Fast Fuzzy Inference in Octave

Piero Molino, Gianvito Pio, Corrado Mencar

Department of Informatics, University of Bari,V. E. Orabona, 4, Bari, 70125, Italy

E-mail: {piero.molino,pio.gianvito}@gmail.com, mencar@di.uniba.it

Abstract

Fuzzy relations are simple mathematical structures that enable a very general representation of fuzzy knowledge, and fuzzy relational calculus offers a powerful machinery for approximate reasoning. However, one of the most relevant limitations of approximate reasoning is the efficiency bottleneck. In this paper, we present a module for fast fuzzy inference through relational composition, with the twofold objective of being general and efficient. We implemented this module in GNU Octave because it is a high-level language targeted to numerical computations. Experimental results show the impressive performance gain when the proposed implementation is used. The module is free available under LGPL licence.

Keywords: GNU Octave; fuzzy relations; fuzzy composition; fuzzy inference.

1. Introduction

Fuzzy Set Theory (FST) is widely recognized as a valid mathematical tool for representing and processing imprecise and gradual knowledge[8]. In particular, FST is capable to represent the semantics of concepts that are usually expressed with natural language terms, which are inherently imprecise. In this way, FST enables the formal representation of linguistically quantified knowledge and provides a mathematical machinery for approximate reasoning[7]. In recent years a number of tools have been developed to express fuzzy knowledge and to provide for deductive inference[*]. Examples of these tools are Xfuzzy[6] (`forja.rediris.es/projects/xfuzzy/`), FisPro[4] (`www.inra.fr/internet/Departements/MIA/M/fispro/`), NEFCLASS[5] (`fuzzy.cs.uni-magdeburg.de/nefclass/`), FuzzyTech (`www.fuzzytech.com/`), Fuzzy Logic Toolbox for Matlab (`www.mathworks.com/products/fuzzy-logic/`) and GUAJE[1] (`www.softcomputing.es/guaje`).

The objective of such tools is to enable knowledge engineers and/or end-users to develop fuzzy expert systems that can be applied in a variety of application domains. Many of available tools are *rule based*, i.e. the knowledge base is expressed in terms of fuzzy rules. This enables an immediate representation of knowledge that can be interpreted in a linguistic fashion. However, rules in fuzzy knowledge bases may have different interpretations[2]; furthermore, fuzzy rule representation is not the unique form for expressing knowledge.

On a more general level, fuzzy rules are par-

[*]Actually, many of such tools have been devised for inductively learning fuzzy knowledge, especially from data; however the analysis of this feature is outside the scope of this paper.

ticular forms of fuzzy *relations* among variables. Thus, fuzzy relations are very general representations of knowledge and fuzzy relational calculus offers a powerful machinery for approximate reasoning. However, one of the most relevant limitations of approximate reasoning is the efficiency bottleneck: while precise reasoning (as in Boolean logic) can be efficiently performed through symbolic computations, in approximate reasoning this is no more possible, since the gradual truth of concepts involved in an inference process must be taken into account. To improve the efficiency of the inference process, either restrictive assumptions on the involved relations need to be made (e.g. only fuzzy rules are allowed) or efficient algorithms for general inference must be devised. The second choice does not require any assumption on the relations used for inference, but requires an effort in designing and implementing the inference algorithms by considering efficiency as a key feature.

In this paper we present a package for fast inference in approximate reasoning that has the twofold objective of being both general and efficient. We implemented this module in GNU Octave (`www.gnu.org/software/octave/`) because it is a high-level language targeted to numerical computations. Furthermore, GNU Octave is a free software and its language is almost identical to the Mathworks^{TM}Matlab$^{®}$language, which is widely known in the FST research community.

In the next Section, a general discussion on fuzzy inference is presented. Next, implementation issues and solutions are described. In Section 4 a quantitative measurement of the efficiency of the proposed procedures is reported. Conclusive remarks end the paper.

2. Fuzzy inference

FST is based on the fundamental concept of fuzzy set defined on a domain (or universe of discourse) U. When a domain can be decomposed as Cartesian product of subdomains, i.e. $U = U_1 \times U_2 \times \cdots \times U_n$, then a fuzzy set on U is called *fuzzy relation* on $U_1, U_2, \ldots U_n$.

A fuzzy relation R maps each element of the domain U to an element of a bounded chain:

$$L = \langle I, \vee, \wedge, 0, 1 \rangle \tag{1}$$

We assume that $I = [0,1] \subset \mathbb{R}$ since a lot of applications are developed on the unitary interval; besides, a numerical interval is the most convenient choice for an implementation in Octave.

The most common choice for the join ($\vee$) and meet ($\wedge$) operators are the maximum (max) and minimum (min) functions, respectively. However, other operators can also be used, among the so-called t-norms (for meet) and t-conorms (for join).

A fuzzy relation

$$R : U_1 \times U_2 \times \cdots \times U_n \mapsto [0,1] \tag{2}$$

defines a connection among variables $x_1, x_2, \ldots, x_n$ defined on their respective domains; the strength of the connection is given by $R(x_1, x_2, \ldots, x_n)$. We restrict our discussion on binary relations, i.e. when $n = 2$. This restriction is not technically limiting, since by defining $X = U_1 \times U_2 \times \cdots \times U_{n-m}$ and $Y = U_{n-m+1} \times \cdots \times U_n$, then the binary relation

$$R : X \times Y \mapsto [0,1] \tag{3}$$

is formally equivalent to (2). Usually, X is called *input domain*, and Y is the *output domain*.

A fuzzy set A defined on X can be considered as a possibility distribution over $n-m$ formal input variables. In other words, a fuzzy set expresses an imprecise (i.e. granular and gradual) fact about the values of these variables. Fuzzy deductive inference takes a fuzzy set A and a fuzzy relation R to provide a new fuzzy set B defined on Y; this fuzzy set restricts the possible values of the formal output variables given the fact A and the relation R:

$$\frac{A, R}{B} \tag{4}$$

As an example, if R encodes a fuzzy rule $\alpha \to \beta$, then fuzzy inference carries out the so-called *Generalized Modus Ponens*

$$\frac{A, \alpha \to \beta}{B} \tag{5}$$

which extends the classical Modus Ponens that is used in Boolean logic.

The basic rule for fuzzy inference in (4) is the relational composition operator, i.e.

$$B = A \circ R \tag{6}$$

In classical set theory, where A, B and R are crisp (i.e. defined on $\{0,1\}$), an element y belongs to B if and only if there exists $x \in A$ and $(x,y) \in R$. By translating this definition in FST, the composition operator can be defined as follows:

$$B(y) = \max_{x \in X} \min \{A(x), R(x,y)\} \tag{7}$$

In a more general setting, (7) is generalized by using a t-norm $\wedge$ in place of the min function, and a t-conorm $\vee$ in place of the max function:

$$B(y) = \vee_{x \in X} (A(x) \wedge R(x,y)) \tag{8}$$

Actually, (8) can be defined when both operands are relations, thus leading to the most general definition of the composition operator:

$$T(w,y) = \vee_{x \in X} (S(w,x) \wedge R(x,y)) \tag{9}$$

where S, R and T are relations on $W \times X$, $X \times Y$ and $W \times Y$ respectively. Eq. (8) specializes (9) by assuming $W = \{\bullet\}$, i.e. a singleton set with a conventional element.

The composition operation (9) is at the basis of fuzzy inference. Hence an efficient implementation should consider this operation as elementary, i.e. not defined in terms of its compound operations. By taking the $\circ$ operator as elementary, fuzzy set operations can be redefined as:

$$A_1 \cap A_2 = A_1 \circ \mathrm{diag}(A_1 \times A_2) \tag{10}$$

and

$$A_1 \cup A_2 = A_1 \circ \mathrm{diag}(A_1 + A_2) \tag{11}$$

being $\mathrm{diag}R(x,y) = R(x,y)$ if $x = y$ and $\mathrm{diag}R(x,y) = 0$ if $x \neq y$. The Cartesian product is defined as $(A_1 \times A_2)(x_1,x_2) = A_1(x_1) \wedge A_2(x_2)$ and the Cartesian co-product is defined as $(A_1 + A_2)(x_1,x_2) = A_1(x_1) \vee A_2(x_2)$.

3. Implementation design

Implementation in Octave takes advantage of the high efficiency of matrix computation. As a consequence, fuzzy sets are represented as vectors and relations as matrices. This requires an implicit ordering of the input and output domains, which is straightforward when they are numeric, while it must be conventionally established when one of them is categorical.

Two different implementations have been designed for a fast computation of the composition operation, which respectively work on full and sparse matrices.[†] The implementation for full matrices (iFM in brief) benefits of the inherent parallelism of the algorithm for computing composition. On the other hand, the implementation for sparse matrices (iSM in brief) is highly efficient when the number of non-zero elements in the operands is small. A checking procedure establishes when the full matrix or the sparse matrix implementation should be applied.

3.1. Implementation for full matrices (iFM)

When this implementation is selected, a systematic approach is applied, which consists in the calculation of all the t-norms/t-conorms required for the composition. Given two input matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times l}$, a total of $n \times m \times l$ t-norm/t-conorm calculations have to be performed. This is the worst case, if compared to the sparse situation explained in the following section. However, this algorithm has the advantage of being easily parallelizable by exploiting multiprocessor systems. Parallelization is achievable because it is possible to know in advance how many operations will be performed and, hence, the computation of the output matrix can be subdivided among all the available processors.

In a system with K available processors, the implementation creates exactly K threads and assigns a portion of the resulting matrix (a subset of rows) to each thread (see fig. 1). This gives the advantage of an almost zero overhead caused by thread handling, because data assignment to each thread is performed only once at the beginning of the procedure and no

[†]Full matrices require an explicit representation of all elements, while sparse matrices require an explicit representation of *non-zero* elements only.

other data assignment operations are performed during the computation.

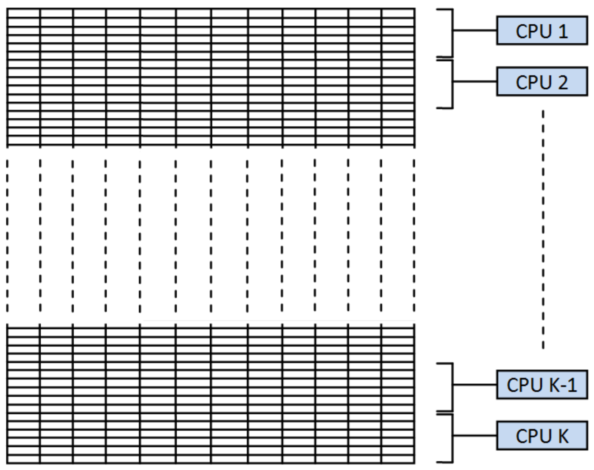

Figure 1: Rows assignment to the available processors. Each processor performs the t-norm/t-conorm calculation for a subset of rows of the resulting matrix.

With this type of implementation, it is possible to perform union and intersection of fuzzy sets in an efficient way. This is accomplished by appropriately locking the cycles, in order to execute only those operations required in (10) and (11). The following pseudo-code illustrates the procedure carried out by each processor.

```
Require: $A \in \mathbb{R}^{n\times m}, B \in \mathbb{R}^{m\times l}$
Require: $i_{\text{start}}, i_{\text{end}}$ {start and end indexes of the result matrix
  assigned to the processor}
Require: $lock$ {=1 for intersection and union, 0 otherwise}
Require: $\otimes : [0,1]^2 \mapsto [0,1]$ {internal operator}
Require: $\oplus : [0,1]^2 \mapsto [0,1]$ {external operator}
Ensure: slice of $C \in \mathbb{R}^{n\times l}$ {result of the composition}
  $inc \leftarrow lock * l + 1$
  for $i = i_{\text{start}} \to i_{\text{end}}$ do
    $j = i * lock$
    repeat
      $s \leftarrow A[i,0] \otimes B[0,j]$
      for $k = 1 \to m-1$ do
        $t \leftarrow A[i,k] \otimes B[k,j]$
        $s \leftarrow s \oplus t$
      end for
      $C[i, j*(1-lock)] \leftarrow s$
      $j \leftarrow j + inc$
    until $j \geqslant l$
  end for
```

When $lock = 0$, the standard composition operator is computed. The index j spans from 0 to $l-1$, thus computing the values of the part of the matrix C assigned to a thread[‡]. On the other hand, when $lock = 1$, the index j is locked to 0, so that C is just a one-dimensional array (corresponding to a fuzzy set). It must be observed that, when intersection or union has to be computed, it is necessary that A and B are vectors, i.e. $m = 1$. In such case, with $lock = 1$ the value of $C[i,0]$ is determined by $A[i,0] \otimes B[0,i]$. If $\otimes$ is a t-norm, then the intersection is calculated; if $\otimes$ is a t-conorm, union is calculated instead. As a final remark, if $lock = 0$ and both A and B are vectors, the Cartesian product (or co-product, depending on the choice of $\otimes$) is computed.

3.2. *Implementation for sparse matrices (iSM)*

Sparse matrices in Octave are stored using the *Column Compressed Storage* (CCS) technique[3]. This technique consists in storing three arrays for each matrix: `data`, `cidx` and `ridx`. The `data` array contains only the nonzero values of the matrix, in columnwise order. The `ridx` array contains the row indexes of the nonzero elements, aligned with the `data` array. The `cidx` array stores the locations in the `data` array that start a column. The number of nonzero elements in the i-th column is given by $\texttt{cidx}[i+1] - \texttt{cidx}[i]$. As a convention, the first element of `cidx` is 0 and the last one is the number of nonzero elements of the matrix.

If the input matrices are sparse it is not useful to calculate the t-norm over the zero elements as the result will always be zero and will not influence the resulting matrix. Hence, in view of an efficient implementation, the computation of the t-norm over those elements should be avoided.

As an example, given the following matrices:

$$A = \begin{bmatrix} 0 & 0.6 & 0 \\ \mathbf{0.5} & \mathbf{0} & \mathbf{0.3} \\ 0 & 0.1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.9 & \mathbf{0} & 0 \\ 0 & \mathbf{0.2} & 0 \\ 0.8 & \mathbf{0.7} & 0.1 \end{bmatrix}$$

‡We assume that the array indexes start from zero.

and max/min for composition, the value of $C[1,1]$ is determined by:

$$C[1,1] = (A[1,0] \wedge B[0,1]) \vee (A[1,1] \wedge B[1,1]) \vee (A[1,2] \wedge B[2,1]) = 0.3$$

We observe that the final value is only due to the computation of $(A[1,2] \wedge B[2,1])$ because both $B[0,1]$ and $A[1,1]$ are null. To get rid of useless t-norm calculations, we consider the CCS representations of $A^\top$ and B:

$$\begin{array}{lclllll} \texttt{data}(A^\top) & = & 0.6 & 0.5 & 0.3 & 0.1 \\ \texttt{ridx}(A^\top) & = & 1 & 0 & 2 & 1 \\ \texttt{cidx}(A^\top) & = & 0 & 1 & 3 & 4 \end{array}$$

and

$$\begin{array}{lcllllll} \texttt{data}(B) & = & 0.9 & 0.8 & 0.2 & 0.7 & 0.1 \\ \texttt{ridx}(B) & = & 0 & 2 & 1 & 2 & 2 \\ \texttt{cidx}(B) & = & 0 & 2 & 4 & 5 & \end{array}$$

To compute $C[1,1]$, the non-zero elements of column 1 of both $A^\top$ and B must be considered. The values of $\texttt{cidx}(A^\top)[1]$ and $\texttt{cidx}(B)[1]$ refer to the elements on the corresponding `data` arrays where such columns begin. In the example, $\texttt{cidx}(A^\top)[1] = 1$ and $\texttt{cidx}(B)[1] = 2$; furthermore, $\texttt{cidx}(A^\top)[2] - \texttt{cidx}(A^\top)[1] = 2$ and $\texttt{cidx}(B)[2] - \texttt{cidx}(B)[1] = 2$, meaning that both columns have two non-zero elements. By looking at `ridx` vectors, we observe that column 1 of $A^\top$ has two non-zero elements in rows 0 and 2 and column 1 of B has two non-zero elements in rows 1 and 2. A simple comparison loop shows that, in column 1, the only row where both $A^\top$ and B have a non-zero value is row 2. The computation of the t-norm on these cells yields the expected result 0.3.

In general, the following procedure is run to compute the composition operator in presence of sparse matrices:

```
**Require:** $A^\top \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{m\times l}$ {in CCS representation}
**Require:** $\otimes : [0,1]^2 \mapsto [0,1]$ {internal operator}
**Require:** $\oplus : [0,1]^2 \mapsto [0,1]$ {external operator}
**Ensure:** $C \in \mathbb{R}^{n\times l}$ {result of composition}
  {due to CCS representation, operations will be done on $A^\top$ and $C^\top$}
  $k_{C^\top} \leftarrow 0$ {data index of $C^\top$}
  **for** $i = 0 \to n-1$ **do**
    $\texttt{cidx}(C^\top)[i] \leftarrow k_{C^\top}$
    **for** $j = 0 \to l-1$ **do**
      $k_{A^\top} \leftarrow \texttt{cidx}(A^\top)[i]$ {starting point of column $i$ in $A^\top$}
      $k_B \leftarrow \texttt{cidx}(B)[j]$ {starting point of column $j$ in $B$}
      $nz_{A^\top} \leftarrow \texttt{cidx}(A^\top)[i+1] - \texttt{cidx}(A^\top)[i]$ {number of non-zero elements in column $i$ of $A^\top$}
      $nz_B \leftarrow \texttt{cidx}(B)[i+1] - \texttt{cidx}(B)[i]$ {number of non-zero elements in column $i$ of $B$}
      $s \leftarrow 0$
      $k_{A^\top}^{\text{end}} \leftarrow k_{A^\top} + nz_{A^\top}$ {limit for $k_{A^\top}$}
      $k_B^{\text{end}} \leftarrow k_B + nz_B$ {limit for $k_B$}
      **while** $k_{A^\top} < k_{A^\top}^{\text{end}} \wedge k_B < k_B^{\text{end}}$ **do**
        **if** $\texttt{ridx}(A^\top)[k_{A^\top}] = \texttt{ridx}(B)[k_B]$ **then**
          $t \leftarrow \texttt{data}(A^\top)[k_{A^\top}] \otimes \texttt{data}(B)[k_B]$
          $s \leftarrow s \oplus t$
          $k_{A^\top} \leftarrow k_{A^\top} + 1$
          $k_B \leftarrow k_B + 1$
        **else if** $\texttt{ridx}(A^\top)[k_{A^\top}] < \texttt{ridx}(B)[k_B]$ **then**
          $k_{A^\top} \leftarrow k_{A^\top} + 1$
        **else**
          $k_B \leftarrow k_B + 1$
        **end if**
      **end while**
      **if** $s \neq 0$ **then**
        $\texttt{ridx}(C^\top)[k_{C^\top}] \leftarrow j$
        $\texttt{data}(C^\top)[k_{C^\top}] \leftarrow s$
        $k_{C^\top} \leftarrow k_{C^\top} + 1$
      **end if**
    **end for**
  **end for**
  $\texttt{cidx}(C^\top)[n] \leftarrow k_{C^\top}$
```

Differently from the implementation for full matrices, the internal loop in this implementation does not necessarily cycle m times; rather, the number of cycles is reduced when zero elements occur in the columns of $A^\top$ or B involved in the computation. Under the sparsity assumption, this greatly reduces the number of computations.

3.3. Implementation selection

The selection of the implementation for computing the composition operator is either in charge of the programmer or can be made automatically, by verifying a number of conditions. In particular, the sparse matrix implementation is selected only when the following conditions are met:

1. Both operand matrices are two-dimensional (i.e. they are not vectors);
2. Sparse matrix implementation is more convenient than full matrix implementation.

Whilst the first condition is immediate to verify, the second one is far more complex. Empirical evidence shows indeed that the most dominant operations (i.e. the operations that require most execution time) in iSM are comparisons made in the branching conditions, likely because of the required access to main memory. However, these operations do not appear in iFM: this makes impractical any attempt to compare the two implementations in terms of dominant operations. In order to define a heuristic strategy for automatic implementation selection, the following steps have been executed.

3.3.1. iFM time estimation

Since iFM computation time is insensitive to matrix density but it only depends on the size of the involved matrices and on the number of available processors, we randomly generated five couples of 1000×1000 matrices to be applied to iFM for their composition. We recorded the average time required for composition and repeated the process by varying the number of involved processors in the set $\{1,2,4\}$. We obtained an average time of 7.16 [secs $\cdot$ processors] and a very small standard deviation (0.02 [secs] with four processors).

3.3.2. iSM time estimation

Similarly to the first step, we generated random matrices to be applied to iSM. However, since iSM computing time is tightly related to matrix densities, we generated five couples of 1000×1000 matrices for each couple of density degrees (d_1, d_2) in $\{.01, .02, .05, .10, .20, \ldots, .80, .90, .95, .98, .99\}^2$. By density degree we define the ratio between non-zero elements and the nominal size of the matrix (num. of rows $\times$ num. of columns). We applied iSM to all these matrices and recorded the average time at varying density degrees.

According to statistical reasoning, it is possible to show that the expected execution time is related to a quadratic model on (d_1, d_2). More specifically, we fitted the model

$$\alpha d_1 d_2 + \beta d_1(1-d_2) + \gamma d_2(1-d_1) + \delta \qquad (12)$$

to the observed execution times, obtaining: $\alpha = 9.98$, $\beta = 2.69$, $\gamma = 2.79$ and $\delta = 0.36$.

The value of α expresses the time required for computation when t-norm/t-conorm must be computed on two cells (including the time for verifying that such functions have to be actually evaluated); the values of β and γ express the time required when t-norms need not to be computed (the values are slightly asymmetrical because the procedure is asymmetrical as the first matrix is accessed from its transposed); finally, the value of δ expresses the overhead of the procedure that does not depend on the size of the matrices.

3.3.3. Comparison

Given two matrices with density degrees d_1 and d_2 respectively, model (12) is used for estimating iSM execution time. iSM is selected if the returned value is smaller than iFM threshold, calculated by dividing the computed value 7.16 by the number of available processors. Otherwise, iFM is selected. When one of the input matrices are in full representation, it is temporarily converted into its sparse equivalent, in order to calculate its density degrees. This implies a small overhead due to the representation conversion; however the benefits deriving from the selection of the proper implementation overcome this extra-time.

The strongest assumption of this approach is its dependency on the machine used for time estimation. We computed these estimations with a state-of-the-art desktop computer (see next Section); however as technology evolves, new measures are needed. Anyhow, the thresholds can be customized to a specific machine by re-running the simulations and updating the constants in the code.

4. Evaluation

All algorithms have been coded in C++ (and compiled with gcc), by using the libraries required for integrating with Octave. Membership degrees are represented as floating point numbers. Fuzzy sets and relations are represented as Octave vectors and matrices respectively.

An empirical analysis has been carried out with the aim of quantifying the performances of the proposed implementations. The test machine was equipped with a four-cores Intel^{TM}Core$^{®}$i5-2500K Processor (4 GHz clock), RAM 16 GB, Microsoft^{TM}Windows$^{®}$7 x64 and GNU Octave 3.2.4.

4.1. iFM evaluation

The first experiment was aimed at evaluating the time required for computing the composition of two matrices with a number of variants. A standard implementation in the Octave programming language has been compared with the proposed iFM with one, two and four cores. In the case of four cores, both single and double precisions have been tested. Comparative results, obtained by averaging five independent runs, are reported in table 1 for 200×200 matrices and in table 2 for 1000×1000 matrices.

Table 1. Comparative results in composing 2 full 200×200 matrices.

Implementation	Time (s)
Octave language	646.484
Proposed, 1 core, single precision	0.073
Proposed, 2 cores, single precision	0.043
Proposed, 4 cores, single precision	0.031
Proposed, 4 cores, double precision	0.032

Table 2. Comparative results in composing 2 full 1000×1000 matrices.

Implementation	Time (s)
Octave language	N.A.
Proposed, 1 core, single precision	7.300
Proposed, 2 cores, single precision	3.513
Proposed, 4 cores, single precision	1.790
Proposed, 4 cores, double precision	2.754

The results clearly show an impressive performance gain when the proposed iFM is used. In the case of 1000×1000 matrices, the implementation in the native Octave language was unable to terminate the task within 23 hours of continuous execution. On the other hand, the proposed implementation is able to carry out the composition in reasonable time.

When more than one core is available (as in many systems, nowadays), performances can be further improved, as depicted in fig. 2. This is motivated by the decomposition of the computation into parallel threads, which almost divides the required time by a factor equal to the number of cores, save for a small overhead for thread preparation. Overhead is more visible in the case of 200×200 matrices, while it becomes negligible when 1000×1000 matrices are involved in the computation.

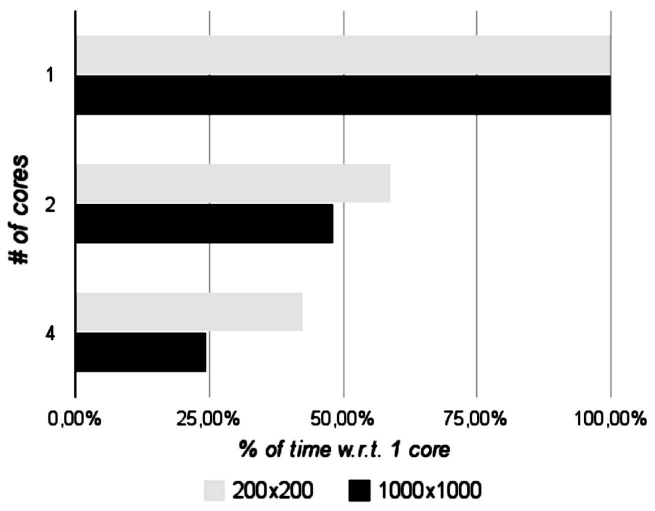

Fig. 2. Performance improvement with multi-core implementation.

When 200x200 matrices are used (table 1) the representation of membership degrees in single or double precision does not affect the required computation time, save for a negligible amount. On the other hand, when larger 1000 x 1000 matrices are applied (table 2), there is a significant increase of required time (about 1 sec.) that can be mainly motivated by the larger representation of each value (64 bits instead of 32). Thus, we preferred single precision to reach a compact representation of the matrices and allow fast processing of large matrices. On the other hand, we did not select a fixed-point representation (which would benefit of faster processing due to integer arithmetic), because most Octave functions process floating point data, and the required transformations from floating point to fixed point would destroy any performance improvement.

4.2. *iSM evaluation*

The second part of the experimentation was devoted at assessing the performances of the sparse matrix implementation, when sparse matrices of different density degrees are provided.

To this pursuit, five sparse matrices of size 1000×1000 have been randomly generated for each couple of density degrees (d_1, d_2) in $\{.01, .02, .05, .10, .20, \ldots, .80, .90, .95, .98, .99\}^2$. Then, the computation time has been recorded for each run, and averages depicted in fig. 3.

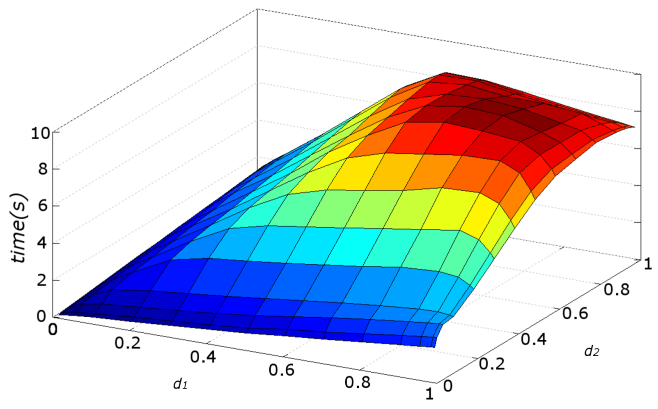

Fig. 3. Execution times of iSM, with different levels of density.

We observe a non-monotonic trend of iSM computation time that shows a peak at $(70\%, 70\%)$. This behavior can be partially related to the distribution time across the branches in the inner selection of the iSM procedure, which results in a non-linear distribution of computation time that can be approximated as the quadratic model (12).

4.3. *Implementation selection*

In order to assess the usefulness of the procedure for automatic selection of the implementation, we recorded the average time for the composition of random 1000×1000 matrices with different density degrees (like the previous experiments, all runs have been repeated for five times). We recorded the average time in three cases: i) by forcing iSM, ii) by forcing iFM, iii) by enabling the automatic selection procedure. All experiments have been repeated by varying the number of operating cores. Results are reported in table 3.

Table 3. Average composition time [secs] with forced and automatic selection of the implementation.

	num. of cores		
Implementation	1	2	4
iSM	4.23	4.23	4.23
iFM	7.30	3.51	1.79
automatic	4.12	2.63	1.60

It is possible to observe that the average time obtained when the automatic selection procedure is always smaller than the time required when a fixed implementation is used. In particular, when just one core is available, the average time is almost identical to iSM time (which is independent on the number of cores). On the other hand, as the number of cores increases, the benefits of iFM parallel computation become apparent, though not in all cases because very sparse matrices are more conveniently dealt with iSM. The automatic selection procedure is capable of well estimating which implementation to use, thus providing an average time that is smaller than iFM even when the number of cores is four.

Of course, this procedure is based on estimations determined by (12), which are not perfect. As an example, with four operating cores, the procedure wrongly selects the best implementation when one matrix is almost full and the other is almost empty. Nevertheless, we observed that the procedure correctly selected the best implementation in 99.5% of the cases for one operating core, 97.3% for two cores and 87.5% for four cores (mostly related to extreme cases). These results make automatic selection a reliable procedure for choosing the most convenient implementation in typical applications of the composition operators.

5. Conclusive remarks

The proposed implementation shows that fast fuzzy inference is possible in Octave, also when very large fuzzy relations are used. Efficiency has been obtained by either profiting of the very common multiprocessor architectures of modern computer systems, or by taking advantages of the possible sparsity of the involved fuzzy relations. Up to now, these two features are exploited alternatively: future developments will try to merge both capabilities for gaining even more efficiency.

Differently from other fuzzy inference systems, which are usually limited to fuzzy rules, the proposed implementation is capable of dealing with several forms of knowledge by using fuzzy relations as elements of knowledge representation. Fuzzy rules are special cases of fuzzy relations, hence fast fuzzy rule-based inference is also admitted. As a consequence, our implementation could be used as an efficient underpinning for more complex fuzzy inference systems, where linguistic variables, knowledge structures and learning schema are involved.

The proposed implementation is freely available as an Octave package under GNU LGPL (v3) licence at `http://octave.sourceforge.net/fl-core/index.html`.

References

1. J. Alonso and L. Magdalena. Generating Understandable and Accurate Fuzzy Rule-Based Systems in a Java Environment. In A. Fanelli, W. Pedrycz, and A. Petrosino, editors, *Fuzzy Logic and Applications (WILF 2011)*, Lecture Notes in Computer Science, pages 212–219. Springer-Verlag Berlin Heidelberg (ISSN: 0302-9743), Trani, Bari (Italy), 2011.
2. D. Dubois and H. Prade. What are fuzzy rules and how to use them. *Fuzzy Sets and Systems*, 84(2):169–185, 1996.
3. I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15:1–14, March 1989.
4. S. Guillaume and B. Charnomordic. Interpretable fuzzy inference systems for cooperation of expert knowledge and data in agricultural applications using FisPro. In *International Conference on Fuzzy Systems*, pages 1–8. IEEE, July 2010.
5. D. Nauck, U. Nauck, and R. Kruse. Generating classification rules with the neuro-fuzzy system NEFCLASS. In *Proceedings of North American Fuzzy Information Processing*, pages 466–470. IEEE, 1996.
6. F. Velo, L. Baturone, S. Solano, and A. Barriga. Rapid design of fuzzy systems with Xfuzzy. In *The 12th IEEE International Conference on Fuzzy Systems, 2003. FUZZ '03.*, pages 342–347. IEEE.

7. L. Zadeh. Fuzzy logic= computing with words. *Fuzzy Systems, IEEE Transactions on*, 4(2):103–111, 1996.
8. L. A. Zadeh. Toward a generalized theory of uncertainty (GTU)an outline. *Information Sciences*, 172(1-2):1–40, June 2005.