The FuzzyLite Libraries for Fuzzy Logic Control

Juan Rada-Vilela, PhD. FuzzyLite Limited. Wellington, New Zealand.

Abstract—**Fuzzy Logic Controllers (FLCs) are mathematical models designed to control systems by means of fuzzy logic. Their simplicity, flexibility, interpretability, and handling of uncertainty have seen them applied to address different problems in a variety of domains. The seminal ideas of FLCs date back to 1965, and today there are more than 20 software libraries that provide such a functionality with different degrees of success. In spite of the widespread usage of FLCs, many of these libraries have not yet been thoroughly compared, hence raising questions about their correctness, performance, and accuracy when having to choose a library among them. In this article, we compare some of the most relevant libraries to design and operate FLCs, namely the FuzzyLite libraries, Matlab, Octave, and jFuzzyLogic. These libraries are evaluated on a set of 20 benchmarks that include Mamdani and Takagi-Sugeno FLCs as well as different membership functions. Our focus is on the performance and accuracy of the libraries, but we also consider the number of features and the amount of source code documentation to rate their overall quality. The results show that the FuzzyLite libraries offer the most accurate results, the highest number of features, the second best performance, and the second most documented source code, thus ranking them first for overall quality. The next libraries in the rankings are Octave, Matlab, and jFuzzyLogic (respectively). Our analysis of results finds explanations for the differences in performance and accuracy between the libraries, which provides useful information not only to further improve their quality, but also for users to make better and more informed decisions when having to choose one.**

Index Terms—**Fuzzy control, Software libraries, Open-source software, Software metrics**

I. Introduction

Fuzzy Logic Controllers (FLCs) are mathematical models designed to control systems by means of fuzzy logic [1]. Specifically, a FLC consists of input variables, output variables, and a set of inference rules that control the relationship between the variables. By using fuzzy logic, uncertainty is inherently represented and accounted for in the design and operation of the controller. For example, the temperature of an office can be controlled by a FLC using two rules stated as "if office is hot then fan is fast" and "if office is cold then fan is slow", where "office" is the input variable representing temperature as "hot" and "cold", and "fan" is the output variable representing the speed of a fan as "fast" and "slow". Thus, the underlying mathematics of FLCs are abstracted by using natural language that incorporates the imprecision and vagueness of human decision making [2]. This simplicity, flexibility, interpretability, and handling of uncertainty has been exploited in machine learning [3], decision making [4], drone control [5], self-driving cars [6], collective robotics [7], sensor networks [8], computer games [9], among others [10]; and more generally in domains like medicine [11], bioinformatics [12], chemistry [13], agriculture [14], and others [15].

The field of fuzzy logic started in 1965 with the seminal work of Lofti Zadeh [1], and today there are more than 20 software libraries to model FLCs [16], [17]. In no particular order, we consider the following to be some of the most relevant libraries today: Matlab and its Fuzzy Logic Toolbox [18], Octave and its Fuzzy Logic Toolkit [19], jFuzzyLogic [16], [17], and the FuzzyLite libraries [20]. We consider these to be relevant libraries mainly because of their relatively high number of features, but also because (a) their source code is available and well designed and documented, (b) they have been maintained for at least five years, and (c) they have an important user base judging by their ranks in search engines and available download metrics. We want to highlight the contributions of these libraries to the scientific community by adopting open-source licenses [21] and, in the case of Matlab, for making the source code commercially available.

The FuzzyLite libraries refer to the fuzzylite and jfuzzylite libraries for the C++ and Java programming languages, respectively. The goal of the FuzzyLite libraries is to easily design and efficiently operate FLCs following an object-oriented programming model without relying on external libraries. Started in 2010 and 2012, the FuzzyLite libraries are the most recent addition to fuzzy logic control software among the libraries here considered. Thus, we are interested in comparing them against some of the most relevant libraries for fuzzy logic control, particularly in terms of performance and accuracy. In addition, considering that a FLC can be configured with a variety of membership functions, we want to rank the performance of the libraries on different configurations in order to provide guidelines that will aid the design of more efficient FLCs. While previous works [16], [17] have compiled information about more than 20 libraries, these works did not focus on performance or accuracy, and did not include the FuzzyLite libraries. Hence, up to date, it is not certain which libraries offer the best performance or the best accuracy, let alone which membership functions are the most efficient.

The overall goal of this article is to introduce the FuzzyLite libraries and compare them against some of the most relevant open-source libraries for fuzzy logic control. Specifically, we will focus on the following objectives:

- Introduce the FuzzyLite libraries and their components.
- Compare the performance and accuracy between the different libraries.
- Identify the best performing configurations of FLCs.
- Rank the libraries according to their overall quality.

The remainder of this article is structured as follows. Section II presents related work and the software libraries. Section III presents an introduction to fuzzy logic control. Section IV presents the FuzzyLite libraries and their components. Section V presents the design of experiments to compare the libraries. Section VI presents the results and discussions. Lastly, Section VII presents the conclusions and future work.

II. Related Work

The most comprehensive comparison of libraries for fuzzy logic control can be found in [16] and [17], where the authors present their own library, namely jfuzzylogic, and a comparison of 26 free and open-source libraries. The libraries were compared on six categories, namely programming language, membership functions, latest release, code availability and usability, functionality, and support for the Fuzzy Control Language (FCL) standard (IEC 61131-7:2000). The authors report the following findings: (a) the libraries were mainly programmed in Java (13/26) and C/C++ (11/26); (b) only five libraries provided more than ten membership functions, and jfuzzylogic provided 14, which was the second highest number among those compared; (c) nine libraries had not released new versions in the last three years or more; (d) the development of only eight libraries appeared to be currently active; (e) the functionality of nine libraries is targeted at specific purposes; and (f) only four of the libraries support the FCL standard, and two of them are based on jfuzzylogic. The authors did not include the FuzzyLite libraries.

A. *Matlab*

The Matlab Fuzzy Logic Toolbox [18] is a component of the Matlab computing environment to design and operate FLCs using the Matlab scripting language or the Fuzzy Inference System (FIS) format. The toolbox also provides a library programmed in C, which we refer to as *cfis*, that allows the FLCs designed in Matlab to be compiled as stand-alone applications. The Matlab computing environment and toolbox are sold separately under proprietary licenses. The source code of the Fuzzy Logic Toolbox is privately available with the purchase of a license. Besides fuzzy logic control, the toolbox provides adaptive neuro-fuzzy modeling [22] and fuzzy data clustering [23]. The current version of the toolbox is R2018b, released in September 2018. The toolbox provides the following features for FLCs. **(2) Controllers**: Mamdani and Takagi-Sugeno. **(14) Membership functions**: triangle, trapezoid, bell, gaussian, gaussian product, pi-shape, sigmoid difference, sigmoid product, sigmoid, s-shape, z-shape, constant, linear, and custom functions. **(3) T-norms**: minimum, product, and custom functions. **(3) S-norms**: maximum, algebraic sum, and custom functions. **(8) Defuzzifiers**: centroid, bisector, smallest of maxima, mean of maxima, largest of maxima, weighted average, weighted sum, and custom defuzzifiers. **(1) Hedges**: not. **(1) Importer**: FIS. **(1) Exporter**: FIS.

B. *Octave*

The Octave Fuzzy Logic Toolkit [19] is a component of the Octave computing environment to design and operate FLCs using the Octave scripting language or the FIS format, both of which are mostly compatible with the Matlab counterparts. The Octave computing environment and toolkit are free and open source, licensed under the GNU General Public License 3. Besides fuzzy logic control, the toolkit provides fuzzy data clustering [23]. The current version of the toolkit is 0.4.5, released in 2014. The toolkit provides the following features for FLCs. **(2) Controllers**: Mamdani and Takagi-Sugeno. **(14) Membership functions**: triangle, trapezoid, bell, gaussian, gaussian product, pi-shape, sigmoid difference, sigmoid product, sigmoid, s-shape, z-shape, constant, linear, and custom functions. **(7) T-norms**: minimum, product, bounded difference, drastic product, einstein product, hamacher product, and custom functions. **(7) S-norms**: maximum, algebraic sum, bounded sum, drastic sum, einstein sum, hamacher sum, and custom functions. **(8) Defuzzifiers**: centroid, bisector, smallest of maxima, mean of maxima, largest of maxima, weighted average, weighted sum, and custom defuzzifiers. **(6) Hedges**: not, somewhat, very, extremely, very very, and custom functions. **(1) Importer**: FIS. **(1) Exporter**: FIS.

C. *jFuzzyLogic*

jFuzzyLogic [16], [17] is a free and open-source library programmed in Java, which supports the FCL to design FLCs. The library is licensed under the GNU Lesser General Public License 3 and the Apache License 2.0. Besides fuzzy logic control, jfuzzylogic provides different algorithms for parameter optimization. The current version of jfuzzylogic is 3.3, released in 2015. The library provides the following features for FLCs. **(2) Controllers**: Mamdani and Takagi-Sugeno. **(14) Membership functions**: triangle, trapezoid, discrete, bell, cosine, gaussian, gaussian product, sigmoid difference, sigmoid, s-shape, z-shape, constant, linear, and custom functions. **(6) T-norms**: minimum, product, bounded difference, drastic product, hamacher product, and nilpotent maximum. **(6) S-norms**: maximum, algebraic sum, bounded sum, drastic sum, hamacher sum, and nilpotent minimum. **(6) Defuzzifiers**: centroid, bisector, smallest of maxima, mean of maxima, largest of maxima, and weighted average. **(1) Hedges**: not. **(1) Importer**: FCL. **(2) Exporter**: FCL, and (not fully operational) C++.

III. Design and Operation of a Fuzzy Logic Controller

In this section, we present an a brief introduction to the design and operation of FLCs using four types of inference, namely Mamdani [24], Larsen [25], Takagi-Sugeno [26], and Tsukamoto [27]. For a rigorous introduction, we recommend the reader to refer to [1], [28]–[30].

The **design** of a FLC can be summarized in three steps. First, modeling the inputs and outputs of a control system as linguistic variables. Second, creating the set of inference rules (based on the linguistic variables) to control the system. Third, configuring the controller according to the type of inference. The **operation** of a FLC consists of three stages. First, the fuzzification stage converts the value $x_i \in \mathbb{R}$ of input variable i into a fuzzy value $\tilde{x}_i$. Second, the inference stage evaluates the rules and aggregates the consequents into the respective fuzzy outputs. Third, the defuzzification stage converts the fuzzy output $\tilde{y}_j$ of output variable j into the output value $y_j \in \mathbb{R}$. The design and operation of a FLC is represented in Figure 1 for a typical example.

The following sections provide more details into the design and operation of FLCs.

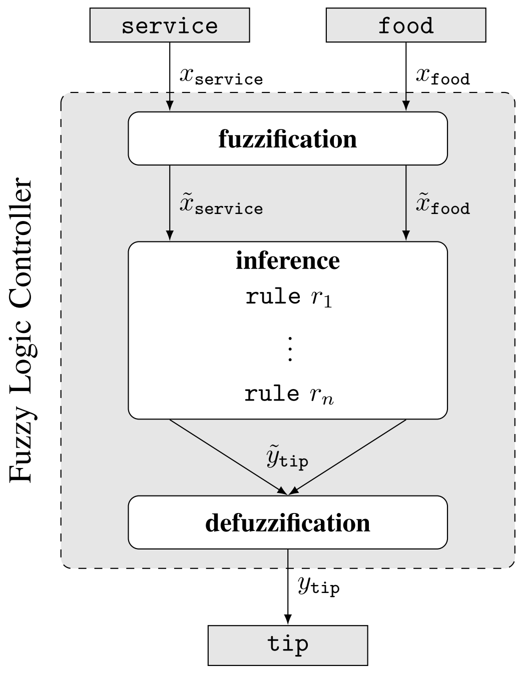


Fig. 1: Tipper example. Design and operation of a FLC to determine the amount to tip at a restaurant based on the quality of the service and the quality of the food.

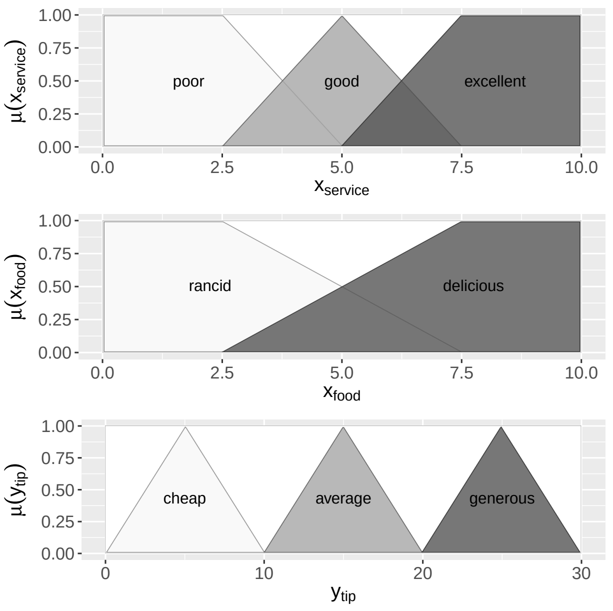


Fig. 2: Design of the linguistic variables for the tipper example.

A. Linguistic Variables

Linguistic variables [29] represent the input variables and output variables of a control system. A linguistic variable consists of linguistic terms that represent the states of the variable, each of which is characterized by a *membership* function $\mu : v \rightarrow \mathbb{R}$. A membership function determines the activation degree of the state it represents for every possible value $v \in \mathbb{R}$ that the variable can take. Thus, a linguistic variable has a *crisp* set of terms when $\mu_j(v) \in \{0, 1\}$ for each term j, and a *fuzzy* set of terms when $\mu_j(v) \in [0.0, 1.0]$ for each term j.

Considering the tipper example, the linguistic variables for `service`, `food`, and `tip` are designed in Figure 2.

B. Rules

The set of rules in FLCs are conditional statements that determine the relation between the input variables and the output variables of a system. Each rule is written in the form "if antecedent then consequent", where the antecedent and consequent contain one or more propositions in the form "variable is term". In the antecedent, multiple propositions are connected by "and" or "or", which are fuzzy operators for *conjunction* and *disjunction*, respectively. In the consequent, the propositions are independent from each other, and are connected by a symbolic "and", which is just a delimiter. The general structure of a rule is presented in Figure 3, where "hedge" refers to linguistic hedges [30] (e.g., *not*, *very*, *somewhat*) that optionally precede the linguistic terms, and the weight $w \in [0.0, 1.0]$ determines the importance of the rule (by default $w = 1.0$).

Considering the tipper example, the set of rules controlling the system are presented in Figure 4.

```
if    variable [hedge]* is term
      [(and|or) variable [hedge]* is term]*
then  variable [hedge]* is term
      [and variable [hedge]* is term]* [with w]?
```

Fig. 3: General structure of a rule, where the elements in bold are keywords, within brackets are optional, ∗-marked may appear zero or more times, and ?-marked may appear once or not at all.

C. Fuzzification

The fuzzification stage computes the activation degrees of the rules based on the antecedents and the importance of the rules. Specifically, the activation degree $\alpha_i \in \mathbb{R}$ of rule i is given by $\alpha_i = \alpha'_i \times w_i$, where $\alpha'_i \in \mathbb{R}$ refers to the evaluation of the antecedent, and w_i is the importance of the rule. The evaluation of the antecedent is the result of computing (1) the translation of each proposition "variable is term" into its corresponding membership function $\mu_{term}(x_{\texttt{variable}})$, and (2) the translation of the fuzzy operators joining the propositions into T-norms and S-norms (for conjunction and disjunction, respectively). Given any two values p and q, the conjunction operation is written as $p \owedge q$, which translates to a T-norm function $\top(p, q) \in [0.0, 1.0]$. Likewise, the disjunction operation is denoted by $p \ovee q$, which translates to an S-norm function $\bot(p, q) \in [0.0, 1.0]$. Typical examples of T-norm and S-norm functions are the minimum and maximum (respectively) between any two values.

Considering the set of rules defined for the tipper example, the activation degree for each rule is computed according to Equation (1), where $x_{\texttt{service}}$ and $x_{\texttt{food}}$ refer to the input values of the variables `service` and `food`, respectively.

$$
\begin{aligned}
\alpha_1 &= \mu_{poor}(x_{\texttt{service}}) \ovee \mu_{rancid}(x_{\texttt{food}}) \\
\alpha_2 &= \mu_{good}(x_{\texttt{service}}) \\
\alpha_3 &= 0.5 \times (\mu_{excellent}(x_{\texttt{service}}) \ovee \mu_{delicious}(x_{\texttt{food}})) \\
\alpha_4 &= \mu_{excellent}(x_{\texttt{service}}) \owedge \mu_{delicious}(x_{\texttt{food}})
\end{aligned} \tag{1}
$$

```
if service is poor or food is rancid then tip is cheap
if service is good then tip is average
if service is excellent or food is delicious then tip is
↪ generous with 0.5
if service is excellent and food is delicious then tip is
↪ generous with 1.0
```

Fig. 4: Set of rules for the tipper example.

D. Inference

The inference stage activates the consequents of the rules and aggregates them into their corresponding output variables. As such, the inference stage produces the fuzzy values for each output variable referenced in the consequents.

Considering the activation degrees from Equation (1), the fuzzy output value of the variable `tip` is given by:

$$\tilde{y}_{\texttt{tip}} = \tilde{f}_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}) = \frac{\alpha_1}{\mathit{cheap}} + \frac{\alpha_2}{\mathit{average}} + \frac{(\alpha_3 + \alpha_4)}{\mathit{generous}} \tag{2}$$

E. Defuzzification

Lastly, the defuzzification stage converts each fuzzy output value $\tilde{y}$ into a crisp value $y \in \mathbb{R}$ by means of a defuzzifier [31]. Depending on the type of inference, a defuzzifier can be based on integration or based on weights. An *integration-based* defuzzifier computes the crisp value by means of integration over the fuzzy value (e.g., centroid). A *weight-based* defuzzifier uses weights and values determined accordingly to compute the crisp value (e.g., weighted average).

Considering the output variable `tip`, its crisp output value is given by Equation (3), where the function $f_{\texttt{tip}}$ is defined accordingly by the defuzzifier.

$$y_{\texttt{tip}} = f_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}) \tag{3}$$

F. Mamdani and Larsen Controllers

Mamdani and Larsen FLCs, or simply Mamdani-based FLCs, are designed with (a) fuzzy operators for implication and aggregation, (b) output variables having fuzzy sets of terms, and (c) an integration-based defuzzifier. The *implication* operator is a T-norm that utilizes the activation degree of the rule to modulate the terms in its consequent. The *aggregation* operator is an S-norm that aggregates the activated terms of the consequents in the output variable. Lastly, the defuzzifier translates the aggregated terms into a crisp output value via integration.

Considering the centroid defuzzifier on the output variable `tip`, its crisp value is given by Equation (4), where $[a, b]$ is the range of the output variable, $z \in \mathbb{R}$ takes values in the discretized range $[a, b]$, and $f'_{\texttt{tip}}$ evaluates the fuzzy output value at z. The function $f'_{\texttt{tip}}$ is given by Equation (5), where the binary operators $\otimes$ and $\oplus$ translate to the T-norm and S-norm used for implication and aggregation, respectively.

$$f_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}) = \frac{\int_a^b z f'_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}, z)\, dz}{\int_a^b f'_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}, z)\, dz} \tag{4}$$

$$\begin{aligned} f'_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}, z) = {} & [\alpha_1 \otimes \mu_{\mathit{cheap}}(z)] \\ & \oplus [\alpha_2 \otimes \mu_{\mathit{average}}(z)] \\ & \oplus [\alpha_3 \otimes \mu_{\mathit{delicious}}(z)] \\ & \oplus [\alpha_4 \otimes \mu_{\mathit{delicious}}(z)] \end{aligned} \tag{5}$$

G. Takagi-Sugeno Controllers

Takagi-Sugeno FLCs are designed with (a) output variables having special terms, and (b) a weight-based defuzzifier. The output variables have terms whose membership functions reflect a constant value or a linear combinations of the input values. The constant value is any $k \in \mathbb{R}$, and the linear combination of input values is in the form $\sum_i c_i x_i + k$, where $c_i, x_i \in \mathbb{R}$ are the coefficient and input value for variable i. The weight-based defuzzifier is computed using the results of the membership functions as values and the activation degrees as weights.

Considering the output variable `tip` containing the constant terms $\mathit{cheap} = 5$, $\mathit{average} = 15$, and $\mathit{generous} = 25$, then its crisp output value computed with the weighted-average defuzzifier is given by:

$$f_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}) = \frac{\begin{array}{c}\alpha_1 \mathit{cheap} + \alpha_2 \mathit{average} \\ + \alpha_3 \mathit{delicious} + \alpha_4 \mathit{delicious}\end{array}}{\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4} \tag{6}$$

H. Tsukamoto Controllers

Tsukamoto FLCs are designed with (a) output variables having monotonic terms, and (b) a weight-based defuzzifier. The terms in the output variables have either monotonically increasing or monotonically decreasing membership functions, i.e., for all $x \leq y$ then $\mu(x) \leq \mu(y)$ or $\mu(x) \geq \mu(y)$, respectively. The weight-based defuzzifier uses the activation degrees as weights, and for the values uses the arguments of the membership functions that produce the respective activation degrees.

Considering the output variable `tip` containing only monotonically increasing (or decreasing) terms, then its crisp output value computed with the weighted-average defuzzifier is given by:

$$f_{\texttt{tip}}(x_{\texttt{service}}, x_{\texttt{food}}) = \frac{\begin{array}{c}\alpha_1 g_{\mathit{cheap}}(\alpha_1) + \alpha_2 g_{\mathit{average}}(\alpha_2) \\ + \alpha_3 g_{\mathit{delicious}}(\alpha_3) + \alpha_4 g_{\mathit{delicious}}(\alpha_4)\end{array}}{\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4} \tag{7}$$

where

$$g_j(\alpha) = \{z \in \mathbb{R} : \mu_j(z) = \alpha\} \tag{8}$$

IV. The FuzzyLite Libraries

The FuzzyLite libraries refer to our free and open-source generic implementations of FLCs, namely fuzzylite and jfuzzylite, available at github.com/fuzzylite. The fuzzylite library was programmed using the C++ language [32] and its

Standard Template Library (STL), following the C++98 standard (ISO/IEC 14882:1998), and recently including features of the C++11 standard (ISO/IEC 14882:2011). The jfuzzylite library is the equivalent of fuzzylite, but it is programmed using the Java language [33] following the Java 6.0 specification. Both libraries, currently in their sixth version, were released in March 2017 and licensed under the GNU General Public License 3 and a commercial license. The goal of the commercial license is to raise funds in order to continue the development of the FuzzyLite libraries.

The FuzzyLite libraries are programmed to be object-oriented, dependency-free, and multi-platform. The object-oriented programming model [34] abstracts the components and operation of FLCs into classes and methods. The dependency-free aspect means that the libraries do not rely on other libraries for their operation. The multi-platform focus enables the libraries to be used in operating systems for the desktop (e.g., Windows, macOS, Linux), mobile (e.g., iOS, Android), and in robots (using ROS [35]).

The FuzzyLite libraries are mature software whose development started in 2010 for fuzzylite and 2012 for jfuzzylite. According to the COCOMO model [36][1], the current version of the fuzzylite library has an estimated development cost of US$888 155 based on 14 949 physical source lines of code, an estimated required effort of 41.10 Person-Months, and a predicted duration of 10.26 months for four developers. Likewise, the estimated cost to develop the jfuzzylite library is US$747 496 based on 12 692 physical source lines of code, an estimated required effort of 34.59 Persons-Months, and a predicted duration of 9.61 months for four developers.

The remainder of this section presents the main components and operation of the FuzzyLite libraries.

A. Linguistic Variables

Linguistic variables [29] are abstract concepts that represent the variables of a control system. A linguistic variable is generalized by the class `Variable`, which basically consists of a name and description, a value $v^t \in \mathbb{R}$ at time t, a range $[a, b] \in \mathbb{R}$ for v^t, a boolean lock l_v that enforces $v^t \in [a, b]$ when $l_v = \texttt{T}$, a boolean indicator to enable the variable, and a set of linguistic terms that represent the states of the variable. A `Variable` represents an input variable or an output variable by means of the subclasses `InputVariable` and `OutputVariable`, respectively. For the purpose of distinction, the value v^t is referred to as x^t when the variable is an input variable, and y^t when it is an output variable.

The `InputVariable` implements an additional method to provide a text representation of the fuzzification of x^t in the form of Equation (2). The `OutputVariable` consists of the fuzzy output value $\tilde{y}^t$ represented by the class `Aggregated`, a defuzzifier to compute the crisp output value y^t, the default value $k \in \mathbb{R}$ that is enforced on y^t when the fuzzy output value is empty (i.e., $\tilde{y}^t = \emptyset$), the value $\dot{y}^t \in \mathbb{R}$ that retains the previous most recent valid output value of the variable, and a boolean lock $l_{\dot{y}}$ that sets $y^t \leftarrow \dot{y}^t$ when $\tilde{y}^t = \emptyset$ (taking precedence over the default value k). Thus, the output value of an `OutputVariable` is determined as shown in Figure 5, where `defuzzify` uses the defuzzifier of the `OutputVariable` to defuzzify $\tilde{y}^t$.

```
if $y^{t-1} \notin \{-\infty, \infty, \texttt{NaN}\}$
    $\dot{y}^t \leftarrow y^{t-1}$                         /*update previous value at time t*/
if $\tilde{y}^t \neq \emptyset$
    $y^t \leftarrow$ defuzzify$(\tilde{y}^t, a, b)$              /*defuzzify current value*/
else
    if $l_{\dot{y}} = \texttt{T}$
        $y^t \leftarrow \dot{y}^t$                         /*use previous value at time t*/
    else
        $y^t \leftarrow k$                                  /*use default value*/
if $l_v = \texttt{T}$
    $y^t \leftarrow \begin{cases} a & \text{if } y^t < a \\ b & \text{if } y^t > b \\ y^t & \text{otherwise} \end{cases}$     /*ensure $y^t \in [a, b]$*/
```

Fig. 5: Algorithm to compute the crisp value of an output variable.

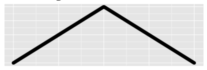
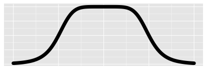
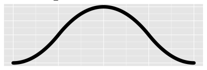
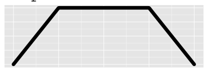
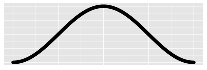
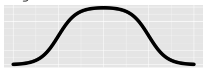
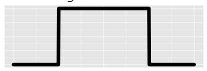
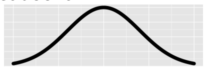
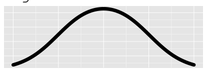
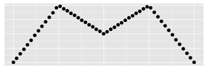
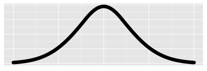
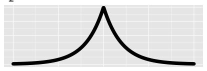
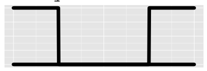
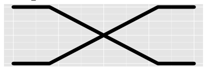
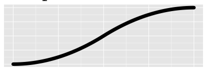
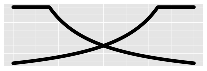
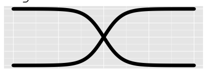
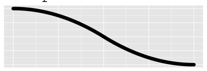
B. Linguistic Terms

Linguistic terms are abstract states of linguistic variables. A linguistic term is generalized by the class `Term`, which basically consists of a name, a height $h \in [0.0, 1.0]$ that multiplies the membership function value, and a `membership` method that computes the membership function value. There are 21 implementations of linguistic terms that provide different membership functions, and we classify them into four groups: basic, extended, edges, and functions. The **basic** group comprise the functions whose computation utilizes only arithmetic operations. The **extended** group comprise the functions whose computation utilizes arithmetic operations and functions (e.g., exponentiation, trigonometric). The **edges** group comprise monotonic functions. The **functions** group comprise linear and custom functions. Table I shows a graphical representation of the available linguistic terms in FuzzyLite.

In addition to these linguistic terms, `Activated` and `Aggregated` are two special terms that are convenient abstractions for the inference and defuzzification stages of the operation of FLCs. An `Activated` term represents the modulation of a linguistic term from a specific proposition contained in the consequent of a rule; that is, an `Activated` term denoted by p contains the activation degree α, the implication operator $\otimes$, and the term to activate μ_i such that its modulation is given by $\mu_p(x) = \alpha \otimes \mu_i$. An `Aggregated` term represents the aggregation of all the activated terms for an output variable; that is, an `Aggregated` term denoted by q contains a set of n activated terms and the aggregation operator $\oplus$ such that $\mu_q(x) = \mu_{p_1}(x) \oplus \ldots \oplus \mu_{p_j}(x) \oplus \ldots \oplus \mu_{p_n}(x)$. Notice that, in the cases of Takagi-Sugeno and Tsukamoto controllers, the implication and aggregation operators of their special terms translate to regular multiplications and sums, respectively.

[1]The COCOMO model is estimated by David A. Wheeler's `SLOCCount` application (dwheeler.com/sloccount/) using coefficients for an organic project, and the estimated annual salary of US$108 059 for a software engineer in 2017 according to ComputerWorld (computerworld.com/salarysurvey/tool/2017/).

TABLE I: Available implementations of linguistic terms in FuzzyLite. The `Binary`, `Concave`, `Ramp`, and `Sigmoid` terms present two opposing configurations. The `Function` term $g(z)$ is a custom function processed at runtime, and is internally represented as a binary expression tree.

basic	extended	
`Triangle` 	`Bell` 	`PiShape` 
`Trapezoid` 	`Cosine` 	`SigmoidDifference` 
`Rectangle` 	`Gaussian` 	`SigmoidProduct` 
`Discrete` 	`GaussianProduct` 	`Spike` 
	edges	
`Binary` 	`Ramp` 	`SShape` 
`Concave` 	`Sigmoid` 	`ZShape` 
	functions	
`Function` $f : z \to g(z)$	`Constant` $f(z) = k$	`Linear` $f(z) = \sum_i c_i x_i + k$

C. Rule Blocks

Rule blocks comprise the sets of rules that control the system. A rule block is represented by the class `RuleBlock`, which consists of a name and description, a set of rules, an activation method, and the fuzzy operators for conjunction, disjunction and implication. By not having the aggregation operator in the rule block, but rather in the aggregated terms of the output variables, a FLC can be configured with multiple rule blocks acting on the same set of output variables.

1) Rules: A rule is represented by the class `Rule`, which basically consists of an `Antecedent`, a `Consequent`, an activation degree α, and a weight w.

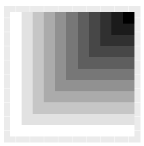
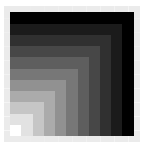
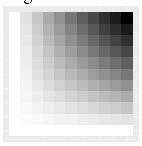
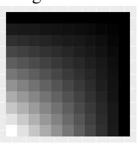
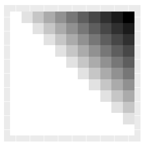
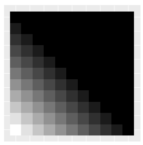
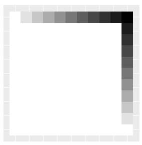
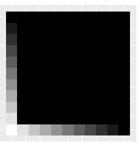
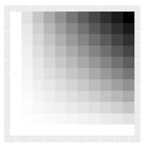
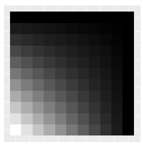
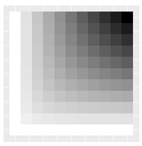
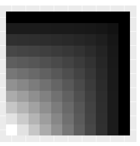
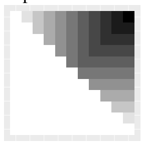
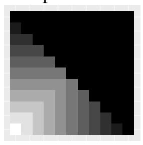
The antecedents of a rule are parsed from text using the Shunting-Yard algorithm [37] and Finite State Machines (FSMs) [38] to produce a binary expression tree of propositions. The operators in the tree refer to the conjunction and disjunction operators used to join the propositions, whereas the operands in the tree refer to the evaluation of the propositions. By using this data structure, an antecedent can have any number of propositions joined by different operators following the conventional order of precedence (from first to last): parentheses, "and", and "or". The evaluation of a proposition "variable is term" results in zero if the variable is disabled, or in the membership function value $\mu_{term}(x_{\texttt{variable}})$ otherwise. If the proposition contains hedges, the right-most hedge is applied on the membership function first, and the remaining hedges are applied on the result from the previous hedge. For example, the proposition "service is seldom not very good" evaluates to $\texttt{seldom}(\texttt{not}(\texttt{very}(\mu_{good}(x_{\texttt{service}}))))$. Besides input variables, the propositions in the antecedent can also utilize output variables, in which case the evaluation of the proposition corresponds to the activation degree of the term in the fuzzy output value.

The consequents of a rule are parsed from text using FSMs to produce a list of propositions, which determines the terms to be activated in the output variables. When a rule is activated, the terms in the propositions are added to new `Activated` terms, passing the respective activation degrees, and the implication operator (in the case of Mamdani-based FLCs). These `Activated` terms are then added to the `Aggregated` terms of the respective output variables.

2) Fuzzy Logic Operators: A rule block requires a conjunction operator if there is at least one rule that utilizes the "and" connective. Likewise, it requires a disjunction operator if a rule utilizes the "or" connective. For Mamdani-based FLCs, the implication and aggregation operators are required, but such is not the case for Takagi-Sugeno and Tsukamoto controllers.

The fuzzy logic operators for conjunction and implication are represented by the class `TNorm`, and those for disjunction and aggregation are represented by the class `SNorm`. Table II shows the 16 implementations of T-norms and S-norms available in FuzzyLite, which includes the `TNormFunction` and `SNormFunction` that allow to create arbitrary functions at runtime.

TABLE II: Available implementations of T-norms and S-norms in FuzzyLite. The figures show the result of each norm for 11 values of $x, y \in [0.0, 1.0]$ equally distributed, where the color white indicates zero, black indicates one, and gray scales indicate the values in between.

Minimum	Maximum	AlgebraicProduct	AlgebraicSum
			
BoundedDifference	BoundedSum	DrasticProduct	DrasticSum
			
EinsteinProduct	EinsteinSum	HamacherProduct	HamacherSum
			
NilpotentMinimum	NilpotentSum	TNormFunction	SNormFunction
		$\top(x, y) \in [0.0, 1.0]$	$\bot(x, y) \in [0.0, 1.0]$

3) Activation Methods: The activation methods refer to the strategies to activate the rules in a rule block. In total, there are seven implementations of activation methods. The `General` activation method activates the rules in the order that were added to the rule block. The `Highest` and `Lowest` activation methods sort the rules by activation degree (ascending and descending, respectively) and only activates the first n rules. The `First` and `Last` activation methods respectively activate the first and last n rules (in insertion order) whose activation degrees are greater than or equal to a given threshold $\theta \in \mathbb{R}$. The `Proportional` activation method computes and normalizes the activation degrees of the rules so that, from a set of n rules, each rule i is activated instead with $\alpha_i^* = \frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j}$ such that $\sum_{i=1}^{n} \alpha_i^* = 1.0$. Lastly, the `Threshold` activation method activates the rules (in insertion order) whose activation degrees satisfy $\alpha \odot \theta$, where $\odot$ is a relational operator $\odot \in \{=, \neq, <, >, \leq, \geq\}$, and $\theta \in \mathbb{R}$ is the threshold.

D. Defuzzifiers

The defuzzifiers [31] convert the fuzzy values of the output variables into crisp values. The defuzzifiers are represented by the abstract class `Defuzzifier`, which is subclassed by two abstract classes for integration-based and weight-based defuzzification, namely `IntegralDefuzzifier` and `WeightedDefuzzifier`.

On the one hand, the defuzzification methods that subclass `IntegralDefuzzifier` approximate the integral over the fuzzy output value at a given resolution r that defaults to $r = 100$. The resolution determines the number of sub-intervals to divide the fuzzy output value into, and hence it balances the trade-off between accuracy and performance. Specifically, a higher resolution provides a more accurate defuzzification at the cost of performance, whereas a lower resolution provides a better performance at the cost of accuracy. The implementations of integration-based defuzzifiers are the following five. The `Centroid` method computes the x-coordinate of the geometric center of the fuzzy value. The `Bisector` method computes the x-coordinate that divides the fuzzy value into two parts of equal area. The `SmallestOfMaximum`, `MeanOfMaximum` and `LargestOfMaximum` respectively compute the smallest, mean, and largest values of the x-coordinate at the global maxima of the fuzzy value.

On the other hand, the defuzzification methods that subclass `WeightedDefuzzifier` utilize the membership function values of the activated terms and their respective activation degrees as weights. In the case of Takagi-Sugeno FLCs, the function values are obtained from the terms using the same `membership` method; whereas in Tsukamoto FLCs, the monotonic terms (i.e., `Concave`, `Ramp`, `Sigmoid`, `SShape`, and `ZShape`) implement the `tsukamoto` method to satisfy Equation (8). The two implementations available of weight-based defuzzifiers are the `WeightedAverage` and `WeightedSum`, which respectively compute the weighted average and weighted sum of the fuzzy output value.

E. Engines

A FLC is represented by the class `Engine`, which consists of a name and description, a set of input variables, a set of output variables, and a set of rule blocks. An `Engine` supports four types of inference (namely Mamdani [24], Larsen [25], Takagi-Sugeno [26], and Tsukamoto [27]), which are automatically determined according to the configuration of the different components. Moreover, our object-oriented design provides the option of configuring a single engine with multiple types of inference simultaneously. For example, we can design a *hybrid* engine with one or more rule blocks performing Mamdani, Takagi-Sugeno, and Tsukamoto inference on a set of output variables configured accordingly.

F. Importers and Exporters

A FLC can be represented in a language different from the programming language that was used to create it. In doing so, we are able to (a) remove the complexity of the programming language, (b) focus specifically on the configuration of the FLC, (c) represent the FLC regardless of implementation, and (d) conveniently adapt the FLC to be stored and retrieved. Thus, an importer is a component that creates and configures FLCs from a specific representation, whereas an exporter describes FLCs using a specific representation.

Importers and exporters subclass `Importer` and `Exporter`, respectively, and the following are available in FuzzyLite. The `FllImporter` and `FllExporter` use the FuzzyLite Language (see Section IV-G). The `FisImporter` and `FisExporter` use the FIS format used by matlab and octave. The `FclImporter` and `FclExporter` use the FCL format used by jfuzzylogic. The `CppExporter` and `JavaExporter` generate the source code implementation of an engine using the fuzzylite and jfuzzylite libraries, respectively. The `FldExporter` evaluates a FLC to generate a plain-text column-based FuzzyLite Dataset (FLD), which can include the input values, output values, and headers for the columns, all separated by a given delimiter. Lastly, the `RScriptExporter` creates an R [39] script using the `ggplot2` [40] library to draw the control surface of each output variable for any given pair of input variables.

G. FuzzyLite Language

The FuzzyLite Language (FLL) is a representation of FLCs that we designed to be simpler, more flexible, more concise, and more efficient to parse than the FIS and FCL formats for describing FLCs. The structure of the FLL is presented in Figure 6, where the indentation and separation of properties are customizable (default is two spaces and carriage return, respectively), comments start with #, the property names and property values are case-sensitive and separated by a colon, parameters are separated by spaces, and `none` indicates an unspecified value. The *`term`* and *`rule`* properties are repeated for each term and rule available in the variable and rule block, respectively. The square brackets around a value (e.g., `[parameter]`) and the pipe between two values (e.g., `a|b`) are not part of the language: the square brackets indicate that the value is optional, whereas the pipe indicates that either one value or the other needs to be present.

Regarding the data types present in place of the property values in Figure 6, a `string` refers to an identifier that contains only letters and numbers, but no spaces or special characters. The `text` refers to arbitrary text in a single line. A `boolean` can be `true` or `false`. A `scalar` is a floating-point value that can also take $-\infty$, ∞, NaN, which are represented by `-inf`, `inf`, and `nan`, respectively. A `Term`

```
# Comment: text
Engine: string
 description: text
InputVariable: string
 description: text
 enabled: boolean
 range: scalar scalar
 lock-range: boolean
 term: string Term [parameters]
OutputVariable: string
 description: text
 enabled: boolean
 range: scalar scalar
 lock-range: boolean
 aggregation: SNorm|none
 defuzzifier: [Defuzzifier [parameter]]|none
 default: scalar
 lock-previous: boolean
 term: string Term [parameters]
RuleBlock: string
 description: text
 enabled: boolean
 conjunction: TNorm|none
 disjunction: SNorm|none
 implication: TNorm|none
 activation: [Activation [parameter]]|none
 rule: if antecedent then consequent
```

Fig. 6: Structure of the FuzzyLite Language.

refers to the name of the class of the linguistic term, and it is optionally followed by parameter values for the configuration of its membership function (see Table III). The `TNorm` and `SNorm` refer to the name of the class for the corresponding T-norm and S-norm. The `Defuzzifier` refers to the name of the defuzzifier. The `Activation` refers to the name of the activation method. Lastly, the `antecedent` and `consequent` are propositions expressed according to Figure 3.

TABLE III: Parameters of the membership functions. Each parameter is of type `scalar`, except for `Function`, whose parameter is expressed as an infix mathematical function. The height of the membership functions can be specified by adding an extra `scalar` value at the end (except for `Constant`, `Linear`, and `Function`).

term	parameters							
	first		second		third		fourth	
`Bell`	center		width		slope			
`Binary`	start		direction					
`Concave`	inflection		end					
`Constant`	value							
`Cosine`	center		width					
`Discrete`	x_0	y_0	$\dots$	x_i	y_i	$\dots$	x_n	y_n
`Function`	$f : x \to \mathbb{R}$							
`Gaussian`	mean		stdev					
`GaussianProduct`	mean		stdev		mean		stdev	
`Linear`	c_0	$\dots$	c_i	$\dots$	c_n	k		
`PiShape`	bottom left		top left		top right		bottom right	
`Ramp`	start		end					
`Rectangle`	start		end					
`Sigmoid`	inflection		slope					
`SigmoidDifference`	left		rising		falling		right	
`SShape`	start		end					
`SigmoidProduct`	left		rising		falling		right	
`Spike`	center		width					
`Trapezoid`	vertex a		vertex b		vertex c		vertex d	
`Triangle`	vertex a		vertex b		vertex c			
`ZShape`	start		end					

H. Factories

The factories refer to the implementation of the factory design pattern [41] for multiple components of the libraries. This design pattern centralizes the instantiation of objects and, with the help of a registry, new objects that are external to the library can be registered and instantiated. The factory design pattern is specially useful for the FLL because, by registering new objects in the factories, the FLL is automatically extended to seamlessly handle the new objects. There are seven factories available: the `ActivationFactory` for activation methods, the `DefuzzifierFactory` for defuzzifiers, the `FunctionFactory` for new functions and operators to be used in the mathematical expressions of `Function`, the `HedgeFactory` for linguistic hedges, the `SNormFactory` and `TNormFactory` for S-norms and T-norms (respectively), and the `TermFactory` for the linguistic terms and corresponding membership functions.

I. Features

To summarize, the features available in FuzzyLite are the following. **(3) Controllers**: Mamdani, Takagi-Sugeno, and Tsukamoto. **(21) Linguistic terms**: (*Basic*) triangle, trapezoid, rectangle, discrete; (*Extended*) bell, cosine, gaussian, gaussian product, pi-shape, sigmoid difference, sigmoid product, spike; (*Edges*) binary, concave, ramp, sigmoid, s-shape, z-shape; and (*Functions*) constant, linear, and custom functions. **(7) Activation methods**: general, proportional, threshold, first, last, lowest, highest. **(8) T-norms**: minimum, algebraic product, bounded difference, drastic product, einstein product, hamacher product, nilpotent minimum, and custom functions. **(8) S-norms**: maximum, algebraic sum, bounded sum, drastic sum, einstein sum, hamacher sum, nilpotent maximum, and custom functions. **(7) Defuzzifiers**: (*Integration-based*) centroid, bisector, smallest of maximum, largest of maximum, mean of maximum; and (*Weight-based*) weighted average, weighted sum. **(7) Hedges**: any, not, extremely, seldom, somewhat, very, and custom functions. **(3) Importers**: FLL, FIS, FCL. **(7) Exporters**: C++, Java, FLL, FLD, FIS, FCL, R script. **(7) Factories**: linguistic terms, activation methods, T-norms, S-norms, defuzzifiers, hedges, and functions and operators.

V. Design of experiments

The fuzzy logic control libraries that we are going to compare are fuzzylite (6.0), jfuzzylite (6.0), matlab (R2016a), cfis (R2016a), octave (4.0.2) with its fuzzy logic toolkit (0.4.5), and jfuzzylogic (3.3). Specifically, we are going to compare the libraries on a set of benchmarks designed to be simple enough such that we can easily trace and explain our metrics of interest, namely performance, rankings, accuracy, and overall quality of the libraries. The *performance* refers to the analysis on the computational time required by the libraries to evaluate the benchmarks. The *ranking* refers to the analysis of the differences in performance between the benchmarks. The *accuracy* refers to the analysis of the numerical differences between the libraries on the results obtained. Lastly, the *overall quality* ranks the libraries from better to worse according to the indicators of performance, accuracy, number of features, and source code documentation.

The set of benchmarks consists of 20 variations of a FLC designed for a self-driving vehicle whose goal is to avoid obstacles (see Figure 7). The FLC is configured with an input variable indicating the location of the obstacle relative to the vehicle, an output variable indicating the direction to steer the vehicle to, and a rule block containing two rules that steer the vehicle towards the opposite direction to the location of the obstacle. Each benchmark defines the output variable with a different membership function available in FuzzyLite, except for the case of the ZSShape benchmark that includes the `ZShape` and `SShape`. The Constant and Linear benchmarks are Takagi-Sugeno FLCs using the weighted-average defuzzifier; whereas the remaining benchmarks are Mamdani FLCs using the centroid defuzzifier with resolution $r = 100$, the minimum T-norm for implication, and the maximum S-norm for aggregation. The availability of the benchmarks in each library is presented in Table IV, and the variations can be found in github.com/fuzzylite/performance.

```
Engine: ObstacleAvoidance
  description: obstacle avoidance for self-driving cars
InputVariable: obstacle
  description: location of obstacle relative to vehicle
  enabled: true
  range: 0.000 1.000
  lock-range: false
  term: left Triangle 0.000 0.333 0.666
  term: right Triangle 0.333 0.666 1.000
OutputVariable: steer
  description: direction to steer the vehicle to
  enabled: true
  range: 0.000 1.000
  lock-range: false
  aggregation: Maximum
  defuzzifier: Centroid 100
  default: nan
  lock-previous: false
  term: left Triangle 0.000 0.333 0.666
  term: right Triangle 0.333 0.666 1.000
RuleBlock: steer-away
  description: steer away from obstacles
  enabled: true
  conjunction: none
  disjunction: none
  implication: Minimum
  activation: General
  rule: if obstacle is left then steer is right
  rule: if obstacle is right then steer is left
```

Fig. 7: Triangle benchmark in FLL

The performance of the libraries is measured by the average time that a FLC spends evaluating the benchmarks over 50 independent runs. The evaluation of a benchmark consists of computing the output values for a set of 100 000 input values equally distributed along the range of the input variable. The input values are preloaded in memory to ensure that the computational performance only measures the operation of the FLC. Each run is executed using the highest priority in the operating system (i.e., using command `nice -20` in macOS) to minimize the influence of other processes. The most relevant differences in performance will be analyzed by auditing the source code of the libraries.

The ranking of the libraries on each benchmark is based on the performance mentioned before, but the analysis will focus on the computational cost of the different membership functions. Hence, for each library, we will be able to classify the membership functions into groups based on the computational performance; thereby providing some experimental guidelines to design more efficient FLCs in each library.

TABLE IV: Availability of benchmarks in each library. The underlined benchmarks are available in all libraries, whereas the rest are not readily available in one or more libraries. The jfuzzylogic library provides a discrete membership function that we use to evaluate the missing membership functions. While jfuzzylogic provides custom functions, the options are limited and hence we also had to discretized the benchmark `Function`.

benchmark	fuzzylite / jfuzzylite	matlab / cfis	octave	jfuzzylogic
Bell	yes	yes	yes	yes
Binary	yes	no	no	discrete
Concave	yes	no	no	discrete
Constant	yes	yes	yes	yes
Cosine	yes	no	no	yes
Discrete	yes	no	no	yes
Function	yes	no	no	discrete
Gaussian	yes	yes	yes	yes
Gaussian Product	yes	yes	yes	yes
Linear	yes	yes	yes	yes
PiShape	yes	yes	yes	discrete
Ramp	yes	no	no	discrete
Rectangle	yes	no	no	discrete
Sigmoid	yes	yes	yes	yes
Sigmoid Difference	yes	yes	yes	yes
SigmoidP.	yes	yes	yes	discrete
Spike	yes	no	no	discrete
Trapezoid	yes	yes	yes	yes
Triangle	yes	yes	yes	yes
ZSShape	yes	yes	yes	discrete

The accuracy of the libraries is measured by the Root Mean Squared Error (RMSE) between the obtained output values and the expected output values on each benchmark. The obtained output values are the result of a single run over the set of input values. In Takagi-Sugeno FLCs, we expect the libraries to produce the same results because the operation of these controllers reduce to an equation that computes the weighted average over a set of values. If any result is different, we will need to audit the source code of the libraries in order to find the reason for the discrepancy. Contrarily, in Mamdani FLCs we expect small differences in the results because the libraries utilize different integration methods to defuzzify the fuzzy output values, despite that the resolution of the defuzzifiers is the same across libraries ($r = 100$). Thus, in order to identify which library is more accurate, we will compute the expected output values using defuzzifiers at a much higher resolution ($r = 100\,000$) to better approximate the true values; and then the libraries whose obtained output values are closer to the better approximations will be therefore more accurate. While the FuzzyLite libraries compare floating-point values using absolute error margins against $\epsilon = 1 \times 10^{-6}$, in the experiments we set $\epsilon = 0.0$ to compare the values just like the other libraries.

The fuzzylite and cfis libraries are compiled for the 64-bit Intel architecture in Release mode with the default `-O3` optimization flag and enabled features of the C++11 and C11 standards, respectively. The jfuzzylite and jfuzzylogic libraries are compiled into uncompressed JAR files. The octave binaries are compiled automatically with the default formula found in the Homebrew package manager (available at brew.sh/), which by default does not enable the experimental Just-In-Time (JIT) compiler and uses the Basic Linear Algebra Subprograms (BLAS) provided by Apple. The experiments settings are presented in Table V.

TABLE V: Experiment settings

benchmarks	20
independent runs	50
input/output values	100 000
computer	MacBook Pro (Mid 2014)
processor	2.5 GHz Intel Core i7
operating system	macOS X 10.11.5
memory	16 GB RAM
C/C++ compiler	Apple LLVM version 7.3.0
Java compiler	Oracle Java JDK 1.8.0_66

VI. Results and Discussions

The results and discussions are structured as follows. In Section VI-A, we present and discuss the performance of the libraries. In Section VI-B, we present and discuss the ranking of the benchmarks. In Section VI-C, we present and discuss the accuracy of the libraries. Finally, in Section VI-D, we present the overall quality of the libraries.

A. *Performance*

The performance of the libraries on the benchmarks is presented in Table VI. The best performing libraries are cfis and matlab, followed by fuzzylite and jfuzzylite, and lastly by jfuzzylogic and octave. According to the one-sided pairwise Wilcoxon rank sum test (at $\alpha = 0.05$ with Holm correction), the differences in performance between each pair of libraries are all statistically significant. In the next sections, we discuss some of the features that explain the performance of fuzzylite and jfuzzylite with respect to the other libraries.

1) fuzzylite vs jfuzzylite: The two libraries utilize the same class hierarchy, data structures, and data types, and yet, overall, fuzzylite is faster than jfuzzylite by $0.645 \div 0.279 = 2.3\text{x}$. We expected fuzzylite to be faster because it is compiled into a native executable, whereas jfuzzylite is compiled into bytecode that is then interpreted by the Java Virtual Machine (JVM). However, there are two benchmarks in which jfuzzylite is actually faster than fuzzylite, namely `Constant` and `Linear` by $0.023 \div 0.012 = 1.9\text{x}$ and $0.025 \div 0.013 = 1.9\text{x}$ (respectively). The better performance of jfuzzylite in these two Takagi-Sugeno benchmarks is thanks to the optimization performed by the JIT compiler in Java, something that we confirmed by performing additional experiments disabling the JIT compiler. Specifically, it is easier for the JIT compiler to optimize the Takagi-Sugeno benchmarks because their defuzzification process translates into a weighted average of a few values (see WeightedAverage::defuzzify); whereas the defuzzification process of the Mamdani benchmarks is harder to optimize because it translates into a weighted average of $r = 100$ values, where r is the resolution of the centroid defuzzifier, and the values are obtained only after evaluating the membership functions (see Centroid::defuzzify).

TABLE VI: Performance of the libraries on the benchmarks. The values represent the average computational time (in seconds) required by each library to evaluate the benchmarks. Smaller values indicate better performance. Empty values represent benchmarks not supported by the library. The common mean is computed over the underlined benchmarks, which are common across libraries; whereas the overall mean is computed over the available benchmarks in each library, ignoring the missing values (if any).

benchmark	cfis	matlab	fuzzylite	jfuzzylite	jfuzzylogic	octave
Bell	0.116	0.162	0.595	1.114	1.628	955.379
Binary			0.170	0.193	2.501	
Concave			0.199	0.269	2.486	
Constant	0.006	0.007	0.023	0.012	0.073	388.558
Cosine			0.396	1.071	1.712	
Discrete			0.383	0.421	1.548	
Function			4.039	9.310	2.407	
Gaussian	0.116	0.163	0.329	0.841	1.174	941.648
GaussianP.	0.116	0.160	0.378	0.858	1.202	1324.361
Linear	0.005	0.007	0.025	0.013	0.067	387.853
PiShape	0.117	0.161	0.229	0.295	2.206	1094.229
Ramp			0.205	0.281	2.392	
Rectangle			0.181	0.219	2.207	
Sigmoid	0.116	0.161	0.313	0.891	1.245	942.106
SigmoidD.	0.117	0.167	0.445	1.562	2.400	1214.323
SigmoidP.	0.116	0.158	0.425	1.547	2.204	1111.997
Spike			0.321	0.836	2.198	
Trapezoid	0.117	0.166	0.208	0.261	0.294	1179.037
Triangle	0.116	0.166	0.200	0.250	0.312	1202.354
ZSShape	0.116	0.165	0.203	0.279	2.204	1074.589
common mean	0.092	0.129	0.279	0.645	0.933	948.402
overall mean	0.098	0.137	0.463	1.026	1.623	984.703
rank	**1st**	2nd	3rd	4th	5th	6th

2) jfuzzylite vs jfuzzylogic: The jfuzzylite library is faster than jfuzzylogic by $0.933 \div 0.645 = 1.5\text{x}$. We expected jfuzzylite to be faster partly because of the simpler class structure, hierarchy, and organization, which helps to optimize for performance; but also because, after reviewing the source code of jfuzzylogic, we identified the following characteristics that slows it down. **Defuzzification**: for Mamdani-based FLCs, the cost of the defuzzification process in jfuzzylogic is higher because (a) the fuzzy output is discretized and stored into an array (see RuleActivationMethod::imply), (b) the array is initialized to zero (see DefuzzifierContinuous::reset), and (c) the array is iterated again to compute the centroid (see DefuzzifierCenterOfGravity::defuzzify) . Differently, the defuzzification process in jfuzzylite computes the centroid of the fuzzy output at the same time it is being discretized (see Centroid::defuzzify), thereby avoiding the memory allocation of the array and the storage of its values. **Reference values**: in jfuzzylogic, every linguistic variable (input and output) maintains and updates a copy of the values of the other variables in the engine in order to use them if needed (see Variable::needEstimateUniverse), but this mechanism is only useful for the output variables that contain `Function` and `Linear` terms. Differently, reference values in jfuzzylite are only maintained and updated in `Function` terms (see Function::membership), as `Linear` terms have direct access to the reference values by index (see Linear::membership). **Discrete terms**: a discrete term with resolution r is defined by a set of n pairs (x_i, y_i) sorted ascending by x_i. Thus, to compute the membership function $\mu(v)$ of a discrete term it is necessary to (1) find the x_i value closest to v, (2) identify x_{i-1} and x_{i+1}, and (3) linearly interpolate the result between $\mu(x_{i-1})$ and $\mu(x_{i+1})$. In jfuzzylogic, the first step utilizes linear search (see MembershipFunctionPieceWiseLinear::membership), which has a computational cost $\mathcal{O}(n)$; whereas jfuzzylite utilizes the more efficient binary search (see Discrete::membership), which has a logarithmic computational cost $\mathcal{O}(\log n)$.

3) fuzzylite vs cfis: The fuzzylite library is slower than cfis by $0.279 \div 0.092 = 3.0$x. It was not surprising to find that cfis outperforms fuzzylite given the following key differences. **Programming paradigm**: fuzzylite is programmed following the object-oriented paradigm, where the FLC and its operation are structured into a set of classes and methods; whereas the cfis library is programmed following the procedural paradigm, where the FLC and its operation are structured into subroutines (defined as static methods) and a few primitive structures that group related variables. Specifically, fuzzylite dynamically allocates objects corresponding to linguistic terms, variables, rules, fuzzy logic operators, and defuzzifiers; whereas cfis utilizes pointers to static methods for the most part (see methods fisComputeInputMfValue, fisFinalOutputMf2, and fisEvaluate). As such, fuzzylite requires not only more memory but also predominantly uses the (slower) heap memory, whereas cfis requires less memory and predominantly uses the (faster) stack memory. Besides memory allocation, fuzzylite is more computationally expensive due to the use of inheritance, templates, exceptions, and the C++ STL, although these characteristics also make fuzzylite importantly more flexible. **Rule activation**: fuzzylite represents the antecedents of each rule in a binary expression tree, whereas cfis represents them in an index-based array whose length and order correspond to the input variables in the engine. As such, the evaluation of the expression tree in fuzzylite is more computationally expensive than the array iteration in cfis, but again fuzzylite provides more flexibility. Specifically, in a single rule, fuzzylite allows to combine conjunction and disjunction operators, group propositions in antecedents *ad hoc*, use multiple hedges, and even use output variables in the antecedents; whereas cfis constrains the antecedents of each rule to input variables, which can be joined exclusively by either conjunction or disjunction, with no support for *ad hoc* groups, and only a limited options of hedges.

4) fuzzylite vs matlab: The fuzzylite library is slower than matlab by $0.279 \div 0.129 = 2.2$x. From all the libraries, we had the most uncertainty on what to expect against matlab. On the one hand, fuzzylite runs natively and the source code has been thoroughly optimized. On the other hand, while matlab interprets the source code, it also has the JIT compiler, and it is popular for its efficient operation on large matrices. After disabling the JIT compiler with the command `feature accel off`, matlab's performance deteriorated, but was still better than fuzzylite. Therefore, it is not only the JIT compiler that favors matlab's performance, but there are other features that makes it faster than fuzzylite. After profiling the benchmarks in matlab, we found that matlab utilizes a precompiled binary file (namely `evalfismex.mexmaci64`) that links matlab to the cfis library. As such, matlab interprets the code to run the benchmarks, but links to the cfis library directly to operate the FLCs. Hence, the linkage between matlab and cfis explains why matlab is slower than cfis by $0.129 \div 0.092 = 1.4$x, but faster than fuzzylite by 2.2x.

5) fuzzylite vs octave: The fuzzylite library is faster than octave by $948.402 \div 0.279 = 3399.3$x. We expected fuzzylite to be faster because, besides being compiled into a native executable, FLCs in octave are interpreted, its JIT compiler is still in experimental stage (and not readily available for use), and octave does not have pre-compiled binaries to link against (unlike matlab). While the difference in performance is very large, we can expect the performance of octave to improve once its JIT compiler leaves the experimental stage and is ready to be used.

6) Current and previous versions of fuzzylite and jfuzzylite: Using the same design of experiments described in Section V, we evaluated the performance of the previous versions of fuzzylite (5.1) and jfuzzylite (5.1). The overall performance of the current versions is better than their previous versions, as shown in Figure VII. According to the one-sided pairwise Wilcoxon rank sum test (at $\alpha = 0.05$ with Holm correction), the difference in performance between each pair of libraries is statistically significant, except between fuzzylite 5 and jfuzzylite 6.

The current version of fuzzylite is faster than its previous version by $0.646 \div 0.281 = 2.3$x, and the current version of jfuzzylite is faster than its previous version by $1.268 \div 0.612 = 2.1$x. The changes responsible for these performance improvements are the following. **Takagi-Sugeno**: the weighted defuzzifiers in previous versions computed the weighted sum and weighted average utilizing instances of `TNorm` and `SNorm` for the products and sums, respectively (e.g., see WeightedAverageCustom::defuzzify). However, the current versions have simplified the weighted defuzzifiers to have their traditional operation using primitive sums and products, thereby reducing their computational cost (e.g., see WeightedAverage::defuzzify). **Membership functions**: the computation of the membership functions for `GaussianProduct`, `PiShape`, `SigmoidProduct`, `ZShape`, and `Discrete` were optimized; especially the `Discrete` term, which now utilizes the more efficient binary search instead of linear search (see Discrete::membership). **Downcasting**: additional logic was included to guarantee safe typecasting of objects (down the inheritance hierarchy of classes) without using the costly `dynamic_cast` in C++ [42] and `instanceof` in Java [43]. In particular, the changes were made to the evaluation of the antecedents of the rules (e.g., see Antecedent::activationDegree), which requires to downcast each variable to determine if it is an input or an output variable, and downcast each node in the binary expression tree to determine if it is an operator (i.e., conjunction or disjunction) or a proposition (e.g., "direction *is* left"). **Inlined methods**: (C++ only) the overhead of method calls was reduced by declaring `inline` all the methods for tolerance-based comparison of floating-point values, amongst other methods (see the Operation class). **Memory allocation**: (C++ only) the heap allocation of `Activated` terms in the fuzzy output was replaced for stack allocation (see the definition of the Aggregated term). **Logging**: (Java only) in the current version, checks are made whether to log or debug information; whereas in the previous version the responsibility was delegated to the logger.

TABLE VII: Performance of the current and previous versions of fuzzylite and jfuzzylite on the benchmarks. The values represent the average computational time (in seconds) required by each library to evaluate the benchmarks. Smaller values indicate better performance. Empty values represent benchmarks not supported by the library. The common mean is computed over the benchmarks available in all libraries. The overall mean is computed over all the benchmarks in each library.

benchmark	fuzzylite 6	fuzzylite 5	jfuzzylite 6	jfuzzylite 5
Bell	0.595	0.608	1.114	1.670
Binary	0.170		0.193	
Concave	0.199	0.277	0.269	0.780
Constant	0.023	0.037	0.012	0.520
Cosine	0.396	0.475	1.071	1.639
Discrete	0.383	5.660	0.421	2.722
Function	4.039	3.940	9.310	10.436
Gaussian	0.329	0.348	0.841	1.349
GaussianP.	0.378	0.641	0.858	2.060
Linear	0.025	0.039	0.013	0.526
PiShape	0.229	0.399	0.295	0.809
Ramp	0.205	0.365	0.281	0.790
Rectangle	0.181	0.259	0.219	0.738
Sigmoid	0.313	0.321	0.891	1.410
SigmoidD.	0.445	0.454	1.562	2.081
SigmoidP.	0.425	0.449	1.547	2.066
Spike	0.321	0.334	0.836	1.349
Trapezoid	0.208	0.327	0.261	0.770
Triangle	0.200	0.328	0.250	0.763
ZSShape	0.203	0.314	0.279	0.780
common mean	0.479	0.820	1.070	1.750
overall mean	0.463	0.820	1.026	1.750
rank	**1st**	2nd	3rd	4th

B. Ranking

The ranking of the benchmarks according to the average computational performance is presented in Table VIII, where the benchmarks in each library have been grouped into four clusters. The average computational performance is presented in Figure 8, providing more details about the ranking.

The goal of ranking and grouping the benchmarks according to their computational performance is to provide guidelines to design more efficient FLCs. Specifically, considering that the performance of each benchmark is intrinsically related to the linguistic terms used therein, these guidelines provide useful information on the tradeoff between the computational complexity and the performance of the underlying membership functions. Besides the differences in performance identified in Section VI-A, the rankings show further differences between the libraries regarding the computation of membership functions and mathematic functions, and also regarding the leverage of certain features inherent to the respective programming languages. The next sections describe the ranking of the benchmarks in each library and the main characteristics influencing the ranking.

1) fuzzylite and jfuzzylite: The ranking of the benchmarks in fuzzylite and jfuzzylite is similar. The **first** group consists of the membership functions for Takagi-Sugeno controllers, namely `Constant` and `Linear`, which are respectively computed from multiple constant values and from

TABLE VIII: Ranking of the benchmarks according to computational performance (from better to worse). In each library, the benchmarks are grouped into four clusters according to the k-means algorithm [44].

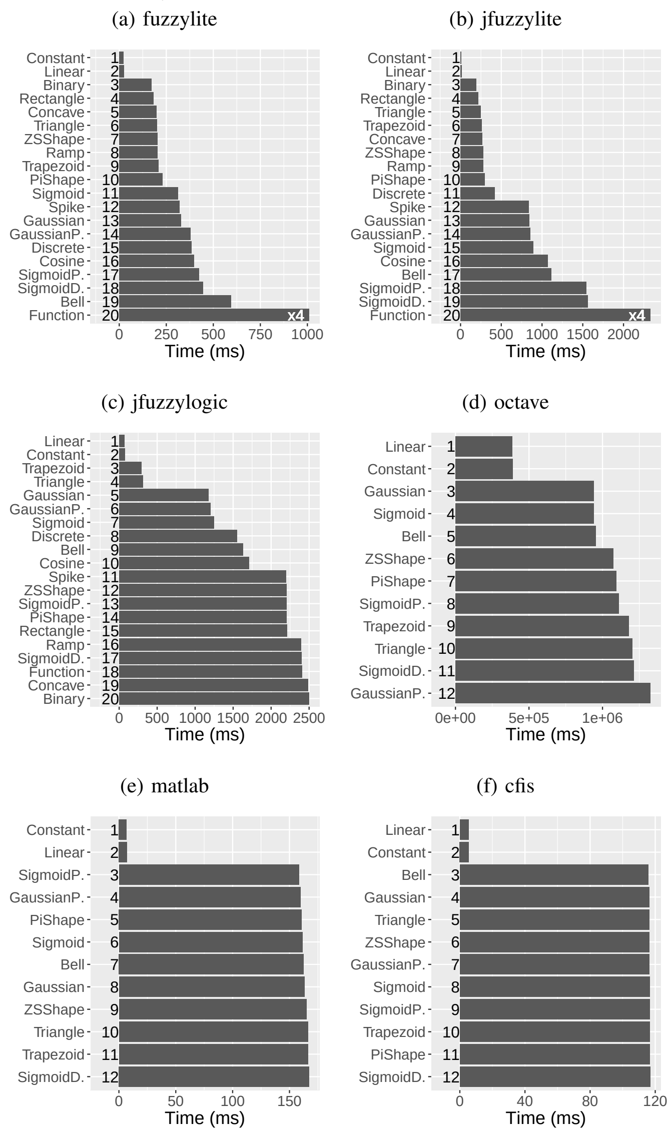
#	fuzzylite	jfuzzylite	jfuzzylogic	octave	matlab	cfis
1	Constant	Constant	Linear	Linear	Constant	Linear
2	Linear	Linear	Constant	Constant	Linear	Constant
3	Binary	Binary	Trapezoid	Gaussian	SigmoidP.	Bell
4	Rectangle	Rectangle	Triangle	Sigmoid	GaussianP.	Gaussian
5	Concave	Triangle	Gaussian	Bell	PiShape	Triangle
6	Triangle	Trapezoid	GaussianP.	ZSShape	Sigmoid	ZSShape
7	ZSShape	Concave	Sigmoid	PiShape	Bell	GaussianP.
8	Ramp	ZSShape	Discrete	SigmoidP.	Gaussian	Sigmoid
9	Trapezoid	Ramp	Bell	Trapezoid	ZSShape	SigmoidP.
10	PiShape	PiShape	Cosine	Triangle	Triangle	Trapezoid
11	Sigmoid	Discrete	Spike	SigmoidD.	Trapezoid	PiShape
12	Spike	Spike	ZSShape	GaussianP.	SigmoidD.	SigmoidD.
13	Gaussian	Gaussian	SigmoidP.			
14	GaussianP.	GaussianP.	PiShape			
15	Discrete	Sigmoid	Rectangle			
16	Cosine	Cosine	Ramp			
17	SigmoidP.	Bell	SigmoidD.			
18	SigmoidD.	SigmoidP.	Function			
19	Bell	SigmoidD.	Concave			
20	Function	Function	Binary			

multiple weighted sums of values. The **second** group consists of the membership functions computed with arithmetic operators (namely $+, -, \times$, and $\div$, in `Binary`, `Concave`, `Ramp`, `Rectangle`, `Trapezoid`, and `Triangle`) and the exponentiation function (namely `pow`, in `PiShape`, `ZShape` and `SShape`). The **third** group consists of the membership functions computed with the exponentiation function of Euler's number (namely `exp`, in `Gaussian`, `GaussianProduct`, `Sigmoid`, `SigmoidDifference`, `SigmoidProduct`, and `Spike`); with the trigonometric cosine function (namely `cos`, in `Cosine`); and with both exponentiation function and absolute function (namely `pow` and `abs`, respectively, in `Bell`). The membership function for `Discrete` terms in jfuzzylite is located in the second group limiting the third group, whereas in fuzzylite is well in the third group. Lastly, the **fourth** group contains only the custom `Function`, which is computed by recursively traversing a binary expression tree. Overall, the computational cost of the benchmarks naturally increases with the complexity of the membership functions.

2) jfuzzylogic: The ranking in jfuzzylogic is somewhat different from fuzzylite and jfuzzylite. The **first** group contains the `Constant` and `Linear` membership functions, which are the most computationally efficient across libraries. However, the first group also contains the `Trapezoid` and `Triangle`, which consist of arithmetic operations, and are ranked in the second group of fuzzylite and jfuzzylite. The **second** group contains the remaining membership functions supported by jfuzzylogic, namely `Bell`, `Cosine`, `Discrete`, `Gaussian`, `GaussianProduct`, and `Sigmoid`, whose membership functions are computed utilizing arithmetic operations, exponentiation functions, trigonometric functions, and linear search (accordingly). Lastly, the **third** and **fourth** groups contain the discretized membership functions of the functions that are not readily supported in jfuzzylogic.

Overall, computational cost increases with complexity, but the performance of the discretized membership functions is not consistent. The `Discrete` membership function (ranking 8th)

Fig. 8: Ranking of the benchmarks in each library according to computational performance (from better to worse). The horizontal axis presents a different scale in each library.



and all the discretized membership functions (ranking 11th or worse) should have the same performance, but instead there are important differences between them, hence classifying them into two clusters. Further research on jfuzzylogic could explore these inconsistencies in performance.

3) octave: The ranking in octave is more consistent than jfuzzylogic on the computational cost of the different membership functions. The **first** group contains the `Constant` and `Linear` benchmarks, like the other libraries. The **second** group consists of terms whose membership functions perform only one exponentiation, namely `Bell`, `Gaussian`, and `Sigmoid`. The **third** group consists of terms whose membership functions are more complex: `SShape`, `ZShape` and `PiShape` each utilizes an anonymous function; `SigmoidProduct` utilizes the exponentiation function twice; `Trapezoid` and `Triangle` each utilizes the maximum function (once) and the minimum function (twice); and the `SigmoidDifference` utilizes the exponentiation function (twice), the maximum function (once), and minimum function (twice). Lastly, the **fourth** group consists of the `GaussianProduct`, whose membership function is computed utilizing two anonymous functions (once each) and the exponentiation function (twice). Overall, the differences in the rankings are mostly affected by the number of functions utilized, and particularly by the number of anonymous functions, which is reported to negatively impact the performance of octave [45, p. 201].

4) cfis and matlab: The ranking in cfis is different from the ranking in matlab, with a few exceptions. Both libraries share the following terms in the same groups: `Constant` and `Linear` terms in the **first** group, `Gaussian` in the **third** group, and `Trapezoid` and `SigmoidDifference` in the **fourth** group. The remaining terms are grouped differently. To explain the differences, we looked into Figure 8, and we found that, excluding `Constant` and `Linear`, the differences in performance between the terms in each library are negligible. Such negligible differences are unexpected because they indicate that the evaluation of a simple function (like minimum and maximum) has a similar computational cost to the evaluation of more complex functions (like trigonometric and exponentiation functions). While unexpected, the negligible differences between the terms are responsible for the large differences in the rankings. Specifically, by having such a similar performance, not only the terms have similar probabilities to occupy any given position, but also small variations lead to larger changes in the ranking. Besides the differences in the rankings, Figure 8 shows that the computational performance of matlab compared to cfis is worse by more than 30 milliseconds, which suggests that this is the computational cost for matlab to link to the cfis library (see Section VI-A4). Overall, instead of the four clusters used to group and rank cfis and matlab, only two groups would suffice: the first group containing the `Constant` and `Linear` terms, and the second group containing the remaining terms. Further research on cfis and matlab could explore the reasons behind such negligible differences in performance between the terms.

C. Accuracy

The accuracy of the libraries on the benchmarks is presented in Table IX. The libraries produce the same results in the Takagi-Sugeno benchmarks. However, in the Mamdani benchmarks, the libraries produce different results mainly due to numerical integration methods and rounding errors.

The most accurate libraries are fuzzylite and jfuzzylite having a common output error of $\delta = \pm 0.006\%$, closely followed by octave at $\delta = \pm 0.008\%$. The matlab and cfis libraries are less accurate at $\delta = \pm 0.055\%$, and jfuzzylogic is the least accurate at $\delta = \pm 1.562\%$. The differences on the accuracy between each pair of libraries are statistically significant according to the one-sided pairwise Wilcoxon rank sum test (at $\alpha = 0.05$ with Holm correction). While the differences are statistically significant, their importance should be judged within the application context. In the next sections, we discuss some of the features that explain the differences in accuracy between fuzzylite and the other libraries.

TABLE IX: Accuracy of the libraries on the benchmarks. The values represent the RMSE (multiplied by 10^4) between the output values obtained with each library (using defuzzifiers with resolution $r = 100$) and a better approximation of the true output values (using fuzzylite and defuzzifiers with resolution $r = 100\,000$). Smaller values indicate more accuracy. Empty values represent benchmarks not supported by the library. The common mean is computed over the underlined benchmarks, which are common across libraries; whereas the overall mean is computed over the available benchmarks in each library, ignoring the missing values (if any). The common and overall errors are presented as percentages of their respective (unscaled) error means relative to the range of the variable.

benchmark	fuzzylite / jfuzzylite	octave	matlab / cfis	jfuzzylogic
Bell	0.406	0.411	3.524	76.248
Binary	15.498			55.498
Concave	0.062			151.037
Constant	0.000	0.000	0.000	0.000
Cosine	0.760			36.656
Discrete	1.721			43.482
Function	1.721			43.482
Gaussian	0.218	0.245	7.713	147.264
GaussianProduct	0.218	0.245	7.713	147.264
Linear	0.000	0.000	0.000	0.000
PiShape	0.710	0.686	0.686	35.275
Ramp	0.562			135.952
Rectangle	15.498			55.498
Sigmoid	0.237	0.265	22.111	902.762
SigmoidDifference	0.430	0.435	2.412	41.983
SigmoidProduct	0.430	0.435	2.412	32.438
Spike	0.263			21.121
Trapezoid	2.439	3.278	3.278	47.096
Triangle	1.722	2.322	2.322	43.501
ZSShape	0.148	0.177	23.084	116.082
common mean	0.630	0.800	5.453	156.235
common error (%)	±0.006	±0.008	±0.055	±1.562
overall mean	2.152	0.708	6.271	106.632
overall error (%)	±0.022	±0.007	±0.063	±1.066
rank	**1st**	2nd	3rd	4th

1) Numerical integration methods: The centroid defuzzifiers in the libraries approximate the integral over the fuzzy output values by using different types of Riemann sums. Specifically, fuzzylite and jfuzzylite utilize the midpoint sum, octave utilizes the trapezoidal sum, and cfis, matlab and jfuzzylogic utilize the left sum. Differential calculus shows that the midpoint sum is definitely more accurate than the left (or right) Riemann sums [46, p.907], and can be more accurate than the trapezoidal sum [47, p.529] under certain conditions. The preference of the midpoint sum over the trapezoidal sum can also be found in other works [48]–[51], and we further support such a preference with the experimental evidence of our case study.

2) Rounding errors: Besides integration methods, rounding errors are another factor that affect the accuracy of the libraries, in particular that of jfuzzylogic. Rounding errors are inherent to digital computing because it is not possible to represent every real number in a finite number of bits [52]. Hence, the programming languages on which the libraries are built, handle the real numbers as approximations using the IEEE Standard for Floating-Point Arithmetic (IEEE-754) with double precision (64 bits). The errors on these approximations are unavoidable, and performing arithmetic operations on these approximations further increases the error [52]. However, in jfuzzylogic, the error is larger because the error accumulates at each iteration of the integration method, whereas in the other libraries the error remains constant. Specifically, considering the integration algorithms in Figure 9, which use Riemann sums of the form $\sum f(x)dx$ to approximate the integral over the fuzzy output values, the error of variable x grows in jfuzzylogic because it is incremented at each iteration using `x+=dx`; whereas the error remains constant in FuzzyLite because it is computed independently at each iteration using `x=minimum+(i+0.5)*dx`. Experimentally, the difference between the two approaches is analogous to the cases of computing the multiplication $0.1 \times 8 = 0.8$ (FuzzyLite) and the sum $0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 0.7999999999999999$ (jfuzzylogic) using any programming language compliant with the IEEE-754.

```
//Common variable
final double dx = (maximum - minimum) / resolution;

//jfuzzylogic: DefuzzifierCenterOfGravity.java
public double defuzzify() {
   double sum = 0, weightedSum = 0;
   for (int i = 0; i < values.length; i++, x += dx) {
      sum += values[i];
      weightedSum += x * values[i];
   }
   return weightedSum / sum;
}

//jfuzzylite: Centroid.java
public double defuzzify(Term term) {
   double sum = 0, weightedSum = 0;
   double x, y;
   for (int i = 0; i < resolution; ++i) {
      x = minimum + (i + 0.5) * dx;
      y = term.membership(x);
      weightedSum += y * x;
      sum += y;
   }
   return weightedSum / sum;
}
```

Fig. 9: Integration methods in jfuzzylogic and jfuzzylite (adapted to our case study).

3) Membership functions: Lastly, we explored the source code of the common membership functions across the libraries in order to determine whether implementation differences could be affecting the results. The differences are indicated in Table X and briefly detailed next, but we found that such differences only affect special cases and do not have any effect on our benchmarks. In all membership functions, FuzzyLite feature a multiplier to change the height of the function, but its value is set to 1.0 for all benchmarks, hence not affecting the results. **Bell**: cfis truncates the function when the slope is negative, whereas the other libraries handle negative slopes. **Sigmoid Difference**: octave enforces boundaries for $f(x) \in [0.0, 1.0]$, and jfuzzylogic enforces $f(x) \geq 0.0$; whereas fuzzylite and cfis do not enforce any boundaries. **Trapezoid** and **Triangle**: cfis and octave enforces boundaries for $f(x) \in [0.0, 1.0]$, whereas fuzzylite and jfuzzylogic do not.

D. Overall quality

The overall quality of the libraries is rated according to their performance, accuracy, features, and documentation. The **performance** and **accuracy** are rated according to the results obtained in Sections VI-A and VI-C, respectively. The **features**

TABLE X: Differences in the implementation of the common membership functions across libraries. In each row, different colors mean different implementations, and equivalent colors mean equivalent implementations.

membership function	fuzzylite / jfuzzylite	jfuzzylogic	matlab / cfis	octave
Bell	░	░	▒	░
Gaussian	░	░	░	░
Gaussian Product	░	░	░	░
Sigmoid	░	░	░	░
Sigmoid Difference	░	▒	░	▓
Trapezoid	░	░	▒	▒
Triangle	░	░	▒	▒

are rated according to the following categories in Table XI, namely: controller types, linguistic terms, activation methods, T-norms, S-norms, defuzzifiers, linguistic hedges, importers, and exporters. The **documentation** is rated according to the proportion of documented source code in the libraries as shown in Table XII.

TABLE XI: Proportion of features readily available in the libraries. The proportions in each category are computed as the ratio between the number of components available in the library and the maximum number of components available in any library.

category	fuzzylite / jfuzzylite	octave	jfuzzylogic	matlab / cfis
controllers	$^{3}/_{3} = 1.00$	$^{2}/_{3} = 0.67$	$^{2}/_{3} = 0.67$	$^{2}/_{3} = 0.67$
terms	$^{21}/_{21} = 1.00$	$^{14}/_{21} = 0.67$	$^{14}/_{21} = 0.67$	$^{14}/_{21} = 0.67$
activation	$^{7}/_{7} = 1.00$	$^{1}/_{7} = 0.14$	$^{1}/_{7} = 0.14$	$^{1}/_{7} = 0.14$
t-norms	$^{8}/_{8} = 1.00$	$^{7}/_{8} = 0.88$	$^{6}/_{8} = 0.75$	$^{3}/_{8} = 0.38$
s-norms	$^{8}/_{8} = 1.00$	$^{7}/_{8} = 0.88$	$^{6}/_{8} = 0.75$	$^{3}/_{8} = 0.38$
defuzzifiers	$^{7}/_{8} = 0.88$	$^{8}/_{8} = 1.00$	$^{6}/_{8} = 0.75$	$^{8}/_{8} = 1.00$
hedges	$^{7}/_{7} = 1.00$	$^{6}/_{7} = 0.86$	$^{1}/_{7} = 0.14$	$^{1}/_{7} = 0.14$
importers	$^{3}/_{3} = 1.00$	$^{1}/_{3} = 0.33$	$^{1}/_{3} = 0.33$	$^{1}/_{3} = 0.33$
exporters	$^{7}/_{7} = 1.00$	$^{1}/_{7} = 0.14$	$^{2}/_{7} = 0.29$	$^{1}/_{7} = 0.14$
mean	0.99	0.62	0.50	0.43
rank	**1st**	2nd	3rd	4th

TABLE XII: Percentage of documented source code computed as $100 \times$ comments/code, where comments and code refer to the number of lines of comments and code in each library, respectively. The number of lines in each library was computed using the application `cloc` available at github.com/AlDanial/cloc/.

#	library	code	comments	percentage	rank
1	octave	4067	5824	143.20%	**1st**
2	fuzzylite	14 949	8671	58.00%	2nd
3	jfuzzylite	12 692	7344	57.86%	3rd
4	matlab	13 657	3889	28.48%	4th
5	jfuzzylogic	16 718	4129	24.70%	5th
6	cfis	1992	312	15.66%	6th

The overall quality of the libraries is presented in Table XIII. The libraries that offer the overall best quality are: fuzzylite and jfuzzylite, followed by octave and matlab, and lastly by cfis and jfuzzylogic. However, this ranking needs to be seen for what it is: an overall ranking which does not necessarily hold for every use case, but provides a general overview of the libraries.

VII. Conclusions and Future Work

FLCs are mathematical models designed to control systems by means of fuzzy logic. The seminal ideas of FLCs were published more than 50 years ago, and today there are more than 20 software libraries to design and operate FLCs. Judging by the number of features, support, availability, and popularity, we consider the following open-source libraries to be some of the most relevant today: FuzzyLite, Octave and its Fuzzy Logic Toolkit, Matlab and its Fuzzy Logic Toolbox, and jFuzzyLogic. These libraries have in common the modeling of FLCs, but their implementations differ in performance, accuracy, number of features, and amount of source code documentation. Based on these differences, we created a measure of overall quality to rank the libraries.

TABLE XIII: Overall quality of the libraries. The libraries are rated in each category with values between 0 and 5, where higher values indicate better ratings. The libraries are rated in each category relative to their ranks, i.e., the best is rated with 5, the second best with 4, and so on. Values in bold indicate the best library in the category.

	fuzzylite	jfuzzylite	octave	matlab	cfis	jfuzzylogic
performance	3.0	2.0	0.0	4.0	**5.0**	1.0
accuracy	**5.0**	**5.0**	4.0	3.0	3.0	2.0
features	**5.0**	**5.0**	4.0	2.0	2.0	3.0
documentation	4.0	3.0	**5.0**	2.0	0.0	1.0
mean	**4.25**	3.75	3.25	2.75	2.50	1.75
rank	**1st**	2nd	3rd	4th	5th	6th

The FuzzyLite libraries, introduced here in detail for the first time, rank best for overall quality because they offer the most accurate results, the highest number of features, the second best performance, and the second most documented source code. Besides these favourable indicators, the FuzzyLite libraries also stand out for simplicity, flexibility, compatibility, and portability thanks to their object-oriented design, FuzzyLite Language and component factories, importers and exporters to other libraries, and availability for desktop, mobile, and robotic platforms. However, the FuzzyLite libraries are missing some features that the other libraries provide, namely fuzzy data clustering and neuro-fuzzy modeling (by Matlab and Octave), and algorithms for parameter optimization (by jFuzzyLogic). We intend to incorporate these features in the future.

The next libraries in the ranking of overall quality are Octave, Matlab, and jFuzzyLogic, respectively. Octave stands out primarily for having the most documented source code, and secondarily for the accuracy of its results; but its performance is rather poor in comparison. Matlab stands out for having the best performance, but lacks features of FLCs that the other libraries provide. Lastly, jFuzzyLogic provides the third highest number of features (more than Matlab), but lags behind in performance and accuracy. Hence, each library has its own advantages, disadvantages, and tradeoffs in terms of performance, accuracy, number of features, and source code documentation.

The performance of the different configurations of Takagi-Sugeno and Mamdani-based FLCs across libraries provide guidelines to design more efficient controllers in each of them. On the one hand, the Takagi-Sugeno FLCs are the most computationally efficient configurations across libraries because their defuzzification process does not approximate the integral over the fuzzy output values, but instead performs a few arithmetic operations. The performance ranking of Takagi-Sugeno FLCs across libraries coincides with the overall ranking, except for with one remarkable exception: jfuzzylite is faster than its homologous fuzzylite thanks to Java's JIT compiler. On the other hand, the performance of Mamdani FLCs is not consistent across libraries because of implementation differences, characteristics of the programming languages, and some reasons to be determined. Specifically, the FuzzyLite libraries present a natural correlation between the complexity of the membership functions and their computational cost (and so does jFuzzyLogic, mostly). Octave's performance is negatively affected by the number of functions (and especially anonymous functions) used to compute the membership functions. Lastly, Matlab's performance does not seem affected by the complexity of the membership functions, which is unexpected and its reasons still need to be determined.

Overall, our study provides important information for the users of the libraries and for the general open-source community. For the users, we expect they will be able to make better and more informed decisions when choosing a library to suit their needs. For the open-source community, we hope our study will be useful to encourage more contributions in order to continue improving the quality of the libraries.

Further research could focus on the following suggestions.

- Implement the fuzzy data clustering, neuro-fuzzy modeling, and type-II FLCs [53] in the FuzzyLite libraries.
- Design more benchmarks to evaluate different configurations of FLCs including more linguistic variables, linguistic terms, and rules, and also Tsukamoto controllers.
- Evaluate the libraries using additional indicators such as memory usage, size of binaries, and external dependencies.
- Compare the libraries using software metrics for coupling, cohesion, and complexity; as well as other metrics that may provide indicators of quality.
- Compare the performance of the fuzzylite library when compiled using the GNU Compiler Collection (`gcc`) and the Microsoft Optimizing Compiler (`cl`).
- Compare the FuzzyLite Language in terms of performance, verbosity, complexity, and flexibility against other alternatives such as the FCL, FIS, and Fuzzy Markup Language [54] when importing and exporting FLCs.
- Further investigate the reasons behind Matlab's best performance, and the few performance inconsistencies in jFuzzyLogic.

Acknowledgments

Thank you to the FuzzyLite community, whose feedback and support has been of great importance to improve the libraries. Thank you to the European Centre for Soft Computing, the Foundation for the Advancement of Soft Computing, Sergio Guadarrama (Google) and Luis Magdalena (Universidad Politécnica de Madrid) for their initial support creating fuzzylite. Thank you to Pablo Cingolani (jFuzzyLogic), Linda Markowsky (Octave Fuzzy Logic Toolkit), Roger Jang (Matlab Fuzzy Logic Toolbox), Edward S. Sazonov (FuzzyEngine), and their respective collaborators for open sourcing their libraries. Special thanks to Mengjie Zhang and Mark Johnston (Victoria University of Wellington, New Zealand); Oscar Cordón (Universidad de Granada, Spain); Maritza Bracho, José G. Sánchez, Carlos Lameda, Belkys Lameda, and Rubén Parma (Universidad Centroccidental Lisandro Alvarado, Venezuela); Yelitza Oviedo, Jorge Rodríguez, Edecio Freitez, Luis Alvarado, Jesús Contreras, and Oswaldo Hernández (Universidad Fermín Toro, Venezuela); and Úber Calderón (Universidad Latina, Costa Rica).

Juan Rada-Vilela received his PhD degree in 2014 from Victoria University of Wellington (New Zealand). Dr Rada-Vilela also holds a Master's degree in Soft Computing and Intelligent Data Analysis from the European Centre for Soft Computing (Spain), a Master of Science in Artificial Intelligence from Universidad Centroccidental Lisandro Alvarado (Venezuela), and a Bachelor in Computer Engineering from Universidad Fermín Toro (Venezuela). His research interests are in Artificial Intelligence and Software Engineering, specifically in Fuzzy Logic Control, Swarm Intelligence, Evolutionary Algorithms, Neural Networks; Software Design and Development, Object-Oriented Programming, and SQL/NoSQL Databases. His research output has been published in seventeen articles in some of the main journals and conferences in the field of Soft Computing. Dr Rada-Vilela is the founder and director of FuzzyLite Limited, which is a company based in Wellington (New Zealand), where he continues to design and develop the FuzzyLite libraries.

References

[1] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[2] İ. Özkan and I. B. Türkşen, *Uncertainty and Fuzzy Decisions*. Springer, 2014, pp. 17–27.

[3] E. Hüllermeier, "Fuzzy methods in machine learning and data mining: Status and prospects," *Fuzzy Sets and Systems*, vol. 156, no. 3, pp. 387–406, 2005.

[4] C. Kahraman, *Fuzzy Multi-Criteria Decision Making: Theory and Applications with Recent Developments*, 1st ed. Springer, 2008.

[5] P. Benavidez, J. Lambert, A. Jaimes, and M. Jamshidi, "Landing of an ardrone 2.0 quadcopter on a mobile base using fuzzy logic," *International Journal of Complex Systems – Computing, Sensing and Control*, vol. 1, no. 1-2, pp. 5–25, 2013.

[6] J. P. Rastelli and M. S. Peñas, "Fuzzy logic steering control of autonomous vehicles inside roundabouts," *Applied Soft Computing*, vol. 35, pp. 662–669, 2015.

[7] G. K. Venayagamoorthy, L. L. Grant, and S. Doctor, "Collective robotic search using hybrid techniques: Fuzzy logic and swarm intelligence inspired by nature," *Engineering Applications of Artificial Intelligence*, vol. 22, no. 3, pp. 431–441, 2009.

[8] K. Kapitanova, S. H. Son, and K.-D. Kang, "Using fuzzy logic for robust event detection in wireless sensor networks," *Ad Hoc Networks*, vol. 10, no. 4, pp. 709–722, 2012.

[9] D. Johnson and J. Wiles, "Computer games with intelligence," in *Proceedings of the 10th IEEE International Conference on Fuzzy Systems*, vol. 3, 2001, pp. 1355–1358.

[10] H. B. Verbruggen, H.-J. Zimmerman, and R. Babŭska, Eds., *Fuzzy Algorithms for Control*. International Series in Intelligent Technologies, 1999.

[11] M. Mahfouf, M. Abbod, and D. Linkens, "A survey of fuzzy logic monitoring and control utilisation in medicine," *Artificial Intelligence in Medicine*, vol. 21, no. 1–3, pp. 27–42, 2001.

[12] Y. Jin and L. Wang, *Fuzzy Systems in Bioinformatics and Computational Biology*, 1st ed. Springer, 2009.

[13] O. N. Jensen, P. Mortensen, O. Vorm, and M. Matthias, "Automation of matrix-assisted laser desorption/ionization mass spectrometry using fuzzy logic feedback control." *Analytical Chemistry*, vol. 69, no. 9, pp. 1706–1714, 1997.

[14] B. Center and B. P. Verma, "Fuzzy logic for biological and agricultural systems," *Artificial Intelligence Review*, vol. 12, no. 1, pp. 213–225, 1998.

[15] J. Alcalá-Fdez and J. M. Alonso, "A survey of fuzzy systems software: Taxonomy, current research trends, and prospects," *IEEE Transactions on Fuzzy Systems*, vol. 24, no. 1, pp. 40–56, 2016.

[16] P. Cingolani and J. Alcalá-Fdez, "jFuzzyLogic: a robust and flexible fuzzy-logic inference system language implementation," in *Proceedings of the IEEE International Conference on Fuzzy Systems*, 2012, pp. 1–8.

[17] ——, "jFuzzyLogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming," *International Journal of Computational Intelligence Systems*, pp. 61–75, 2013.

[18] MathWorks, *Fuzzy Logic Toolbox (TM) User's Guide*. Natick, MA: The MathWorks, Inc., 2016. [Online]. Available: mathworks.com/help/pdf_doc/fuzzy/index.html

[19] L. Markowsky and B. Segee, "The octave fuzzy logic toolkit," in *Proceedings of the International Workshop on Open-Source Software for Scientific Computation*, 2011, pp. 118–125.

[20] J. Rada-Vilela, "fuzzylite: a fuzzy logic control library in C++," in *Proceedings of the Open Source Developers Conference*, 2013. [Online]. Available: fuzzylite.com/downloads

[21] S. Sonnenburg, M. L. Braun, C. S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.-R. Müller, F. Pereira, C. E. Rasmussen, G. Rätsch, B. Schölkopf, A. Smola, P. Vincent, J. Weston, and R. Williamson, "The need for open source software in machine learning," *Journal of Machine Learning Research*, vol. 8, pp. 2443–2466, 2007.

[22] J.-S. Jang, "ANFIS: adaptive-network-based fuzzy inference system," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 3, pp. 665–685, 1993.

[23] M.-S. Yang, "A survey of fuzzy clustering," *Mathematical and Computer Modelling*, vol. 18, no. 11, pp. 1–16, 1993.

[24] E. H. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International Journal of Man-Machine Studies*, vol. 7, no. 1, pp. 1–13, 1975.

[25] M. Mas, M. Monserrat, J. Torrens, and E. Trillas, "A survey on fuzzy implication functions," *IEEE Transactions on Fuzzy Systems*, vol. 15, no. 6, pp. 1107–1121, 2007.

[26] T. Takagi and M. Sugeno, "Fuzzy identification of systems and its applications to modeling and control," *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-15, no. 1, pp. 116–132, 1985.

[27] R. Shoureshi and Z. Hu, "Tsukamoto-type neural fuzzy inference network," in *Proceedings of the American Control Conference*, vol. 4, 2000, pp. 2463–2467.

[28] C. C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller — Parts I and II," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, no. 2, pp. 404–435, 1990.

[29] L. A. Zadeh, "The concept of a linguistic variable and its application to approximate reasoning - Part I," *Information Sciences*, vol. 8, no. 3, pp. 199–249, 1975.

[30] ——, "The concept of a linguistic variable and its application to approximate reasoning - Part II," *Information Sciences*, vol. 8, no. 4, pp. 301–357, 1975.

[31] W. V. Leekwijck and E. E. Kerre, "Defuzzification: criteria and classification," *Fuzzy Sets and Systems*, vol. 108, no. 2, pp. 159–178, 1999.

[32] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[33] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java (TM) Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.

[34] G. Booch, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison Wesley Longman Publishing Co., Inc., 2004.

[35] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[36] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, and B. Steece, *Software Cost Estimation with Cocomo II with Cdrom*, 1st ed. Prentice Hall PTR, 2000.

[37] E. Dijkstra, "Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60," Stichting Mathematisch Centrum, Tech. Rep., 1961. [Online]. Available: oai.cwi.nl/oai/asset/9251/9251A.pdf

[38] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2000.

[39] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: R-project.org/

[40] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009. [Online]. Available: ggplot2.org/

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[42] M. Gregoire, *Professional C++*, 3rd ed. Wrox Press Ltd., 2014.

[43] C. Hunt and B. John, *Java Performance*, 1st ed. Prentice Hall Press, 2011.

[44] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. University of California Press, 1967, pp. 281–297.

[45] J. W. Weaton, D. Bateman, S. Hauberg, and R. Wehbring, *GNU Octave: Free Your Numbers*, 2016. [Online]. Available: gnu.org/software/octave/octave.pdf

[46] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007.

[47] G. Dahlquist and A. Björck, *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics, 2008, vol. 1.

[48] P. C. Hammer, "The midpoint method of numerical integration," *Mathematics Magazine*, vol. 31, no. 4, pp. 193–195, 1958.

[49] P. J. Davis and P. Rabinowitz, "Chapter 2 - approximate integration over a finite interval," in *Methods of Numerical Integration*, 2nd ed., P. J. Davis and P. Rabinowitz, Eds. Academic Press, 1984, pp. 51–198.

[50] M. Ortiz and E. P. Popov, "Accuracy and stability of integration algorithms for elastoplastic constitutive relations," *International Journal for Numerical Methods in Engineering*, vol. 21, no. 9, pp. 1561–1576, 1985.

[51] J. F. Kraaijevanger, "B-convergence of the implicit midpoint rule and the trapezoidal rule," *BIT*, vol. 25, no. 4, pp. 652–666, Dec. 1985.

[52] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[53] N. N. Karnik, J. M. Mendel, and Q. Liang, "Type-2 fuzzy logic systems," *IEEE Transactions on Fuzzy Systems*, vol. 7, no. 6, pp. 643–658, 1999.

[54] G. Acampora and V. Loia, "Fuzzy markup language: A new solution for transparent intelligent agents," in *IEEE Symposium on Intelligent Agent (IA)*, 2011, pp. 1–6.