jFuzzyLogic: a Java Library to Design Fuzzy Logic Controllers According to the Standard for Fuzzy Control Programming

Pablo Cingolani [1], Jesús Alcalá-Fdez [2]

[1] School of Computer Science, McGill University,
McConnell Engineering Bldg, Room 318,
Montreal, Quebec, H3A-1A4, Canada
E-mail: pablo.cingolani@mcgill.ca

[2] Department of Computer Science and Artificial Intelligence,
Research Center on Information and Communications Technology (CITIC-UGR),
University of Granada, Granada, 18071, Spain
E-mail: jalcala@decsai.ugr.es

Received 13 December 2012
Accepted 1 March 2013

Abstract

Fuzzy Logic Controllers are a specific model of Fuzzy Rule Based Systems suitable for engineering applications for which classic control strategies do not achieve good results or for when it is too difficult to obtain a mathematical model. Recently, the International Electrotechnical Commission has published a standard for fuzzy control programming in part 7 of the IEC 61131 norm in order to offer a well defined common understanding of the basic means with which to integrate fuzzy control applications in control systems. In this paper, we introduce an open source Java library called jFuzzyLogic which offers a fully functional and complete implementation of a fuzzy inference system according to this standard, providing a programming interface and Eclipse plugin to easily write and test code for fuzzy control applications. A case study is given to illustrate the use of jFuzzyLogic.

Keywords: Fuzzy Logic Control, Fuzzy Control Language, Fuzzy Logic, IEC 61131-7, Open Source Software, Java Library

1. Introduction

Expert Control is a field of Artificial Intelligence that has become a research topic in the domain of system control.

Fuzzy Logic Control [1,2,3] is one of the topics within Expert Control which allows us to enhance the capabilities of industrial automation. It is suitable for engineering applications in which classical control strategies do not achieve good results or when it is too difficult to obtain a mathematical model. Fuzzy Logic Controllers (FLCs) are a specific model of Fuzzy Rule Based Systems (FRBSs) that provide a tool which can convert the linguistic control strategy based on expert knowledge into an automatic control strategy. They usually have two characteristics: the need for human operator experience, and a strong non-linearity. Nowadays, there are many real-world applications of FLCs such as mobile robot navigation [4,5], air conditioning controllers [6,7], domotic control[8,9], and industrial applications[10,11,12].

FLCs are powerful when solving a wide range of problems, but their implementation requires a certain programming expertise. In the last few years, many fuzzy logic software tools have been developed to minimise this requirement. Although many of them are commercially distributed, for example MATLAB Fuzzy logic toolbox[a], a few are available as open source software. Open source tools can play an important role as is pointed out in [13].

Recently, the International Electrotechnical Commission published the 61131 norm (IEC 61131) [14], which has become well known for defining the Programmable Controller Languages (PLC) and is commonly used in industrial applications. In part 7 (IEC 61131-7) of this norm Fuzzy Control Language (FCL) is defined, offering common understanding of the basic means with which to integrate fuzzy control applications in control systems and providing a common language with which to exchange portable fuzzy control programs among different platforms. This standard has a world-wide diffusion and is independent of systems manufactures, which has many advantages: easy migration to and from several hardware platforms from different manufacturers; protection of investment at both the training and application-level; conformity with the requirements of the Machinery Directive EN60204; and reusability of the developed application.

In this paper, we present an open source Java library called jFuzzyLogic[b] which allows us to design and to develop FLCs following the standard for FCL. jFuzzyLogic offers a fully functional and complete implementation of a fuzzy inference system (FIS), and provides a programming interface (API) and an Eclipse plugin in order to make it easier to write and test FCL code. This library brings the benefits of open source software and standardization to the fuzzy systems community, which has several advantages:

- Standardization reduces programming work. This library contains the basic programming elements for the standard IEC 61131-7, removing the need for developers to attend to boiler plate programming tasks.
- This library extends the range of possible users applying FLCs. This provides a complete implementation of FIS following the standard for FCL, reducing the level of knowledge and experience in fuzzy logic control required of researchers. As a result researchers with less knowledge will be able to successfully apply FLCs to their problems when using this library.
- The strict object-oriented approach, together with the modular design used for this library, allows developers to extend it easily.
- jFuzzyLogic follows a platform-independent approach, which enables it to be developed and run on any hardware and operating system configuration that supports Java.

This paper is arranged as follows. The next section introduces the basic definitions of the FLCs and the published PLCs in the IEC 61131 norm. In Section 3 we review some non-commercial fuzzy softwares and the main benefits that the jFuzzyLogic offers with respect to other softwares. Section 4 presents jFuzzyLogic: its main features and components. In Section 5, a FLC is used in order to illustrate how jFuzzyLogic can be used in a control application. Finally, Section 6 draws some conclusions and indicates future work.

2. Preliminaries

In this section, we first introduce the basic definitions of the FLCs, and then we present the published PLCs by the IEC 61131 norm.

2.1. Fuzzy logic controller

FLCs, as initiated by Mamdani and Assilian [15,16], are currently considered to be one of the most important applications of the fuzzy set theory proposed by Zadeh [17]. This theory is based on the notion of the fuzzy set as a generalization of the ordinary set characterized by a membership function μ that takes values from the interval [0, 1] representing degrees of membership in the set. FLCs typically define a

[a] `http://www.mathworks.com`

[b] `http://jfuzzylogic.sourceforge.net`

non-linear mapping from the system's state space to the control space. Thus, it is possible to consider the output of a FLC as a non-linear control surface reflecting the process of the operator's prior knowledge.

A FLC is a kind of FRBS which is composed of: A Knowledge Base (KB) that comprises the information used by the expert operator in the form of linguistic control rules; a Fuzzification Interface, which transforms the crisp values of the input variables into fuzzy sets that will be used in the fuzzy inference process; an Inference System that uses the fuzzy values from the Fuzzification Interface and the information from the KB to perform the reasoning process; and the Defuzzification Interface, which takes the fuzzy action from the inference process and translates it into crisp values for the control variables. Figure 1 shows the generic structure of a FLC.

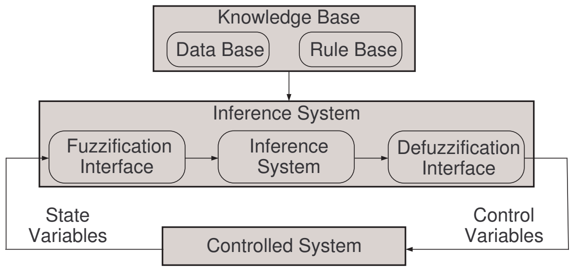


Fig. 1. Generic structure of a FLC.

The KB encodes the expert knowledge by means of a set of fuzzy control rules. A fuzzy control rule is a conditional statement in which the antecedent is a condition in its application domain, the consequent is a control action to be applied in the controlled system and both, antecedent and consequent, are associated with fuzzy concepts; that is, linguistic terms. The KB includes two components: the Data Base (DB) and the Rule Base (RB). The DB contains the definitions of the linguistic labels; that is, the membership functions for the fuzzy sets. The RB is a collection of fuzzy control rules, comprised by the linguistic labels, representing the expert knowledge of the controlled system.

The Fuzzification Interface establishes a mapping between each crisp value of the input variable and a fuzzy set defined in the universe of the corresponding variable. This interface works as follows:

$$A' = F(x_0)$$

where x_0 is a crisp value defined in the input universe U, A' is a fuzzy set defined in the same universe and F is a fuzzifier operator.

The Inference System is based on the application of the Generalized Modus Ponens, an extension of the classical Modus Ponens, proposed by Zadeh in which:

$$\begin{array}{l} \text{If } X \text{ is } A \text{ then } Y \text{ is } B \\ X \text{ is } A' \\ \hline Y \text{ is } B' \end{array}$$

where X and Y are linguistic variables, A and B are fuzzy sets, and B' is the output fuzzy set inferred. To do this, the system firstly obtains the degree of matching of each rule by applying a conjunctive operator (called an aggregation operator in the IEC-61131-7 norm), and then infers the output fuzzy sets by means of a fuzzy implication operator (called activation operator in the IEC-61131-7 norm). The Inference System produces the same amount of output fuzzy sets as the number of rules collected in the KB. These groups of fuzzy sets are aggregated by an aggregation operator (called an accumulation operator in the IEC-61131-7 norm), but they must be transformed into crisp values for the control variables. This is the purpose of the Defuzzification Interface. There are two types of defuzzification methods [18,19] depending on the way in which the individual fuzzy sets B' are aggregated:

- Mode A: Aggregation First, Defuzzification After. The Defuzzification Interface performs the aggregation of the individual fuzzy sets inferred, B', to obtain the final output fuzzy set. Usually, the aggregation operator is the minimum or the maximum.
- Mode B: Defuzzification First, Aggregation After. This avoids the computation of the final fuzzy set by considering the contribution of each rule output individually, obtaining the final control action by taking a calculation (an average, a weighted sum or a selection of one of them) of a concrete crisp characteristic value associated with each of them.

More complete information on FLCs can be found in [1,2,3].

2.2. *IEC 61131 languages*

The International Standard IEC 61131 [14] applies to programmable controllers and their associated peripherals such as programming and debugging tools, Human-machine interfaces, etc, which have as their intended use the control and command of machines and industrial processes. In part 3 (IEC 61131-3) of this standard, the syntax and semantics of a unified suite of five programming languages for programmable controllers is specified. The languages consist of two textual programming languages: Instruction List (IL) and Structured Text (ST); and three graphical programming languages: Ladder diagram (LD), Function block diagram (FBD) and Sequential function chart (SFC).

IL is its low level component and it is similar to assembly language: one instruction per line, low level and low expression commands. ST, as the name suggests, intends to be more structured and it is very easy to learn and understand for anyone with a modest experience in programming. LD was originated in the U.S. and it is based on graphical presentation of Relay Ladder Logic [20]. FBD is widely used in the process industry and it represents the system as a group of interconnected graphical blocks, as in the electronic circuit diagrams. SFC is based on GRAFCET (itself based on binary petri nets [21]).

All the languages are modular. The basic module is called the Programmable Organization Unit (POU) and includes Programs, Functions or Function Blocks (FBs). A system is usually composed of many POUs, and each of these POUs can be programmed in a different language. For instance, in a system consisting of two functions and one FB (three POUs), one function may be programed in LD, another function in IL and the FB may be programmed in ST. The norm defines all common data types (e.g. BOOL, REAL, INT, ARRAY, STRUCT, etc.) as well as ways to interconnect POUs, assign process execution priorities, process timers, CPU resource assignment, etc.

The concepts of the Program and Functions are quite intuitive. Programs are a simple set of statements and variables. Functions are calculations that can return only one value and are not supposed to have state variables. An FB resembles a very primitive object. It can have multiple input and multiple output variables, can be enabled by an external signal, and can have local variables. Unlike an object, an FB only has one execution block (i.e. there are no methods). The underlying idea for these limitations is that you should be able to implement programs using either text-based or graphic-based languages. Having only one execution block allows the execution to be easily controlled when using a graphic-based language to interconnect POUs.

In part 7 of the standard IEC 61131 (IEC61131-7), FCL is also defined in order to deal with fuzzy control programming. The aim of this standard is to offer the companies and the developers a well defined common understanding of the basic means with which to integrate fuzzy control applications in control systems, as well as the possibility of exchanging portable fuzzy control programs across different programming systems. jFuzzyLogic is focused on this language.

FCL has a similar syntax to ST but there are some very important differences. FCL exclusively uses a new POU type, FIS, which is a special type of a FB. All fuzzy language definitions should be within an FIS. Moreover, as there is no concept of execution order concept there are no statements. For instance, there is no way to create the typical "Hello world" example since there is no *printed* statement.

An FIS is usually composed of one or more FBs. Every `FUNCTION_BLOCK` has the following sections: i) input and output variables are defined in the `VAR_INPUT` and `VAR_OUTPUT` sections respectively; ii) fuzzification and defuzzification membership functions are defined in the `FUZZIFY` and `DEFUZZIFY` sections respectively; iii) fuzzy rules are written in the `RULEBLOCK` section.

Variable definition sections are straightforward; the variable name, type and possibly a default value are specified. Membership functions either in `FUZZIFY` or `DEFUZZIFY` are defined for each linguistic term using the `TERM` statement followed by a function definition. Functions are defined as piece-wise linear functions using a series of points $(x_0,y_0)(x_1,y_1)...(x_n,y_n)$, for instance, `TERM` `average := (10,0) (15,1) (20,0)` defines a triangular membership function. Only two membership functions are defined in the IEC standard: singleton and piece-wise linear. As shown in Section 4, jFuzzyLogic significantly extends these concepts.

An FIS can contain one or more `RULEBLOCK`, in which fuzzy rules are defined. Since rules are intrinsically parallel, no execution order is implied or warranted by the specified order in the program. Each rule is defined using standard "`IF condition THEN conclusion [WITH weight]`" clauses. The optional `WITH weight` statement allows weighting factors for each rule. Conditions tested in each `IF` clause are of the form "variable IS [NOT] linguistic_term". This tests the membership of a *variable* to a *linguistic_term* using the membership function defined in the corresponding `FUZZIFY` block. An optional NOT operand negates the membership function (i.e. $\overline{m}(x) = 1 - m(x)$). Obviously, several conditions can be combined using `AND` and `OR` connectors.

A simple example of an FIS using FCL to calculate the tip in a restaurant is shown below, where Figure 2 shows the membership functions for this example. More complete information on the IEC 61131 norm can be found in [14].

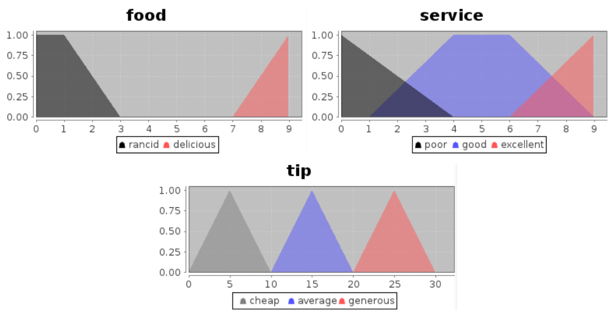


Fig. 2. Membership functions for tipper example.

```
FUNCTION_BLOCK tipper
VAR_INPUT
service, food : REAL;
END_VAR

VAR_OUTPUT
tip : REAL;
END_VAR

FUZZIFY service
TERM poor := (0, 1) (4, 0) ;
TERM good := (1, 0) (4,1) (6,1) (9,0);
TERM excellent := (6, 0) (9, 1);
END_FUZZIFY

FUZZIFY food
TERM rancid := (0, 1) (1, 1) (3,0);
TERM delicious := (7,0) (9,1);
END_FUZZIFY

DEFUZZIFY tip
TERM cheap := (0,0) (5,1) (10,0);
TERM average := (10,0) (15,1) (20,0);
TERM generous := (20,0) (25,1) (30,0);
METHOD : COG; // Center of Gravity
END_DEFUZZIFY

RULEBLOCK tipRules
Rule1: IF service IS poor OR food IS rancid
       THEN tip IS cheap;
Rule2: IF service IS good THEN tip IS average;
Rule3: IF service IS excellent AND food IS delicious
       THEN tip IS generous;
END_RULEBLOCK
END_FUNCTION_BLOCK
```

3. Comparison of fuzzy logic software

In this section we present a comparison of non-commercial fuzzy software (Table 1). We center our interest on free software because of its important role in the scientific research community [13]. This comparison is not intended to be comprehensive or exhaustive. It shows the choices a researcher in the field faces when trying to select a fuzzy logic software package.

We analyze 26 packages (including jFuzzyLogic), mostly from SourceForge or Google-Code, which are considered to be some of the most respectable software repositories. The packages are analyzed in the following categories:

- *FCL support.* Only four packages ($\sim 17\%$) claim to support IEC 61131-7 specification. Notably, two of them are based on jFuzzyLogic. Only two packages that support FCL are not based on our software. Unfortunately, neither of them seem to be maintained by their developers any longer. Furthermore, one of them has some code taken from jFuzzyLogic.
- *Programming language.* This is an indicator of code portability. Their languages of choice were mainly Java and C++/C (column *Lang.*). As Java is platform independent it has the advantage of

Table 1: Comparison of open fuzzy logic software packages. Columns describe: Project name (Name), IEC 61131-7 language support (IEC), latest release year (Rel.), main programming language (Lang.), short description from website (Description), number of membership functions supported (MF) and Functionality (notes). Name* : package is maintained, compiles correctly, and has extensive functionality.

Name	IEC	Rel.	Lang.	Description	MF	Notes
Akira [22]	No	2007	C++	Framework for complex AI agents.	4	
AwiFuzz [23]	Yes	2008	C++	Fuzzy logic expert system	2	Does not compile
DotFuzzy [24]	No	2009	C#	.NET library for fuzzy logic	1	Specific
FFLL [25]	Yes	2003	C++	Optimized for speed critical applications.	4	Does not compile
Fispro* [26]	No	2011	C++/Java	Fuzzy inference design and optimization	6	
FLUtE [27]	No	2004	C#	A generic Fuzzy Logic Engine	1	Beta version
FOOL [28]	No	2002	C	Fuzzy engine	5	Does not compile
FRBS citeFRBS2011	No	2011	C++	Fuzzy Rule-Based Systems	1	Specific
Funzy [29]	No	2007	Java	Fuzzy Logic reasoning	2^*	Specific
Fuzzy Logic Tools* [30]	No	2011	C++	Framework fuzzy control systems,	12	
FuzzyBlackBox [31]	No	2011	-	Implementing fuzzy logic	-	No files released
FuzzyClips [32]	No	2004	C/Lisp	Fuzzy logic extension of CLIPS	$3+2^*$	No longer maintained
FuzzyJ ToolKit [33]	No	2006	Java	Fuzzy logic extension of JESS	15	No longer maintained
FuzzyPLC* [34]	Yes	2011	Java	Fuzzy controller for PLC Siemens s226	$11+14^*$	Uses jFuzzyLogic
GUAJE* [35]	No	2012	Java	Development environment		Uses FisPro
javafuzzylogicctrltool [36]	No	2008	Java	Framework for fuzzy rules	-	No files released
JFCM [37]	No	2011	Java	Fuzzy Cognitive Maps (FCM)	-	Specific
JFuzzinator [38]	No	2010	Java	Type-1 Fuzzy logic engine	2	Specific
jFuzzyLogic*	Yes	2012	Java	FCL and Fuzzy logic API	$11+14^*$	This paper
jFuzzyQt* [39]	Yes	2011	C++	jFuzzyLogic clone	8	
libai [40]	No	2010	Java	AI library, implements some fuzzy logic	3	Specific
libFuzzyEngine [41]	No	2010	C++	Fuzzy Engine for Java	1	Specific
Nefclass [42]	No	1999	C++/Java	Neuro-Fuzzy Classification	1	Specific
nxtfuzzylogic [43]	No	2010	Java	For Lego Mindstorms NXT	1	Specific
Octave FLT* [44]	No	2011	Octave	Fuzzy logic for Toolkit	11	
XFuzzy3* [45]	No	2007	Java	Development environment	6	Implements XFL3 specification language

portability. C++ has an advantage in speed and also allows for easier integration in industrial controllers.

- *Functionality.* Eight packages ($\sim$ 29%) were made for specific purposes, marked as 'specific' (column *Notes*, Table 1). Specific code usually has limited functionality, but it is simpler and has a faster learning curve for the user.
- *Membership functions.* This is an indicator of how comprehensive and flexible the package is. Specific packages include only one type of membership function (typically trapezoid) and/or one defuzzification method (data not shown). In some cases, arbitrary combinations of membership functions are possible. These packages are marked with an asterisk. For example, '$M+N^*$' means that the software supports M membership functions plus another N which can be arbitrarily combined.
- *Latest release.* In nine cases ($\sim$ 33%) there were no released files for the last three years or more (see *Rel.* column in the Table 1). This may indicate that the package is no longer maintained, and in some cases the web site explicitly mentions this.
- *Code availability and usability.* Five of the packages ($\sim$ 21%) had no files available, either because the project was no longer maintained or because the project never released any files at all. Whenever the original sites were down, we tried to retrieve the projects from alternative mirrors. In three cases ($\sim$ 13%) the packages did not compile. We performed minimal testing by simply following the instructions, where available, and made no effort to correct any compilation problems.

To summarise, only eight of the software packages ($\sim$ 33%) seemed to be maintained, compiled correctly, and had extensive functionality. Only two of them (FuzzyPLC and jFuzzyQt) are capable of parsing FCL (IEC 61131-7) files and both are based on jFuzzyLogic. From our point of view users need an open source software which provides the basic programming elements of the standard IEC 61131-7 and is maintained in order to take advantage of the benefits of open source software and standardization. In the next section we will describe jFuzzyLogic in detail.

4. JFuzzyLogic

JFuzzyLogic's main goal is to facilitate and accelerate the development of fuzzy systems. We achieve this goal by: i) using standard programming language (FCL) that reduces learning curves; ii) providing a fully functional and complete implementation of FIS; iii) creating API that developers can use or extend; iv) implementing an Eclipse plugin to easily write and test FCL code; v) making the software platform independent; and vi) distributing the software as open source. This allows us to significantly accelerate the development and testing of fuzzy systems in both industrial and academic environments.

In these sections we show how these design and implementation goals were achieved. This should be particularly useful for developers and researchers looking to extend the functionality or use the available API.

4.1. jFuzzyLogic implementation

jFuzzyLogic is fully implemented in Java, thus the package is platform independent. ANTLR[46] was used to generate Java code for a lexer and parser based on our FCL grammar definition. This generated parser uses a left to right leftmost derivation recursive strategy, formally know as "LL(*)".

Using the lexer and parser created by ANTLR we are able to parse FCL files by creating an Abstract Syntax Tree (AST), a well known structure in compiler design. The AST is converted into an Interpreter Syntax Tree (IST), which is capable of performing the required computations. This means that the IST can represent the grammar, like and AST, but is also capable of performing calculations. The parsed FIS can be evaluated by recursively transversing the IST.

4.2. Membership functions

Only two membership functions are defined in the IEC standard: singleton and piece-wise linear. jFuzzyLogic also implements other commonly used membership functions:

- Cosine : $f(x|\alpha,\beta) = cos[\frac{\pi}{\alpha}(x-\beta)], \forall x \in [-\alpha,\alpha]$
- Difference of sigmoidals: $f(x|\alpha_1,\beta_1,\alpha_2,\beta_2) = s(x,\alpha_1,\beta_1) - s(x,\alpha_2,\beta_2)$, where $s(x,\alpha,\beta) = 1/[1+e^{-\beta(x-\alpha)}]$
- Gaussian : $f(x|\mu,\sigma) = e^{(x-\mu)^2/2\sigma^2}$
- Gaussian double : $f(x|\mu_1,\sigma_1,\mu_2,\sigma_2) =$

$$\begin{cases} e^{(x-\mu_1)^2/2\sigma_1^2} & x < \mu_1 \\ 1 & \mu_1 \leqslant x \leqslant \mu_2 \\ e^{(x-\mu_2)^2/2\sigma_2^2} & x > \mu_2 \end{cases}$$

- Generalized bell : $f(x|\mu_1,a,b) = \frac{1}{1+|(x-\mu)/a|^{2b}}$
- Sigmoidal : $f(x|\beta,t_0) = \frac{1}{1+e^{\beta(x-t_0)}}$
- Trapezoidal : $f(x|m_{in},l_{ow},h_{igh},m_{ax}) =$

$$\begin{cases} 0 & x < m_{in} \\ \frac{x-m_{in}}{l_{ow}-m_{in}} & m_{in} \leqslant x < l_{ow} \\ 1 & l_{ow} \leqslant x \leqslant h_{igh} \\ \frac{x-h_{igh}}{m_{ax}-h_{igh}} & h_{igh} < x \leqslant m_{ax} \\ 0 & x > m_{ax} \end{cases}$$

- Triangular: $f(x|m_{in},m_{id},m_{ax}) =$

$$\begin{cases} 0 & x < m_{in} \\ \frac{x-m_{id}}{l_{ow}-m_{id}} & m_{in} \leqslant x \leqslant m_{id} \\ \frac{x-m_{id}}{m_{ax}-m_{id}} & m_{id} < x \leqslant m_{ax} \\ 0 & x > m_{ax} \end{cases}$$

- Piece-wise linear : Defined as the union of consecutive points by an affine function.

Furthermore, jFuzzyLogic enables arbitrary membership functions to be built by combining mathematical expressions. This is implemented by parsing an Interpreter Syntax Tree (IST) of mathematical expressions. IST is evaluated at running time, thus enabling the inclusion of variables into the expressions. Current implementation allows the use of the following functions:Abs, Cos, Exp, Ln, Log, Modulus, Nop, Pow, Sin, Tan, as well as addition, subtraction, multiplication and division.

It should be noted that, as mentioned in section 2.2, IEC standard does not define or address the concept of fuzzy partition of a variable. For this reason, fuzzy partitions are not directly implemented in jFuzzyLogic. In order to produce a complete partition, we should indicate the membership function for each linguistic term.

4.3. Aggregation, activation & accumulation

As mentioned in section 2.2, rules are defined inside the `RULEBLOCK` statement in an FIS. Each rule block also specifies aggregation, activation and accumulation methods. All methods defined in the norm are implemented in jFuzzyLogic. It should be noted that we adhere to the definitions of Aggregation, Activation and Accumulation as defined by IEC 61131-7, which may differ from the naming conventions found in other references.

Aggregation methods (sometimes be called "combination" or "rule connection methods") define the t-norms and t-conorms playing the role of AND & OR operators. Needless to say, each set of operators must satisfy De Morgan's laws. These are shown in Table 2.

Table 2. Aggregation methods.

Name	x AND y	x OR y
Min/Max	$min(x,y)$	$max(x,y)$
Bdiff/Bsum	$max(0, x+y-1)$	$min(1, x+y)$
Prod/PobOr	$x\,y$	$x+y-x\,y$

Drastic	if$(x == 1) \to y$
	if$(y == 1) \to x$
	otherwise $\to 0$
		if$(x == 0) \to y$
		if$(y == 0) \to x$
		otherwise $\to 1$

Nil potent	if$(x+y > 1) \to min(x,y)$ otherwise $\to 0$	if$(x+y < 1) \to max(x,y)$ otherwise $\to 1$

Activation method define how rule antecedents modify rule consequents; i.e., once the IF part has been evaluated, how this result is applied to the THEN part of the rule. The most common activation operators are Minimum and Product (see Figure 3). Both methods are implemented in jFuzzyLogic.

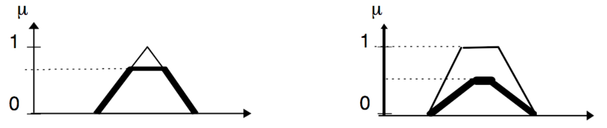


Fig. 3. Activation methods: Min (left) and Prod (right).

Finally, the accumulation method defines how the consequents from multiple rules are combined within a Rule Block (see Figure 4). Accumulation methods implemented by jFuzzyLogic defined in the norm include:

- Maximum : $\alpha_{cc} = max(\alpha_{cc}, \delta)$
- Bound sum: $\alpha_{cc} = min(1, \alpha_{cc} + \delta)$
- Normalized sum: $\alpha_{cc} = \frac{\alpha_{cc}+\delta}{max(1, \alpha_{cc}+\delta)}$
- Probabilistic OR : $\alpha_{cc} = \alpha_{cc} + \delta - \alpha_{cc}\,\delta$

where α_{cc} is the accumulated value (at point x) and $\delta = m(x)$ is the membership function for defuzzification (also at x).

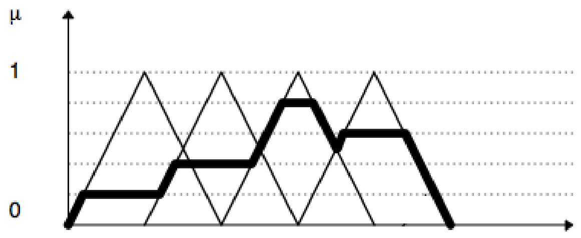


Fig. 4. Accumulation method: Combining consequents from multiple rules using Max accumulation method.

4.4. Defuzzification

The last step when evaluating an FIS is defuzzification. The value for each variable is calculated using the selected defuzzification method, which can be:

- Center of gravity : $\frac{\int x\mu(x)dx}{\int \mu(x)dx}$
- Center of gravity singleton : $\frac{\sum_i x_i \mu_i}{\sum_i \mu_i}$
- Center of area : $u \mid \int_{-\infty}^{u} \mu(x)dx = \int_{u}^{\infty} \mu(x)dx$
- Rightmost Max : $arg\max_x [\mu(x) = max(\mu(x))]$
- Leftmost Max : $arg\min_x [\mu(x) = max(\mu(x))]$
- Mean max : $mean(x) \mid \mu(x) = max(\mu(x))$

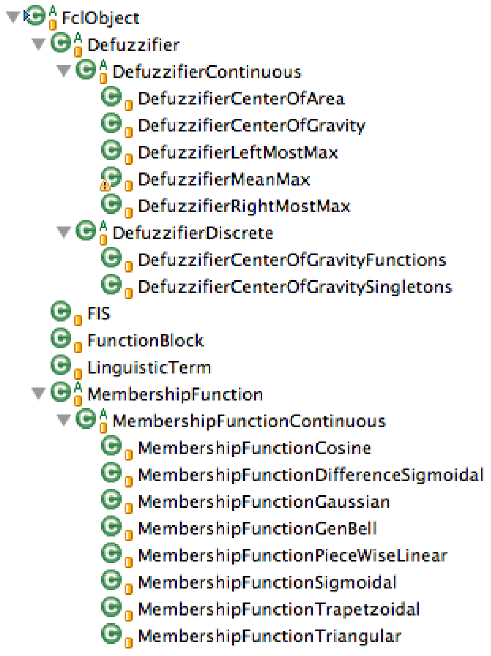
When dealing with simple membership functions, such as trapezoidal and piece-wise linear, defuzzicication can be computed easily by applying known mathematical equations. Although it can be carried out very efficiently, it unfortunately cannot be applyed to arbitrary expressions.

Due to the flexibility in defining membership functions, we use a more general method. We discretize membership functions at a number of points and use a more computational intensive, numerical integration method. The default number of points used for discretization, one thousand, can be adjusted according to the precision-speed trade-off required for a particular application. Inference is performed by evaluating membership functions at these discretization points. In order to perform a discretization, the "universe" for each variable, has to be estimated. The universe is defined as the range in which the variable has a non-neglectable value. Each membership function and each term is taken into account when calculating a universe for each variable. Once all the rules have been analyzed, the accumulation for each variable is complete.

4.5. API extensions

Some of the extensions and benefits provided by jFuzzyLogic are described in this section.

Modularity. The modular design allows us to extend the language and the API easily. It is possible to add custom aggregation, activation or accumulation methods, defuzzifiers, or membership functions by extending the provided object tree (see Figure 5).



```
FclObject
  Defuzzifier
    DefuzzifierContinuous
      DefuzzifierCenterOfArea
      DefuzzifierCenterOfGravity
      DefuzzifierLeftMostMax
      DefuzzifierMeanMax
      DefuzzifierRightMostMax
    DefuzzifierDiscrete
      DefuzzifierCenterOfGravityFunctions
      DefuzzifierCenterOfGravitySingletons
  FIS
  FunctionBlock
  LinguisticTerm
  MembershipFunction
    MembershipFunctionContinuous
      MembershipFunctionCosine
      MembershipFunctionDifferenceSigmoidal
      MembershipFunctionGaussian
      MembershipFunctionGenBell
      MembershipFunctionPieceWiseLinear
      MembershipFunctionSigmoidal
      MembershipFunctionTrapetzoidal
      MembershipFunctionTriangular
```

Fig. 5. Part of jFuzzyLogic's object tree. Used to provide API extension points.

Dynamic changes. We have defined an API which supports dynamic changes made to an FIS: i) variables can be used as membership function parameters; ii) rules can be added or deleted from rule blocks, iii) rule weights can be modified; iv) membership functions can use combinations of predefined functions.

Data Types. Due to the nature of fuzzy systems and in order to reduce complexity, jFuzzyLogic considers each variable as a *REAL* variable which is mapped to a *double* Java type.

Execution order. By default it is assumed that an FIS is composed of only one FB, so evaluating the FIS means evaluating the default FB. If an FIS has more than one FB, they are evaluated in alphabetical order by FB name. Other execution orders can be implemented by the user, which allows us to define hierarchical controllers easily.

4.6. Optimization API

An optimization API was developed in order to automatically fine tune FIS parameters. Our goal was to define a very lightweight and easy to learn API, which was flexible enough to be extended for general purpose usage.

The most common parameters to be optimized are membership functions and rule weights. For instance, if a variable has a fuzzifier term "`TERM rancid := trian 0 1 3`", there are three parameters that can be optimized in this membership function (whose initial values are 0, 1 and 3 respectively). Using the API, we can choose to optimize any subset of them. Similarly, in the rule "`IF service IS good THEN tip IS average`" we can optimize the weight of this rule (implicit "WITH 1.0" statement). This API consists of the following objects:

- *ErrorFunction*: An object that evaluates a Rule Block and calculates the error. Extending ErrorFunction is the bare minimum required to implement an optimization using one of the available optimization methods.
- *OptimizationMethod*: An optimization method object is an abstraction of an algorithm. It changes *Parameter* based on the performance measured using an ErrorFunction.

- *Parameter*: This class represents a parameter to be optimized. Any change to a parameter, will make the corresponding change to the FIS, thus changing the outcome. There are two basic parameters: ParameterMembershipFunction and ParameterRuleWeight, which allow for changes to membership functions and rule weights respectively. Other parameters could be created in order to, for instance, completely rewrite rules. We plan to extend them in future versions. Most users will not need to extend *Parameter* objects.

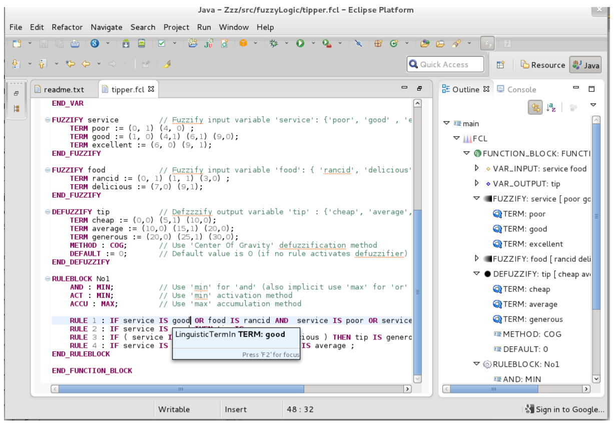
For most optimization applications, extending only one or two objects is enough (i.e. *ErrorFunction*, and sometimes *OptimizationMethod*). We provide template and demo objects to show how this can be done, all of which are included in our freely available source code.

A few optimization algorithms are implemented, such as gradient descent, partial derivative, and delta algorithm [47]. As noted above, other algorithms can be easily implemented based on these templates or by directly extending them. In the examples provided it is assumed that error functions can be evaluated anywhere in the input space. This is not a limitation in the API, since we can always develop an optimization algorithm and the corresponding error function that evaluates the FIS on a learning set.

4.7. Eclipse plugin

Eclipse is one of the most commonly used software development platforms. It allows us to use a specific language development tool by using the Eclipse-plugin framework. We developed a jFuzzyLogic plugin that allows developers to easily and rapidly write FCL code, and test it. Our plugin was developed using Xtext, a well known framework for domain specific languages based on ANTLR.

The plugin supports several features, such as syntax coloring, content assist, validation, program outlines and hyperlinks for variables and linguistic terms, etc. Figure 6 shows an example of the plugin being used to edit FCL code, where the left panel shows an editor providing syntax coloring while adding content assist at cursor position, and the right panel shows the corresponding code outline.

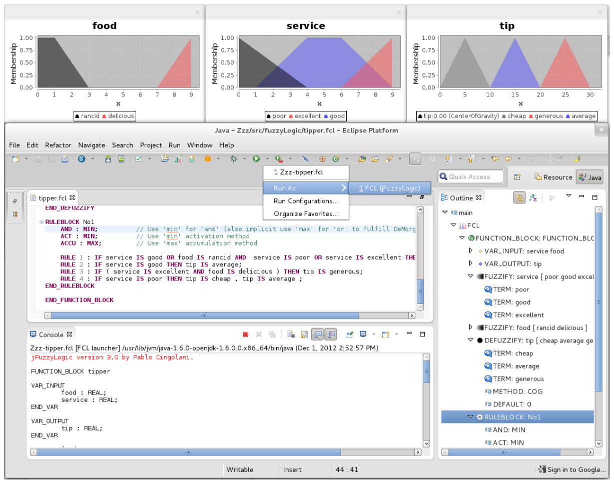

Java – Zzz/src/fuzzyLogic/tipper.fcl – Eclipse Platform
File Edit Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
readme.txt
tipper.fcl
Outline
Console
LinguisticTermIn TERM: good
Press 'F2' for focus
Writable
Insert
48 : 32
Sign in to Google...

```
END_VAR

FUZZIFY service          // Fuzzify input variable 'service': {'poor', 'good' , 'e
    TERM poor := (0, 1) (4, 0) ;
    TERM good := (1, 0) (4,1) (6,1) (9,0);
    TERM excellent := (6, 0) (9, 1);
END_FUZZIFY

FUZZIFY food             // Fuzzify input variable 'food': { 'rancid', 'delicious'
    TERM rancid := (0, 1) (1, 1) (3,0) ;
    TERM delicious := (7,0) (9,1);
END_FUZZIFY

DEFUZZIFY tip            // Defzzzify output variable 'tip' : {'cheap', 'average',
    TERM cheap := (0,0) (5,1) (10,0);
    TERM average := (10,0) (15,1) (20,0);
    TERM generous := (20,0) (25,1) (30,0);
    METHOD : COG;        // Use 'Center Of Gravity' defuzzification method
    DEFAULT := 0;        // Default value is 0 (if no rule activates defuzzifier)
END_DEFUZZIFY

RULEBLOCK No1
    AND : MIN;           // Use 'min' for 'and' (also implicit use 'max' for 'or'
    ACT : MIN;           // Use 'min' activation method
    ACCU : MAX;          // Use 'max' accumulation method

    RULE 1 : IF service IS good OR food IS rancid AND  service IS poor OR service
    RULE 2 : IF service IS
    RULE 3 : IF ( service I                                 ious ) THEN tip IS generc
    RULE 4 : IF service IS                                  IS average ;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

```
main
  FCL
    FUNCTION_BLOCK: FUNCTI
      VAR_INPUT: service food
      VAR_OUTPUT: tip
      FUZZIFY: service [ poor gc
        TERM: poor
        TERM: good
        TERM: excellent
      FUZZIFY: food [ rancid deli
      DEFUZZIFY: tip [ cheap av
        TERM: cheap
        TERM: average
        TERM: generous
        METHOD: COG
        DEFAULT: 0
      RULEBLOCK: No1
        AND: MIN
```

Fig. 6. FCL code edited in the Eclipse plugin.

When an FCL program is running, this plugin shows membership functions for all input and output variables. Moreover, the output console shows the FCL code parsed by jFuzzyLogic (Figure 7) .



```
END_DEFUZZIFY

RULEBLOCK No1
    AND : MIN;           // Use 'min' for 'and' (also implicit use 'max' for 'or' to fulfill DeMorg
    ACT : MIN;           // Use 'min' activation method
    ACCU : MAX;          // Use 'max' accumulation method

    RULE 1 : IF service IS good OR food IS rancid AND  service IS poor OR service IS excellent THE
    RULE 2 : IF service IS good THEN tip IS average;
    RULE 3 : IF ( service IS excellent AND food IS delicious ) THEN tip IS generous;
    RULE 4 : IF service IS poor THEN tip IS cheap , tip IS average ;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

```
Zzz-tipper.fcl [FCL launcher] /usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/bin/java (Dec 1, 2012 2:52:57 PM)
jFuzzyLogic version 3.0 by Pablo Cingolani.

FUNCTION_BLOCK tipper

VAR_INPUT
        food : REAL;
        service : REAL;
END_VAR

VAR_OUTPUT
        tip : REAL;
END_VAR
```

Fig. 7. Eclipse plugin when an FCL program is running.

5. A case study

In this section we present an example of the creation of a FLC controllers with jFuzzyLogic. This case study is focused on the development of the wall-following robot as explained in [48]. Wall-following behavior is well known in mobile robotics. It is frequently used for the exploration of unknown indoor environments and for the navigation between two points in a map.

The main requirement of a good wall-following controller is to maintain a suitable distance from the wall that is being followed. The robot should also move as fast as possible, while avoiding sharp movements, making smooth and progressive turns and changes in velocity.

5.1. *Robot fuzzy control system*

In our fuzzy control system, the input variables are: i) normalized distances from the robot to the right (*RD*) and left walls (*DQ*); ii) orientation with respect to the wall (*O*); and iii) linear velocity (*V*).

The output variables in this controller are the normalized linear acceleration (*LA*) and the angular velocity (*AV*). The linguistic partitions are shown in Figure 8, which are comprised by linguistic terms with uniformly distributed triangular membership functions giving meaning to them.

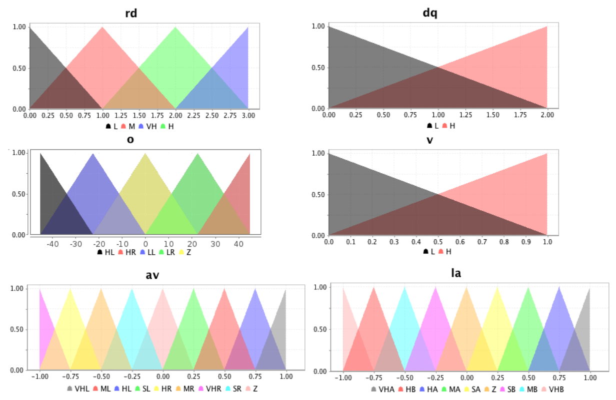


Fig. 8. Membership functions for a wall-following robot.

In order to implement the controller, the first step is to declare the input and output variables and to define the fuzzy sets. Variables are defined in the *VAR_INPUT* and *VAR_OUTPUT* sections. Fuzzy sets are defined in *FUZZIFY* blocks for input variables and *DEFUZZIFY* blocks for output variables.

One *FUZZIFY* block is used for each input variable. Each *TERM* line within a *FUZZIFY* block defines a linguistic term and its corresponding membership function. In this example all membership functions are triangular, so they are defined using the *'trian'* keyword, followed by three parameters defining the left, center and right points.

Output variables define their membership functions within *DEFUZZIFY* blocks. Linguistic terms and membership functions are defined using the *TERM* keyword as previously described for input variables. In this case we also add parameters to select the defuzzification method. The statement *'METHOD : COG'* indicates that we are using 'Center of gravity'. The corresponding FCL code generated for the first step is as follows:

```
VAR_INPUT
rd : REAL; // Right distance
dq : REAL; // Distance quotient
o  : REAL; // Orientation. Note: 'or' is a reserved word
v  : REAL; // Velocity
END_VAR

VAR_OUTPUT
la : REAL; // Linear acceleration
av : REAL; // Angular velocity
END_VAR

FUZZIFY rd
TERM L  := trian 0 0 1;
TERM M  := trian 0 1 2;
TERM H  := trian 1 2 3;
TERM VH := trian 2 3 3;
END_FUZZIFY

FUZZIFY dq
TERM L := trian 0 0 2;
TERM H := trian 0 2 2;
END_FUZZIFY

FUZZIFY o
TERM HL := trian -45 -45 -22.5;
TERM LL := trian -45 -22.5 0;
TERM Z  := trian -22.5 0 22.5;
TERM LR := trian 0 22.5 45;
TERM HR := trian 22.5 45 45;
END_FUZZIFY

FUZZIFY v
TERM L := trian 0 0 1;
TERM H := trian 0 1 1;
END_FUZZIFY

DEFUZZIFY la
TERM VHB := trian -1 -1 -0.75;
TERM HB  := trian -1 -0.75 -0.5;
TERM MB  := trian -0.75 -0.5 -0.25;
TERM SB  := trian -0.5 -0.25 0;
TERM Z   := trian -0.25 0 0.25;
TERM SA  := trian 0 0.25 0.5;
TERM MA  := trian 0.25 0.5 0.75;
TERM HA  := trian 0.5 0.75 1;
TERM VHA := trian 0.75 1 1;
METHOD : COG; // Center of Gravity
DEFAULT := 0;
END_DEFUZZIFY
```

```
DEFUZZIFY av
TERM VHR := trian -1 -1 -0.75;
TERM HR  := trian -1 -0.75 -0.5;
TERM MR  := trian -0.75 -0.5 -0.25;
TERM SR  := trian -0.5 -0.25 0;
TERM Z   := trian -0.25 0 0.25;
TERM SL  := trian 0 0.25 0.5;
TERM ML  := trian 0.25 0.5 0.75;
TERM HL  := trian 0.5 0.75 1;
TERM VHL := trian 0.75 1 1;
METHOD : COG;
DEFAULT := 0;
END_DEFUZZIFY
```

These membership functions can be plotted by running jFuzzyLogic with the FCL file generated as argument (e.g. `java -jar jFuzzyLogic.jar robotWCOR.fcl`).

The second step is to define the rules used for inference. They are defined in *RULEBLOCK* statements. For the wall-following robot controller, we used 'minimum' connection method (*AND : MIN*), minimum activation method (*ACT : MIN*), and maximum accumulation method (*ACCU : MAX*). We implemented the RB generated in [48] by the algorithm WCOR [49]. Each entry in the RB was converted to a single FCL rule. Within each rule, the antecedent (i.e. the *IF* part) is composed of the input variables connected by *'AND'* operators. Since there is more than one output variable, we can specify multiple consequents (i.e. *THEN* part) separated by semicolons. Finally, we add the desired weight using the *'with'* keyword followed by the weight. This completes the implementation of a controller for a wall-following robot using FCL and jFuzzyLogic. The FLC code generated for the second step is as follows:

```
RULEBLOCK rules
AND  : MIN; // Use 'min' for 'and' (also implicit use
            //'max' for 'or' to fulfill DeMorgan's Law)
ACT  : MIN; // Use 'min' activation method
ACCU : MAX; // Use 'max' accumulation method

RULE 01: IF rd is  L and dq is L and o is LL
    and v is L THEN la is VHB , av is VHR with 0.4610;
RULE 02: IF rd is  L and dq is L and o is LL
    and v is H THEN la is VHB , av is VHR with 0.4896;
RULE 03: IF rd is  L and dq is L and o is  Z
    and v is L THEN la is   Z , av is  MR with 0.6664;
RULE 04: IF rd is  L and dq is L and o is  Z
    and v is H THEN la is  HB , av is  SR with 0.5435;
RULE 05: IF rd is  L and dq is H and o is LL
    and v is L THEN la is  MA , av is  HR with 0.7276;
etc;
END_RULEBLOCK
```

The corresponding whole FCL file for this case study is available for download as one of the examples provided in the jFuzzyLogic package[c]. The corresponding Java code that uses jFuzzyLogic to run the FCL generated for WCOR is:

```
public class TestRobotWCOR {
  public static void main(String[] args)
  throws Exception {
    FIS fis = FIS.load("fcl/robot.fcl", true);
    FunctionBlock fb = fis.getFunctionBlock(null);
    // Set inputs
    fb.setVariable("dp", 1.25);
    fb.setVariable("o", 2.5);
    fb.setVariable("rd", 0.3);
    fb.setVariable("v", 0.6);
    // Evaluate
    fb.evaluate();
    // Get output
    double la = fb.getVariable("la").getValue());
    double av = fb.getVariable("av").getValue());
  }
}
```

This can also be done using the command line option "-e", which assigns values in the command line to input variables alphabetically (in this case: "dp", "o", "rd" and "v") and then evaluates the FIS. Here we show the command, as well as part of the output:

```
java -jar jFuzzyLogic.jar -e robot.fcl 1.2 2.5 0.3 0.6

FUNCITON_BLOCK robot
  VAR_INPUT          dq = 1.200000
  VAR_INPUT           o = 2.500000
  VAR_INPUT          rd = 0.300000
  VAR_INPUT           v = 0.600000
  VAR_OUTPUT         av = 0.061952
  VAR_OUTPUT         la = -0.108399
   ...(rule activations omitted)
```

Moreover, this utility also produces plots of membership functions for all variables, as well as the defuzzification areas for the output of all variables. Figure 9 shows the defuzzification areas for "av" and "la" in light grey.

When evaluation of multiple input values is required, values can be provided as a tab-separated input file using the command line option `-txt file.txt`. This approach can be easily extended for other data sources, such as databases, online acquisition, etc.

[c]`http://jfuzzylogic.sourceforge.net/html/example_java.html`

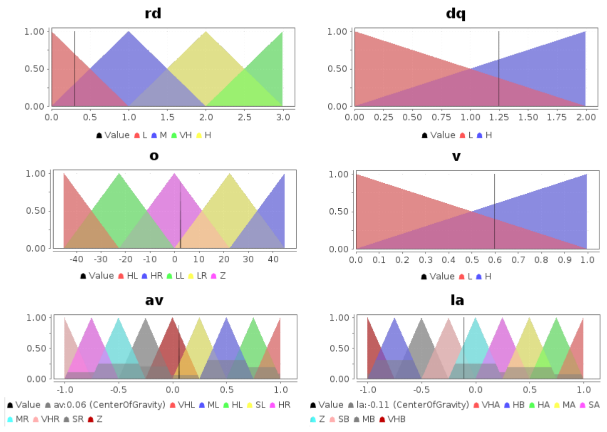


Fig. 9. Membership functions and defuzzification areas (light grey) for `robots.fcl` example.

Finally, we can use the jFuzzyLogic Eclipse pluging to see membership functions for all input and output variables and the FCL code parsed by jFuzzyLogic (Figure 10).

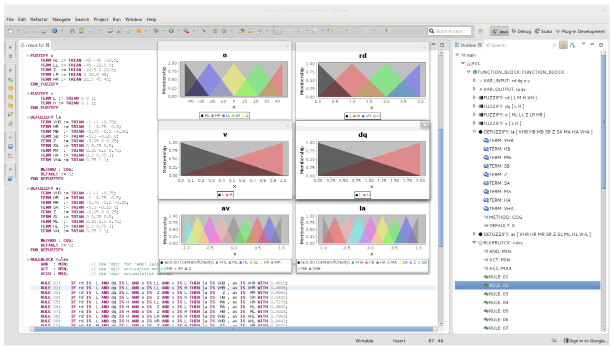


Fig. 10. jFuzzyLogic Eclipse pluging with `robots.fcl` example.

5.2. *Parameter optimization*

Continuing our case study, we show an application of the optimization API shown in section 4.6. We apply a parameter optimization algorithm to our FLC for a wall-following robot.

To provide an example, we optimize the membership functions of input variables “dg” and “v”. Each variable has two `TRIAN` membership functions, and each triangular membership function has three parameters. Thus, the total number of parameters to optimize is 12. The following code is used to perform the optimization:

```
// Load FIS form FCL
FIS fis = FIS.load("robot.fcl");
RuleBlock ruleBlock = fis.getFunctionBlock(null)
                                .getFuzzyRuleBlock(null);
// Get variables
Variable dg = ruleBlock.getVariable("dq");
Variable v = ruleBlock.getVariable("v");

// Add variables to be optimized to parameter list
ArrayList<Parameter> parameterList = new
                                 ArrayList<Parameter>();
parameterList.addAll(Parameter
                  .parametersMembershipFunction(dq));
parameterList.addAll(Parameter
                  .parametersMembershipFunction(v));

// Create optimizaion object and run it
ErrorFunction errFun = new ErrorFunctionRobot
                                     ("training.txt");
optimization = new OptimizationDeltaJump(ruleBlock
                             , errFun, parameterList);
optimization.optimize();
```

An appropriate error function was defined in object ErrorFunctionRobot, which is referenced in the previously shown code. The error function evaluates the controller on a predefined learning set, which consists of 5070 input and output values.

The resulting membership functions from this optimization are shown in Figure 11. It is easy to see that the optimized membership functions differ significantly from the originally defined ones.

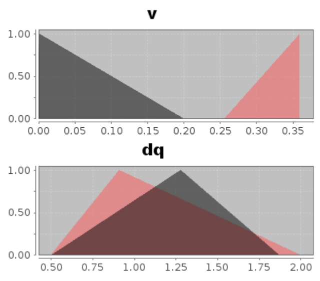


Fig. 11. Membership functions after optimization.

The implemented algorithm performs a global optimization. Accordingly, the improvements to the overall RMS error was reduced from 15% in the first iteration, to 3% on the second iteration and only 0.5%. on the third one. Further improvements could be obtained by allowing more parameters in the optimization process, at the expense of computational time.

6. Conclusions

In this paper, we have described an open source Java library called jFuzzyLogic that allows us to design and to develop FLCs according to the standard IEC 61131-7. jFuzzyLogic offers a fully functional and complete implementation of an FIS and provides an API and Eclipse plugin to easily write and test FCL code. Moreover, this library relieves researchers of much technical work and enables researchers with little knowledge of fuzzy logic control to apply FLCs to their control applications.

We have shown a case study to illustrate the use of jFuzzyLogic. In this case, we developed a FLC for wall-following behavior in a robot. The example shows how jFuzzyLogic can be used to easily implement and to run a FLC. Moreover, we have shown how we can use jFuzzyLogic to tune a FLC.

The jFuzzyLogic software package is continuously being updated and improved. Future work includes: i) using defuzzifier analytic solutions wherever possible (e.g. FIS consisting exclusively of trapezoidal and triangular sets), as it would considerably speed-up processing; ii) adding FIS constraints and developing ways to include them as part of the language extensions; and iii) at the moment, we are developing an implementation of an FCL to C/C++ compiler, allowing easy implementation of embedded control systems.

Acknowledgments

jFuzzyLogic was designed and developed by P. Cingolani. He is supported in part by McGill Uninversity, Genome Quebec. J. Alcala-Fdez is supported by the Spanish Ministry of Education and Science under Grant TIN2011-28488 and the Andalusian Government under Grant P10-TIC-6858. We would like to thank R. Sladek and M. Blanchette for their comments. Special thanks to P.J. Leonard for his contributions to the project.

References

1. C.C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller parts i and ii," *IEEE Transactions on Systems, Man, and Cybernetics*, **20**, 404–435 (1990).
2. H. Hellendoorn D. Driankov and M. Reinfrank, "An Introduction to Fuzzy Control," *Springer-Verlag* (1993).
3. P.P. Bonissone, "Fuzzy logic controllers: An industrial reality," *In Computational Intelligence: Imitating Life, IEEE Press*, 316–327 (1994).
4. B.E. Eskridge and D.F. Hougen, "Extending adaptive fuzzy behavior hierarchies to multiple levels of composite behaviors," *Robotics And Autonomous Systems*, **58:9**, 1076–1084 (2010).
5. Ch.-F. Juang and Y.-Ch. Chang, "Evolutionary-group-based particle-swarm-optimized fuzzy controller with application to mobile-robot navigation in unknown environments," *IEEE Transactions on Fuzzy Systems*, **19:2**, 379–392 (2011).
6. R. Alcalá, J. Alcalá-Fdez, M.J. Gacto, and F. Herrera, "Improving fuzzy logic controllers obtained by experts: a case study in HVAC systems," *Applied Intelligence*, **31:1**, 15–30 (2009).
7. E. Cho, M. Ha, S. Chang, and Y. Hwang, "Variable fuzzy control for heat pump operation," *Journal of Mechanical Science and Technology*, **25:1**, 201–208 (2011).
8. F. Chávez, F. Fernández, R. Alcalá, J. Alcalá-Fdez, G. Olague, and F. Herrera, "Hybrid laser pointer detection algorithm based on template matching and fuzzy rule-based systems for domotic control in real home enviroments," *Applied Intelligence*, **36:2**, 407–423 (2012).
9. G. Acampora and V. Loia, "Fuzzy control interoperability and scalability for adaptive domotic framework," *IEEE Transactions on Industrial Informatics*, **1:2**, 97 – 111 (2005).
10. J. Otero, L. Sánchez, and J. Alcalá-Fdez, "Fuzzy-genetic optimization of the parameters of a low cost system for the optical measurement of several dimensions of vehicles," *Soft Computing*, **12:8**, 751–764 (2008).
11. O. Demir, I. Keskin, and S. Cetin, "Modeling and control of a nonlinear half-vehicle suspension system: A hybrid fuzzy logic approach," *Nonlinear Dynamics*, **67:3**, 2139–2151 (2012).
12. Y. Zhao and H. Gao, "Fuzzy-model-based control of an overhead crane with input delay and actuator saturation," *IEEE Transactions on Fuzzy Systems*, **20:1**, 181 –186 (2012).
13. S. Sonnenburg, M.L. Braun, Ch.S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.-R. Muller, F. Pereira, C.E. Rasmussen, G. Ratsch, B. Scholkopf, A. Smola, P. Vincent, J. Weston, and R. Williamson, "The need for open source software in machine learning," *Journal of Machine Learning Research*, **8**, 2443–2466 (2007).
14. "International Electrotechnical Commission technical committee industrial process measurement and control. IEC 61131 - Programmable Controllers," *IEC* (2000).
15. E.H. Mamdani, "Applications of fuzzy algorithms for control a simple dynamic plant," *Proceedings of the Institution of Electrical Engineers*, **121:12**, 1585–1588 (1974).

16. E.H. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International Journal of Man-Machine Studies*, **7**, 1–13 (1975).
17. L.A. Zadeh, "Fuzzy sets," *Information and Control*, **8**, 338–353 (1965).
18. L.X. Wang, "Adaptive Fuzzy Systems and Control. Design and Stability Analysis," *Prentice-Hall* (1994).
19. O. Cordón, F. Herrera, and A. Peregrín, "Applicability of the fuzzy operators in the design of fuzzy logic controllers," *Fuzzy Sets and Systems*, **86**, 15–41 (1997).
20. E.W. Kamen, "Ladder logic diagrams and plc implementations," *In Industrial Controls and Manufacturing, Academic Press*, 141–164 (1999).
21. W. Reisig, "Petri nets and algebraic specifications," *Theoretical Computer Science*, **80:1**, 1–34 (1991).
22. G. Pezzulo and G. Calvi, "Designing and implementing mabs in akira," *In P. Davidsson, B. Logan, and K. Takadama, editors, Multi-Agent and Multi-Agent-Based Simulation, Lecture Notes in Computer Science, Springer Berlin Heidelberg*, **3415**, 49–64 (2005), `http://www.akira-project.org/`.
23. "Awifuzz - fuzzy logic control system," `http://awifuzz.sourceforge.net/` (2006).
24. "Dotfuzzy," `http://www.havana7.com/dotfuzzy/` (2009).
25. M. Zarozinski, "An open source fuzzy logic library," *In AI Game Programming Wisdom, Charles River Media*, 90–103 (2002), `http://ffll.sourceforge.net/`.
26. Serge Guillaume and Brigitte Charnomordic, "Learning interpretable fuzzy inference systems with fispro," *International Journal of Information Sciences*, **181:20**, 4409–4427 (2011), `http://www.inra.fr/mia/M/fispro/`.
27. "Flute: Fuzzy logic ultimate engine," `http://flute.sourceforge.net/` (2004).
28. Ronald Hartwig, Carsten Labinsky, Sven Nordhoff, Bernd Landorff, Peter Jensch, and Joerg Schwanke, "Free fuzzy logic system design tool: Fool," *In 4th European congress on intelligent techniques and soft computing (EUFIT)*, **3**, 2274–2277 (1996), `http://rhaug.de/fool/`.
29. "Funzy," `http://code.google.com/p/funzy/` (2007).
30. A. Barragán and J.M. Andújar, "Fuzzy Logic Tools. Reference Manual v1.0," *Universidad de Huelva publicaciones* (2011), `http://uhu.es/antonio.barragan/category/temas/fuzzy-logic-tools`.
31. "Fuzzyblackbox," `http://fuzzyblackbox.sourceforge.net/` (2011).
32. Togai InfraLogic, "Fuzzyclips," `http://www.ortech-engr.com/fuzzy/fzyclips.html` (2004).
33. R.A. Orchard, "Fuzzy reasoning in jess: The fuzzyj toolkit and fuzzyjess," *3rd International Conference on Enterprise Information Systems (ICEIS)*, **2**, 533–542 (2001), `http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyJDocs/index.html`.
34. "Fuzzyplc," `http://fuzzyplc.sourceforge.net/` (2011).
35. J.M. Alonso and L. Magdalena, "Generating understandable and accurate fuzzy rule-based systems in a java environment," *9th International Workshop on Fuzzy Logic and Applications (WILF), Lecture Notes in Artificial Intelligence, Springer-Verlag*, **6857**, 212–219 (2011), `http://sourceforge.net/p/guajefuzzy/wiki/Home/`.
36. "javafuzzylogicctrltool," `http://code.google.com/p/javafuzzylogicctrltool/` (2008).
37. "Java fuzzy cognitive maps," `http://jfcm.megadix.it/` (2011).
38. "Jfuzzinator," `http://jfuzzinator.sourceforge.net/` (2010).
39. "jfuzzyqt - C++ fuzzy logic library," `http://jfuzzyqt.sourceforge.net/` (2011).
40. "libai," `http://libai.sourceforge.net/` (2010).
41. "libfuzzyengine," `http://libfuzzyengine.git.sourceforge.net/git/gitweb-index.cgi` (2010).
42. Detlef Nauck and Rudolf Kruse, "Nefclass - a neuro-fuzzy approach for the classification of data," *ACM Symposium on Applied Computing*, 461–465 (1995), `http://fuzzy.cs.uni-magdeburg.de/nefclass/`.
43. "nxtfuzzylogic," `http://code.google.com/p/nxtfuzzylogic/` (2010).
44. "Fuzzy logic toolkit for octave," `http://pdb.finkproject.org/pdb/package.php/fuzzy-logic-toolkit-oct324` (2011).
45. I. Baturone, F.J. Moreno-Velo, S. Sánchez-Solano, A. Barriga, P. Brox, A. Gersnoviez, and M. Brox, "Using xfuzzy environment for the whole design of fuzzy systems," *IEEE International Conference on Fuzzy Systems*, 1–6 (2007), `http://www2.imse-cnm.csic.es/Xfuzzy/Xfuzzy_3.0/index.html`.
46. T. Parr, "The definitive ANTLR reference: building domain-specific languages" (2007).
47. R.O. Duda, P.E. Hart, and D.G. Stork, "Pattern classification," *John Willey & Sons* (2001).
48. M. Mucientes, J. Alcalá-Fdez, R. Alcalá, and J. Casillas, "A case study for learning behaviors in mobile robotics by evolutionary fuzzy systems," *Expert Systems with Applications*, **37:2**, 1471–1493 (2010).
49. R. Alcalá, J. Alcalá-Fdez, J. Casillas, O. Cordón, and F. Herrera, "Hybrid learning models to get the interpretability-accuracy trade-off in fuzzy modelling," *Soft Computing*, **10:9**, 717–734 (2006).