

Fast Implementations of Fuzzy Arithmetic Operations Using Fast Fourier Transform (FFT)

Olga Kosheleva, Sergio D. Cabrera, Glenn A. Gibson
Department of Electrical and Computer Engineering
The University of Texas at El Paso
El Paso, TX 79968, USA
{olga,cabrera}@ece.utep.edu

Misha Koshelev
1003 Robinson
El Paso, TX 799602, USA
mkosh@cs.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. In such applications, it makes perfect sense to restrict ourselves to simplified approximate 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 uses Fast Fourier Transform (FFT) to reduce computation time from $O(n^2)$ to $O(n \log(n))$ (where n is the number of points x at which we know the membership functions $\mu(x)$). To compute FFT even faster, we propose to use special hardware.

The results of this paper were announced in [8].

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 $\mu_r(x) \in [0, 1]$ 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, we 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 (by 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., [5]).
- 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

1.1.1 Fuzzy Data Processing

We want to use the expert (fuzzy) knowledge about the values r_1, \dots, 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 $r = f(r_1, \dots, r_n)$ 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 $r = w_1 \cdot r_1 + \dots + w_n \cdot r_n$ with known weights w_i .

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

1.1.2 Fuzzy Arithmetic Operations

In the computers, usually, only elementary arithmetic operations (+, −, ·, /) 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 $x_1 \cdot (x_2 + x_3)$ 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* $\circ = +, -, \cdot$, to fuzzy numbers. The formulas for these operations come from the *extension principle* (see, e.g., [6]): In particular, if we use an algebraic product $a \cdot b$ as a fuzzy analogue of $\&$, we arrive at the following formula for $t = r \circ s$:

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

In particular, for $\circ = +$, we have

$$\mu_t(x) = \sup_y (\mu_r(y) \cdot \mu_s(x - y)). \quad (2)$$

1.1.3 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 $(\sigma_t)^{-2} = (\sigma_r)^{-2} + (\sigma_s)^{-2}$ [6, 9]. These are computationally very simple formulas to implement.

There are simple formulas for several other cases (see, e.g., [6] and references therein).

1.1.4 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 to 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?*

1.2 What We Are Planning to Do

In this paper, we design a new, faster, algorithm for performing fuzzy arithmetic operations.

2 Fast Arithmetic Operations with Fuzzy Numbers

2.1 Addition: the New Algorithm and Theoretical Estimates of Its Running Time

2.1.1 Idea

Let us first describe how to compute the expression (2). In our computation, we will use the well-known fact that for non-negative numbers μ_1, \dots, μ_n , we have

$$\max(\mu_1, \dots, \mu_n) = \lim_{p \rightarrow \infty} (|\mu_1|^p + \dots + |\mu_n|^p)^{1/p}$$

(see, e.g., [7]). Therefore, for sufficiently large p , we have

$$\max(\mu_1, \dots, \mu_n) \approx (|\mu_1|^p + \dots + |\mu_n|^p)^{1/p};$$

the larger p , the better the quality of this approximation.

Applying this approximate formula to the values $\mu_r(y) \cdot \mu_s(x - y)$ maximized in the formula (2), we come up with an approximate formula $\mu_t(x) \approx T(x)^{1/p}$, where we denoted

$$T(x) = \sum_y (\mu_r(y) \cdot \mu_s(x - y))^p.$$

The formula for $T(x)$ can be rewritten as:

$$T(x) = \sum_y (\mu_r(y))^p \cdot \mu_s(x - y)^p.$$

In the natural assumption that the values y are equally spaced, with step h , this sum becomes a *convolution* of two functions: $R(x) = (\mu_r(x))^p$ and $S(x) = (\mu_s(x))^p$. Now, we can use the following two ideas to compute $T(x)$ fast:

- It is known that the Fourier transform of the convolution $R * S$ of two functions R and S is equal to the product of their Fourier transforms.
- Fourier transform can be computed in time $O(n \log(n))$ [12, 16]; the corresponding algorithms are called *Fast Fourier Transform* (FFT, for short).

In view of these two facts, we can use the following algorithm to compute the membership function that expresses the sum of two given fuzzy numbers:

2.1.2 Resulting Algorithm

Given: the values $\mu_r(x)$ and $\mu_s(x)$ for n equally spaced values x .

Algorithm: First, we pick a large number p (the larger p , the better the results of our computations). Then, we do the following:

1. For each of n values x , we compute the values $R(x) = (\mu_r(x))^p$ and $S(x) = (\mu_s(x))^p$.
2. We apply FFT to the functions $R(x)$ and $S(x)$ and get their Fourier transforms $\hat{R}(\omega)$ and $\hat{S}(\omega)$ (for n different values ω).
3. We multiply $\hat{R}(\omega)$ and $\hat{S}(\omega)$; let us denote the corresponding product by $\hat{T}(\omega)$.
4. We apply inverse Fast Fourier transform to the product $\hat{T}(\omega)$ (computed on the previous step). As a result, we get a function $T(x)$.
5. Finally, we reconstruct $\mu_t(x)$ as $(T(x))^{1/p}$.

2.1.3 This Algorithm is Asymptotically Faster

Let us estimate the number of computational steps of this algorithm. Steps 1, 3, and 5 require linear time ($O(n)$ steps each, so, $O(n)$ total). Steps 2 and 4 involve FFT and therefore, require the time $O(n \log(n))$. Therefore, the total number of computational steps is equal to $O(n) + O(n \log(n)) = O(n \log(n))$, which is $\ll O(n^2)$.

2.2 New Algorithm for Adding Fuzzy Numbers: Is The Theoretical Improvement Indeed Achieved in Practice?

2.2.1 For Small Number of Values n , the Traditional Algorithm is Better

In the previous section, we have shown that for our algorithm, the total number of computational steps is asymptotically smaller than for the traditional method. This fact, however, does not necessarily mean that our algorithm is faster for all n ; indeed:

- computations that directly implement the extension principle utilize simple operations (maximum, difference, and product);
- computations from our algorithm use more complicated (thus, slower) operations, such as exponentiation and FFT.

To compare the two algorithms, we will compute the actual number of steps for different n .

2.2.2 Choosing n

Let us first choose *the values of n* for which we will compare the two algorithms. The Fast Fourier Transform (FFT) algorithm runs best for $n = 2^k$; actually, it was originally invented for $n = 2^k$, and FFT for $n \neq 2^k$ works as follows: we extend the sequence of n values to 2^k by adding zeros, and apply the original FFT algorithm. Because of this, for every n from 2^{k-1} to 2^k , FFT takes almost exactly the same time as FFT for $n = 2^k$. Since FFT is the major part of our algorithm, we will, therefore, consider only $n = 2^k$, i.e., $n = 1, 2, 4, \dots, 1024$.

2.2.3 Computational Complexity of the Traditional Algorithm

In Section 1.1.4, we have already estimated the number of steps for the algorithm that directly implements the extension principle: if we store n values of each membership function, then, to compute each of n values of $\mu_t(x)$, we need n multiplications ($\mu_r(y) \cdot \mu_s(x - y)$), and $n - 1$ comparisons to compute the maximum of these n values: totally, $2n - 1$ operations per resulting value. Totally, we need $n \cdot (2n - 1) = 2n^2 - n$ operations.

Comment. The reason why we are analyzing the existing algorithm is that we want to show that our new algorithm is faster. Thus, what we are really interested in is *computation time*. The reason why we are estimating the number of computational steps is that computation time depends on the times of different basic computer operations and is, therefore, highly dependent on the choice of a computer. To avoid this dependency, we can *estimate* the computation time by multiplying the total number of computation steps and the average time of a computer operation.

In most computers, multiplication of two real numbers takes much longer time than the comparison of these two numbers, and the time for multiplication is closer to the average operation time. Thus, if we simply add the total numbers of multiplications and comparisons, and then multiply the result by the average operation time, we may get an *overestimate* of the actual computation time. Therefore, if we show that the number of computation steps of the new algorithm is smaller than this overestimate we will not be able to conclude that the new algorithm is necessarily better. To make the comparison convincing in the absence of the actual precise estimate, we must compare the number of computation steps for the new algorithm with an *underestimate* of the number of computation steps required for the traditional method. To get such an overestimate, we will count only the multiplications: there are n^2 of them. Since multiplication usually takes longer than the average computer operation, the product of the total number of multiplications and of the average operation time is an underestimate of the actual computation time.

Our computer experiments (explained below) show that this underestimate is close to the actual computation time.

2.2.4 Computational Complexity of the New Algorithm

For the new algorithm, we need the following times:

- On the first part of the new algorithm, we raise $2n$ values (n values of μ_r and n values of μ_s) to p -th power. Raising a value z to p -th power is easiest for $p = 2^l$; for such values, it takes l multiplications: first, we compute $z^2 = z \cdot z$, then $z^4 = z^2 \cdot z^2$, etc., until we get $z^p = z^{2^l} = z^{2^{l-1}} \cdot z^{2^{l-1}}$. Therefore, this first part requires $l \cdot n$ computational steps.
- The fastest known FFT algorithm (iterative FFT [1]) consists of $\log_2(n)$ stages. Each of these stages, in its turn, consists of $n/2$ “butterflies”. A butterfly is an operation that takes two complex numbers a_k and b_k and returns two new complex numbers $a_k + \omega_n^k b_k$ and $a_k - \omega_n^k b_k$. The coefficient $(\omega_n)^k$ (called a *twiddle factor*) depends only on n and k (and does not depend on the data to which we apply FFT); therefore, for fixed n , we can *pre-compute* the values ω_n^k for all k , store them in an array, and get them from the array every time we need them. If we do that, then the number of computational steps required for each butterfly consists of one multiplication, one addition, and one subtraction of complex numbers. Every operation with complex numbers consists, in its turn, in several operations with real numbers:
 - The product $z = z_1 \cdot z_2$ of two complex numbers $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ is equal to $z = x + iy$, where $x = x_1x_2 - y_1y_2$ and $y = x_1y_2 + x_2y_1$; this computation requires 4 multiplications and 2 additions, totally, 6 arithmetic operations.
 - The sum $z = z_1 + z_2$ of two complex numbers $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ is equal to $z = x + iy$, where $x = x_1 + x_2$ and $y = y_1 + y_2$; this computation requires 2 additions.
 - Similarly, the difference $z = z_1 - z_2$ between two complex numbers requires 2 subtractions.

Totally, a butterfly requires 10 elementary arithmetic operations with real numbers. Since FFT consists of $\log_2(n) \cdot (n/2)$ butterflies, the FFT algorithm requires $\log_2(n) \cdot (n/2) \cdot 10 = 5n \log_2(n)$ operations.

If the number n of stored values varies, then we cannot store ω_n^k , we have to compute it every time.

- The time necessary to compute an elementary operation (like sin or cos) is usually estimated as 10 elementary operations with real numbers, so, computing ω_n requires 20 steps.

- Computing each of the consequent powers of ω_n ($\omega_n^2, \omega_n^3, \dots, \omega_n^{n-1}$) requires one complex multiplication, so, we need $n - 2$ complex multiplications. Each of these complex multiplications consists of 6 elementary operations with real numbers, so, totally, we need $6(n - 1)$ elementary operations.

Totally, if we do not pre-compute twiddle factors, FFT needs $20 + 6(n - 1) = 6n + 14$ additional steps.

- Computing $\hat{T}(\omega)$ takes n complex multiplications, i.e., $6n$ elementary steps.
- Inverse FFT takes exactly the same time as FFT.
- Computing p -th root $z^{1/p}$ is usually done by applying a “to the power of” subroutine, which computes a^b as $a^b = (e^{\log(a)})^b = \exp(b \cdot \log(a))$; so, $z^{1/p}$ is computed as $\exp((1/p) \log(z))$. The value $1/p$ can be pre-computed; so, to get each of n values of $\mu_t(x)$, we need one log, one multiplication, and one exp. Counting each function as 10 arithmetic operations, we get 21 computational steps for each of n values; totally, $21n$ steps.

Overall, the new algorithm requires $l \cdot n + 5n \log_2(n) + 21n = n(l + 5 \log_2(n) + 21)$ computational steps ($6n + 14$ more steps if we do not pre-compute the twiddle factors).

2.2.5 Results and Conclusions

For pre-computed twiddle factors, the resulting numbers of computational steps for the traditional and the new algorithms are given in the following table:

n	Traditional method	New method $p = 4$	New method $p = 8$
2	4	56	58
4	16	132	136
8	64	304	312
16	256	688	704
32	1,024	1,536	1,568
64	4,096	3,392	3,456
128	$1.7 \cdot 10^4$	$0.74 \cdot 10^4$	$0.75 \cdot 10^4$
256	$6.3 \cdot 10^4$	$1.6 \cdot 10^4$	$1.6 \cdot 10^4$
512	$2.5 \cdot 10^5$	$0.35 \cdot 10^5$	$0.35 \cdot 10^5$
1,024	$1.0 \cdot 10^6$	$0.075 \cdot 10^6$	$0.076 \cdot 10^6$
2,048	$4.2 \cdot 10^6$	$0.16 \cdot 10^6$	$0.16 \cdot 10^6$

The conclusion is:

The new algorithm is indeed better for reasonable n : the new algorithm is faster starting with $n = 64$, and for $n = 1,024$, it is more than ten times faster.

If we have to pre-compute the twiddle factors, then the results for the new algorithm will be slightly worse:

n	Traditional method	New method $p = 4$	New method $p = 8$
2	4	82	84
4	16	170	174
8	64	366	374
16	256	798	814
32	1,024	1,742	1,774
64	4,096	3,790	3,854
128	$1.7 \cdot 10^4$	$0.82 \cdot 10^4$	$0.83 \cdot 10^4$
256	$6.3 \cdot 10^4$	$1.8 \cdot 10^4$	$1.6 \cdot 10^4$
512	$2.5 \cdot 10^5$	$0.38 \cdot 10^5$	$0.38 \cdot 10^5$
1,024	$1.0 \cdot 10^6$	$0.081 \cdot 10^6$	$0.082 \cdot 10^6$
2,048	$4.2 \cdot 10^6$	$0.17 \cdot 10^6$	$0.17 \cdot 10^6$

The conclusion is:

Even with the computation of twiddle factors, the new algorithm is still faster starting with $n = 64$, and for $n = 1,024$, it is still more than ten times faster.

Comment. The traditional algorithm computes the desired membership function *precisely*, while the new algorithm gives only an *approximation*. This is OK, because the input data (values of the membership functions) are expert estimates and are, therefore, inherently imprecise. With imprecise inputs, it makes no sense to compute the result with a larger accuracy than the inputs themselves. Our numerical experiments has shown that with this inaccuracy of input data in mind, the algorithms for $p = 4$ and $p = 8$ are pretty accurate.

In case for some inputs, we would want to get more accurate estimates, we can use larger values of p . By choosing $p = 2^l$, we can still get reasonably fast algorithms (faster than the traditional method). For still larger p , we can compute a^p in a way similar to our method of computing $a^{1/p}$, i.e., as $\exp(p \cdot \log(a))$. This computation takes slightly longer than computing a^4 or a^8 by multiplications, but the resulting number of computational steps does not depend on p and is still smaller than for the traditional method. Thus, although we cannot compute the resulting membership function *precisely*, we can compute it as accurately as necessary without increasing the number of computation steps.

2.2.6 Theoretical Conclusions Experimentally Confirmed

To check these estimates, we implemented both traditional and new algorithms in C++, and compared their running time on the PC (486-based Epson laptop ActionNote 890C). It is difficult to measure the running time exactly on a

Windows-based PC, so we could only get crude time estimates. The measured running time turned out to be pretty much independent on the actual choice of the membership functions, so, to compute the running times, we used the simple linear functions $\mu_r = \mu_s$, for which the consequent n values of $\mu_r(x)$ are $0, 1/n, 2/n, \dots, (n-1)/n$. To compute FFT, we used a C++ program from [2]; this FFT program does not pre-compute twiddle factors. The resulting running times were in line with the second table (corresponding to the case of no pre-computing):

n	Traditional method (msec)	New method (msec)
16	1.2	4.4
32	3.2	8.8
64	14	16
128	56	36
256	230	75
512	900	160
1024	3,700	340
2048	15,000	700

2.3 Other Arithmetic Operations

The ability to compute the sum of two fuzzy numbers fast leads to the fast algorithms for other arithmetic operations:

2.3.1 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, we can apply the above algorithm to compute the desired membership function for $t = r - s = r + (-s)$.

2.3.2 Multiplication

If the quantities r and s both take only positive values, then, to compute $r \cdot s$, 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 $\mu_{\ln(r)}(x) = \mu_r(\ln(x))$ and $\mu_{\ln(s)}(x) = \mu_s(\ln(x))$.
- Applying the algorithm presented above, we compute the membership function $\mu_{\ln(t)}$ for $\ln(t) = \ln(r) + \ln(s)$.
- Finally, from $\mu_{\ln(t)}$, we compute $\mu_t(y)$ as $\mu_t(y) = \mu_{\ln(t)}(\exp(y))$.

If the quantities r and s can take both positive and negative values, then, from the extension principle, we can conclude that for $x > 0$, the value of $\mu_t(x)$ is equal to the maximum of two numbers:

$$\mu_t(x) = \max(\mu_t^{++}(x), \mu_t^{--}(x)), \quad (3)$$

where

$$\mu_t^{++}(x) = \sup_{y>0}(\mu_r(y) \cdot \mu_s(x/y)),$$

and

$$\mu_t^{--}(x) = \sup_{y<0}(\mu_r(y) \cdot \mu_s(x/y)).$$

To compute the values $\mu_t^{++}(x)$, we can use the above procedure for $r > 0$ and $s > 0$. To compute $\mu_t^{--}(x)$, we can use the formula $r \cdot s = (-r) \cdot (-s)$:

- First, we compute the membership functions for $-r$ and $-s$ (just like we did that for addition).
- Then, we compute the membership function for $(-r) \cdot (-s) = r \cdot s$ using the multiplication algorithm for two positive values.

After we compute $\mu_t^{++}(x)$ and $\mu_t^{--}(x)$, we compute $\mu_t(x)$ for $x > 0$ using the formula (3).

The values of $\mu_t(x)$ for $x < 0$ can be computed by using a similar formula

$$\mu_t(x) = \max(\mu_t^{+-}(x), \mu_t^{-+}(x)), \quad (4)$$

where

$$\mu_t^{+-}(x) = \sup_{y>0}(\mu_r(y) \cdot \mu_s(x/y)),$$

and

$$\mu_t^{-+}(x) = \sup_{y<0}(\mu_r(y) \cdot \mu_s(x/y)).$$

To compute $\mu_t^{+-}(x)$, we can use the formula

$$-t = r \cdot (-s),$$

and to compute $\mu_t^{-+}(x)$, the formula $-t = (-r) \cdot s$.

2.3.3 Division

Division $t = r/s$ can be expressed as $t = r \cdot (1/s)$. 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, we use the algorithm for multiplication to compute the membership function for

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

2.3.4 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 $O(n \log(n))$.

3 What If A t-Norm (&-Operation) Is Different From Algebraic Product?

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

In the above formulas, we used $\min(a, b)$ as a fuzzy “and” operation. In principle, other “and”-operations $f_{\&}(a, b)$ (called *t-norms*) can also be used:

Definition [6, 11]. A function $f_{\&} : [0, 1] \times [0, 1] \rightarrow [0, 1]$ is called a *t-norm* if it satisfies the following four conditions:

- $f_{\&}(1, a) = a$ for all a ;
- $f_{\&}(a, b) = f_{\&}(b, a)$ for all a and b ;
- $f_{\&}(a, f_{\&}(b, c)) = f_{\&}(f_{\&}(a, b), c)$ for all a, b , and c ;
- if $a \leq a'$ and $b \leq b'$, then $f_{\&}(a, b) \leq f_{\&}(a', b')$.

Comment. The usual notation for a t-norm is $T(a, b)$. We decided, however, to use a (less frequently used) notation $f_{\&}(a, b)$ to avoid confusion with the quantity $T(x)$ introduced in Section 2.

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

$$\mu_t(x) = \sup_y f_{\&}(\mu_r(y), \mu_s(x - y)). \quad (5)$$

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

3.2.1 Idea

It is known that if an &-operation satisfies some reasonable conditions, then it can be represented in the form

$$f_{\&}(a, b) = \psi^{-1}(\psi(a) \cdot \psi(b)) \quad (6)$$

for some continuous strictly increasing function $\psi : [0, 1] \rightarrow [0, 1]$:

Definition [6, 11].

- A t -norm $f_{\&}(a, b)$ is called *Archimedean* if it is continuous and $f_{\&}(a, a) < a$ for all $a \in (0, 1)$.
- An Archimedean t -norm is called *strictly Archimedean* if it is strictly increasing for $a, b \in (0, 1)$.

Proposition [13, 10, 6, 11].

- For every continuous strictly increasing function $\psi : [0, 1] \rightarrow [0, 1]$, the function $f_{\&}(a, b) = \psi^{-1}(\psi(a) \cdot \psi(b))$ is a strictly Archimedean t -norm.
- If $f_{\&}(a, b)$ is a strictly Archimedean t -norm, then there exists a continuous strictly increasing function $\psi : [0, 1] \rightarrow [0, 1]$ for which $f_{\&}(a, b) = \psi^{-1}(\psi(a) \cdot \psi(b))$.

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,

$$\psi(\mu_t(x)) = \sup_y \psi(f_{\&}(\mu_r(y), \mu_s(x - y))). \quad (7)$$

From (6), 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:

$$\psi(\mu_t(x)) = \sup_y \psi(\mu_r(y)) \cdot \psi(\mu_s(x - y)). \quad (8)$$

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

$$\nu_t(x) = \sup_y (\nu_r(y) \cdot \nu_s(x - y)), \quad (9)$$

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:

3.2.2 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 $\nu_s(x)$; 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.

3.2.3 Computational Complexity

The resulting algorithm requires

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

computational steps.

3.3 Other Arithmetic Operations

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

4 New Architecture for FFT Can Help

In [4, 3, 14, 15], we have proposed a new computer architecture that enables, among other things, to compute FFT really fast (faster than the previously proposed schemes).

Since the main computational part of the proposed algorithm is computing FFT, this architecture can definitely be of help.

In particular, the use of FFT speeds up our algorithm and thus, decreases the “threshold” value of n starting from which our algorithm is faster.

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. 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 V. Kreinovich, R. N. Lea, R. Mesiar, and to all the participants of the 1996 International IEEE Conference on Fuzzy Systems for the encouragement, and to the anonymous referees for valuable suggestions.

References

- [1] Th. H. Cormen, Ch. L. Leiserson, R. L. Rivest, *Introduction to algorithms* (MIT Press, Cambridge, MA, 1990).
- [2] R. Davies, *Newmat08a, a matrix library in C++*.
<http://nz.com/webnz/robert/index.html>.
- [3] G. A. Gibson, *Six-month Report - Investigations of Modularly Configurable Attached Processor with Intelligent Memories*, Technical report to ONR, Attachments A and B, Office of Naval Research, March 1995.
- [4] G. A. Gibson, V. P. Singh, S. J. Singh, Y. C. Liu, Y. C. Chang, and S. D. Cabrera, MCM Implementation of Modularly Configurable Attached Processors, *IEEE International Computer Symposium* (Taiwan, Dec. 1994) 465–472.

- [5] K. Hirota and M. Sugeno, *Industrial Applications of Fuzzy Technology in the World* (World Scientific, Singapore, 1996).
- [6] G. Klir and B. Yuan, *Fuzzy sets and fuzzy logic: theory and applications* (Prentice Hall, Upper Saddle River, NJ, 1995).
- [7] A. N. Kolmogorov and S. V. Fomin, *Introductory Real Analysis* (Dover, N.Y., 1975).
- [8] O. Kosheleva, S. D. Cabrera, G. A. Gibson, and M. Koshelev, Fast Implementations of Fuzzy Arithmetic Operations Using Fast Fourier Transform (FFT), *Proceedings of the 1996 IEEE International Conference on Fuzzy Systems* (New Orleans, September 8–11, 1996) Vol. 3, 1958–1964.
- [9] V. Kreinovich, C. Quintana, and L. Reznik, Gaussian membership functions are most adequate in representing uncertainty in measurements, *Proceedings of NAFIPS'92: North American Fuzzy Information Processing Society Conference, Puerto Vallarta, Mexico, December 15–17, 1992* (NASA Johnson Space Center, Houston, TX, 1992) 618–625.
- [10] C. H. Ling, Representation of associative functions, *Publ. Math. Debrecen* **12** (1965) 189–212.
- [11] H. T. Nguyen and E. A. Walker, *A first course in fuzzy logic* (CRC Press, Boca Raton, Florida, 1997).
- [12] A. V. Oppenheim and R. W. Schaffer, *Discrete-Time Signal Processing* (Prentice Hall, Englewood Cliffs, NJ, 1989).
- [13] B. Schweizer and A. Sklar, Associative functions and abstracts semigroups, *Publ. Math. Debrecen* **10** (1963) 69–81.
- [14] S. J. Singh, B. W. Gremel, V. P. Singh, and G. A. Gibson, Design Considerations for implementing a Modularly Configurable Attached Processor in a Multichip Module, *Multi-Chip Module Conference MCMC'95* (Santa Cruz, CA, January 1995) 62–68.
- [15] S. J. Singh, B. W. Gremel, V. P. Singh, and G. A. Gibson, Design Issues in a CMOS Implementation of Modularly Configurable Attached Processor, *Int'l J. of Electronics* **78**(5) (1995) 945–958.
- [16] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform* (SIAM, Philadelphia, 1992).