquestionable, programming in the microcontroller assembly language can be even more advantageous when issues like processing of a large number of fuzzy control rules as well as switching between coarse and fine control regimes are considered. Nonetheless, assembly language is highly specific for a MCU family, and thus for generality sake, the widely used high level language C is the choice for implementing the FEUP Fuzzy Tool for embedded systems.

As mentioned before, the fuzzy controller performs a static non-linear mapping between the input and outputs. Bearing this in mind, the control surface describes all the essence of the fuzzy controller, therefore porting this characteristic to an embedded system enable us to emulate a fuzzy controller for a particular system. This is the foundation upon which is built the approach to generate C code for microcontrollers.

The rule base is intentionally built to keep the relations between with at most two variables in an antecedent that produce a unique consequence, and as discussed before this represent a natural way to built MIMO controllers as a layered structure of DISO or SISO controllers. With this in mind, it is natural to use this practice for the embedded application too.

For the most complex case in which the rule has a two variables in its antecedent fuzzy transfer function of the controller is a surface on a three dimensional state space, see Figure 5.1. Discretizing or sampling this surface with a given resolution on each state variable one is able to encode the resulting sampled surface on a matrix structure.

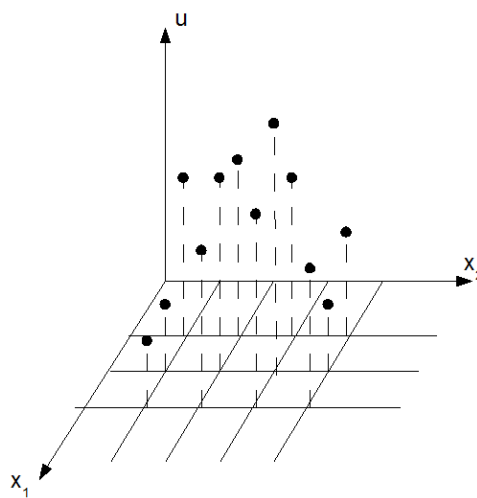


Figure 5.1 Illustration of the sampling of a control surface.

The intersection of a given row and column gives the respective control output for a given pair of samples of the input state variables. Values that do not correspond to a given sampled are interpolated. When a given set of rules have only one variable on their antecedent, the same reasoning applies and the structure that holds the sampled values is a vector.

5.2.1 - Aitken-Neville interpolation formula and the dual interpolation

The Aitken-Neville form allows obtaining the interpolating polynomial at a given point in a recursive fashion, considering successive interpolation nodes and their respective interpolation values.

Let m be an integer between 0 and n , and k another integer between 0 and $n-m$ and define $p_{m,k}$ as the polynomial function, with degree less or equal than k , that interpolates the values $(y_i)_{i=m..m+k}$ on the nodes $(x_i)_{i=m..m+k}$. The polynomial $p_{m,k+1}$ is constructed from the polynomials $p_{m,k}$ and $p_{m+1,k}$ by the recursive expression

$$p_{m,k+1}(x) = \frac{(x-x_{m+k+1}) \cdot p_{m,k}(x) + (x_m - x) \cdot p_{m+1,k}(x)}{x_m - x_{m+k+1}} \quad (5.1)$$

Let $z_{ij}=f(x_i, y_j)$ be the known values of a function $f = \mathbb{R}^2 \rightarrow \mathbb{R}$, where $(x_i)_{i=0..n}$ and $(y_j)_{j=0..m}$ are all distinct and it's desired to obtain an approximation of this function. This is an interpolation problem in \mathbb{R}^2 that can be solved by using interpolation on \mathbb{R} . To do this, method known by dual interpolation is used. It is based on performing polynomial interpolation with one variable at the time. One way to accomplish this task is by first interpolating on x , obtaining for each j a polynomial p_j interpolating the values $(z_{ij})_{i=0..n}$ on the nodes $(x_i)_{i=0..n}$. Then, it is obtained the polynomial that interpolate the values $p_j(x)$ on the values $(y_j)_{j=0..m}$ that gives the approximation of the function $f(x, y)$ for each (x,y) pair.

With this recursive formula and the notion of dual interpolation, it's possible to use a polynomial function of any degree to approximate the any value of the control surface. Splines interpolation can also be used to obtain these values but such implementation would consume additional computing power and limit its application as they require MCU that sustains such computational burden.

Fuzzy Control Surfaces are generally smooth. The requirement is that for each crisp value of a linguistic variable there is matched with at least one fuzzy set defined over its universe of discourse. This brings about the condition of adjacency of fuzzy sets. In other words, there must be enough overlap between fuzzy sets in order to guaranty the smoothness of the control surface [8], [9]. On reference [36] is theoretically proven that for SISO and DISO fuzzy controller with linearly distributed fuzzy partition over the state space will result on a smooth control surface. With a smooth surface the computational burden of finding a higher degree interpolating polynomial between samples is eased. Also having the possibility to define the resolution on both sampling variables brings about the ability to specify a state space partition suited for linear interpolation between samples as the smoothness of the control surface implies soft rate of change in a given interval.

5.2.2 - Fuzzy controller coding

Retaking the previous discussion about the implementation of a efficient fuzzy controller on a microcontroller system, we know now that the control surface can be approximated as much as desired; been a compromise between the memory available, the computational capacity, and the real time constraints intrinsic from the controlled system.

The FEUP Fuzzy Tool is able to export in a c-coded format any controller by delivering a set of header files and its corresponding c-coded file. These files contain the sampled function embedded in a multidimensional array (matrix) which represents a look-up table. This look up table is hard coded or embedded into the source code of its look-up function as its local variable. For MIMO controllers (composed of many layered SISO and DISO controllers) this implies the existence of various functions that performs similar task with the only difference in the look-up table and its intrinsic parameters. It is of course desirable to create a unique look-up general function that performs this task, given the look-up table as a parameter. But one of the restriction that place the c language is that when a multidimensional array is passed as a parameter to a function, the dimension of this structure must be known *a-priori*. This poses a problem because of the many parameters involved in the creation of such data structure that directly or indirectly affects its dimension (like the sampling resolution, the universe of discourse for a given variable), thus resulting in a variable size and context-dependent data structure. One solution could be the usage of pointers (or nested pointer if we are dealing with multidimensional arrays), but this implies loading the data structure at run-time to the RAM memory and also the storage of those values at the program memory resulting in an excessive waste of memory space. The problem increases drastically if it is desired to use floating point arithmetic with double precision.

Thus in order to keep the speed of computation as the primary and most important requirement, a particular MIMO controller will consist of a series of basic fuzzy controller each one will derive a particular look-up function with the sampled surface hard coded on it.

Let's now take a look at the implementation of the look-up function. The first thing that must be noted is that given the different requirement of interpolating a $f = \mathbb{R} \rightarrow \mathbb{R}$ function (only linear interpolation) with a $f = \mathbb{R}^2 \rightarrow \mathbb{R}$ function (dual interpolation) the implementation is also separated although the first case is a particular and simpler problem of the second case. From now on, attention is devoted only to the general case of the sampled function been stored in a bi-dimensional array.

The dimension of rows and columns of the matrix that embodies the sampled surface represents the quantity of samples with respect of a given variable. For example, if the row dimension is thirty two, and given the sampling resolution and the universe of discourse, or domain, of that variable, the quantization resulted in thirty two samples along that direction of the state space. Given two crisp values for those variables, say (x,y), a useful question that

arises: what are the indexes of those samples that must be used to perform the interpolation and deliver the approximate value of the control surface.

In the C programming language all arrays are indexed starting from zero. So it is first considered the case where the domain of the variable is any interval comprised between $[0, +\infty)$. In this case the nearest smallest sample index of an input value x is given by

$$index = \left\lfloor x / sampling_resolution \right\rfloor. \quad (5.2)$$

Here we used the floor function that returns the nearest returns the largest integer smaller than or equal to the argument value. This will return a value always greater than or equal to zero.

If the domain is extended to include negative values, i.e., any interval comprised between $[\delta, +\infty)$, $\delta < 0 \wedge \delta \in \mathbb{N}$, then the nearest smallest sample index of an input value x is given by

$$index = \left\lfloor x / sampling_resolution \right\rfloor + \delta. \quad (5.3)$$

The quantity δ represents the number of samples that are contained between the sample representing the origin and that representing the lower bound of the variable's domain. Obviously this quantity must be an integer and for that to be true, two properties must be satisfied. The first is that the domain's lower bound must be an integer. The reasons will become clear shortly, and thus the domain of a variable for sampling purposes will always be adjusted to be bounded by integer values. Also the sampling resolution must be an integral part of the unity, that is, the quotient between the unity and the sampling resolution must be an integer value. This way the origin is always included as one of the samples, because if the domain is limited by integer numbers there is always an integer number of samples between the origin and the domain's lower bound that defines the offset quantity δ used for the index computation. Even if the user defines a sampling resolution or a variable domain that do not fulfill these properties the process of automatic generation of programming code takes care of these aspects by itself.

The reasoning used before for a single variable must be used in the same way in order to get the row and column index for a given input values. This way all the samples values needed to perform the interpolation are defined. Name "i" and "j" the respective indexes, that samples accessed by the indexes pair (i,j) , $(i+1,j)$, $(i,j+1)$, $(i+1,j+1)$ give the necessary values to perform dual linear interpolation for a given sample and obtain the approximation of the control output. It should be noted that whether the obtained indexes exceed the matrix dimension, they must be adjusted to the lower/upper bounds that defines the matrix dimension. Also the computational complexity is very low, because the control computations are composed by very few comparisons, memory fetching instructions, addition and multiplications when compared with the lengthy operation of matrix computations including fuzzy set algebra, the fuzzification, inference and defuzzification.

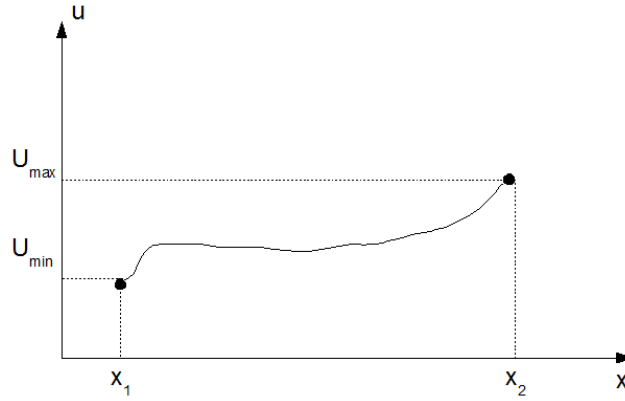


Figure 5.2 A simplified one-dimensional representation of a possible nonlinear control law.

As was mentioned before, the storage of floating point numbers with double precision demands large amounts of memory capacity and for some applications the represents and increased cost or some kind of obstacle that are not justifiable. To circumvent this problem FEUP Fuzzy Tool provides an option which makes possible to encode the samples stored into the look table as integers represented on a single byte, reducing by a factor of eight the memory requirements.

Within the FEUP Fuzzy Tool the user is able to specify whether the sampled values come into the double floating point precision format or if they are encoded as integer values. The process of encoding those values is performed as follows: consider Figure 5.2 which depicts a control law for a unidimensional, which dimension was assumed this way to simplify the explanation. The maximum and minimum values are denoted by U_{max} and U_{min} respectively and automatically defines the interval boundaries within all the control values lie. A single byte encodes 256 different integer values that range from 0 to 255. This way, and within the interval of control values, we are able to discriminate at most 256 levels. Each level is uniquely described by a binary label that represents a particular value within the aforementioned interval. So the task consist on finding the appropriate integer value that discriminate a particular value from the others, that classify the floating point number according the level it belongs.

Define the following transformation

$$\begin{cases} q = \frac{U_{max} - U_{min}}{255} \\ u_{int} = \text{Int}\left(\frac{u_{float} - U_{min}}{q}\right) \\ u_{float} = (u_{int} \cdot q) + U_{min} \end{cases} \quad (5.4)$$

where $\text{Int}()$ is the function that returns the integer part of its argument. The transformation (5.4) represents the way that the integer equivalent of the floating point value is obtained. Also it shows the transformation that transforms the integer representation back a floating point value. It's obvious that from the inverse transformation only an approximate value is

obtained and not the original one. This is because values that are only different by a quantity less than $q/2$ are mapped to the same integer and then mapped back to the same floating point value by the inverse transformation. This process is similar to the A/D transformation where q represents the resolution or quantum.

As can be expected, the smaller the control interval the more accurate the results. This brings about the issue of using normalized universe of discourse. As stated by [9], the impact of input scaling factors on phase trajectories can be quite significant. There is evidence that with a normalized universe the rules activated by an input increases. In general, it is much easier to achieve the desired control quality with a larger number of activated fuzzy rules. That is why the scaling of inputs should be done carefully so that we can use the full fuzzy rule base. Because of inadequate scaling, the fuzzy rule table may be imperfectly partitioned, which may cause many of the rules to remain inactive even though the rule base is complete.

This mapping function enables to use the previous defined procedure without any substantial change. The only difference is that when retrieving a value from the encoded look-up table, the transformation of the integer back to floating point must be performed according to (5.4). This in turn implies that within the look-up function the values of q and U_{\min} must be stored in order to fulfill this procedure. When the user decide to use this option all this variables are automatically included at the look-up table.

Also it should be noted that on the C programming language an integer is made up of 2 bytes, thus in practice instead of integers it is used the unsigned `char` type, and together with typecasting operation provides all that is needed.

All this functionality is actually delivered in a series of c-compatible files automatically created by the FEUP Fuzzy Tool. These c-coded files are the following

- The `ffuzzylib.h` header file that provides the definition (function prototype) whose utilization is for control purpose, as it should be expected. This header file must be included on main program file. This header includes another header file called `ffconstants.h` (see below).
- The `ffuzzylib.c` provides the actual implementation of the control (look-up) functions to be linked to the project.
- The `ffconstants.h` has the definition of all constant (like sampling resolution, matrix dimensions, upper/lower bound of the variable domain, etc.) used by the functions implemented on the `ffuzzylib.c` file. The values defined on this file are used by the compiler's pre-processor to substitute every occurrence of the label, which represents the constant, prior to compilation. This way, by the use of the `#define` C pre-processor directive, it is bypassed the need of storage for this values and also human understandable code is obtained. Every constant has a unique label that is only known by the function that needs it.

```

C source code Editor
File Edit View Ctrl Surface Parameters

#include <math.h>
#include "ffuzzylib.h"

inline double aitken_Neville(double pol_interp1, double pol_interp2, double x, double x1, double x2)
{
    return (pol_interp1*(x-inter_nod2)-pol_interp2*(x-inter_nod1))/(x-inter_nod2-x-inter_nod1);
}

double surfl_LookUp(float x, float y){
    int row_index, col_index;
    double result, result2;
    static const double surfl[9][9]={ { -0.89000, -0.88480, -0.87167, ...
    if (x<surfl_MIN_DOMAINX) x=surfl_MIN_DOMAINX;
    if (x>surfl_MAX_DOMAINX) x=surfl_MAX_DOMAINX;
    if (y<surfl_MIN_DOMAINY) y=surfl_MIN_DOMAINY;
    if (y>surfl_MAX_DOMAINY) y=surfl_MAX_DOMAINY;
    x/=surfl_SAMPLING_STEPX;
    y/=surfl_SAMPLING_STEPLY;
    row_index=(int)floor(x)+surfl_ROW_OFFSET;
    col_index=(int)floor(y)+surfl_COL_OFFSET;
    if (row_index>=surfl_MAX_ROW) row_index=surfl_MAX_ROW-1;
    if (col_index>=surfl_MAX_COL) col_index=surfl_MAX_COL-1;
    result=aitken_Neville(surfl[row_index][col_index],surfl[row_index+1][col_index],x,surfl_MIN_DOMAINX,surfl_MAX_DOMAINX);
    result2=aitken_Neville(surfl[row_index][col_index],surfl[row_index][col_index+1],y,surfl_MIN_DOMAINY,surfl_MAX_DOMAINY);
    result2*=(y-(float)(col_index-surfl_COL_OFFSET));
    return result2-result;
}

```

Figure 5.3 Snapshot of the C code Editor

A file saved with a user predefined name that contains a code example that uses the first of the look up functions defined on the ffuzzylib.h for illustrative purpose. This test code is included in a `#if ... #endif` block.

5.3 - The C code Editor

The C code editor is a FEUP Fuzzy Tool form whose purpose is aid the user through the automatic creation and edition of the programming code for embedded systems. This form is presented at Figure 5.3. It has a series of specific functions and procedures responsible to implement the automatic construction of the microcontroller's programming code whose structure was described on the previous section.

The C code editor supports syntax highlighting and word-wrap, and it includes code completion and templates components for the C programming language. Although it can be used as a general purpose programming code editor, it is intended to provide an inside on the code automatically created for embedded systems programming and also facilitate the user to modify some particular detail of the presented code.

For brevity and simplicity sake, instead of describing every implemented routine to explain the functionality of this feature, the system behavior and requirement analysis will be described with the aid of the Use Case diagram shown at Figure 5.4.

The cornerstone for the automatic code creation lies sampling the control surface which belongs to a particular combination of inputs and output variable that defines a fuzzy controller. To fulfill this task and to obtain modularity on the resulting code that implements

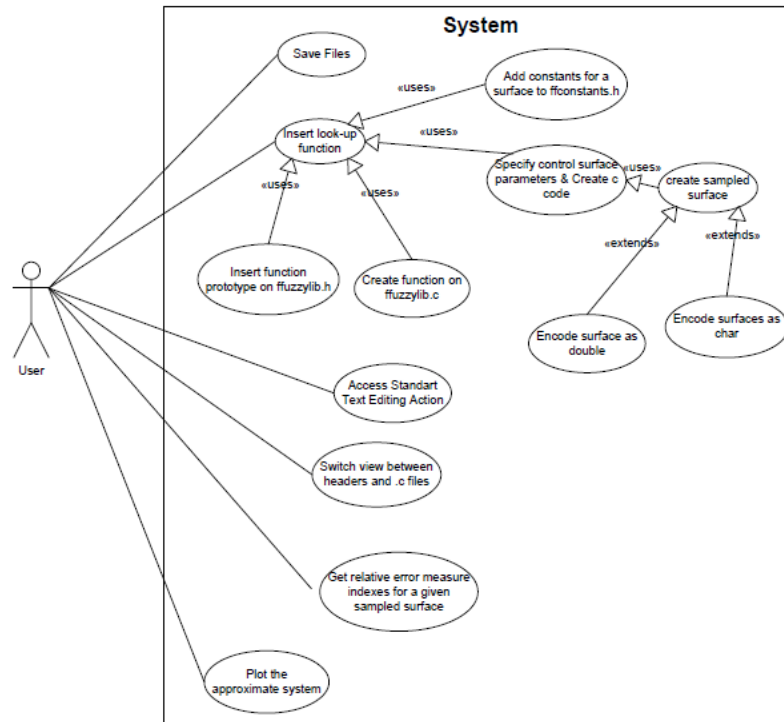


Figure 5.4 Use-Case UML diagram. It depicts the behavior of the system for the C code generation functionality.

the C code Editor, a data structure containing detailed information about the sampling process must be defined. As will be explained along the discussion, different functions use this data structure and its fields to perform several different tasks. This structure must contain information about the sampling resolution of both input variables, the maximum and minimum value of the domain over which is defined the variable, and the actual samples of the control surface which later must be translated onto the statement that declares and initialize the look-table as multidimensional array at the ffuzzylib.c file.

Other properties, that are defined through the previous specified ones and whose values are needed more than once by different operations, are also declared. The purpose is to speed up the availability of the values for those properties (like the row and column dimension of the look-up matrix, or the sample offset that defines the index of the sample representing the zero value of a variable). Thus the `TSampledSurface` data structure is defined

```

TSampledSurface = record
    var1, var2, varout: integer;
    data: array of array of double;
    Var1Min, Var2Min: integer;
    Var1Max, Var2Max: integer;
    row_size: integer;
    col_size: integer;
    offsetX, offsetY: integer;
    resolutionX, resolutionY: double;
end.
  
```

In order for this form to function properly, the user must be able to specify the variables that constitute the controller and automatically defines through them the part of total rule table (in case the fuzzy system is composed by many controllers) that wants to be translated for embedded applications. To offer text edition support basic actions such as copy, paste, cut, select all, undo, redo, etc. must be provided. If it is the case of a MIMO controller, the user must be able to actually encode as many fuzzy controllers as necessary. Also some assessment is necessary in order to quantify the performance of the approximated controller that is being created. In order to actually export the programming code, saving function must be implemented. All this requirements define the behavior of the system to this particular situation and is represented at Figure 5.4.

When the user wants to access the C code Editor is asked to specify: the input and output variables, the inference model (Mamdani or Takagi-Sugeno), the sampling resolution of both variables (or for the first one if the controller is a SISO type), and an oversampling factor to create an oversampled surface to compute relative error measures (absolute measures cannot be used because the transfer function for inputs-output mapping is not analytical), and the type of encoding of the look-up table, that is, whether the values are acquired as floating point with double precision type or as integer type for memory economy. See Figure 5.5 where the process just described is presented by the parameter form.

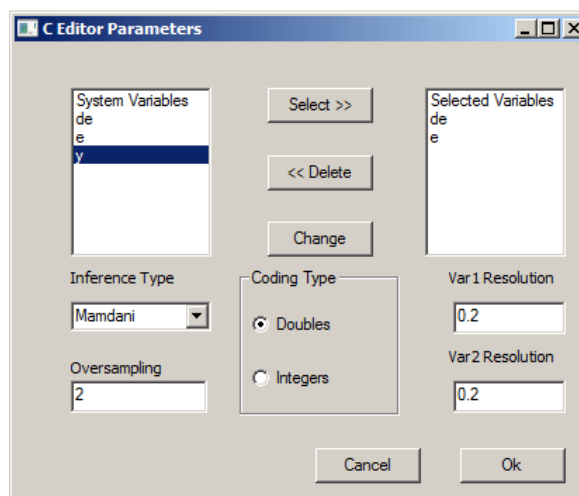


Figure 5.5 User Interface for parameter selection to automate the process of programming code generation.

Inside the Use Case diagram this comprises a function that is implicitly accessed every time that the user wants to insert a new variable, the use case is called “Insert look-up function”. It should be noted that the oversampled function is used to create a relative error surface that is used to present some statistic about the accuracy of the sampled surface. Furthermore, this functionality it’s just implemented to the first sampled control surface that has been inserted. If one wants to insert other look-up functions to create MIMO controllers

they should be previously studied and the parameters that defines them should be also determined or known *a priori*. The error measures provided are maximum error value, mean, standard deviation and the root mean square error (RMSE) a good measure of accuracy.

To perform the sampling operation a procedure like `SampleSurface(surface: TSampledSurface; System: TFFSystem)` is automatically called to fill the data property of the `TSampledSurface` data structure passed by reference as a parameter. The actual insertion of the statement that declares the look-up function on the `ffuzzylib.c` is made by a function like

```
function CreateLookUpFunc(name: string; surface: TSampledSurface;
coding: Tcoding; func_header: string): string;
```

where the user specifies the desired name to the function through the “name” parameter. Depending on the desired encoding type for the look-up table the function will deliver an integer or double type encoded look up table. The insertion of a new look-up function implies the addition of its header at the `ffuzzylib.h` and the reference passed parameter `func_header` contains such definition ready to be added. Also becomes necessary to define the constants values used by a particular look-up function on the `ffconstants.h`, every time a new instance of this function is added. A function like

```
function CreateSurfaceConst(name: string; surface: TSampledSurface):
string;
```

is automatically called to perform this task. The name parameter is used to construct the label that indentifies a constant and it embeds the name of the function that the constant belongs. The Use Case diagram at Figure 5.4 summarizes the previous discussion. This particular use case scenario is responsible for the automatic creation of the c code.

When the user desires to include another basic controller to conceive a MIMO controller, the process is pretty much as above. The respective control surface will be sampled and added to the `ffuzzylib.c` along with the repercussions that occurs at the `ffuzzylib.h` and `ffconstants.h` files.

The user should be able edit or modify every file that composes the microcontroller’s code. In order to deliver this task string data containers should be used to preserve the modifications and content of every specific file and show them on the text plain when requested by the user. This data containers are the implemented through the `TStringList` class, provided by the Lazarus unit `Classes`, and is able to manage arrays or collections of strings. This corresponds to the use case scenario of “switch between files”.

Despite of the error measures provided to the user, the ability to see a 3D plot of the approximated surface is also implemented. It uses the values from data field of the `TSampledSurface` and the dual interpolation method described at the previous section to plot the approximation. Finally, the user is able to exports the created code by saving button

that can be accessed through the file menu tab. One is asked to provide a name for the main program name, as the other files already has a named strictly defined, and all files are saved on the same system directory.

5.4 - Validation Test

Two major concerns arises when trying to test the aforementioned approach, the first is in terms of syntax correctness according to the ANSI C programming language, and the second one in terms of functionality and obtained results when put inside an embedded controller.

To validate both requirements two programming platforms where used, the first one is Bloodshed Dev-C++ [37], a full-featured Integrated Development Environment (IDE) for the C/C++ programming language. It uses Mingw port of GCC (GNU Compiler Collection) as its compiler. This tool enables to validate the ported code on a personal computer as a standard c-coded console application.

At the other extreme, it is necessary to test the program on a concrete microcontroller. The ATmega8 microcontroller from the AVR family was used as a bench test that pursues this validation. AVR implements an 8-bit RISC processor with a modified Harvard architecture which was developed by Atmel. Detailed information about these MCU can be found at [38]. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage and it also support in circuit programming, a very appealing feature. The AVR is one of the most widely used MCU families nowadays with a very wide spectrum of applications. AVRs have a large following due to the free and inexpensive development tools available, including reasonably priced development boards and free development software. AVR Studio [39] is a professional Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows environments. This software was used for testing and debugging the FEUP Fuzzy Tool code and program the ATmega8. To test the FEUP Fuzzy Tool embedded application programming library with the ATmega8 a small program was conceived and using the Synchronous/Asynchronous Serial Peripherals (USART) interface, the control output value was sent via RS232 connection to a personal computer that was monitoring the control variable result for a particular set of inputs. The following description explains the implemented controller that was subject to test on the previous platforms.

The fuzzy rule table shown at Figure 4.9 resembles a typical PI fuzzy controller. The equation giving a conventional PI-controller is

$$u = K_p \cdot e + K_I \cdot \int e \, dt, \quad (4.5)$$

where K_p and K_I are the proportional and the integral gain coefficients. When the derivative, with respect to time, of the above expression is taken, it is transformed into an equivalent expression

$$\dot{u} = K_p \cdot \dot{e} + K_I \cdot e. \quad (4.6)$$

The PI-like fuzzy controller consist of rule of the form

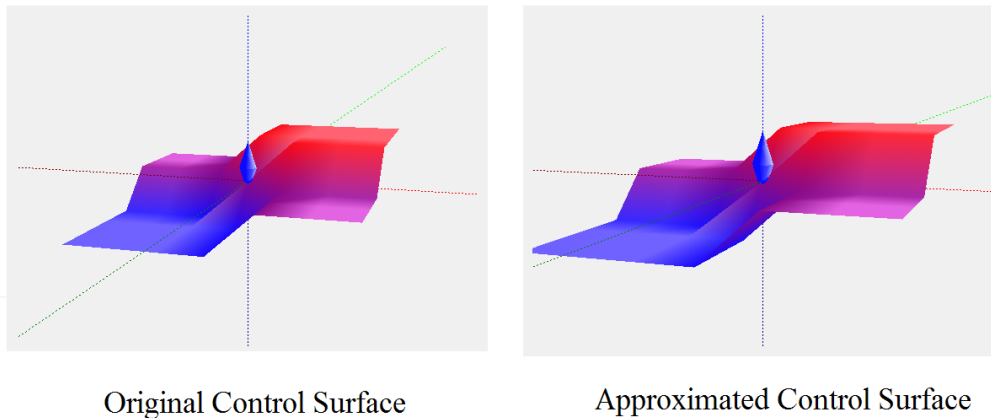


Figure 5.6 Control Surfaces created through fuzzy inference (left) and by samples and dual interpolation (right).

If e is <property symbol> and Δe is <property symbol> then Δu is <property symbol>.

In this case, to obtain the value of the control output variable $u(k)$, the change-of-control output Δu is added to $u(k-1)$. It is to be stressed here that this takes places outside the PI-like fuzzy controller, and is not reflected in the rules themselves.

To make this controller general the universe of discourse of inputs and outputs fuzzy variables are normalized universes and appropriate scaling factor must be conceived for a particular application. This controller was ported and translated to the c-coded program for the AVR microcontroller. The sampling resolution on both variables was 0.2. This methodology shows the advantages in using normalized universes where both memory storage (for the look up table) and good resolution can be achieved. Sampling a universe whose upper and lower bound are far greater than the unity demands large memory capacity in order to obtain a high resolution and a good approximation for the control output. At Figure 5.6 the original and approximated surfaces can be evaluated.

As might be expected the difference are hardly noticeable. Also Table 5.1 summarizes few results obtained by the ported code and the inference on the FEUP Fuzzy Tool analyzed through the rule scope.

Table 5.1 Comparative results of the PI fuzzy controller when implemented on an MCU and on a standard PC.

$e(k)$	$\Delta e(k)$	Output C-coded application	Output FEUP Fuzzy Tool	Approximation Error
-1	-1	-0.8900	-0.8900	0.0000
-0.9	-0.45	-0.8819	-0.8799	0.0020
0	0	0.0000	0.0000	0.0000
0.35	0.23	0.5200	0.5552	-0.0352
1	1	0.8900	0.8900	0.0000

The results are practically the same because a very good approximation for the control surface was obtained. Therefore, the approach employed to develop a fuzzy controller for embedded systems was proven successful. Very good results can be achieved as demonstrated and real-time constraints can be satisfied because of the low computational burden placed on the MUC when compared with the laborious approach of fuzzy arithmetic, especially in processors with limited math support.

5.5 - Summary

An optimized methodology to generate programming code for embedded application of fuzzy controllers was covered throughout the chapter. The results were validated through different platforms with different compilers implementations. One of the test platforms was one of the most widely employed microcontroller family, the ATmega8 microcontroller unit from the AVR family. Through the tests was proven the correctness of the delivered code in terms of syntax compatibility with the C language standard and in terms of functionality, as it delivers on both platforms results similar to those obtained by the fuzzy inference with FEUP Fuzzy Tool software on a standard personal computer. Later on, the FEUP Fuzzy tool will be used to control various simulated processes, thus by having similar results on both embedded controller and with the toolbox solely, one can extrapolate the validity of the embedded controller.

Chapter 6

Self-Learning fuzzy architectures

It is often difficult or impossible to accurately model complex processes or systems using a conventional mathematical approach when the knowledge comes in a shallow form or in no form at all. The basic problem to be studied in this chapter is based on how to construct a fuzzy system from numerical data. Ideally, one uses prior knowledge develop a model and predict the outcome, but for some cases there are systems where little is known or where experimental analyses are too costly to perform, this approach fails to support the design process making the modeling task extremely difficult. This is in contrast to our previous discussion, where we used linguistics as the starting point to specify a fuzzy system.

On the one hand fuzzy modeling is very practical and can be used to develop a model for the system using the limited available information. Some of the available algorithms for developing a fuzzy model from numerical data are: batch least squares, recursive least squares, gradient method, learning from example (LFE), modified learning from example (MLFE), and clustering method. More information about these methods can be found at [10], [42]. The gradient method is of particular importance and is the one approach that will be employed to fulfill the learning capability of a fuzzy system. On the other hand neural networks are well known by its learning capabilities when forced to adapt to a new environment. Thus it is desirable to use the fundamental concepts of this field and applied together with the fuzzy logic theory to exploit the advantages from both approaches.

Relationships between fuzzy systems and neural networks are built based two perspectives. First, techniques from one area can be used in the other. Second, in some cases the functionality is identical. Some label the intersection between fuzzy systems and neural networks with the term “fuzzy-neural” or “neuro-fuzzy” to highlight that techniques from both fields are being used. A feedforward network is traditionally viewed as a nonlinear network whose transfer function can be tuned by changing the weights, biases, and parameters of the activation functions. The fuzzy system is also a tunable nonlinear transfer function whose shape can be changed by tuning, for example, the membership functions.

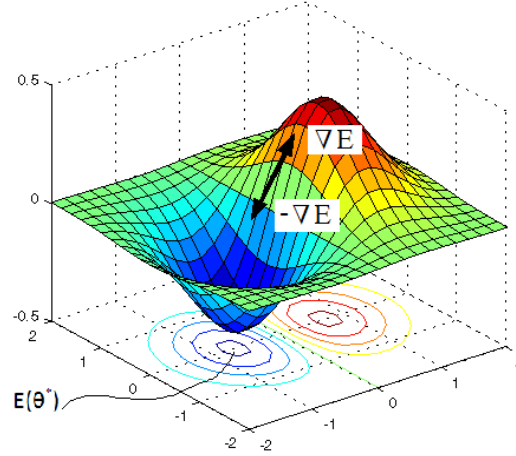


Figure 6.1 Illustration of the concepts of the gradient steepest descent method.

Bearing this similitude in mind, the following approach is possible: Gradient methods can be used for training fuzzy systems to perform system identification or to act as estimators or predictors in the same way as neural networks are trained. Historically, the gradient training of neural networks, also known as “backpropagation training”, was introduced well before the gradient training of fuzzy systems, thus the idea for training fuzzy systems this way was inspired from the field of neural networks.

The objective of this chapter is to show how to construct fuzzy systems from numerical data, and introduce the implementation of self-learning archetypes on the FEUP Fuzzy Tool. For notational purposes, the following convention will be adopted along the document: vectors are denoted by bold letters whereas scalar quantities are not.

6.1 - The gradient steepest descent method

The steepest descent method pertains to a class of gradient-based optimization techniques, capable of evaluating an optimization path according to an objective function's derivative information. For a real-valued objective function $E(\theta)$ defined on an n -dimensional input space $\theta = [\theta_1, \theta_2, \dots, \theta_n]^T$. The main concern of the method is to finding a possibly local minimum point $\theta = \theta^*$ that minimizes $E(\theta)$ (See Figure 6.1).

In general, a given objective function $E(\theta)$ may have a nonlinear form with respect to an adjustable parameter θ . Due to the complexity of $E(\theta)$, an iterative algorithm is often employed to explore the input space efficiently. In iterative descent methods, the next point θ_{k+1} is determined by a step down from the current point θ_k in a direction vector d :

$$\theta_{k+1} = \theta_k + \eta d, \quad (6.1)$$

where η is some positive step size regulating to what extent to proceed in that direction. In neuro-fuzzy literature, the step size η is also called learning rate.

When the straight downhill direction \mathbf{d} is determined on the basis of the gradient $\mathbf{g}(\theta)$ of an objective function $E(\theta)$, such descent methods are called gradient-based descent methods. The gradient of a differentiable function $E: \mathbb{R}^n \rightarrow \mathbb{R}$ at θ is the vector denoted as $\mathbf{g}(\theta)$ and defined by

$$\mathbf{g}(\theta) = \nabla E(\theta) \triangleq \left[\frac{\partial E(\theta)}{\partial \theta_1}, \frac{\partial E(\theta)}{\partial \theta_2}, \dots, \frac{\partial E(\theta)}{\partial \theta_n} \right]^T \quad (6.2)$$

In general, based on a given gradient, downhill directions adhere to the following condition for feasible descent directions.

$$D_v E(\theta) = \mathbf{g} \cdot \mathbf{d} = \|\mathbf{g}\| \cdot \|\mathbf{d}\| \cdot \cos(\phi_k) < 0, \quad (6.3)$$

where $D_v E(\theta)$ is the directional derivative of $E(\theta)$ along direction \mathbf{d} , and ϕ_k denotes the angle between \mathbf{g}_k and \mathbf{d} at the current point θ_k . The descent direction condition (6.3) does not guarantee convergence of the algorithms because it can likely to get stuck on a local minimum.

6.2 - Artificial neural networks

Artificial neural networks (ANN) are a way of using physical hardware or computer software to model computational properties analogous to those of biological neurons such as the ability to learn and discriminate. In a more formal view, an adaptive network is a network structure whose overall input-output behavior is determined by a collection of modifiable parameters. Specifically, the configuration of an adaptive network is composed of a set of interconnected nodes, where each node defines a static node function of its incoming signals to generate a single node output and each connection specifies the direction of signal flow from one node to another. Usually a node function is a parameterized function with modifiable parameters; by changing these parameters, we change the node function as well as the overall behavior of the adaptive network.

Adaptive networks are generally classified into two categories on the basis of the type of connections they have: feedforward and recurrent. A feedforward type of network is shown in Figure 6.2; here the output of each node propagates from the input side to the output side unidirectionally. If there is a feedback link that forms a circular path in a network, then the network is recurrent; see also Figure 6.2.

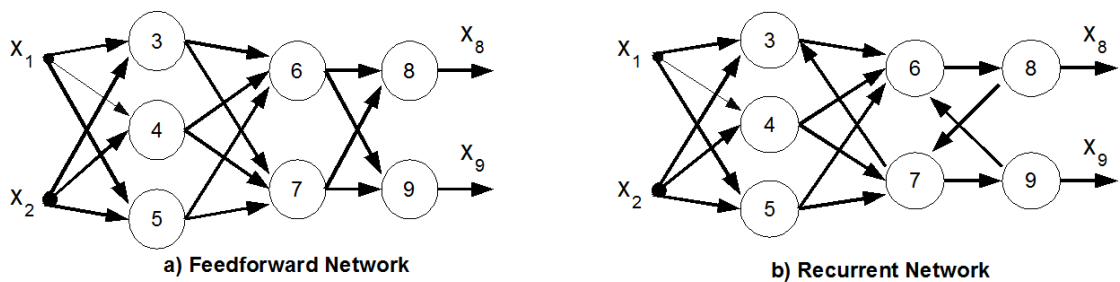


Figure 6.2 The feedforward and recurrent ANN architectures.

The feedforward network is the kind of network treated in this chapter. Conceptually is able to performs a static mapping between its input and output spaces; this mapping may be either a simple linear relationship or a highly nonlinear one, depending on the network structure, that is node arrangement, connections, and so on, or also the functionality of each node. A network for achieving a desired nonlinear mapping that is regulated by a data set consisting of desired input-output pairs of a target system to be modeled. This data set is usually called the training data set, and the procedures used for adjusting the parameters and improve the network's performance are often referred as adaptation or learning algorithms.

The main approach used for training feedforward networks is based on gradient steepest descent method discussed previously and consist on recursively obtain a gradient vector in which each element is defined as the derivative of an error measure with respect to a parameter. As mentioned before this learning paradigm is also known by backpropagation training or backpropagation learning rule. Once the gradient is obtained, the parameters are updated following some derivative based optimization rule as the one described at the previous section.

Suppose that a given feedforward adaptive network has L layers and layer l ($l = 0, 1, \dots, L$; $l = 0$ represents the input layer). Let $N(l)$ be the number of nodes from layer l . The output and function of node i [$i = 1, \dots, N(l)$] in layer l is represented as $x_{l,i}$ and $f_{l,i}$ respectively. Since the output of a node depends on the incoming signals and the parameter set of the node, we have the following general expression for the node function $f_{l,i}$.

$$x_{l,i} = f_{l,i}(x_{l-1,1}, \dots, x_{l-1,N(l)}, \alpha, \beta, \gamma \dots), \quad (6.4)$$

where α, β, γ are the parameters of this node.

Assuming that the given training data set has P entries, the error measure for the p th ($1 \leq p \leq P$) entry of the training data is then defined as the sum of squared errors.

$$E_p = \frac{1}{2} \cdot \sum_{i=1}^{N(l)} (d_i - x_{l,i})^2, \quad (6.5)$$

where d_i is the i th component of the desired output vector and $x_{l,i}$ is the actual i th component of the output vector, the factor $\frac{1}{2}$ is used for convenience in calculating the derivative of the function. To use steepest descent to minimize the error measure, first is necessary to obtain the gradient vector.

It becomes obvious that E_p depends on $x_{l,i}$ indirectly, since a change in $x_{l,i}$ will propagate through indirect paths to the output layer and thus produce a corresponding change in the value of E_p . Thus, a small change within a node's parameter affects the output of that particular network, which in turn influences the network's behavior. The set of parameters of the network then forms an n -dimensional space whose vectors indirectly influence the magnitude of the error function E_p . Bearing this in mind the gradient vector is defined as the derivative of the error measure with respect to each parameter and with the aid of the chain rule we obtain

$$\frac{\partial E_p}{\partial \alpha} = \frac{\partial E_p}{\partial x_{l,i}} \cdot \frac{\partial f_{l,i}}{\partial \alpha} = \epsilon_{l,i} \cdot \frac{\partial f_{l,i}}{\partial \alpha}. \quad (6.6)$$

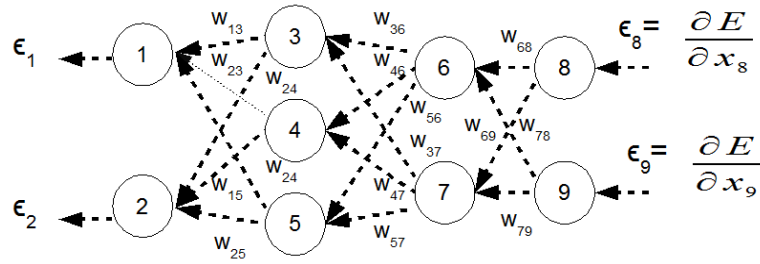


Figure 6.3 Signal backpropagation through a feedforward ANN.

The entity $\epsilon_{l,i}$ is called error signal and may be viewed as a form of derivative information starting from the output layer and going backward layer by layer until the input layer is reached. Note that if the parameter α is shared between different nodes, then Equation (6.6) should be changed to a more general form

$$\frac{\partial E_p}{\partial \alpha} = \sum_{x^* \in S} \frac{\partial E_p}{\partial x^*} \cdot \frac{\partial f^*}{\partial \alpha}, \quad (6.7)$$

where S is the set of nodes containing α as a parameter; and x^* and f^* are the output and function, respectively, of a generic node in S , see Figure 6.3.

Accordingly, for steepest descent minimization, the update formula for the generic parameter α is

$$\Delta \alpha = -\eta \frac{\partial E_p}{\partial \alpha}. \quad (6.8)$$

6.3 - The Perceptron neuron model

The device known as the perceptron is a mathematical function conceived as an abstraction of biological neurons. It was invented by psychologist Frank Rosenblatt in 1957 at the Cornell Aeronautical Laboratory. It was derived from a biological brain neuron model, proposed by McCulloch and Pitts back in 1943 which was the first neuron model. It represented an attempt to illustrate some of the fundamental properties of intelligent systems in general.

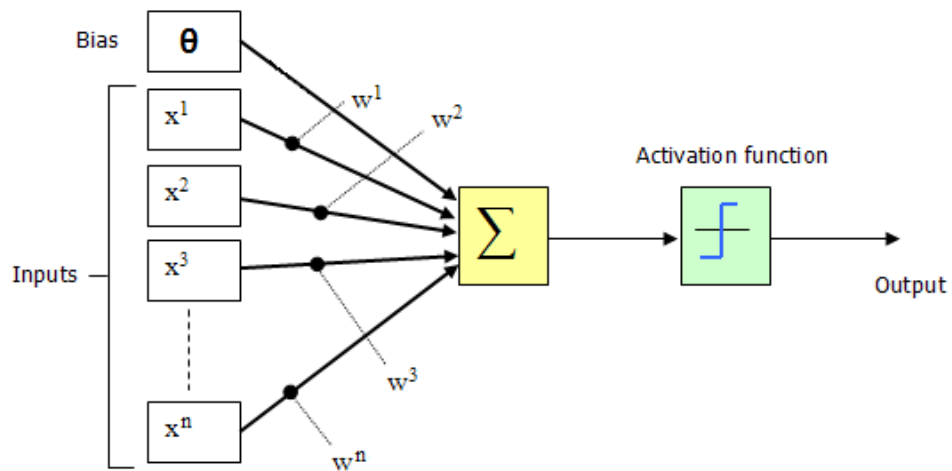


Figure 6.4 The perceptron model.

Rosenblatt believed that the connectivity that develops in biological networks contains a large random element and developed a theory of statistical separability that he used to characterize the gross properties of these somewhat randomly interconnected networks.

The perceptron model can be seen at Figure 6.4. It is a simple structure, having just one nonlinear function of a weighted sum of several data inputs x_1, x_2, \dots, x_n . For computational efficiency, we can introduce the bias connection weight w_0 in place of the threshold value, thus the transfer function of the perceptron is

$$\begin{aligned} o &= f(\sum_{i=1}^n w_i x_i - \theta) \\ &= f(\sum_{i=1}^n w_i x_i + w_0), \quad w_0 = -\theta, \\ &= f(\sum_{i=0}^n w_i x_i) = \mathbf{w}^T \cdot \mathbf{x} \end{aligned} \quad (6.9)$$

The activation function can take various forms and must be a smooth, bounded and differentiable function for training purposes. The most commons are

- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$.
- Hyperbolic tangent: $f(x) = \tanh(x/2) = \frac{1-e^{-x}}{1+e^{-x}}$.
- Identity function: $f(x) = x$.

The sigmoidal and the hyperbolic tangent activations functions are often used on regression and classification problems. Generally, it maps all or a part of the input pattern into a binary value $X_i \in [1, 0]$ or a bipolar value $X_i \in [-1, 1]$. Perceptrons can differentiate patterns only if the patterns are linearly separable, thus the goal of the perceptron is to correctly classify the set of externally applied stimuli x_1, x_2, \dots, x_n into one of two classes. The notion of linear separable will become clear shortly. The decision rule for the classification is to assign the each of the extreme values that the activation function can take to each of the classes respectively. To develop insight into the behavior of a pattern classifier, it is customary to plot a map of the decision regions in the m -dimensional signal space spanned by the m input variables x_1, x_2, \dots, x_n . In the simplest form of the perceptron there are two decision regions separated by a hyperplane defined by

$$\sum_{i=1}^n w_i x_i - \theta = 0. \quad (6.10)$$

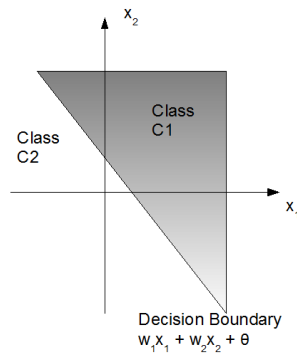


Figure 6.5 Graphical interpretation about the classification process of two classes in a plane

This is illustrated in Figure 6.5 for the case of two input variables x_1 and x_2 for which the decision boundary takes the form of a straight line.

A point (x_1, x_2) that lies above the boundary line is assigned to class C1 and a point (x_1, x_2) that lies below the boundary line is assigned to class C2. Note also that the effect of the bias θ is merely to shift the decision boundary away from the origin.

The synaptic weights w_1, w_2, \dots, w_n of the perceptron can be adapted on an iteration-by-iteration basis. Starting with a set of random connection weights, the basic learning algorithm for a single-layer perceptron repeats the following steps until the weights converge:

1. Select an input vector \mathbf{x} from the training data set.
2. If the perceptron gives an incorrect response, modify all connection weights w_i $\Delta w_i = \eta t_i x_i$ according to where t_i is a target output and η is a learning rate.

The foregoing learning rule can be applied as well to updating a threshold $\theta = -w_0$ according to Equation (6.9). The value for the learning rate η can be a constant throughout training; or it can be a varying quantity proportional to the error. A η that is proportional to the error usually leads to faster convergence but can cause unstable learning. The preceding learning algorithm above is roughly based on gradient descent; Rosenblatt proved that there exists a method for tuning the weights that is guaranteed to converge to provide the required output if and only if such a set of weights exist. This is called the perceptron convergence theorem. A detailed explanation about this algorithm can be found at [43].

6.4 - Backpropagation Multilayer Perceptron

The backpropagation Multi Layer Perceptron (MLP) network is an adaptive feedforward network whose nodes represent the perceptron model previously introduced. It normally uses three or more layers of neurons (nodes). Its power resides in the fact that it can classify data that is not linearly separable, or separable by a hyperplane. The net input \bar{x} of a node is defined as the weighted sum of the incoming signals plus a bias term. For instance, the net input and output of node j are given by

$$\begin{cases} \bar{x}_j = \sum_i w_{ji} x_i - \theta_j \\ x_j = f(\bar{x}_j) = (1 + e^{-\bar{x}_j})^{-1} \end{cases} \quad (6.11)$$

These networks are by far the most commonly used ANN structures for applications in a wide range of areas, such as pattern recognition, signal processing, data compression, and automatic control, speech and image recognition, and machine translation software, but they have also seen applications in other fields such as cyber security.

The sigmoid function will be our choice for the activation function of each neuron throughout the discussion. The derivative of this function is given by

$$\begin{aligned} f'(x) &= f(x) \cdot (1 - f(x)) \\ f(x) &= (1 + e^{-x})^{-1} \end{aligned} \quad (6.12)$$

Learning occurs in the perceptron by changing connection weights (or synaptic weights) after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning, and is carried out through backpropagation. Retaking our previous discussion about this topic, to find the gradient vector for the MLP with sigmoid activation function the error term from equation (6.6) is given with the aid of the chain rule and equations (6.11) and (6.12) by the expression

$$\bar{\epsilon}_i = \begin{cases} -(d_i - x_i) \frac{\partial x_i}{\partial \bar{x}_i} = (d_i - x_i)x_i(1 - x_i), & \text{if node } i \text{ is an output node} \\ \frac{\partial x_i}{\partial \bar{x}_i} \sum_{j,i < j} \frac{\partial E_p}{\partial \bar{x}_j} \frac{\partial \bar{x}_j}{\partial x_i} = x_i(1 - x_i) \cdot \sum_{j,i < j} \bar{\epsilon}_j w_{ij}, & \text{otherwise} \end{cases} \quad (6.13)$$

where d_i is the desired output for node i and w_{ij} is the connection weight from node i to j . If w_{ij} is zero then there is no direct connection. Then the weight update w_{ki} follows from (6.8) as

$$\Delta w_{ki} = -\eta \frac{\partial E_p}{\partial w_{ki}} = -\eta \frac{\partial E_p}{\partial \bar{x}_i} \cdot \frac{\partial \bar{x}_i}{\partial w_{ki}} = \eta \bar{\epsilon}_i \bar{x}_k \quad (6.14)$$

where η is a learning rate that affects the convergence speed and stability of the weights during learning. The backpropagation algorithm has a slow rate of convergence; one way to increase the speed of convergence is to use a technique called momentum. When calculating the weight-change value, Δw_{ki} , we add a fraction of the previous change. This additional term tends to keep the weight changes going in the same direction, hence the term momentum. The weight change equations on the output layer then become

$$\Delta w_{ki} = -\eta \bar{\epsilon}_i \bar{x}_k + \alpha \Delta w_{ki}^{prev} \quad (6.15)$$

Where Δw_{ki}^{prev} is the previous update amount, and the momentum constant α , in practice, is usually set to something between 0.1 and 1. The addition of the momentum term smoothes weight updating and tends to resist erratic weight changes due to gradient noise or high spatial frequencies in the error surface.

6.5 - Multilayer perceptron data structure

This section describes the implementation of the Backpropagation Multilayer Perceptron network in the FEUP Fuzzy Tool. It starts with a requirement analysis followed by the description of the data structure realization.

The first thing to note is the massively parallel processing of every ANN models that is not easily mimicked by a traditional computer system. Thus, it becomes necessary to adapt the sequential behavior of computer processing in order to be able to simulate this type of a parallel-processing system. It is important also to consider the aspects of code portability and reusability early in the implementation of the ANN model, as those models will serve as a basis for the implementation of advance hybrid models, as will be shown later on. Thus attention is paid on the characteristics that are common to most of the ANN models, and

implement those characteristics as data structures that will allow the ANN simulator to migrate to the widest variety of network models possible.

6.5.1 - Requirement analysis

One of the first observations to make is that it is necessary to design the data structure so that it can be sized adequately. The ability to specify the number of processing elements and its organization must be made at run-time, as is not desirable to reprogram and recompile an ANN application every time the network structure change.

Another observation is that the implementation of the ANN poses a high computational complexity. To understand why this is so, let's consider how a uniprocessor computer will simulate a neural network. The program will have to be written to allow the CPU to time multiplex between units in the network; that is, each unit in the ANN model will share the CPU for some period. As the computer visits each node, it will perform the input and output mapping computation before moving on to the next unit. As we have already seen, this computation is normally a sum-of-products calculation, a very time-consuming operation if there are a large number of inputs at each node. Compounding the problem, the sum-of-products calculation is done using floating-point numbers, since the network simulation is essentially a digital representation of analog signals. Thus, the CPU will have to perform two floating-point operations (a multiplication and an addition) for every input to each unit in the network. Given the large number of nodes in some networks, each with many possible inputs, it is easy to see that the computer must be capable of undertake this burden in order to simulate an ANN of moderate size in a reasonable amount of time. Even assuming the computer has the floating-point hardware needed to improve the performance of the simulator, it's desirable to optimize the computer's ability to perform this computation by designing this data structures appropriately.

The observation made earlier that data will be processed as a sum of products implies that the network data ought to be arranged in groups of linearly sequential arrays. The rationale behind this arrangement is that it is much faster to step through an array of data sequentially than it is to have to look up the address of every new value, as would be done if a linked-list approach were used. This grouping also is much more memory efficient than is a linked-list data structure, since there is no need to store pointers in the arrays.

Base on the network structure presented earlier, to simulate this structure using data organized in arrays, one can model the connections and node outputs as values in two arrays, which we will call weights and outputs respectively. The data in these arrays will be sequentially arranged with a one to one correspondence according to the item being modeled. Specifically, the output from the first input unit will be stored in the first location in the outputs array, the second in the second, and so on. Similarly, the weight associated with the connection between the first input unit and the unit of interest, w_{1i} , will be located as the first value in the i th weights array, weights. Notice that there will be many such arrays

in the network, each containing a set of connection weights. The index here indicates that this array is one of these connection arrays, specifically the one associated with the inputs to the i th network unit.

The process needed to compute the aggregate input at the i th unit in the upper layer is as follows: it begins by setting two pointers to the first location of the outputs and weights arrays, and setting a local accumulator to zero. One then performs the computation by multiplying the values located in memory at each of the two array pointers, adding the resulting product to the local accumulator, incrementing both of the pointers, and repeating this sequence for all values in the arrays.

If a neuron from the lower layer does not have a connection with a neuron from the upper layer that connection would have a zero weight, so the contribution to the total input of the node that it feeds would be zero. Therefore, the only reason for the existence of this connection would be that it acts as a placeholder, allowing the weights array to maintain its one-to-one correspondence of location to connection. The cost of this implementation is the amount of time consumed performing a useless multiply-accumulate operation, and, in very sparsely connected networks, the large amount of wasted memory space, although this a very rare case because this network could be replaced by a smaller and fully connected network.

Until now, it has been defined a possible structure for performing the input computations at each node in the network, this structure need to be extended to model the entire network. Since the array structure tends to be efficient for computing input values at run time on most computers, the implementation of the connection weights and node outputs as dynamically allocated arrays seems to be appropriated. Similarly, any additional parameters required by the different networks and associated with individual connections will also be modeled as arrays that coexist with the connection-weights arrays. It is imperative to model a higher-level structure that binds together the previous concepts and enables the access of these arrays in an efficient manner. The following assumptions about how information is processed in a "standard" neural network will provide a base to conceive such structure:

- Units in the network can always be accommodated into layers in which units have similar characteristics, even if there is only one unit in some layers.
- All the units in any layer must be completely processed before the computation of units from a posterior layer takes place.
- The number of layers that our network simulator will support is indefinite, limited only by hardware.
- The processing done at each layer will involve the input connections to a node, and will only rarely involve output connections from a node.

Based on these assumptions, a layer will consist of a record that contains pointers to the various arrays that store the information about the nodes on that layer.

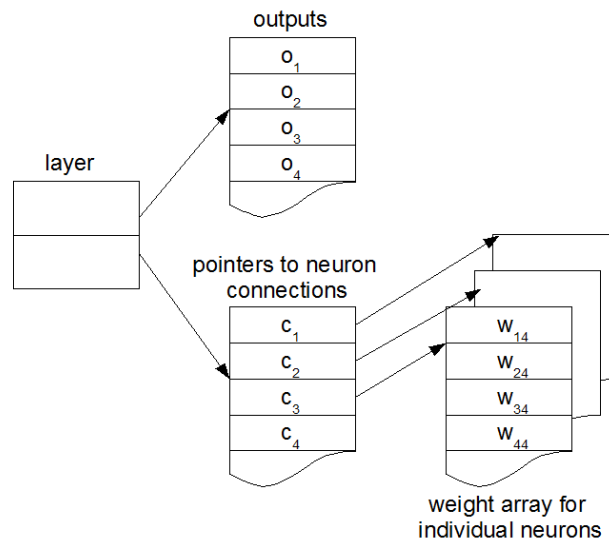


Figure 6.6 Structural view of the layer model. Adapted from [44].

Such a layer model is presented in Figure 6.6. Notice that, whereas the layer record will locate the node output array directly, the connection arrays that represent the connection weight for a particular neuron are accessed indirectly through an intermediate array of pointers, in which each position gives access to the weight connections for the related neuron. This resembles a matrix or bi-dimensional array structure. Since each node on the layer will produce exactly one output, the outputs for all the nodes on any layer can be stored in a single array. However, each node will also have many input connections, each with weights unique to that node. Therefore one must construct a data structure that allows input-weight arrays to be identified uniquely with specific nodes on the layer. The intermediate weight-pointer array satisfies the need to associate input weights with the appropriate node, while allowing the input weights for each node to be modeled as sequential arrays, thus maintaining the desired efficiency in the network data structures. This provides the basis for the implementation of the MLP network, and allows for networks of arbitrary size and complexity, while optimizing the data structures for efficient run-time operation.

6.5.2 - Model implementation

Based on our knowledge of how the MLP operates and the previous discussion, the class `TBPNNetwork` shown at the class diagram on Figure 6.7 is proposed. An inspection of the MLP class reveals that this structure is designed to allow us to create networks with any number of hidden layers through the usage of dynamic arrays of the structure `TLayer`.

The data structure `TLayer` defines the layer structure and is also shown at the class diagram of Figure 6.7. Note that the MLP has two modes of operation, i.e., forward and backward signal propagation, and different information is needed in each phase. Thus, the layer structure contains two different sets of arrays: one set used during forward propagation, and one set used during error propagation.

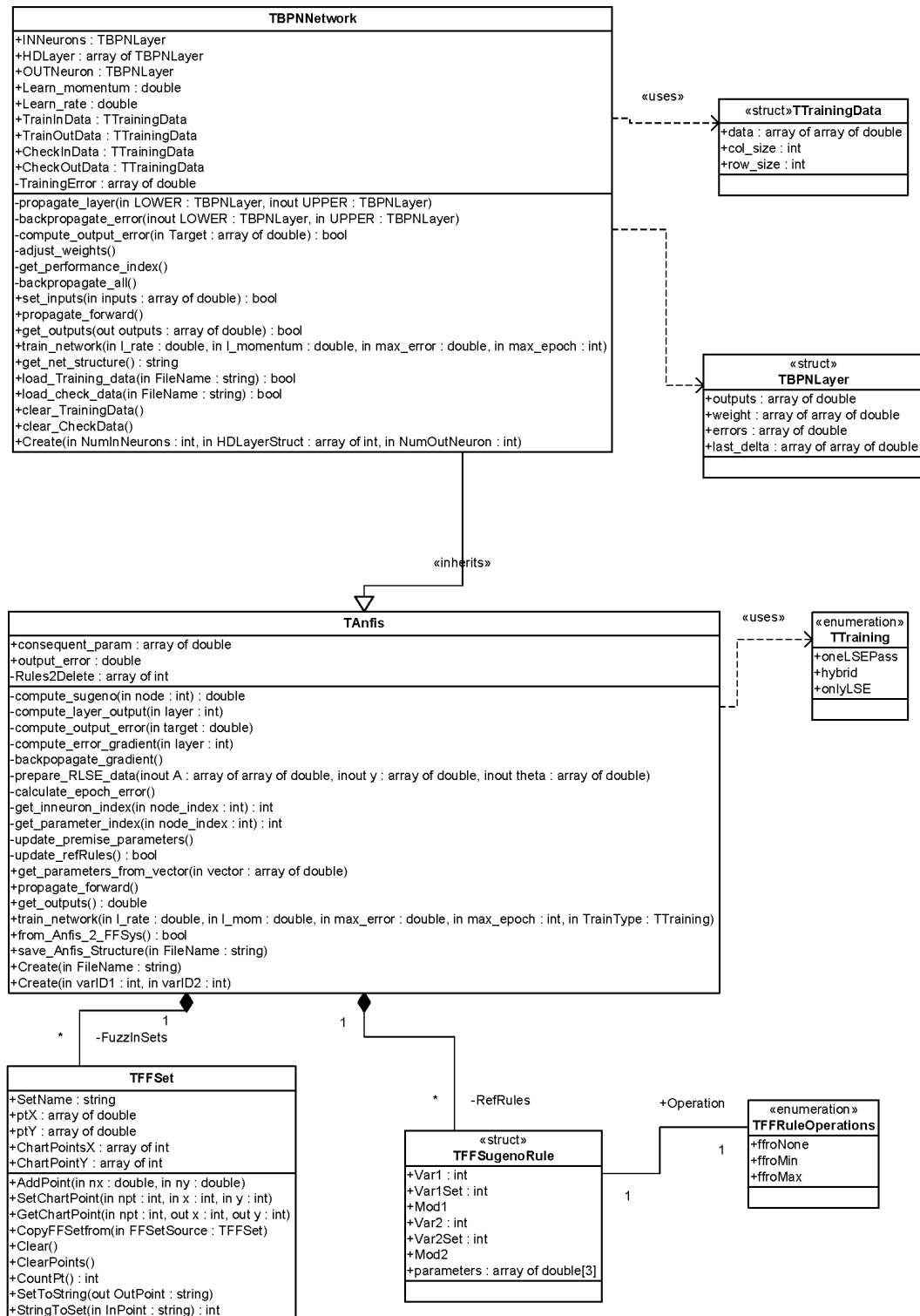


Figure 6.7 UML class diagram of the neuro-fuzzy structures inside the FEUP Fuzzy Tool

The weight connections of the layer are stored within a matrix (two dimensional array or array of arrays) inside the layer, and each row of that matrix stores at each column the value of the weight connection respective to the neuron of the previous layer, i.e. $\text{weight}[i][j]$

represent the weight connection from neuron j from the layer L to the neuron i of the layer $L+1$. When using momentum to speed up the training phase, the previous updating values should be stored. To accomplish this, another matrix called `last_delta` is used to store these values and their order is in one-to-one correspondence with the weight matrix, thus `last_delta[i][j]` correspond to the weight connection `weight[i][j]`. During the forward-propagation phase, the network will use the information contained in the outputs and weights arrays, however, during the backpropagation phase, the MLP requires access to an array of error terms (one for each of the units on the layer) and to the list of change parameters used during the previous learning pass (stored on a connection basis). Without loss of generality the bias parameter of each neuron is not considered for the implemented model.

The class' constructor `Create()` receive three parameters, two integer values and an array of integer. The integer values represents the numbers of neurons of the input and output layers. The array parameter has as many elements as the desired number of hidden layers, and each integer element of the array specifies the number of neuron or nodes for that particular layer. Weights are initialized with small, random values between ± 0.5 creating a fully connected network.

In a MLP, the signals flow is bidirectional, but in only one direction at a time. During training, there are two types of signals present in the network: during the first half-cycle, modulated output signals flow from input to output; during the second half-cycle, error signals flow from output layer to input layer. In the production mode, only the forward propagated signal is utilized. In this network model, the input units are fan-out processors only. That is, the units in the input layer perform no data conversion on the network input pattern. They simply act to hold the components of the input vector within the network structure.

Forward signal propagation occurs according to the following sequence of activities

1. Locate the first processing unit in the layer immediately above the current layer.
2. Set the current input neuron total to zero.
3. Compute the product of the first input connection weight and the output from the transmitting unit.
4. Add that product to the cumulative total.
5. Repeat steps 3 and 4 for each input connection.
6. Compute the output value for this unit by applying the output sigmoid function.
7. Repeat steps 2 through 6 for each unit in this layer.
8. Repeat steps 1 through 7 for each layer in the network.

Based on these steps, to accomplish the forward propagation the class `TBPNNetwork` contains four methods. The first function is the method `set_inputs(inputs: array of real)` and it will serve as the interface and assumes that the user has defined an array of floating-point numbers that indicate the pattern to be applied to the network as inputs. If the

user specifies an input array that differs in terms of dimensions to the number of input neurons it will return the Boolean false value to notify the user. The next method, `propagate_layer(LOWER, UPPER: TBPNNLayer)`, performs the forward signal propagation between any two adjacent layers, that is from the lower towards the upper layer. This routine embodies the calculations done by equation (6.11) for each neuron of the upper layer and the steps 1 through 7 of the forward signal propagation process. With the aid of the previous routine the `propagate_forward()` method performs the forward signal propagation for the entire network. It assumes the input layer contains a valid input pattern, placed there by a higher-level call to `set_inputs()`. Finally, the method `get_outputs(outputs: array of real)` retrieves the output values generated by the network and copy them into an external array specified by the calling entity. This routine is the complement of the set-input routine described earlier. The function will return a Boolean false value if the calling entity provides an array of incompatible size to the neurons of the output layer.

The previously defined methods are also used on the training mode. Once an output value has been calculated for every unit in the network, the values computed for the units in the output layer are compared to the desired output pattern, element by element. At each output unit, an error value is calculated. These error terms are then fed back to all other units in the network structure accomplishing the backpropagation algorithm through the following sequence of steps:

1. Locate the first processing unit in the layer immediately below the output layer.
2. Set the current error total to zero.
3. Compute the product of the first output connection weight and the error provided by the unit in the upper layer.
4. Add that product to the cumulative error.
5. Repeat steps 3 and 4 for each output connection.
6. Multiply the cumulative error by the derivative of the sigmoid function, i.e., equation (6.12).
7. Repeat steps 2 through 6 for each unit on this layer.
8. Repeat steps 1 through 7 for each layer.
9. Locate the first processing unit in the layer above the input layer.
10. Compute the weight change value for the first input connection to this unit by adding a fraction of the cumulative error at this unit to the input value to this unit.
11. Modify the weight change term by adding a momentum term equal to a fraction of the weight change value from the previous iteration.
12. Save the new weight change value as the old weight change value for this connection.

13. Change the connection weight by adding the new connection weight change value to the old connection weight.
14. Repeat steps 10 through 13 for each input connection to this unit.
15. Repeat steps 10 through 14 for each unit in this layer.
16. Repeat steps 10 through 15 for each layer in the network.

Once again based on these steps that describes the backpropagation process and because we allowed an extra array to contain error terms associated with each unit within a layer (similar to our data structure for unit outputs) the error-propagation procedure can be accomplished in four routines. The first will compute the error term for each unit on the output layer `compute_output_error(const TARGET: array of real)`, where `target` is the desired output vector. The second will backpropagate errors from a layer with known errors to the layer immediately below (steps 1 to 7 of the backpropagation process), this is accomplished by the method named `backpropagate_error(upper, lower: TBPNNLayer)`. To backpropagate the error term through all the layers the method `backpropagate_all()` is used. The last one, the `adjust_weights()` method, will use the error term at any unit to update the output connection values from that unit. This last method implements the steps 9 through 16 of the backpropagation process.

There are some other auxiliary methods that needed when training the network, namely the loading of a training data set and checking data set, and the network performance measure. The data structure `TTrainingData` contains a bidimensional array (matrix) that stores a collection of data, and also has fields that provide the dimension of such collection (see Figure 6.7). The class `TBPNNNetwork` has four attributes of this type at the private section, which represent containers for the input and output training data set and also for the checking data set. The name of this attributes are `TrainOutData` , `TrainOutData`, `CheckInData` and `CheckOutData`. The rows of the data matrix represent a single data tuple of the input or output data collection, and have the same dimension as the number of neurons of the input or output layer respectively.

The method `load_Training_data(FileName: string)` reads the specified file that implicitly contain the training data (inputs pattern and desired output) for the network learning and divides it into the training input data (patterns), which are stored in the attribute `TrainInData`, and desired output data, which is stored on the class' attribute `TrainOutData`. The data field is dynamically sized within the function and the number of rows from both structures is the same at the end of the process. The method returns a Boolean value as a signal of the success or failure of the procedure. Every line of the specified file should contain a floating point number for each input plus the output neurons. If there is a mismatch on the number of values at any line and the number of neurons of the input and output layers, the function will stop the loading and will return the Boolean false

value. The method that perform the same operation for the checking data is the `load_check_data(FileName: string)`.

In general, one can use as many data as you have available to train the network, although is not mandatory to use it all. From the available training data, a small subset is often all that you need to train a network successfully. The remaining data can be used to test the network to verify that the network can perform the desired mapping on input vectors it has never encountered during training.

The MLP is good at generalization, that is, given several different input vectors, all belonging to the same class, a MLP will learn to key off of significant similarities in the input vectors. Irrelevant data will be ignored. In contrast to generalization, the MLP will not extrapolate well. If a MLP is inadequately or insufficiently trained on a particular class of input vectors, subsequent identification of members of that class may be unreliable. The training data must cover the entire expected input space. During the training process, select training-vector pairs randomly from the set, if the problem lends itself to this strategy. In any event, training the network completely with input vectors of one class, and then switch to another class is not desirable because the network will "forget" the previous training.

In order to measure the efficiency of the network when subjected to training a performance index must be computed. The method `get_performance_index()` will use the data contained on the `CheckInData` (input patterns) and `CheckOutData` (desired output to the correspondent pattern) to evaluate the error of the network for the entire data set. Every input pattern, that is each row of the `CheckInData` data, is forward propagated through the network and its output value is compared with the desired output vector of the `CheckOutData`. The difference of the desired and actual output is computed for the complete data collection and then the root mean square error (RMSE) is computed as a performance index of the network.

With all the previous introduced methods there is one last thing that is needed to have a complete model of the MLP network is the actual training function. As mentioned before this function will use the entire training data collection with the backpropagation algorithm with momentum (to speed the convergence of the algorithm) to teach the neural network how to perform a particular mapping. The method that implements this algorithm is

```
train_network(l_rate,    L_momentum,    max_error:    real;    max_epoch:
integer);
```

where `l_rate` is the learning rate, `l_momentum` the momentum constant, `max_error` is the desired performance index to achieve, and `max_epoch` is the maximum number of epochs that are allowed to achieved the desired performance. An epoch is defined as a training cycle where the entire data set has been presented to the network and the weight connections have been actualized correspondingly. The `train_network` method is then a top-level routine that calls the signal-propagation procedures in the correct sequence and is defined as

```

train_network(l_rate, L_momentum, max_error: real; max_epoch: integer) begin
  while network_error > max_error or epoch <= max_epoch
  do
    for i:=0 to TrainInData.row_size-1 do
      set_inputs(TrainInData.data[i]);
      propagate_forward;
      compute_output_error(TrainOutData.data[i]);
      backpropagate_error();
      adjust_weights();
    end_for
    get_performance_index();
  end_while
end.

```

6.6 - Model validation: the XOR problem

The simplest and most well-known pattern recognition problem in neural network literature is the exclusive-OR (XOR) problem. The task is to classify a binary input vector to class 0 if the vector has an even number of 1's, or assign it to class 1. For a three-input binary XOR problem, the desired behavior is regulated by a truth table:

Table 6.1 Truth table of the XOR operation for three inputs and the ANN classification.

X	Y	Z	Output Class	MLP network Approximation	Approximation Error
0	0	0	0	0.1459	-0.1459
0	0	1	1	0.9867	0.0133
0	1	0	1	0.9864	0.0136
0	1	1	0	0.0171	-0.0171
1	0	0	1	0.9950	0.0050
1	0	1	0	0.0150	-0.0150
1	1	0	0	0.0137	-0.0137
1	1	1	1	0.8692	0.1308

The XOR problem is not linearly separable. In other words, we cannot use a single-layer perceptron to construct a straight line to partition the two-dimensional input space into two regions, each containing only data points of the same class. The MLP data structure was used to address this problem with 3-6-6-1 structure (three input nodes, six nodes for the first and second hidden layers and one output node). The size of the network was found by trial-and-error procedure. Networks of inferior size fail to recognize the all the patterns that characterizes the problem, that is some of them where correctly classified but others don't

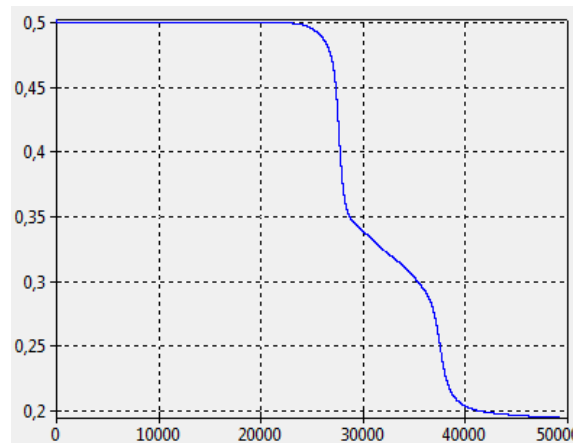


Figure 6.8 Training error evolution for the MLP to approximate the XOR function.

because the network was incapable to make enough partitions on the hyperspace for correct pattern recognition. The results are also summarized at Table 6.1. As can be seen the results are pretty close to those that are expected.

The training was performed with the backpropagation algorithm with momentum as expected, with a learning rate of 0.3, learning momentum of 0.1 and a desired performance error lesser or equal than 0.1. The training evolution is summarized in Figure 6.8. First of all it is noticeable the slow convergence of the algorithm because of the elevated number of epochs needed. Also the desired error criterion was not satisfied and the algorithm stopped at 50,000 epochs (this was specified as an upper bound for the number of epoch that has to be performed). It can be seen that for this network structure this low error index cannot be attained because of the slow convergence rate shown in the last 10,000 epochs where the error converge is converging to 0.18. Also the first 20,000 epochs shown little, if any, improvement. This is due to the random initialization of the weight connections that brings some kind of inertia to the learning process.

To finalize, as expected the results are in accordance with the theory validating the implemented data structure model that simulates the multilayer perceptron network.

6.7 - Neuro-Fuzzy architectures

6.7.1 - Introduction

The term neuro-fuzzy system refers to combinations of techniques from neural networks and fuzzy systems fields. Although generally assumed to be the realization of a fuzzy system through connectionist networks, this term is also used to describe some other configurations including:

- Deriving fuzzy rules from trained Radial Basis Function (RBF) networks. Fuzzy logic based tuning of neural network training parameters.
- Fuzzy logic criteria for increasing a network size.

- Realizing fuzzy membership function through clustering algorithms in unsupervised learning in Self Organizing Maps (SOMs) and neural networks.
- Representing fuzzification, fuzzy inference and defuzzification through multi-layers feed-forward connectionist networks.

Neuro-fuzzy methods are usually applied when is required to solve a function approximation problem or where the manual design process should be supported or replaced by an automatic learning process. The manual design of a fuzzy system requires specification of various design parameters for each variable and a set of fuzzy rules. If the fuzzy system's performance is somehow not satisfactory, the structure, parameters or both must be modified accordingly. This can be a very lengthy and error-prone process that is effectively based on trial and error. In order to support this design process learning techniques based on sample data became a popular research topic at the beginning of the 1990s when fuzzy systems in the guise of fuzzy controllers first became successful and widely known. The history of neuro-fuzzy systems can be roughly structured into feed-forward systems for control and function approximation.

From the control engineering perspective the principles and architectures are similar to the neuro-fuzzy approaches for function approximation, the main difference resides in the learning mechanism. While function approximation models can use supervised learning based on a training set, controllers need to discover a model in a setting where target outputs are not known. Neuro-fuzzy controllers therefore use reinforcement learning and require either a model of or direct feedback from the process they are supposed to control. This by no means imply that architectures that uses supervised learning are inadequate for controlling purposes because they are also used as controllers as will be shown later on. Among the neuro-fuzzy architectures ARIC (Approximate Reasoning-based Intelligent Control) is one of the first neuro-fuzzy controllers and was suggested by Berenji in 1992. This model implements a fuzzy controller by using several specialized feedforward neural networks. The architecture of ARIC is similar to an adaptive critic, a special neural controller learning by reinforcement. Another important architecture from the control domain is NEFCON (Neuro-fuzzy Control) is a model for neural fuzzy controllers proposed by Nauck in 1994 [4]. This is probably the first neuro-fuzzy system that tries to introduce the notion of interpretability by preventing identical linguistic terms being represented by more than one membership function. Like ARIC, the learning algorithm for NEFCON is based on the idea of reinforcement learning but instead of an adaptive critic network it uses a fuzzy rule base to describe a fuzzy error.

Many neuro-fuzzy systems for function approximation are inspired on the Takagi-Sugeno fuzzy systems, because it is best suited for learning purposes through gradient based methods if differentiable membership function are used. ANFIS (adaptive network-based fuzzy inference system) is one of the first and still one of the popular neuro-fuzzy systems. It was proposed by Jyh-Shing Roger Jang in 1991 [11], [45], [46] and [47]. ANFIS is a neuro-fuzzy

method to determine the parameters of a Sugeno-type fuzzy model which is represented as a special feed-forward network. Although recognized as a universal approximator, the modeling capacity of the archetype has found applications on the control field as a controller or in plant model identification.

Neuro-fuzzy classification systems became more popular in the second half of the 1990s. They are a special case of function approximators and their output is typically a fuzzy classification of a pattern, i.e. a vector of membership degrees that indicates membership to different classes. With the rising interest in data mining, fuzzy classifiers became more and more important in the fuzzy system community. Some of the architectures used for this purpose are: the NNDFR Model (Neural Network Driven Fuzzy Reasoning) by Takagi and Hayashi in 1991, the neuro-fuzzy model FuNe-I proposed in 1992, NEFCLASS (1995), Fuzzy RuleNet: (1995) and the NEFPROX (Neuro-Fuzzy Function Approximator) model. More information of those models can be found at [45].

There is still ongoing research in the area of adaptive control. Good progress has been achieved in combining reinforcement learning methods with neuro-fuzzy architectures. The same holds for applications in classification that became more and more important with the growing interest in data mining. Although questions about the complete automation of the learning process and how to guarantee a certain level of interpretability remain to be important issues.

6.7.2 - Adaptive Neural-Based Fuzzy Inference System - ANFIS

As mentioned before, one of the most appealing approaches from the neuro-fuzzy field is the ANFIS, because it fuses concepts from both ANN and fuzzy systems fields in a natural way giving birth to an interesting structure with very wide range of applications. This is the selected archetype that brings the learning capacity to the FEUP Fuzzy Tool and can be easily translated to an equivalent fuzzy system. Figure 6.9 show the structure of a two input ANFIS. This ANFIS structure is equivalent to the first-order Sugeno fuzzy model introduced in Chapter 4.

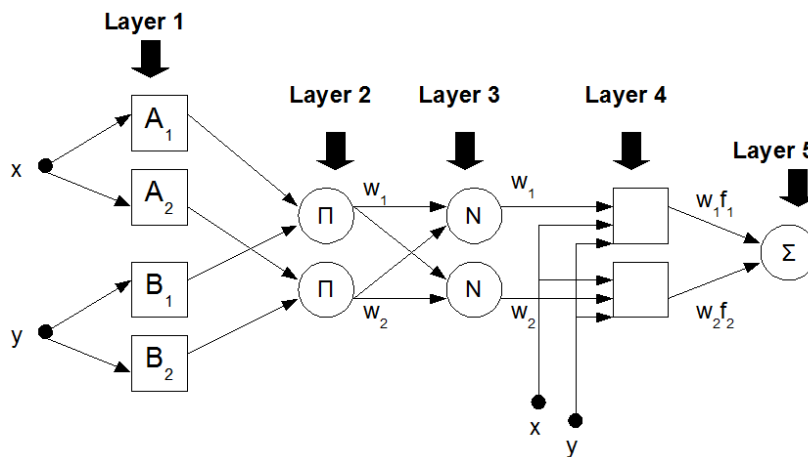


Figure 6.9 The ANFIS structure for a dual-input-single-output Sugeno Model.

The first order model was selected because it is implemented in FEUP Fuzzy Tool and it facilitates the model explanation. It is a five-layered-feedforward network, whose the nodes' function are specific for the layer function:

Layer 1: Every node i in this layer is an adaptive node with a node function

$$\begin{cases} x_{1,i} = \mu_{A_i}(x), & \text{for } i = 1,2 \\ x_{1,i} = \mu_{B_{i-2}}(x), & \text{for } i = 3,4 \end{cases} \quad (6.15)$$

where x (or y) is the input to node i , and A_i (or B_{i-2}) is a linguistic label (such as "small" or "large") associated with this node. In other words, $O_{1,i}$ is the membership grade of a fuzzy set A (A_1, A_2, B_1 or B_2) and it specifies the degree to which the given input, x or y , satisfies the quantifier A . Here the membership function for A can be any appropriate parameterized membership function. Each node is associated with a fuzzy proposition of the Sugeno model rules.

Layer 2: Every node in this layer is a fixed node labeled Π , whose output is the product of all the incoming signals

$$x_{2,i} = w_i = \mu_{A_i}(x) \cdot \mu_{B_i}(x), \text{ for } i = 1,2. \quad (6.16)$$

Each node output represents the firing strength of a rule. In general, any other T-norm operators that perform fuzzy AND can be used as the node function in this layer if multiple conjunctive antecedents are considered. For the case of considering disjunctive antecedents any S-norm operator should be used although the most common is the algebraic sum (see section 2.4.4) because of is derivable. Here it is only considered the case of multiple conjunctive antecedents as is the most encountered in practice.

Layer 3: Every node in this layer is a fixed node labeled N . The i th node calculates the ratio of the i th rule's firing strength to the sum of all rules' firing strengths. The outputs of this layer are called normalized firing strengths.

$$x_{3,i} = \bar{w}_i = \frac{w_i}{w_1 + w_2}, \text{ for } i = 1,2. \quad (6.17)$$

Layer 4: Every node represent the consequent of a rule from the sugeno model affected by the respective rule firing strength. Each node i in this layer is an adaptive node with a node function

$$x_{4,i} = \bar{w}_i f_i = \bar{w}_i (p_i x + q_i y + r_i), \quad (6.18)$$

where \bar{w}_i is a normalized firing strength from layer 3 and $\{p_i, q_i, r_i\}$ is the parameter set of this node. Parameters in this layer are referred to as consequent parameters.

Layer 5: The single node in this layer is a fixed node labeled Σ , which computes the overall output as the summation of all incoming signals

$$x_{5,i} = \sum_i \bar{w}_i f_i = \sum_i \frac{w_i f_i}{w_i}. \quad (6.19)$$

As can be seen the network performs the same function as the Sugeno model. For the Mamdani fuzzy inference system with max-min composition, a corresponding ANFIS can be constructed if discrete approximations are used to replace the integrals in the centroid defuzzification scheme introduced in Chapter 4. However, the resulting ANFIS is much more

complicated than the Sugeno ANFIS. The extra complexity in structure and computation of Mamdani ANFIS with max-min composition does not necessarily imply better learning capability or approximation power.

6.7.3 - Hybrid learning rule: combining the steepest descent method and the Least-Squares estimation

Though the gradient method can be applied to identify the parameters in an adaptive network, the method is generally slow and likely to become trapped in local minima. Instead, the hybrid learning rule which combines the gradient method and the least squares estimate (LSE) to identify parameters results in a more efficient way to train the network. The output of the ANFIS can be expressed as

$$x_{5,1} = F(\vec{I}, S), \quad (6.20)$$

where \vec{I} is the set of input variables and S is the set of parameters. Using a function H such that the composite function $H \circ F$ is linear in some of the elements of S , then these elements can be identified by the least squares method. In other words the parameter set S can be decomposed into two sets

$$S = S_1 \oplus S_2, \quad (6.21)$$

(where \oplus represents direct sum) such that $H \circ F$ is linear in the elements of S_2 , then upon applying H to (6.20) it is obtained a function which is linear in the elements of S_2 . If the values of elements of S_1 are considered fixed, we can plug P training data into (6.20) and obtain a matrix equation:

$$A\theta = y, \quad (6.22)$$

where θ is an unknown vector whose elements are parameters in S_2 with dimension $\dim(\theta) = M$, and the dimensions y and A , are $\dim(y) = Px1$, and $\dim(A) = PxM$, respectively. The number of training data pairs P is usually greater than the number of linear parameters M , which results in an overdetermined linear system of equation and does not has an exact solution. The least squares estimate (LSE) of θ , θ^* , intends to minimize the squared error $\|A\theta - y\|^2$ and whose solution uses pseudo-inverse of θ . If the columns of the matrix A are linearly independent, then the matrix $A^T A$ is non-singular and the general solution [48] is given by

$$\theta^* = (A^T A)^{-1} A^T y. \quad (6.23)$$

However, dealing with the matrix inverse which implies expensive in computation and, moreover, it becomes ill-defined if $A^T A$ is singular. One way to contour this problem is to use the sequential method of computation for the LSE. Specifically, let the i th row vector of matrix A defined in (6.22) be a^T and the i th element of y be y^T , then θ can be calculated iteratively using the sequential formulas

$$\begin{cases} P_{k+1} = P_k - \frac{P_k a_{k+1} a_{k+1}^T P_k}{1 + a_{k+1}^T P_k a_{k+1}}, \\ \theta_{k+1} = \theta_k + P_{k+1} a_{k+1} (y_{k+1} - a_{k+1}^T \theta_k) \end{cases} \quad (6.24)$$

where P_k is often called the covariance matrix and the least squares estimate θ^* is equal to θ_p . The initial conditions to bootstrap (6.24) are $\theta_0 = 0$ and $P_0 = \gamma I$, where γ is a positive large number and I is the identity matrix of dimension $M \times M$. The detailed derivation of these formulas and initial conditions is given at [11], [49] but they can also be easily obtained if the Least-Squares estimator is interpreted as the Kalman filter for the process

$$\begin{cases} \theta(k+1) = \theta(k) \\ y(k) = a^T \theta(k) + e(k) \end{cases} \quad (6.25)$$

where k is a time index, $e(k)$ is random noise, $\theta(k)$ is the state to be estimated, and $y(k)$ is the observed output.

The gradient method and the least squares estimate can be combined to update the parameters in an adaptive network. Each epoch of this hybrid learning procedure is composed of a forward pass and a backward pass. In the forward pass, it is supplied input data and functional signals go forward to calculate each node output until the matrices A and y in (6.22) are obtained, and the parameters in S_2 are identified by the sequential least squares formulas in (6.24). After identifying parameters in S_2 , the functional signals keep going forward till the error measure is calculated. In the backward pass, the error rates, i.e., the derivative of the error measure with respect to each node output, propagate from the output end toward the input end, and the parameters in S_1 are updated by the gradient method.

Given the ANFIS architecture shown at Figure 6.9, it is observed that given the values of premise parameters, the overall output can be expressed as linear combinations of the consequent parameters. More precisely, the output f in Figure 6.9 can be rewritten as

$$f = \bar{w}_1 f_1 + \bar{w}_2 f_2 = (\bar{w}_1 x) p_1 + (\bar{w}_1 y) q_1 + (\bar{w}_1) r_1 + (\bar{w}_2 x) p_2 + (\bar{w}_2 y) q_2 + (\bar{w}_2) r_2 \quad (6.26)$$

which is linear in the consequent parameters. Then the following relation can be mapped

- S = set of total parameters
- S_1 = set of premise parameters
- S_2 = set of consequent parameters

The functions H and F are the identity function and the function of the fuzzy inference system, respectively.

Therefore the hybrid learning algorithm can be applied directly. More specifically, in the forward pass of the hybrid learning algorithm, the input signals go forward till the fourth layer and the consequent parameters are identified by the least squares estimate, afterwards in the backward pass, the error rates propagate backward and the premise parameters are updated by the gradient descent.

For given fixed values of parameters in S_1 , the parameters in S_2 thus found are guaranteed to be the global optimum point in the S_2 parameter space due to the choice of the squared error measure. Not only can this hybrid learning rule decrease the dimension of the search space in the gradient method, but, in general, it will also cut down substantially the convergence time.

There are three methods to update the parameters of particular interest to implement the training algorithm for the ANFIS and are listed below according to their computation complexities:

- **Gradient Descent and One Pass of LSE:** The LSE is applied only once at the very beginning to get the initial values of the consequent parameters and then the gradient descent takes over to update all parameters.
- **Gradient descent and LSE:** This is the proposed hybrid learning rule.
- **Sequential LSE Only:** The ANFIS is linearized with respect to the premise parameters and the Kalman filter algorithm is employed to update all parameters, equation (6.24).

The FEUP Fuzzy tool as a unit (a source-code module from which an object Pascal coded program is constructed) that provides support for the computation of the RLSE given by equation (6.24). It was designed to support the hybrid training of the ANFIS.

6.7.4 - ANFIS data structure

As explained on the previous sections the ANFIS pertains to the class of feedforward networks, and like the previously implemented MLP, the ANFIS shares a common basis for the signal propagation. The only difference resides on the fact that with the MLP the parameters that can be modified during the network learning are the weight connections, whereas the on the ANFIS structure a connection between nodes, if exist, has a value equal to the unity, otherwise is zero. Therefore, the ANFIS learns or modifies its performance through changes in nodes' parameters and whose nodes pertain exclusively to the first and fourth layers (that is why these nodes are represented by square instead of a circle on Figure 6.9, to indicate they are adaptive nodes).

Beside this difference their behavior is pretty much the same in terms of data structures. The object oriented programming paradigm offers a concept to deal with this situation when two objects have similar structures and behaves differently or when some kind of hierarchic relation exist between them, that is the concept of inheritance and polymorphism.

Here the ANFIS is intended to inherit from the MLP. This way the public attributes (structure) and methods (behavior) of the MLP are passed to the ANFIS. Some of the methods of the MLP are common to the ANFIS and some are not. Those methods that define a different behavior should be "reintroduced" or redefined at the ANFIS class through the concept of polymorphism. The complete definition for the neuro-fuzzy architectures is shown in Figure 6.7 where the definition and implementation of the ANFIS can be analyzed, through the `TAnfis` class. The methods that posses a common names on both `TBPNNetwork` and `TAnfis` classes indicate the aforementioned polymorphic behavior as stated by the UML diagram class syntax. Besides the input, hidden and output layer common to the `TBPNNetwork` structure the `TAnfis` structure needs to store the fuzzy sets that defines the function for the nodes of the first layer, equation (6.15). These fuzzy sets are stored within a dynamic array of the

class `TFFSet`, in which the sets preserve one-to-one correspondence with the respective node's position within layer. Also the parameters used at the forth layer, that represent the consequence of the rules defined for the Sugeno inference model, are stored through a matrix (bi-dimensional array). It should be noted that only rules with multiple conjunctive antecedent are supported.

In terms of forward signal propagation it should be noted that the ANFIS has different activation function on its constituent neurons. The `compute_layer_output()` method is intended to substitute the `propagate_layer` method of the `TBPNNetwork`, and is an auxiliary routine that perform the forward propagation of a specific layer. It receives an integer as a parameter that represents the number of the layer that is intended to be propagated and uses the output of the previous layer to calculates the total input, then computes the result using the type of activation function that characterize the layer (equations (6.15) to (6.19)). In the process it make uses of some other auxiliary methods like the `compute_sugeno()` to complete this task. The purpose of these functions is to achieve modularity and understandability of the programming coded for future utilization. To finalize the procedure `propagate_forward()` is the actual class interface that makes possible the propagation of an input vector through the whole network. Similar to the `TBPNNetwork` the output of the `TAnfis` can be retrieved from the output node through the method. The backpropagation error routines as before are similar to those implemented at the `TBPNNetwork`.

The method `compute_error_gradient()` performs the same operation as the `compute_layer_output()` but in the backward direction, and instead of using the activation function, it passes the derivative of the error with respect to the nodes outputs as stated at the previous section and according with equations (6.6) and (6.7). Some of this functions will result in complicated formulas, and when hardcoded into the `computed_error_gradient()` will result in a non modular or understandable code. To overcome this problem, like it was done before, a series of auxiliary functions were implemented to fulfill this task, namely the methods `compute_do2_do1()`, `compute_do3_do2()`, `compute_output_error()`. Finally the procedure that will backpropagate the error from the output layer back to the first layer is `backpopagate_gradient()`.

The ANFIS makes uses of the Hybrid training algorithm explained at the previous section and the inherited training method from the `TBPNNetwork` class must be overridden. Both these methods declaration is similar in terms of passed parameters, that is the learning rate, momentum constant, desired network overall error and maximum number of epochs are passed as parameters, but it also includes another parameter of the `TTraining` type (see Figure 6.7) to specify the type of hybrid training required (according to the variations for the hybrid training algorithm). Whatever the variation every epoch in the hybrid training is

composed of two phases (except when just the LSE is applied that takes only one training epoch). The first phase, when the Least-Squares estimation takes place, the data matrix, parameter vector and output vector must be constructed with the appropriate dimensions (according to the linear matrix model from equation (6.22)) to initiate the recursive LSE algorithm. The input and output data must be loaded prior to the training, through the use of the inherited methods for loading training and checking data defined on the `TBPNNNetwork` class. Then according to the hybrid training algorithm, every input data must be propagated until the third layer, and the output of this layer, for every input pattern applied to the network, is used to construct the data (**A**) matrix. The desired output training data is used to construct directly the output vector of the least-squares model (**y**). Finally, the parameter vector is initiated with the value 0 over all its entries as stated by the RLSE algorithm. These operations are performed with the method `prepare_RLSE_data()`. When the data is ready the function that performs the RLSE algorithm, provided by the RLSE unit inside the FEUP Fuzzy Tool, is called and thus finalizes the first phase of the training epoch.

The second phase of the training is characterized by the application of the backpropagation algorithm. Once the error is back propagated all the way back to the first layer, the parameters that define every fuzzy set must be adapted accordingly. Because the fuzzy sets are defined as piecewise linear function for a particular input the sets the membership function's parameters that are needed to update depends on the value of the input to the node, see Figure 6.10. As represented at that picture the parameter p and q must be updated according to the formulas (6.7) and (6.8). The method `get_parameter_index()` is used to identify the first parameter, parameter p in the context of the picture, within the ordered array `ptX` of that particular fuzzy set. The second is automatically defined at the next position that array. Then depending on the type of the variation used for the hybrid training these two phases are repeated until the network reaches a desired performance or the maximum number of epochs is exceeded. The method `update_premise_parameters()` is used at the end of every training epoch to update the membership function parameters.

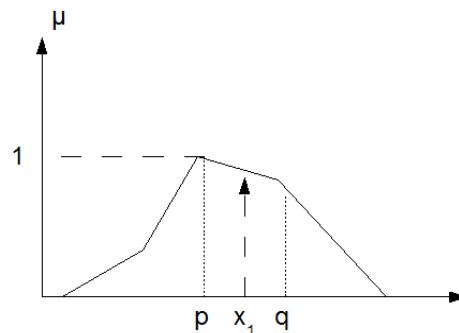


Figure 6.10 An ideal fuzzy set represented by a piecewise linear function defined by parameters p and q among others. For that particular x_1 input the membership function can be seen as a line segment.

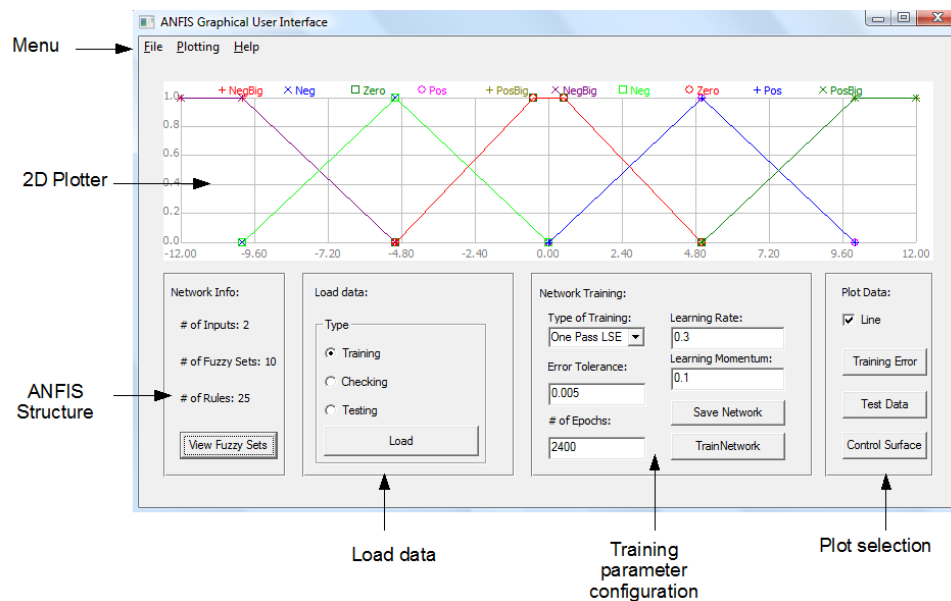


Figure 6.11 The ANFIS GUI.

As happens with the `TBPNNNetwork` class, in order to measure the network performance for the `TAnfis` during the training phase, the method `calculate_epoch_error()` computes the RMSE for the checking data set. During each epoch, this performance measure is stored within a dynamic array for later uses for statistical analysis or for graphics (plots) purposes.

6.7.5 - The ANFIS graphical user interface

A graphical user interface was created at the FEUP Fuzzy Tool to manage the ANFIS functionality and to facilitate the interaction of the user with the data structure. Almost the entire functionality of the `TAnfis` class can be accessed through this form. Figure 6.11 depicts the form. As can be seen, it offers a complete interface to interact with the ANFIS providing features like information about the network structure, loading mechanism for learning data sets, training parameterization and also the choice for the type of training, and a series of graphic routines that uses the 2D plotter or the control surface to see the ANFIS transfer characteristic according to a set of inputs. The shape of fuzzy set can be evaluated before and after the training through the 2D plotter, as well as the evolution of the network error during the training. These routines can also be accessed through the form menu at the top.

Every time the user accesses this functionality a new instance of the `TAnfis` class is created (run-time), and its lifetime and operation is extended until the user leaves the section. The class can be instantiated in two different ways. One way implies the creation of the ANFIS structure from the already implemented dual-input-single-output fuzzy system with Sugeno inference model, and is intended to tune the implemented fuzzy system through learning from an available training data set which specifies the desired performance. An internal data structure realized through a dynamic

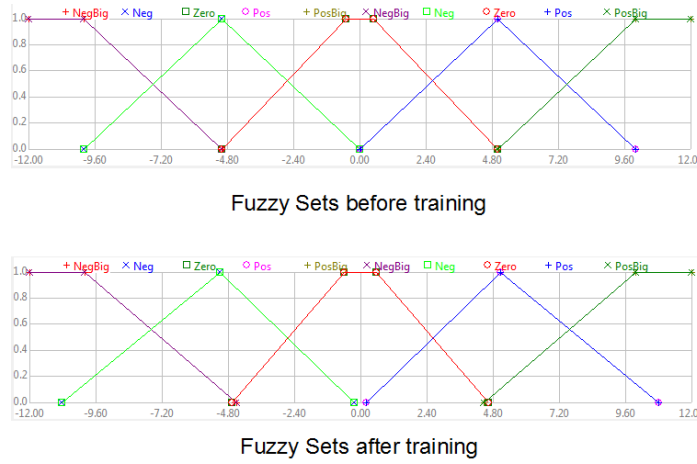


Figure 6.12 Initial and post-training distribution of fuzzy sets for the sinc function approximation

array of integers called `Rules2Delete` (private section of the `TAnfis` class) keeps a record of the relative position inside the rule base (for the Sugeno model) of the rules that resembles the equivalent ANFIS network structure. This structure is used by the `TAnfis`' method `from_Anfis_2_FFSys()` as an index to access the rule base, after the training has been completed, to create a sugeno's type fuzzy system equivalent to the trained ANFIS. This actualization procedure occurs automatically when the user request to leave the section and just before the `TAnfis` instance is destroyed. This implies that one of the purposes of this functionality is to provide support for the controller adjustment according to knowledge that is embedded in numerical data.

The second way of instantiation supports the construction of a more general structure for the ANFIS, instead of the dual-input-single-output structure described previously. The user must provide the appropriate file type, which is passed as a parameter to the class constructor. The file is then parsed, and the ANFIS structure is created according to the specification. This file defines an understandable syntax, and is intended to facilitate the process of instantiation for a multi-input-single-output ANFIS. When a network is created this way the user can save the structure for future utilization or if the intention is for programming usage. The method `save_Anfis_Structure()` perform this task and creates the same type file mentioned before. This routine can be used at the GUI by clicking the save button.

6.7.6 - Model Validation: Approximating the two-input sinc function

As a validation test, the ANFIS structure implemented on the FEUP Fuzzy Tool is used to model a two-dimensional sinc equation defined by

$$f = \text{sinc}(x, y) = \frac{\sin(x) \cdot \sin(y)}{xy} \quad (6.27)$$

From the evenly distributed grid points of the input range $[-10, 10] \times [-10, 10]$ of the preceding equation, 121 training data pairs were obtained. The ANFIS used here contains 25

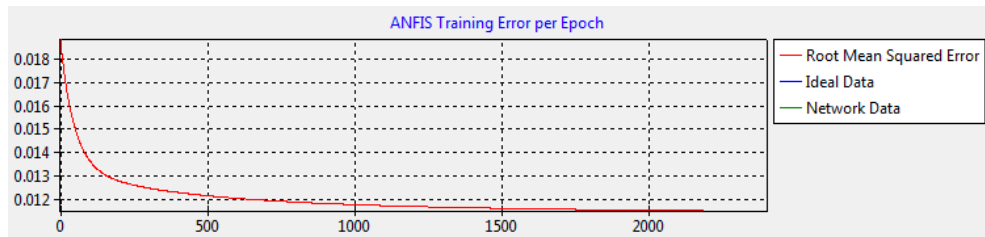


Figure 6.13 Training log for the sinc function approximation

rules, with five membership functions assigned to each input variable defined over the universe of discourse as in Figure 6.12. Note that there are ten fuzzy sets depicted and not five as one can be misled to believe because the sets from both variables are overlaid.

The training method selected was the Gradient Descent and One Pass of LSE with learning rate of 0.3, momentum constant equal to 0.1, the desired error as 0.005, and 2400 as the maximum number of epochs. Figure 6.13 shows the evolution of the training error per epoch of training. One can appreciate the drastic evolution of the error after the first epoch, where the recursive least-squares were computed, diminishing to 0.019 approximately. Also Figure 6.14 depicts the sinc approximation that is quite good but its lacks softness around the small crests. After 700 epochs the situation has improved remarkably and a better view of the function is obtained. At the end of the training, after 2400 epochs, the training error achieved the lowest value 0.01151 and one can appreciate the beauty of the approximation where the detail and symmetry of the sinc function are prevalent. The shape and distribution of the fuzzy sets after training is also shown at Figure 6.12. Once again the symmetry is latent with the distribution of the sets from both variables along their respective universes, evolving in the same manner with the gradient training. Note also that these sets preserve their previous overlaying and resemble the symmetry of the sinc function along each spatial direction. Similar results could have been achieved with only 1500 epochs as can be seen on the training log of Figure 6.13 because error evolution is very slow after this training iteration.

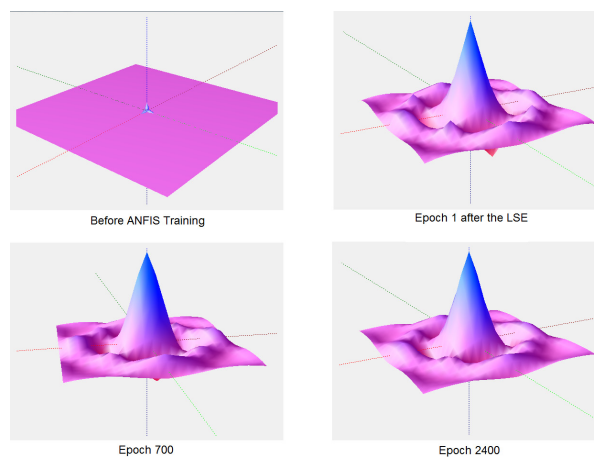


Figure 6.14 3D plot of the sinc function approximated by the ANFIS in various training epochs

This example is accessible through the main form from the GUI of the FEUP Fuzzy Tool. It validates the proposed data structure of the ANFIS implementation and the correctness of the training algorithm and updating parameter procedures and functions.

6.8 - Summary

This chapter reviews the fundamental of artificial neural network paradigm and explains the junction of this field with the fuzzy logic theory and the implementations of these structures on the FEUP Fuzzy tool in order to support learning mechanism. The backpropagation algorithm, which is based on the gradient steepest descent optimization method, was introduced and represents the basis of the supervised learning. The artificial neuron model, perceptron, was briefly introduced along with the generalization to the multilayer perceptron, a neural network that belongs to the feedforward class.

A detailed discussion took place about the requirements and the implemented model to simulate the multilayer perceptron within the FEUP Fuzzy tool. This data structure was validated through the traditional pattern recognition problem, the XOR problem.

The Adaptive Network-based Fuzzy Inference System ANFIS is an interesting approach to support learning for fuzzy systems and allows model identification through the analysis of data. This archetype was implemented at the FEUP Fuzzy Tool as a data structure that shares a common blueprint with the implementation of the multilayer perceptron. Moreover is a generalization of this data structure realized through the concept of inheritance that is familiar within the object oriented programming paradigm. The hybrid training method is employed to train this type of network. Also the basic theory behind this neuro-fuzzy model and the details about its implementation is given throughout the section. The model was validated with a function approximation problem, namely the approximation of the two-dimensional sinc function. The results were very satisfactory proving the consistency of the constructed models.

The addition of these architectures marks the FEUP Fuzzy Tool as one of the most advanced available tools that explores the advanced concepts of fuzzy logic and control.

Chapter 7

Results - Simulated Problems

The primary goal of control engineering is to distill and apply knowledge about how to control a process so that the resulting control system will reliably and safely achieve high-performance operation. The problem of control is also one of decision making, that is, given the observation of the state of the process to decide from encoded knowledge what action to take. Knowledge based systems and, in particular rule-based approaches, are ideally suited for such a decision making, task. On the one hand one may use deep knowledge, perhaps in the form of qualitative models along with qualitative reasoning for such a decision making. On the other, a simple rule-base derived from the experience of a process operator can also be used.

The fuzzy controller is one such simple rule-based control system. The original purpose of a fuzzy logic control is to mimic the behavior of a human operator able to control a complex plant satisfactorily. The complex plant in question could be a chemical reaction process, a subway train, or a traffic signal control system.

To construct a fuzzy controller, we need to perform knowledge acquisition, which takes a human operator's knowledge about how to control a system and generates a set of fuzzy if-then rules as the backbone for a fuzzy controller that behaves like the original human operator. Usually we can obtain two types of information from a human operator: linguistic information and numerical information.

- Linguistic information from an experienced human operator. It can usually summarize his or her reasoning process in arriving at final control actions or decisions as a set of fuzzy if-then rules with imprecise but roughly correct membership functions; this corresponds to the linguistic information supplied by human experts, which is obtained via a lengthy interview process plus a certain amount of trial and error.

- Numerical information acquired by a human operator when working. It is possible to record the sensor data observed by the human and the human's corresponding actions as a set of desired input-output data pairs. This data set can be used as a training data set in constructing a fuzzy controller. Prior to the emergence of neuro-fuzzy approaches, most design methods used only the linguistic information to build fuzzy controllers; this approach is not easily formalized and is more of an art than an engineering practice. Following this approach usually involves manual trial-and-error tweaking processes to fine-tune the membership functions.

Whatever the knowledge available about dynamic systems, it needs not to be expressed mathematically it can also be encoded in the form of rules. Most design methods for fuzzy controller relies on the linguistic information. This approach usually involves manual trial-and-error tweaking processes to fine-tune the membership functions. With fuzzy learning architectures, controllers can further take advantage of the numerical information for input-output data pairs and refine the controllers' performance in a systematic way. In other words, linguistic information can be used to identify the structure of a fuzzy controller, and then use numerical information to identify the parameter such that the fuzzy controller can reproduce the desired action more accurately.

This chapter intends to validate the application of the data structures that will implement knowledge-based intelligent controllers on two examples. The first example reflects the creation of a fuzzy controller based only on the knowledge available in the form of cause-effect rules, which describes the way that the system should behave when faces a specific situation. This is de kind of control problem that resembles a decision making process. The second example will implement a control strategy based on learning or adaptive controllers. The knowledge available comes in the form of numerical data.

7.1 - Fuzzy control of an autonomous mobile robot

When designing an autonomous system, two important challenges are frequently encountered. The first is related to the nonlinear nature of the control formulation, which often does not have a nice analytical formulation to work with. And the second deals with how human approach solution of such problems. Often the human experience and approach can best represented with a set of linguistic rules. Fuzzy logic controllers can mimic experts and is particularly suited for this kind of applications as discussed in the previous chapter.

This example describe the application of a knowledge base controller, namely a fuzzy controller, to control a differential-driven robot's navigation through a corridor, having the wall as a navigation landmark for absolute positioning purposes. The intention of this example is two-fold: on the one hand its serves to demonstrate the applicability of the FEUP Fuzzy Tool on real control application, and on the other hand to provide an illustrative simulation example for educational purposes.

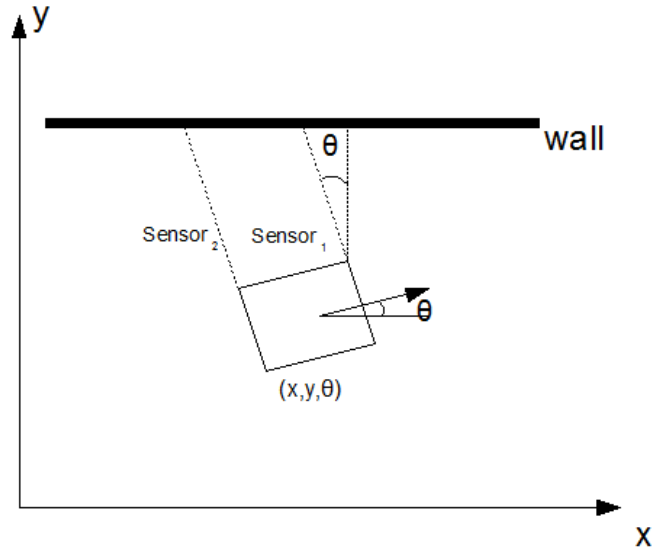


Figure 7.1 The wall following problem description and input denotations.

Figure 7.1 will result helpful in describing the problem scenario. The robot base dimensions are 0.5m for both width and length; it possesses two “ideal” sensors whose function is to measure the relative distance to the wall. These sensors are placed at same side of the robot. It can be designated to follow at any specified distance from the right or left wall. The robot scanning regions and the input parameter denotations are also shown in Figure 7.1. Only two input parameters are considered, road offset and orientation error. The sensor configuration provide two measures from which we are able to infer the distance and the orientation error that are given respectively by the equations

$$\theta = \arctan(\text{Length}_{base}, d_1 - d_2) \quad (7.1)$$

$$d_{error} = d_{ref} - \left(\frac{\text{Width}_{base} + 3d_2 - d_1}{2} \right) \quad (7.2)$$

where d_1 and d_2 are the distance measure by the sensors respectively, θ is the orientation error, d_{error} is the road offset and d_{ref} is the desired relative distance to the wall. These two equations are derived through the trigonometric analysis of the robot position and represent the input variables to the fuzzy controller. The control goal here is to align the robot parallel with the wall at a specified distance. For instance, when the robot is heading farther away from the specified distance trail toward the wall, the controller is supposed to turn the robot with a large positive degree.

The membership functions are constructed as in Figure 7.2 where normalized universes of discourse are considered and scaling factors are used inside the fuzzy controller and also act as tuning parameters. At [24] there is a detailed discussion about the tuning process of fuzzy controller's scaling factors. The optimized scaling values considered for the controller are $\pi/2$ for the orientation error input, 1 for the input distance error and 1.35 for the control output.

Table 7.1 shows the rule base of the fuzzy controller.

Table 7.1 Fuzzy rule table for the autonomous robot fuzzy controller

Orientation Error/ Distance Error	Neg	Zero	Pos
Neg	-	Out-	Out--
Zero	Out+	Out=	Out-
Pos	Out++	Out+	-

As can be seen this rule is obtained through knowledge that comes from intuition about the action that must be taken in a given situation, resembling a decision making problem as mentioned before.

The simulation is based on the kinematic description of a differential-driven robot [50] without considering the dynamics involved in the speed response. Moreover, the linear velocity is considered to be constant and equal to 0.55m/s, as seems to be a reasonable value, and the angular velocity is proportional to the control signal.

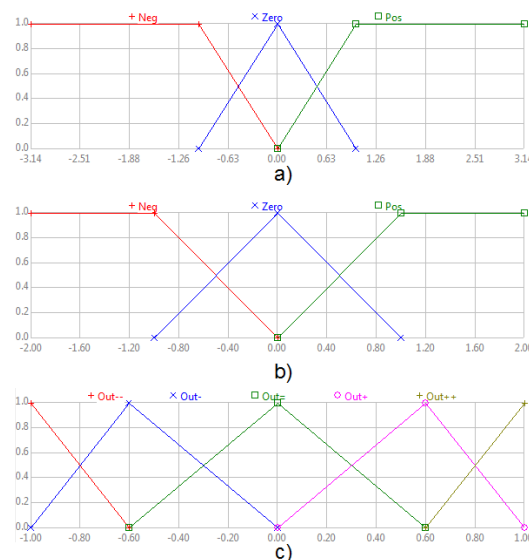


Figure 7.2 Fuzzy sets configuration for the autonomous robot fuzzy controller

Figure 7.3 shows the simulation environment inside the FEUP Fuzzy Tool's GUI. The consistency of the fuzzy controller is demonstrated at the upper left corner where a 2D scenario is used to show the robot behavior. This particular situation where there is a sudden change in the wall structure was intentionally chosen to test the robot behavior with a drastic change of the reference landmark. Note that only seven fuzzy rules were used to obtain this performance. Also the design time of the controller is dramatically reduced using the fuzzy logic approach because the source of knowledge easily lends itself to be translated in the form of rules.

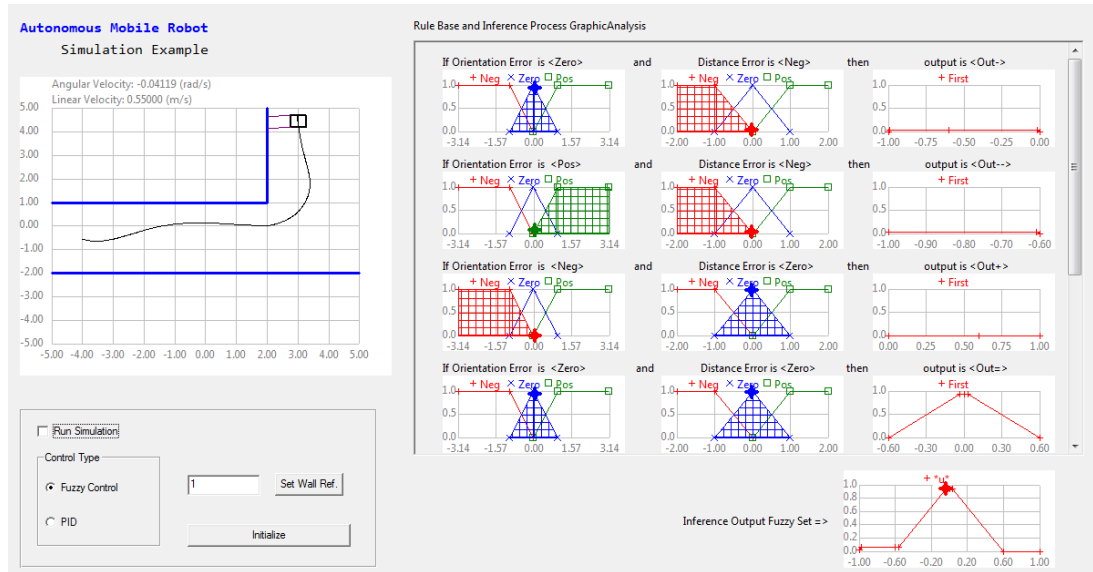


Figure 7.3 Simulation environment for the autonomous robot control.

The inference process can be analyzed at every instant inside the GUI as shown at the right-hand side of Figure 7.3. The Inference process can be completely evaluated because the results of every single rule and the whole rule base are addressed here for a particular set of inputs. The example is intended as an educative approach to illustrate the controller's action at every instant and gain a better understanding about the fuzzy control process. It will help students to have a nicely first contact with the fuzzy logic control approach.

7.2 - Neuro-Fuzzy control of a nonlinear plant

This section intends to provide an example of the control of a nonlinear plant described by the difference equation

$$y(k+1) = \frac{y(k)u(k)}{1+y(k)^2} - \tan(u(k)). \quad (7.3)$$

This problem was inspired from [11]. The state and control action are respectively denoted by $y(k)$ and $u(k)$. As can be seen the nonlinear nature of the plant makes the controlling design truly a challenge, and traditional methods like root locus analysis for compensator design or state feedback, etc. fails to provide a basic structure to control such complicated system. It is assumed the dynamics of the plant to be unknown and the goal is to use the inverse learning approach with an ANFIS controller to control this system. This example is available at the GUI for teaching purposes.

7.2.1 - Inverse Learning

The inverse learning as a controller design methodology that composed by a learning phase, where an adaptive estimation technique is used to model the inverse dynamics of the plant. The obtained neuro-fuzzy model, which represents the inverse dynamics of the plant, is then used to generate control actions in the application phase. These two phases can

proceed simultaneously; hence this design method fits perfectly in the classical adaptive control scheme.

Let's assume that the number of state variables is known and all state variables are measurable and that the system satisfies the condition of reachability and controllability [48], [51]. Under these assumptions the dynamic of the plant is given by an equation of the form

$$x(k+1) = f(x(k), u(k)), \quad (7.4)$$

where $x(k+1)$ is the state at time $k+1$, $x(k)$ is the state at time k , and $u(k)$ is the control signal at time k , that for simplicity sake is assumed to be a scalar quantity. Similarly, the state at time $k+2$ is expressed as

$$x(k+2) = f(x(k+1), u(k+1)) = f(f(x(k), u(k)), u(k+1)), \quad (7.5)$$

In a general form this composition of functions can be translated to

$$x(k+n) = F(x(k), U), \quad (7.6)$$

where n is the order of the plant, F is a multiple composite function of f , and U is the control actions from k to $k+n-1$, which is equal to $[u(k), u(k+1), \dots, u(k+n-1)]^T$. The previous equation stresses the fact that it takes only n steps for the system to be transferred from state $x(k)$ to $x(k+n)$, given the previous assumptions about controllability and reachability.

We will also assume that the plant is a minimum phase system, that is, the inverse dynamics of the plant do exist, and U can be expressed as an explicit function of $x(k)$ and $x(k+n)$.

$$U = G(x(k), x(k+n)). \quad (7.7)$$

This equation essentially says that there exists a unique input sequence U , specified by the mapping G , that can drive the plant from state $x(k)$ to $x(k+n)$ in n time steps.

If the system is linear, i.e., the dynamics of the system is given by $x(k+1) = Ax(k) + Bu(k)$ then the state $x(k+n)$ is obtained by the equation

$$x(k+n) = A^n x(k) + WU \quad (7.8)$$

where $W = [A^{n-1}B \dots AB \ B]$ is the controllability matrix. The derivation of the preceding equation can be found at [48], [51]. Given the assumption of controllability, every state transformation has a unique control sequence U , which can be calculated as

$$U = W^{-1}[x(k+n) - A^n x(k)]. \quad (7.9)$$

Although the inverse mapping G in Equation (7.7) exists by assumption, it does not always have an analytically close form. Therefore, instead of looking for methods of solving Equation (7.7) explicitly, an adaptive network or ANFIS with $2n$ inputs and n outputs can be used to approximate the inverse mapping G according to the generic training data pairs $[x(k)^T, x(k+n)^T; U^T]$. When the adaptive network approximately imitates the input-output mapping of the inverse dynamics G , then given the current state $x(k)$ and the desired future state $x_d(k+n)$, the adaptive network will generate an estimated \hat{U} .

After n steps, this control sequence can bring the state $x(k)$ to the desired state $x_d(k+n)$. If the future desired state $x_d(k+n)$ is not available in advance, the current desired state $x_d(k)$

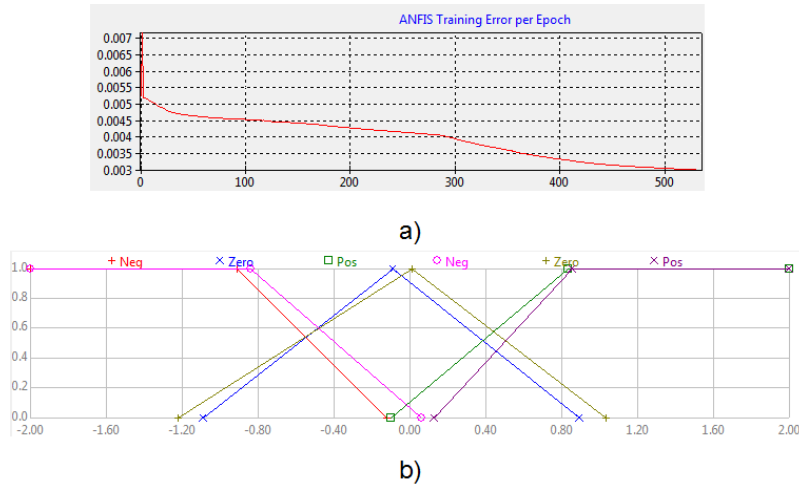


Figure 7.4 Training log (a), pre- and post-training configuration of fuzzy sets (b) for the inverse learning problem.

can be used instead. This implies that the current desired state will appear after n time steps and the whole system behaves like a pure n -step time delay system. When the approximation of G is not close to the original mapping, the control sequence \hat{U} cannot bring the state to $x_d(k+n)$ in exactly the next n time step.

The rationale behind inverse learning assumes the existence of inverse dynamics for a plant, which is not generally valid. Furthermore, the minimization of the network error $\|U - \hat{U}\|^2$ does not guarantee minimization of the overall system error $\|x(k) - x_d(k)\|^2$.

Retaking our previous conversation about controlling the dynamic system expressed by equation (7.3) through the ANFIS controller, the first task at hand is to teach the ANFIS the inverse mapping of this system. Each of the inputs has three fuzzy sets uniformly distributed over the interval $[-1; 1]$ and together 9 rules used to specify the ANFIS structure. The collected data was based on a computer programs that will generate a hundred training data pairs of the form $[y(k), y(k+1); u(k)]$, where the inputs $u(k)$, $k = 1$ to 99, are uniformly distributed random numbers between -1 and 1. The other components were obtained systematically by applying the sequence $u(k)$ in the equation (7.3) with the initial condition of $y(0)=0$.

The ANFIS then was training with the pure hybrid training, with learning rate of 0.3, momentum constant of 0.1 and a desired error of 0.003. A little more than 500 epochs elapsed to achieve the disered performance At Figure 7.4 can be seen the network performance evolution and the distribution of fuzzy sets after training. Figure 7.5 shows the controller's transfer function.

It should be noted that the distribution of the training data pose a problem. Ideally, is desirable that that the training data distributes across the input space of the controller in a somewhat uniform manner.

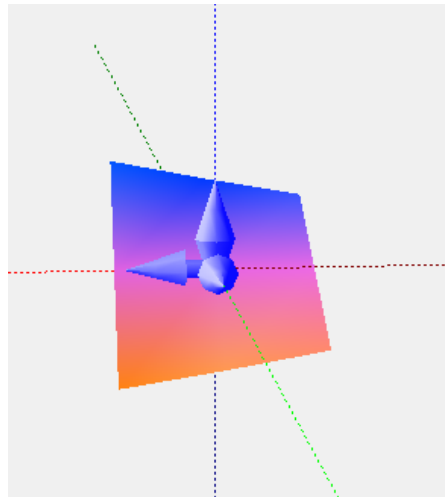


Figure 7.5 Transfer characteristic of the neuro-fuzzy controller.

However, this may not be possible due either to the scarcity of the data in some cases. The response of the controlled system when asked to follow the $\cos(t)$ reference function, for 5 seconds of simulation with a sampling period of 100 ms, can be observed at Figure 7.6.

Some conclusions can be addressed through this picture. The performance of the controller is outstanding given its simplicity (equivalent to a Sugeno model with nine rules). Note that, at a given time, the state error is never bigger in magnitude than 0.03. Also note from Figure 7.7 (which is a magnified view of the system response) that the system behaves truly as a pure delay system with a delay approximately of 100ms between the desired and real state. This is because the current desired state given by the function $\cos(t)$ that is used as a reference and it takes 1 time step, according with (7.6) to reach the desired state.

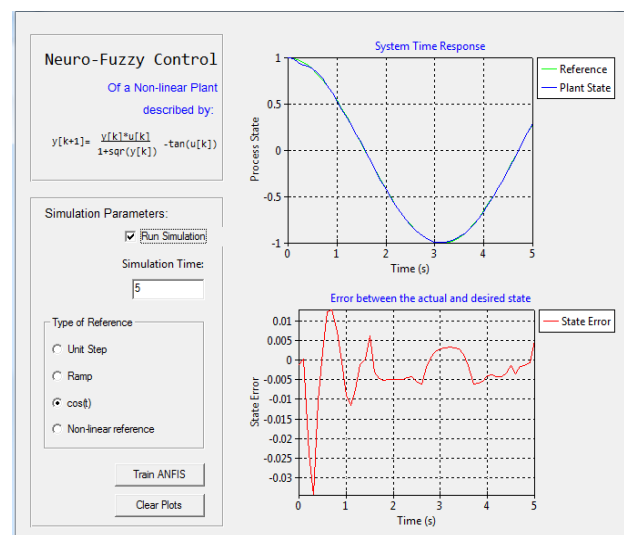


Figure 7.6 Simulation environment nonlinear System + ANFIS controller with a $\cos(t)$ reference.

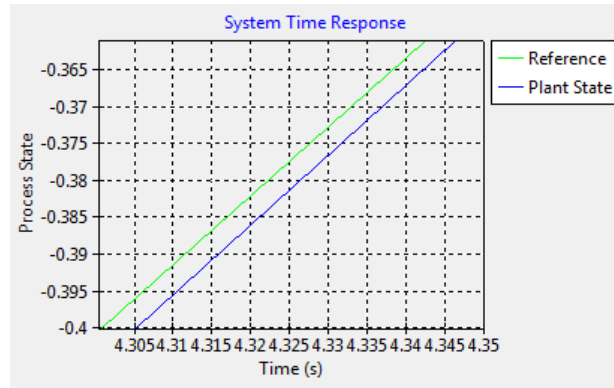


Figure 7.7 Zoom of the plot in Figure 7.6 to visualize the systems as a pure one step delay system.

This example validates the application of the `TAnfis` class onto the control filed and shows its usefulness as an adaptive structure. The main advantage of the method relies on the fact that it is not needed to know the dynamic of the system in advance; its identification is embedded at the ANFIS training.

7.3 - Summary

This chapter summarizes the most important aspects about the utilization of the FEUP Fuzzy Tool as a neuro-fuzzy controller. The first application was as a controller of the behavior of a differential-driven robot in the wall following problem inside a corridor. The controller is created through knowledge that comes in terms of rules and is embedded as a structured cause-effect relation like the fuzzy if-then statement. The controller's design process is deeply facilitated with the fuzzy logic approach.

The other example uses an adaptive network-based structure like ANFIS to control the nonlinear dynamics of a process. Traditional methods fail to support the controller's design mainly because they are based on analytical methods that assume the availability of a mathematical model of the process, and the only knowledge available about the process comes in terms of numerical information. The ANFIS controller demonstrates its supremacy in this situation. Moreover, the inverse learning method is particularly suited for adaptive nonlinear control and outstanding results have been attained with its application in this matter.

Chapter 8

Conclusion and Future work

FEUP Fuzzy Tool is an on-going project whose main purpose is to provide a toolbox for fuzzy systems, and it has been used as an educational tool for introducing the concepts of fuzzy set theory over the last years at the Faculty of Engineering of the University of Oporto, Portugal. This tool also allows an introduction to intelligent control systems based on linguistic variables that are very interesting to use because they reduce the time to produce a system controller. The tool is usable in a GUI mode to produce and edit fuzzy control rules and as a controller that optionally produces visual feedback of the controller variables and associated decision process.

In spite of the potential that comes with the toolbox, by the time this dissertation started two important requirements were of primary concern, namely the portability and learning issues. The majority of software tools implements the basic concepts about fuzzy logic and fuzzy control but lacks the support of more advanced concepts that result useful for the majority of the application.

The more important practical aspects relative to the development of these functionalities are summarized next:

- Conception of an optimized method of fuzzy inferencing for embedded application, the results of this method is approximate; however the approximation is only limited by hardware constraints.
- Development of an automatic code generation mechanism for embedded system.
- Design of an editor that supports C code syntax highlighting as a graphical interface for the automatically generated programming code for embedded systems.
- Implementation of a data structure that simulates the Backpropagation Multilayer Perceptron. The training routine is the backpropagation algorithm with momentum. The pattern recognition capability of this structure can also be used, together with clustering methods, to identify or create membership functions for

systems where it is desired to classify data sets of inputs and corresponding outputs, and the relationship between them may be highly nonlinear or not known at all.

- Implementation of the ANFIS neuro-fuzzy model to support learning of fuzzy systems. This model is built from a junction of the data structures that comes from the previously implemented fuzzy core engine with the MLP neural network. Three variation of the hybrid training are supported and it has been demonstrated experimentally its supremacy over the backpropagation algorithm.
- Development of a graphical interface to support the interaction with the ANFIS data structure.

The implemented programming code is intended to preserve the main internal factors that affect software quality: understandability, reusability, modularity, verifiability (or easy to test) and maintainability. It should be noted that all the developed software was experimentally tested through its application in a variety of examples and simulation environments. The most important ones or that have an educational relevance were included at the GUI.

The requirements were successfully and completely satisfied. The FEUP Fuzzy Tool has now found privileged position compared the current state of the art fuzzy logic software. Moreover, the introduction of the neural network paradigm brings lots of opportunities for future development and broadens the horizons for the library application. Therefore, there is a convergence to transform the fuzzy toolbox to a soft computing toolbox, although as this moment, much work needs to be devoted to fulfill this aspiration. Probably future versions of the library will reveal themselves to be a more clear sight of this ideal.

From an educational view, it offers aid to the learning process addressing the main concepts of the soft-computing through a variety of examples that comes together as an extension to the toolbox. Also provides a user friendly GUI and is built under the object oriented programming paradigm which supports a high level of abstraction for ease of utilization.

This work is essentially characterized by the junction of a multidisciplinary environment because it explores different areas that ranges from the soft-computing and artificial intelligence, to the computer science and software engineering with applications in the fields of control engineering, robotics, pattern recognition, image processing, system identification and adaptive signal processing. These topics were somehow explored throughout the document.

8.1 - Future Trends

When it comes to future developments there is a wide horizon of possible choices, primary because the vision about the toolbox starts to shift the ideology of being purely fuzzy logic software to enter in a more general framework called soft computing.

There are improvements already planned on the fuzzy core engine, particularly the inclusion of some other important defuzzification methods, T-norm and S-norm operators for fuzzy set algebra. Also there is an increasing interest with implementing a communication protocol based on an industrial *de-facto* standard, the Modbus over the Ethernet, in order to enlarge the number of compatible hardware platforms supported.

One interesting idea to enrich the portability issue would be the ability to handle files that adhere to the IEC 61131-7 standard, which is a part of the standard for Programmable logic controllers (PLC) published by the International Electrotechnical Commission (IEC). This part represents the specifications for the Fuzzy Control Language (FCL) syntax. Programmable logic controllers have become dominant control devices in process automation. Capable of operating as standalone devices or elements of distributed control system networks, PLCs represent a very good basis for various control solutions. The portability of the FEUP Fuzzy Tool to a PLC bring further success to the toolbox on both the industrial and the educational areas as it can be used many other disciplines, especially those that belongs to the automation area.

When it come to the network architectures, other more efficient training methods, like the quick propagation algorithm or the Levenberg-Marquardt method can be implemented for the MLP data structure. The implementation of other ANN structures is also critical, namely the self-organizing Kohonen maps. In terms of the ANFIS, support for multiple disjunctive antecedents must be implemented, especially when the S-norm operator used is the algebraic sum. Also the MIMO extension of the ANFIS architecture, called Coactive ANFIS or simply CANFIS [11], would result in an interesting approach that further fuses the Fuzzy and ANN approach. The foundations for this implementation are already present in the FEUP Fuzzy Tool classes.

References

- [1] Lopes F., Ferreira A., Sousa A. "New Fuzzy Logic Library - Implementation of a Fuzzy Logic Controller on a PC under Soft Real Time Constraints", *3rd International Conference on Hands-on Science: Science Education and Sustainable Development*, 2006, pp. 83-89.
- [2] L. A. Zadeh, "Fuzzy sets", *Information and Control* 8, 1965, pp. 338-353.
- [3] Eklund P., "Fuzzy Logic in Northern Europe: Industrial Applications and Software Developments", *IEEE World Congress on Computational Intelligence*, vol.2, June 1994, pp. 712-715.
- [4] Nauck, D., "GNU Fuzzy", *IEEE International Conference on Fuzzy Systems*, July 2007, pp. 1-6.
- [5] H. Kiendl, M. Krabs, T. Scheel, "Fuzzy Control in Germany: A Survey", *IEEE World Congress on Computational Intelligence*, vol.2, June 1994, pp. 716-720.
- [6] Magdalena, L., "Soft computing for students and for society [Technology Review]", *IEEE Computational Intelligence Magazine*, vol.4, pp. 47-50, February 2009.
- [7] Pegoraro E., Sousa A., "FEUP Fuzzy Tool - Renewed and Extended for Intelligent Control and Education", *CISTI 2009 - 4º Conferência Ibérica de Sistemas e Tecnologias da Informação*, April 2009.
- [8] Driankov D., Hellendoorn H., Reinfrank M., *An Introduction to Fuzzy Control*, 2nd Edition, Springer-Verlag, Berlin 1996.
- [9] Kovacic Z., Bogdan S., *Fuzzy Controller Design: Theory and Applications*, Taylor & Francis Group, FL 2006.
- [10] Ross T., *Fuzzy Logic with Engineering applications*, 2nd Edition, John Wiley & Sons Ltd, Chichester: 2004.
- [11] Jang R., Sun T., Mizutani E., *Neuro-Fuzzy and Soft Computing: A computational approach to learning and machine intelligence*, Prentice Hall, New Jersey: 1997.
- [12] Mamdani E. H., Assilian S., "An experiment in linguistic synthesis with a fuzzy logic controller", *International Journal of Man-Machine Studies*, Vol. 7, 1975, pp. 1-13.

- [13]Free Fuzzy Logic Library. Online Through <http://ffll.sourceforge.net/index.html>. Accessed in June 2009.
- [14]Togai InfraLogic, Inc. Online Through <http://www.ortech-engr.com/fuzzy/togai.html>. Accessed in June 2009.
- [15]FuzzyCLIPS. Online Through <http://www.ortech-engr.com/fuzzy/fzyclips.html>. Accessed in June 2009.
- [16]StarFLIP++. Online Through <http://www.dbai.tuwien.ac.at/proj/StarFLIP/>. Accessed in June 2009.
- [17]NRC FuzzyJ Toolkit for the Java(tm) Platform. Online Through <http://www.cs.vu.nl/~ksprac/2002/doc/fuzzyJDocs/index.html>. Accessed in June 2009.
- [18]Lawrence O. Hall, Richard J. Hathaway, "Software Review", in *IEEE Transactions on Fuzzy Systems*, vol. 4, February 1996, pp. 82-85.
- [19]Fuzzy Logic Toolbox - MATLAB. Online through <http://www.mathworks.com/products/fuzzylogic/>. Accessed in June 2009.
- [20]fuzzyTECH. Online through <http://www.fuzzytech.com/>. Accessed in June 2009.
- [21]Feng Chuan, Sun Zengqi, Su Ling, "Design and Implementation of Scilab Fuzzy Logic Toolbox", *IEEE International Symposium on Computer Aided Control Systems Design*, vol.4, September 2004, pp. 147-151.
- [22]Scilab - fuzzy logic Toolbox. Online Through http://www.scilab.org/contrib/index_contrib.php?page=displayContribution&fileID=249. Accessed in June 2009.
- [23]Wolfram Mathematica Fuzzy Logic Toolbox. Online Through <http://www.wolfram.com/products/applications/fuzzylogic/>. Accessed in June 2009.
- [24]D. R. López, et al., "XFUZZY: a design environment for fuzzy systems", *IEEE International Conference on Fuzzy Systems*, Anchorage - Alaska, May 1998, pp. 1060-1065.
- [25]Xfuzzy Home Page. Online Through <http://www.imse.cnm.es/Xfuzzy/>. Accessed in June 2009.
- [26]Fide - Fuzzy Inference Development Environment. Online Through <http://www.aptronix.com/fide/fide.htm>. Accessed in June 2009.
- [27]Lazarus Wiki. Online through <http://wiki.lazarus.freepascal.org/>. Accessed on June 2009.
- [28]GLScene for Lazarus. Online through <http://wiki.lazarus.freepascal.org/GLScene>. Accessed on June 2009.

- [29]Glebovsky E., Ivanov V., Karpov Y., "Computer Modeling and Simulation in Higher Education", *APRU Distance Learning and the Internet 2006 Conference*, 2006, pp. 201-210.
- [30]Mamdani E.H., "Twenty years of Fuzzy Control: Experiences Gained and Lessons Learnt", *Second IEEE International Conference on Fuzzy Systems*, vol.1, 1993, pp. 339-344.
- [31]Kuo-Yang Tu, Tsu-Tian Lee, Wen-Jieh Wang, "Design of a multi-layer fuzzy logic controller for multi-input multi-output systems", *Fuzzy Sets and Systems* 111, 2000, pp. 199-214.
- [32]Chopra S., Mitra R., Kumar V., "Neural Network Tuned Fuzzy Controller for MIMO System", *International Journal of Intelligent Technology*, vol. 2, November 2006, pp. 78-85.
- [33]M. Sugeno, T. Takagi, "Fuzzy identification of systems and its applications to modeling and control", *IEEE Transactions on Systems, Man, and Cybernetics*, 1985, pp. 116- 132.
- [34]Yamakawa T., "A Fuzzy Inference Engine in Nonlinear Analog Mode and Its Application to a Fuzzy Logic Control", *IEEE Transactions on Neural Networks*, vol. 4, May 1993, pp. 496-522.
- [35]Giacalone B., Lo Presti M., Di Marco F., "Hardware Implementation Versus Software Emulation of Fuzzy Algorithms in Real Applications", *IEEE World Congress on Computational Intelligence*, vol. 4, May 1998, pp 7-12.
- [36]Han-Xiong Li, Gatland H. B., "A New Methodology for Designing a Fuzzy Logic Controller", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, SYSTEMS, MAN, March 1995, pp. 505-512.
- [37]BloodShed Dev++. Online through <http://www.bloodshed.net/devcpp.html>. Accessed on June 2009.
- [38]Dhananjay V. Gadre, *Programming and customizing the AVR microcontroller*, McGraw-Hill, USA 2001.
- [39]Atmel Products - Tools & Softwarw Online through http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725. Accessed on June 2009.
- [40]Tunstel E., Jamshidi M., "On Embedded Fuzzy Controllers", *1st World Automation Congress*, Maui, August 1994, pp 14-17.
- [41]Coimbra de Matos A., *Apontamentos de Análise Numérica*, Faculdade de Engenharia da Universidade do Porto, Porto, Setembro de 2005.
- [42]Passino K., Yurkovich S., *Fuzzy Control*, Addison-Wesley Longman, California: 1998, chapter 5.

- [43]Haykin s., *Neural Networks: a comprehensive foundation*, 2nd Edition, Prentice Hall, Delhi 2005.
- [44]Freeman J., Skapura D., *Neural Networks Algorithms, Applications and Programming Techniques*, Addison-Wesley, USA 1991.
- [45]Nauck D., Nurnberger A., "The Evolution of Neuro-Fuzzy Systems" *Annual Meeting of the North American Fuzzy Information Processing Society*, June 2005, pp. 98-103.
- [46]Jang R., "ANFIS: Adaptive-Network-Based Fuzzy Inference System", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 23, May/June 1993, pp 665-685.
- [47]Jang R., Chuen-Tsai Sun, "Neuro-Fuzzy Modeling and Control", *Proceedings of the IEEE*, vol. 83, March 1995, pp. 378-406.
- [48]Katayama T., *Subspace Methods for System Identification*, Springer-Verlag, London 2005.
- [49]Ljung L., *System Identification: Theory for the user*, Prentice Hall, New Jersey 1987.
- [50]Siegwart R., Nourbakhsh I., *Introduction to Autonomous Mobile Robots*, MIT Press, Massachusetts 2004.
- [51]Ogata K., *Discrete-Time Control Systems*, 2nd Edition, Prentice Hall, New Jersey 1995.
- [52]Ng K.C., Trivedi M., "A Neuro-Fuzzy Controller for Mobile Robot Navigation and Multirobot Convoying", *IEEE Transactions on Systems, Man, and Cybernetics*, vol.28, December 1998, pp. 829-840.
- [53]Borenstein1 J., Everett2H. R., Feng L. (April 1996). *Where am I? Sensors and Methods for Mobile Robot Positioning* [Online]. Available: <http://www.eng.yale.edu/ee-labs/morse/main.htm>.
- [54]Hess J., "Applications of Fuzzy Logic and Software Development in Switzerland", *IEEE World Congress on Computational Intelligence*, vol.2, June 1994, pp. 721-726.