URC97055

Fast Fuzzy Arithmetic operations

Michael Hampton[1] and Olga Kosheleva[2]

Departments of [1]Computer Science and
[2]Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, TX 79968, USA
emails[1] mhampton@cs.utep.edu
[2] olga@ece.utep.edu

Abstract

In engineering applications of fuzzy logic, the main goal is not to simulate the way the experts really think, but to come up with a good engineering solution that would (ideally) be better than the expert's control, [n such applications, it makes per feet sense to restrict ourselves to simplified approxi m ate expressions for membership functions. If we need to perform arithmetic operations with the resulting fuzzy numbers, then we can use simple and fast algorithms that are known for operations with simple membership functions.

In other applications. especially the ones that are related to humanities, simulating experts is one of the main goals. In such applications, we must use membership functions that capture every nuance of the expert's opinion; these functions are therefore complicated, and fuzzy arithmetic operations with the corresponding fuzzy numbers become a computational problem.

In this paper, we design a new algorithm for performing such operations. This algorithm is applicable in the case when negative logarithms — $\log(\mu(x))$ of membership [unctions $\mu(x)$ are convex, and reduces computation time from $O(n^2)$ to $O(n\ log(n))$ (where n is the number of points x at which wc know the membership functions $\mu(x)$).

1 Formulation of the Problem

Depending on the goal, applications of fuzzy logic can be naturally divided into two classes:

- Engineering applications like *fuzzy control* in which fuzzy logic is used as a tool for achieving a certain goal: a better (smoother and safer) control of a car, a better heating, etc. In such applications, the expert's knowledge described by fuzzy rules is *used* not to *simulate* the way experts solve these problems, but to *design better control strategies*,
- Applications to humanities (psychology, linguistics, etc.) in which fuzzy logic is used to *describe* and simulate the human behavior, the human decision-making processes, etc., and thus *predict* the way humans will react in different situations.

In both types of applications, we have to deal with *fuzzy numbers* r, i.e., quantities whose values we do not know precisely, and instead, we only have expert (fuzzy) knowledge about these values. This knowledge is usually described in terms of *membership functions* $\mu_r(x)$ that assign to every real number x the expert's degree of belief $\mu_r(x) \in [(), 1]$ that the actual (unknown) value of the quantity r is equal to x.

The formalism (membership functions) is the same, but, depending on the application, wc treat these membership functions differently:

- In engineering applications, we do not need to describe the *exact* opinion of the experts, because we are going to *improve* this description (b-j some fine-tuning) anyway. Therefore, it is quite sufficient to use membership functions that *approximately* describe expert's opinions. To simplify computations, usually, the simplest approximations are used, most often triangular or trapezoid membership functions (see, e.g., [2]).

- In humanities applications, if we use oversimplified approximations to membership functions, we will end up having very crude models of human behavior For such applications, we, therefore, need accurate descriptions of membership functions. and these descriptions can be very complicated.

1.1 Fuzzy Data Processing and Fuzzy Arithmetic Operations: If We Must Use Precise Membership Functions, We Have a Computational Problem

Fuzzy data processing. We want to use the expert (fuzzy) knowledge about the values $r_1, \ldots, r_n$ of some quantities to *predict* the value of some quantity r that is related to r_i. In this paper, we will consider the simplest case when "related" means that we know the exact form of the dependency $r = f(r_1, \ldots, r_n)$ between r_i and r, and the only uncertainty in r is caused by the uncertainty in the values of r_i.

For example, when we formalize the expert's opinion about possible candidates for a position, we may know that this opinion depends on the values of n characteristics r_i of the candidate, we have expert (fuzzy) knowledge about the values of r_i, and we know that the final opinion depends on the total evaluation $r = w_1 \cdot r_1 + \ldots + w_n r_n$ with known weights w_i.

In such situations, we must transform the fuzzy knowledge about the values r_i into a fuzzy knowledge about $r = f(r_1, \ldots, r_n)$. This transformation is called *fuzzy data processing.*

Fuzzy arithmetic operations. In the computers, usually, only elementary arithmetic operations $(+, -, \cdot, /)$ are hardware supported. Therefore, every data processing algorithm written in a high-level programming language is *parsed*, i.e., represented as a sequence of elementary arithmetic operations. For example, computing an expression $x_1(x_2 + x_3)$ is decomposed into two steps: Computing $x_2 + x_3$ and multiplying the result by x_1.

In view of this decomposition, in order to implement. an arbitrary data processing algorithm with fuzzy inputs, it is sufficient to be able to apply *elementary arithmetic operations* $o = +, -, \cdot,$ to fuzzy numbers. The formulas for these operations come from the *extension principle (see, e.g.,* [3]): In particular, if we use an algebraic product $a \cdot b$ as a fuzzy analogue of &, we arrive at the following formula for $t = r \circ s$:

$$\mu_t(x) = \sup_{y,z\ y \circ z = x} (\mu_r(y)\ \mu_s(z)). \tag{1}$$

In particular, for $o = +$, we have

$$\mu_t(x) = \sup_y (\mu_r(y) \cdot \mu_s(x - y)). \tag{2}$$

For simple membership functions, fuzzy arithmetic operations are computationally easy. For example, if we use Gaussian membership functions

$$\mu_r(x) = \exp((x - a_r)^2/(\sigma_r)^2),$$

$$\mu_s(x) = \exp((x - a_s)^2/(\sigma_s)^2),$$

then *(2) leads to* a Gaussian *membership function for* t: $\mu_t(x) = \exp((x - a_t)^2/(\sigma_t)^2)$ *with*

$$a_t = \frac{a_r(\sigma_r)^{-2} + a_s(\sigma_s)^{-2}}{(\sigma_r)^{-2} + (\sigma_s)^{-2}}$$

and $(\sigma_t)^{-2} = (\sigma_r)^{-2} + (\sigma_s)^{-2}$ [3, 5]. These are computationally very simple formulas to implement.

There are simple formulas for several other cases (see, e.g., [3] and references therein).

For complicated membership functions, fuzzy arithmetic operations are computationally complicated. When we cannot use approximating simple expressions, then we cannot use simplified formulas that stem from the use of these expressions, and therefore, we have to use the formula (2). This formula is straightforward, so, we can simply use it to compute $\mu_t(x)$. To find out how long it would take to compute $\mu_t(x)$, let us estimate the number of computational steps that are required to compute $\mu_t(x)$.

Of course, in reality, we can only know the values of $\mu_r(x)$ and $\mu_s(x)$ for finitely many values x. Let us denote the total number of such values by n. In this case, it is reasonable to compute only n values of $\mu_t(x)$. For each of these n values, according to the formula (2), we must find the largest of n products. Computing each product takes 1 elementary computational step, computing the largest of n numbers requires that we do n -- 1 comparisons. So, the total number of computation steps that needs to be done to compute one value of $\mu_t(x)$ is $2n - 1 = O(n)$.

If we have n parallel processors at our disposal. then we can use each processor Lo compute its **own** value of $\mu_t(x)$ and thus, compute *all* these values in linear time.

In many real-life situations, however, we only have one computer. In such situations, to compute *all n* **values** of the desired membership function $\mu_t(x)$, we need $O(n^2)$ computational steps.

The more accurately we wish to represent the expert's opinion, the larger n we need to take. For large $n, O(n^2)$ is too long. Can *we perform fuzzy arithmetic operations faster?*

In [4], an *approximate* algorithm is given that performs arithmetic operations with fuzzy numbers in time $O(n\log(n))$.

1.2 What We Are Planning to Do

In this paper, we design a new fast algorithm that computes the precise value of the resulting membership functions in $O(n\log(n))$ time.

This algorithm is applicable when the negative logarithms $-\log(\mu(x))$ of the membership functions $\mu(x)$ are convex. This class of membership functions includes many important classes such as Gaussian membership functions.

2 Fast Addition of Fuzzy Numbers

2.1 Main Idea

Let us describe, step-by-step, how we can simplify the problem of computing the sum of two fuzzy numbers.

First simplification: reformulation in discrete terms. We only know the membership functions p,(x) and $\mu_s(x)$ in finitely many points. and usually, these points are of the type $x_i = i$ Ax. **In** this case, the formula (2) takes the following form:

$$t_i = \max_j(r_j\, s_{i-j}), \tag{3}$$

where we denoted $t_i = \mu_t(i\ \Delta x), r_i = \mu_r(i.\ \Delta x)$, *and* $s_i = \mu_s(i\Delta x)$.

Further simplification: reducing multiplication to addition. The formula (3) can be simplified even further if we recall that the equality $t = \mathrm{r}\cdot\mathrm{s}$ is equivalent to 'T = R + S, where $T = -\ln(t)$, R = $-\ln(r)$, and S = – in(s). In view of this equivalence, and taking into consideration the fact that – in(z) is a strictly decreasing function, we can reformulate the formula (3) as follows:

$$T_i = \min_j(R_j + S_{i-j}), \tag{4}$$

where we denoted $T_i = -\ln(t_i), R_i = -\ln(r_i)$, and $S_i = -\ln(s_i)$. We will describe how, given the two sequences R_i and S_i, we will be able to compute the elements T_i fast. Then, if we know the values $r_i = \mu_r(i\cdot\Delta x)$ and $s_i = \mu_s(i\,\Delta x)$, we will be able to compute the values R_i and S_i, compute $T_i = -\ln(t_i)$, and then reconstruct the desired values $t_i = \mu_t(i\,\Delta x)$ **as** $t_i = \exp(-T_i)$.

How to compute the formula (4)?

Final simplification: a local criterion for the maximum. For a given i, when does the sum $\sum_j = R_j + S_{i-j}$ attains its minimum'? If it does attain the minimum for some j, this means that the value of this sum for this particular j is not larger than the values of this sum for $j-1$ and for $j+1$: $\sum_j \le \sum_{j-1}$ and $\sum_j \le \sum_{j+1}$. If we denote $D_j = \sum_j - \sum_{j-1}$, then these two inequalities take the form

$$D_j \le 0; \quad D_{j+1} \ge 0. \tag{5}$$

We can use binary search to find the desired j. Since the function $-\ln(\mu_r(x))$ is convex, the sequence R_j is also convex, and therefore, the differences $R_j - R_{j-1}$ are monotonically increasing with j. Similarly, the differences $S_{i-(j-1)} - S_{i-j}$ are strictly *decreasing* with j. Therefore. the difference $D_j = (R_j - R_{j-1}) - (S_{i-j} - S_{i-(j-1)})$ is increasing with j.

Hence, we can find the desired value j that satisfies the condition (5) by **using** *binary search:* This will be an iterative process on which, on each step, wc will have lower and upper bounds for the desired value j. We start with the lower and upper bounds that encompass all possible values of j. Then, on each iteration. we:

- take a midpoint $m = (\texttt{lower} + \texttt{upper})\texttt{div}\,2$ between the current lower and the upper bounds;
- compute D_m for this midpoint m, and
- compare the resulting value D_m with 0.

Depending on the result of this comparison, we do the following:

- If $D_m = 0$, then, due to the monotonicity of the sequence D_m, we have $D_{m+1} \geq D_m = 0$, i.e., $D_{m+1} \geq 0$. Hence, this m satisfies the condition (5). Using monotonicity of D_j, one can easily show that in this case,
  - either m is the only value for which (5) in true (in which case, it is the only possible minimum of $\sum_m$),
  - or $D_j = 0$ not only for $j = m$, but also for several values of j that are neighboring to m, in which case, there arc several minima with exactly the same value of $\sum_j$.

  In both cases, the value of $\sum_m$ for the midpoint m *is* the desired minimum.
- If $D_m > 0$, this means, due to monotonicity of the sequence D_j, that $j < m$. In this case, we can take m *as* the new value of the variable upper.
- Similarly, if $D_m < 0$, this means, due to monotonicity of the sequence D_j, that $m < j$. In this case, *we* can take m as the new value of the variable lower.

This algorithm takes $O(n \log(n))$ steps. On each iteration of the binary search, we reduce the size in half. In k iteration, we go down from n to $\leq n/2^k$ possible values. When $n/2^k \leq 1$, we are down to a single point, and thus, wc have localized the desired j. The inequality $n/2^k \leq 1$ is achieved when $k \approx \log_2(n)$, so, we need $O(\log(n))$ points to find the desired j and thus, to compute the desired value of T_i for this particular i.

To compute the values of T_i for 71 different i's, we thus need $n \cdot O(\log(n)) = O(n \log(n))$ computational steps.

2.2 Resulting Algorithm

GIVEN: the values p,(r) and $\mu_s(x)$ for n equally spaced values $x_i = \text{r'} \cdot \Delta x$.

A*LGORITHM:*

- First, for each of n values x_i, we compute the values $R_i = -\ln(\mu_r(x_i))$ and $S_i = -\ln(\mu_s(x_i))$.
- For each i, we:
  - apply binary search to find the index j for which the non-decreasing sequence $D_j = (R_j - R_{j-1}) - (S_{i-j} - S_{i-(j-1)})$ passes from the non-positive to non-negative values; compute T_i as $R_j + S_{i-j}$ for this very j;
  - compute $\mu_t(x_i)$ as $\exp(-T_i)$.

3 Algorithms for Other Arithmetic Operations

3.1 Subtraction

To compute $t = r$ –s, we can represent it as $t = r + (-s)$. Since we know the membership function $\mu_s(x)$ for s, we can easily compute the membership function $\mu_{-s}(x)$ for –s as $\mu_{-s}(x) = \mu_s(-x)$. Then, wc can apply the above algorithm to compute the desired membership function for $t = r - s = r + (-s)$.

3.2 Multiplication

If the quantities r and s both take only positive values, then, to compute r s, we can use the formula $r \cdot s = \exp(\ln(r) + \ln(s))$:

- From the membership functions for r and s, we can easily compute the membership functions for $\ln(r)$ and $\ln(s)$ as $\mu_{\ln(r)}(x) = \mu_r(\ln(x))$ arid $\mu_{\ln(s)}(x) = \mu_s(\ln(x))$.
- Applying the algorithm presented above, we compute the membership function $\mu_{\ln(t)}$ for $\ln(t)$ = In(r)+ $\ln(s)$.
- Finally, from $\mu_{\ln(t)}$, we compute $\mu_t(y)$ as $\mu_t(y) = \mu_{\ln(t)}(\exp(y))$.

3.3 Division

Division $t = r/s$ *can he* expressed as $t = r(1/s)$. So, to divide two fuzzy numbers, we can use the following algorithm:

- First, we compute the membership function for $1/s$ as $\mu_{1/s}(x) = \mu_s(1/x)$.
- Then, wc use the algorithm for multiplication to compute the membership function for

$$t = \mathrm{r} \cdot (1/s) = r/s.$$

3.4 Computational Complexity

For all **these** operations, the major part is computing the sum of fuzzy numbers that takes $O(n$ log(n)) steps. Therefore, the computational complexity of computing the difference, product, or ratio of two fuzzy numbers is also $O(n \log(n))$.

4 What If A t-Norm ($\&$–Operation) Is Different From Algebraic Product?

4.1 Fuzzy Arithmetic Operations: Case of a General t-Norm

For an arbitrary &—operation $f_{\&}(a,$ b), the extension principle for addition leads to the following formula:

$$\mu_t(x) = \sup_Y f_{\&}(\mu_r(y), \mu_s(x - y)). \tag{3}$$

4.2 Strictly Archimedean t-Norms and Reduction to the Case of Algebraic Product

Idea. [t is known (see, e.g., [3]), that if an &–operation satisfies some reasonable conditions, then it **can** represented i n the form

$$f_{\&}(a, b) = \psi^{-1}(\psi(a) \cdot \psi(b)) \tag{4?}$$

for some strictly increasing function $\psi : [0,1] \to$ **[0,** $1]$ ($\&$–operations **that satisfy these "reasonable"** conditions are called *strictly Archimedean*).

Since the function ψ is strictly increasing, the value $f_{\&}(\mu_r(y), \mu_s(x-y))$ is the largest iff the value $\psi(f_{\&}(\mu_r(y), \mu_s(x-y)))$ is the largest, so.

$$\psi(\mu_t(x)) = \sup_Y \psi(f_{\&}(\mu_r(y), \mu_s(x-y)). \tag{5}$$

From (4), we conclude that $\psi(f_{\&}(\mu_r(y), \mu_s(x-y))) = \psi(\mu_r(y)) \cdot \psi(\mu_s(x-$ y)). Therefore, (7) can be rewritten as:

$$\psi(\mu_t(x)) = \sup_y \psi(\mu_r(y))\, \psi(\mu_s(x - \mathrm{y})). \tag{6}$$

If we denote $\nu_r(x) = \psi(\mu_r(x)), \nu_s(x) = \psi(\mu_r(x))$, and $\nu_t(s) = \psi(\mu_t(x))$, then this formula will take the form

$$\nu_t(x) = \sup_Y(\nu_r(y)\, \nu_s(x - \mathrm{Y}))! \tag{7}$$

which is exactly like the formula (2) that we already know how to compute fast. From $\nu_t(x) = \psi(\mu_t(x))$, we can compute $\mu_t(x)$ by applying an inverse function ψ^{-1}: $\mu_t(x) = \psi^{-1}(\nu_t(x))$.

So, to compute $\mu_t(x)$, we can apply the following algorithm:

Algorithm.

- For every x, compute $\nu_r(x) = \psi(\mu_r(x))$ and $\nu_s(x) = \psi(\mu_s(x))$. This takes $O(n)$ steps.
- Apply the algorithm (described in the previous section) to $\nu_r(x)$ and ν_s (x); this algorithm will take $O(n$ log(n)) computational steps and return $nu_t(x)$.
- Apply the inverse function ψ^{-1} to $\nu_t(x)$, resulting in $\mu_t(x) = \psi^{-1}(\nu_t(x))$. This is done value-by-value. so, for $O(n)$ values of x, it takes $O(n)$ steps.

Computational Complexity. The resulting algorithm requires

$$o(n) + O(n\log(n)) + O(n) = O(n\log(n))$$

computational steps.

4.3 Other Arithmetic Operations

For other arithmetic operations with fuzzy numbers (–, . . /), we have a similar reduction to the case of algebraic product that leads Lo similar $O(n$ log(n)) algorithms.

Acknowledgments. This work was supported by the Office of Naval Research Grant No. *N00014-93-1-1343* and, partially, by the National Science Foundation Grant No. CDA 9522903, and by the NASA Pan American Center for Environmental and Earth Studies (PACES). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agencies.

The authors are thankful to Ann Gates, Vladik Kreinovich, Luc Longpré, and Scott Starks for their encouragement.

References

[1] Th. H. Cormen, Ch. L. Leiserson, R,. L. Rivest, *Introduction to algorithms*, MIT Press. Cambridge, MA, *1990.*

[2] K. Hirota and M. Sugeno. *industrial Applications of Fuzzy Technology in the World.* World Scientific, Singapore, 1996.

[3] G. Klir and B. Yuan, *Fuzzy sets and fuzzy logic: theory* and *applications.* Prentice Hall, Upper Saddle River, NJ, 1995.

[4] O. Kosheleva, S. D. Cabrera, G. A. Gibson. and M. Koshelev, "Fast Implementations of Fuzzy Arithmetic Operations Using Fast Fourier Transform (F FT)", *Proceedings of the 1996 IEEE International Conference on Fuzzy Systems, New* Orleans. September 8-11, 1996, Vol. 3, pp. 1958-1964.

[5] V. Kreinovich, C. Quintana, and I.. Reznik. Gaussian membership functions arc most adequate in representing uncertainty in measurements. *Proceedings of /VA FIPS'92: North A merican Fuzzy Information* Processing Society *Conference, Puerto Vallarta,* Mexico, *December 15-17, 1992,* NASA Johnson Space Center, Houston, TX, 1992, pp. 618-625.