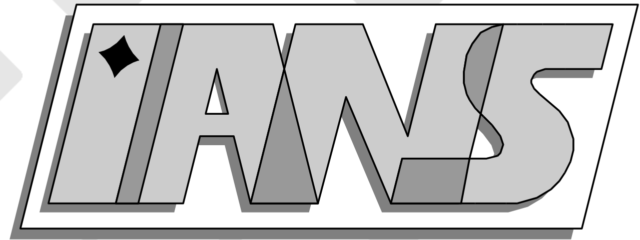
Universität Stuttgart



An Efficient Implementation of the Transformation Method of Fuzzy Arithmetic

Andreas Klimke

Berichte aus dem Institut für Angewandte Analysis und Numerische Simulation

Extended Preprint 2003/009

Universität Stuttgart

An Efficient Implementation of the Transformation Method of Fuzzy Arithmetic

Andreas Klimke

Berichte aus dem Institut für Angewandte Analysis und Numerische Simulation

Extended Preprint 2003/009

Institut für Angewandte Analysis und Numerische Simulation (IANS)
Fakultät Mathematik und Physik
Fachbereich Mathematik
Pfaffenwaldring 57
D-70 569 Stuttgart

E-Mail: ians-preprints@mathematik.uni-stuttgart.de
WWW: http://preprints.ians.uni-stuttgart.de

ISSN **1611-4176**

© Alle Rechte vorbehalten. Nachdruck nur mit Genehmigung des Autors.
IANS-Logo: Andreas Klimke. LaTeX-Style: Winfried Geis, Thomas Merkle.

An Efficient Implementation of the Transformation Method of Fuzzy Arithmetic

Andreas Klimke

Abstract

The transformation method has been proposed for the simulation and analysis of systems with uncertain parameters. Here, several aspects of an efficient implementation are presented: fast processing of discretized fuzzy numbers through multi-dimensional arrays, elimination of recurring permutations, automatic decomposition of models, treatment of single occurrences of variables through interval arithmetic, and a monotonicity test based on automatic differentiation. All algorithms have been implemented in MATLAB.

AMS subject classification: 03E71, 90C70, 26E50

Key words: constrained fuzzy arithmetic, interval arithmetic, functions of fuzzy numbers

---

1 Introduction

Fuzzy-parameterized models have become increasingly popular in engineering sciences as a tool to analyze systems with respect to uncertain model parameters. A fuzzy-parameterized model can help in determining the range of results, as well as in quantifying the influence of the uncertainty of each fuzzy parameter on the overall uncertainty of the model output if combined with a sensitivity analysis. Basis of the evaluation of such models is the theory of fuzzy arithmetic. The extension principle defined by Zadeh [25] provides a formal approach to extend real-valued arithmetic to arithmetic of fuzzy numbers. The implementation of this principle, however, is a rather non-trivial task of nonlinear programming if performed in an exact manner.

Standard fuzzy arithmetic. Classical fuzzy arithmetic methods, such as arithmetic based on LR-fuzzy numbers according to Dubois and Prade [8], fuzzy arithmetic based on interval arithmetic according to Kaufmann and Gupta [17], or similar variations (e.g. [9]) provide a way to compute fuzzy-valued expressions in manageable ways in terms of ease of applicability and computational complexity (linear). The latter methods, however, have one important common characteristic, which turns out to be a serious drawback: Each variable is considered independently in each occurrence, even if the same variable occurs multiple times in the given expression. This can lead to a large overestimation of the result, which has been rigorously demonstrated [14, 7, 23, 24, 18].

Constrained fuzzy arithmetic. To avoid the overestimation effect (which is also commonly referred to as the “dependency effect” in interval arithmetic literature) of the classical methods, Dong and Wong [7] proposed a more accurate, non-overestimating approach called FWA algorithm. In its initial version, the algorithm was only applicable to functions that were monotonic with respect to all of its fuzzy variables. An extension of the FWA algorithm was presented by Wood et al. [23]. In addition to a reduction of the complexity of the algorithm in certain cases, an enhancement was proposed to correctly compute non-monotonic functions. Wood’s algorithm requires an additional routine to locate internal extrema, which can be done either analytically or numerically with an algorithm suited for global optimization of non-linear functions. Klir [18] suggested a new kind of theoretical framework to take dependencies of fuzzy parameters into account, which he calls “constrained fuzzy arithmetic”. The latter methods can be interpreted as implementations of constrained fuzzy arithmetic. Similar implementations were presented in [2, 20, 24].

More recently, the transformation method has been proposed by Hanss [14] as a practical approach to evaluate fuzzy-parameterized models without the need of an external optimization routine. Due to its exponential complexity, whose degree depends on the number of fuzzy input parameters, it can be concluded that an implementation will be inferior compared to iterative methods derived from global optimization unless significant improvements can be made. In this paper, we will show that this is indeed possible.

The rest of the article is organized as follows: In the next section, we briefly recall the transformation method. In the main part of the article starting with Section 3.1, we first show a simple but computationally efficient implementation based on multi-dimensional array processing in MATLAB (Section 3.2). We then successively show how the algorithm can be improved by avoiding recurring permutations (Section 3.3), by decomposing the model into smaller sub-problems (Section 3.4), and finally, by enhancing the transformation method by techniques derived from interval arithmetic (Section 3.5). Each section contains an analysis of the effects of the modification with respect to algorithmic complexity and computation time, illustrated by numerical examples. In Section 4, we compare our approach to a recent publication [20] to show the effectiveness of our algorithms.

2 The transformation method

We are briefly recalling the implementation of fuzzy arithmetic using the transformation method as a main topic of [14]. To avoid confusion, we adhere to the same notation.

With the general transformation method, each fuzzy parameter $\tilde{p}_i$, $i = 1, \ldots, n$ is first being decomposed into α-cuts, leading to a set P_i of $(m+1)$ intervals $X_i^{(j)}$, $j = 0, 1, \ldots, m$, of the form

$$P_i = \left\{ X_i^{(0)}, X_i^{(1)}, \ldots, X_i^{(m)} \right\} \tag{1}$$

with

$$X_i^{(j)} = \left[a_i^{(j)}, b_i^{(j)} \right] , \quad a_i^{(j)} \leq b_i^{(j)} , \quad i = 1, 2, \ldots, n , \quad j = 0, 1, \ldots, m . \tag{2}$$

The μ-axis is subdivided into m segments, equally spaced by $\Delta\mu = 1/m$. The $(m+1)$ levels of membership μ_j are then given by

$$\mu_j = \frac{j}{m} , \quad j = 0, 1, \ldots, m . \tag{3}$$

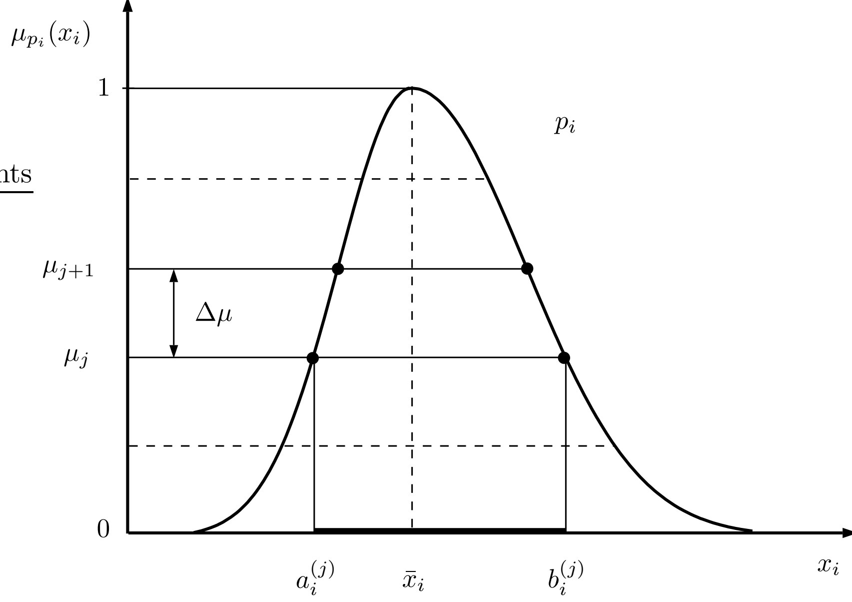


Figure 1: Decomposition of the ith uncertain parameter into intervals [14]

For an illustration of the decomposition, see Fig. 1.
The intervals of each level of membership μ_j, $j = 0, 1, \ldots, m$, can now be transformed into arrays $\widehat{X}_i^{(j)}$ of the following form:

$$\widehat{X}_i^{(j)} = \big(\overbrace{\gamma_{1,i}^{(j)}, \gamma_{2,i}^{(j)} \ldots, \gamma_{(m+1-j),i}^{(j)}, \ldots, \gamma_{1,i}^{(j)}, \gamma_{2,i}^{(j)} \ldots, \gamma_{(m+1-j),i}^{(j)}}^{(m+1-j)^{i-1} \quad \text{(m+1-j)-tuples}} \big) \tag{4}$$

with

$$\gamma_{l,i}^{(j)} = \underbrace{\big(c_{l,i}^{(j)}, \ldots, c_{l,i}^{(j)}\big)}_{(m+1-j)^{n-i} \text{ elements}} \tag{5}$$

and

$$c_{l,i}^{(j)} = \begin{cases} a_i^{(j)} & \text{for} \quad l = 1 & \text{and} \quad j = 0, 1, \ldots, m \ , \\ \frac{1}{2}\left(c_{l-1,i}^{(j+1)} + c_{l,i}^{(j+1)}\right) & \text{for} \quad l = 2, 3, \ldots, m-j & \text{and} \quad j = 0, 1, \ldots, m-2 \ , \\ b_i^{(j)} & \text{for} \quad l = m-j+1 & \text{and} \quad j = 0, 1, \ldots, m \ , \end{cases} \tag{6}$$

Note that $a_i^{(j)}$ and $b_i^{(j)}$ are the lower and upper bounds of the interval $X_i^{(j)}$ at the membership level μ_j for the ith uncertain model parameter. Each of the transformed arrays $\widehat{X}_i^{(j)}$ has a total number of $(m + 1 - j)^n$ entries.

Assuming that the fuzzy-parameterized model is given by the arithmetical expression f with the functional form

$$\tilde{q} = f\left(\tilde{p}_1, \tilde{p}_2, \ldots, \tilde{p}_n\right) \ , \tag{7}$$

its estimation is then carried out by evaluating the expression separately at each of the positions of the arrays using the conventional arithmetic for crisp numbers. Thus, if the

output $\tilde{q}$ of the system can be expressed in its decomposed and transformed forms by the arrays $\widehat{Z}^{(j)}$, $j = 0, 1, \dots, m$, the kth element ${}^k\hat{z}^{(j)}$ of the array $\widehat{Z}^{(j)}$ is given by

$$ {}^k\hat{z}^{(j)} = f\left({}^k\hat{x}_1^{(j)}, {}^k\hat{x}_2^{(j)}, \dots, {}^k\hat{x}_n^{(j)} \right) , \tag{8} $$

where ${}^k\hat{x}_i^{(j)}$ denotes the kth element of the array $\widehat{X}_i^{(j)}$. Finally, the fuzzy-valued result $\tilde{q}$ of the problem can be obtained in its decomposed form

$$ Z^{(j)} = \left[a^{(j)}, b^{(j)} \right] , \quad j = 0, 1, \dots, m , \tag{9} $$

by re-transforming the arrays $\widehat{Z}^{(j)}$ according to the recursive formulae

$$ \begin{array}{l} a^{(j)} = \min\limits_k \left(a^{(j+1)}, {}^k\hat{z}^{(j)} \right) \\ b^{(j)} = \max\limits_k \left(b^{(j+1)}, {}^k\hat{z}^{(j)} \right) \end{array} , \quad j = 0, 1, \dots, m-1 , \tag{10} $$

and

$$ a^{(m)} = \min_k \left({}^k\hat{z}^{(m)} \right) = \max_k \left({}^k\hat{z}^{(m)} \right) = b^{(m)} . \tag{11} $$

3 Implementation concepts

3.1 General assumptions

Prior to introducing specific implementation concepts, we would like to give a brief summary of the types of fuzzy-parameterized models that can be handled by the proposed algorithms in this paper. In detail, we will allow the following characteristics:

- The model may have several independent fuzzy-valued input parameters $\tilde{p}_i$ with the membership functions $\mu_{\tilde{p}_i}(x_i)$, $i = 1, 2, \dots, n$, and
- several fuzzy-valued output parameters $\tilde{q}_r$ with the membership functions $\mu_{\tilde{q}_r}(z_r)$, $r = 1, 2, \dots, N$, that are obtained as the result of the evaluation of the model.
- In addition to the fuzzy input and output parameters, the model may have an arbitrary number of crisp input and output parameters.
- The model is composed of expressions or functions f_r, $r = 1, 2, \dots, M$, that perform some operations on the fuzzy input variables.
- Each function must be representable as a computer program that calculates numerical values. It may be composed of an arbitrary number of nested sub-functions.
- Each function must be composed of assignments, standard arithmetic operations, and common elementary functions, such as hyperbolic, trigonometric, or exponential functions.
- The functions may contain loops.

Some general restrictions apply:

- The functions may only contain branches (occurring, for example, within `if-then-else` control structures) if the branching conditions do not depend on any of the fuzzy input parameters.
- The functions may be non-differentiable, except for the topics presented in Section 3.5.2, which considers automatic differentiation to detect monotonicity. In this case, at least piecewise differentiability is required.

In the following, we often show error plots to measure the performance of the presented algorithms. For clarification, we give the definition of the relative error of a fuzzy number in Appendix A.2.

Having defined this general framework, we can now move on to describing our implementation of the transformation method.

3.2 An efficient standard implementation using a multi-dimensional array structure

In this section, we describe an alternative formulation of the transformation method by representing the vector arrays $\widehat{X}_i^{(j)}$ of the transformed intervals as multi-dimensional arrays. To view all possible combinations of the interval bounds as a multi-dimensional array was originally suggested by Dong and Wong for their Fuzzy Weighted Averages (FWA) algorithm [7]. For the general transformation method, however, we must also consider points that lie within the interval bounds.

While the results of the multi-dimensional formulation are exactly identical to the original formulation of the transformation method as presented in [14], the new representation makes the creation and indexing of the arrays less complex for the user and also computationally more efficient, since powerful existing packages for multi-dimensional array processing can be used.

To simplify the notation for the multi-dimensional arrays, we can take advantage of its repetitive contents and its regular structure. We introduce the following notation:

$$\widehat{X}_{D,d} = [s_1, s_2, \ldots, s_t]_{D,d} \,, \tag{12}$$

with D denoting the number of dimensions of the array and d denoting the dimension along which the t different entries s_t of the array vary. Along all other array dimensions, these entries are simply replicated, and the resulting array is of the size t for $D = 1$, $(t \times t)$ for $D = 2$, $(t \times t \times t)$ for $D = 3$, and so on.

To illustrate the notation, two examples for two-dimensional arrays are given below. Another example using three dimensions is given in Fig. 2.

Example 1: $\widehat{X}_{2,1} = [1, 2]_{2,1}$ results in the two-dimensional ($D = 2$) array

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}.$$

Note that the $t = 2$ different array entries s_1 and s_2 vary along the dimension $d = 1$ (the rows).

Example 2: $\widehat{X}_{2,2} = [1, 2, 3, 4]_{2,2}$ results in the array

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

Note that in this case, the $t = 4$ different array entries $s_1, \ldots, s_4$ vary along the dimension $d = 2$ (the columns).

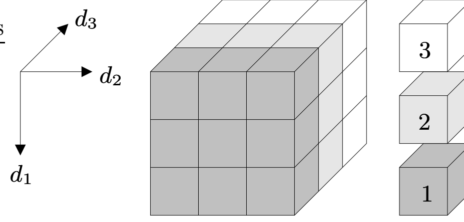


Figure 2: Example: three-dimensional array $\widehat{X}_{3,3} = [1, 2, 3]_{3,3}$.

To extract an array entry ${}^k\hat{x}_{D,d}$, $k = (k_1, k_2, \ldots, k_D)$, we have

$$ {}^k\hat{x}_{D,d} = s_{k_i \ | \ i=d}. \tag{13}$$

The number of array dimensions D corresponds to the number of fuzzy parameters n, with each dimension $d = i$ belonging to a particular fuzzy variable $\tilde{p}_i$. With multi-dimensional arrays and the introduced notation, the array $\widehat{X}_i^{(j)}$ of the general transformation method can be written down as follows:

$$\widehat{X}_{n,i}^{(j)} = \left[c_{1,i}^{(j)}, \ldots, c_{m-j+1,i}^{(j)} \right]_{n,i} \tag{14}$$

with

$$c_{l,i}^{(j)} = \begin{cases} a_i^{(j)} & \text{for} \quad l = 1 & \text{and} \quad j = 0, 1, \ldots, m, \\ \frac{1}{2}\left(c_{l-1,i}^{(j+1)} + c_{l,i}^{(j+1)} \right) & \text{for} \quad l = 2, 3, \ldots, m-j & \text{and} \quad j = 0, 1, \ldots, m-2, \\ b_i^{(j)} & \text{for} \quad l = m-j+1 & \text{and} \quad j = 0, 1, \ldots, m. \end{cases} \tag{15}$$

For the reduced transformation method, we obtain

$$\widehat{X}_{n,i}^{(j)} = \left[a_i^{(j)}, b_i^{(j)} \right]_{n,i}. \tag{16}$$

The multi-dimensional output arrays $\widehat{Z}^{(j)}$ can easily be obtained by computing each entry according to

$$ {}^k\hat{z}^{(j)} = F\left({}^k\hat{x}_{n,1}^{(j)}, {}^k\hat{x}_{n,2}^{(j)}, \ldots, {}^k\hat{x}_{n,n}^{(j)} \right), \ \forall\ k = (k_1, k_2, \ldots, k_n),\ k_i = 1, 2, \ldots, (m+1-j). \tag{17}$$

Fig. 3 shows implementations of the general and the reduced transformation method in MATLAB applying the above-described technique. MATLAB's `shiftdim` and `repmat` commands are used to construct the full multi-dimensional matrices for each input parameter. These full arrays are passed to the analytical function which operates element-wise over the arrays using MATLAB's built-in arithmetic array operators.

Although this implementation runs very fast, there is also a drawback: For each of the n fuzzy input parameters, $8n2^n$ (reduced) and $8n(m+1)^n$ (general, for the lowest of the $(m+1)$ α-cuts) bytes are required just to allocate the input arrays of the function when using double-precision floating point numbers. An alternative implementation is not to explicitly construct the arrays, but to perform the function evaluations element-by-element or in smaller chunks instead (this could be easily achieved by slightly modifying the algorithms above). However, one can expect performance loss with increasing number of function calls. An algorithm similar to the reduced transformation method that performs the function evaluations element-by-element without explicitly constructing the arrays is the Level Interval Algorithm (LIA) introduced by Wood, Otto, and Antonsson [23].

```
function fzr = rtrm(func, varargin)
% RTRM Fast, vectorized implementation of the
% reduced transformation method (Hanss 2002)
% using multi-dimensional arrays.
%    FZR = RTRM(FUNC, FZ1, FZ2, ..., FZN)
%    calls the reduced transformation method
%    with N fuzzy numbers in alpha-cut repre-
%    sentation. The function FUNC must be an
%    analytical expression using \textsc{Matlab}'s
%    array arithmetic (or logic) operators.
%
%    Example:
%    f = inline('x.^2 - y');
%    fz1 = [0 5; 1 4; 2 3; 2.5 2.5];
%    fz2 = [1 3; 1.5 2.5; 1.75 2.25; 2 2];
%    fzr = rtrm(f, fz1, fz2)
%    plot(fzr,linspace(0,1,4))

n = nargin - 1;
m = size(varargin{1},1) - 1;
fzr = repmat([inf -inf],m+1,1);
for j = m+1:-1:1
  for i = 1:n
    repvec = 2*ones(n,1);
    repvec(i) = 1;
    x{i} = repmat(shiftdim(...
           varargin{i}(j,:)',1-i),repvec);
  end
  z = reshape(feval(func,x{:}),2^n,1);
  fzr(j,1) = min(min(z),fzr(min(j,m)+1,1));
  fzr(j,2) = max(max(z),fzr(min(j,m)+1,2));
end
```

```
function fzr = gtrm(func, varargin)
% GTRM Fast, vectorized implementation of the
% general transformation method (Hanss 2002)
% using multi-dimensional arrays.
% Syntax see HELP RTRM

n = nargin - 1;
m = size(varargin{1},1) - 1;
c_old = zeros(2,n);
fzr = repmat([inf -inf],m+1,1);
for j = m+1:-1:1
  t = max(m+2-j,2);
  c = zeros(t,n);
  for i = 1:n
    c(1,i) = varargin{i}(j,1);
    c(2:m-j+1,i) = 0.5*(c_old(1:m-j,i)+...
                   c_old(2:m+1-j,i));
    c(end,i) = varargin{i}(j,2);
    repvec = t*ones(n,1);
    repvec(i) = 1;
    x{i} = repmat(shiftdim(c(:,i),1-i),repvec);
  end
  c_old = c;
  z = reshape(feval(func,x{:}),t^n,1);
  fzr(j,1) = min(min(z),fzr(min(j,m)+1,1));
  fzr(j,2) = max(max(z),fzr(min(j,m)+1,2));
end
```

Figure 3: MATLAB implementation, reduced (left) and general (right) transformation method

3.3 Avoiding additional function evaluations for recurring combinations

With the transformation method, the fuzzy parameters are restricted to convex fuzzy numbers. This implies that the decomposition of each fuzzy number results in intervals that are subsets of the next-lower one, i.e. $X_i^{(j)} \subset X_i^{(j-1)}$, $j = 1, \ldots, m$. We find that the decomposition scheme of the general transformation method sometimes produces recurring points, depending on the shape of the membership function, which in turn generate recurring combinations. This is especially true for fuzzy numbers with a symmetric triangular membership function (see Fig. 4). One could remove these recurring combinations from the evaluation procedure and therefore save computation time. Going one step further, we can try to reuse as many points for different α-cuts as possible by selecting the inner points suitably, i.e. for any α-cut, we consider only inner points that have already occurred in a higher-level α-cut. Consequently, instead of Eq. 15, we use

$$c_{l,i}^{(j)} = \begin{cases} a_i^{(j)} & \text{for} \quad l = 1 & \text{and} \quad j = 0, 1, \ldots, m, \\ c_{l-1,i}^{(j+2)} & \text{for} \quad l = 2, 3, \ldots, m-j & \text{and} \quad j = 0, 1, \ldots, m-2, \\ b_i^{(j)} & \text{for} \quad l = m-j+1 & \text{and} \quad j = 0, 1, \ldots, m, \end{cases} \tag{18}$$

to compute the discretization points.

For symmetric triangular membership functions, the obtained discretization is identical to the original formulation, since for two subsequent α-cuts,

$$c_{l-1,i}^{(j+2)} = \frac{1}{2}(c_{l-1,i}^{(j+1)} + c_{l,i}^{(j+1)})$$

for $l = 2, 3, \ldots, m-j$ and $j = 0, 1, \ldots, m-2$ (see Fig. 4(**b**)). For other, arbitrarily shaped membership functions, the distribution of the points is less regular, but of similar density. Fig. 5 illustrates the new scheme.

We achieve a reduction of the computational complexity of the general transformation method due to a decrease of the required function evaluations. However, the less regular distribution of the inner points results in less accurate results compared to the original formulation if the number of α-cuts is kept constant. The trade-off between these two properties is further analyzed in the following regarding the criteria algorithmic complexity, convergence, and computation time.

3.3.1 Algorithmic Complexity

The overall complexity of the general transformation method is given by the total number of permutations that the function is evaluated for. It can be determined for both variants of the method according to Table 1, depending on the number of α-cuts $(m+1)$ and the number of uncertain parameters n. In general, one can say that if the number of α-cuts $(m+1)$ is large compared to the number of uncertain parameters, the new scheme offers a significantly better complexity. A few examples are given in Table 2.

PSfrag replacements

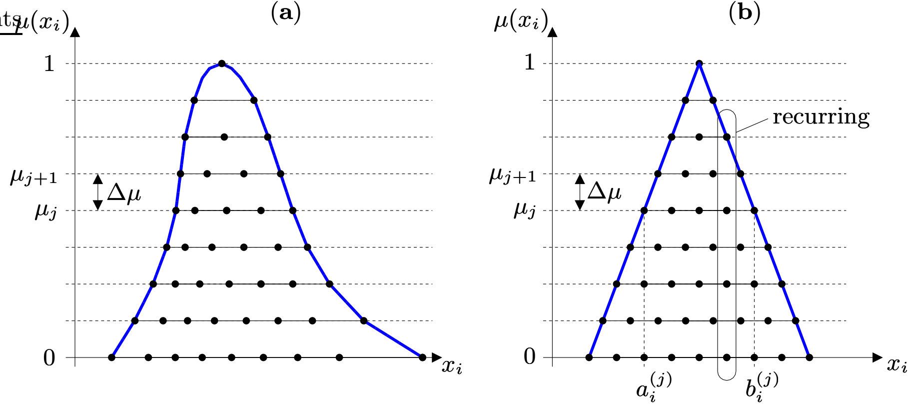


Figure 4: Standard decomposition: (**a**) arbitrary, (**b**) symmetric triangular shape

PSfrag replacements

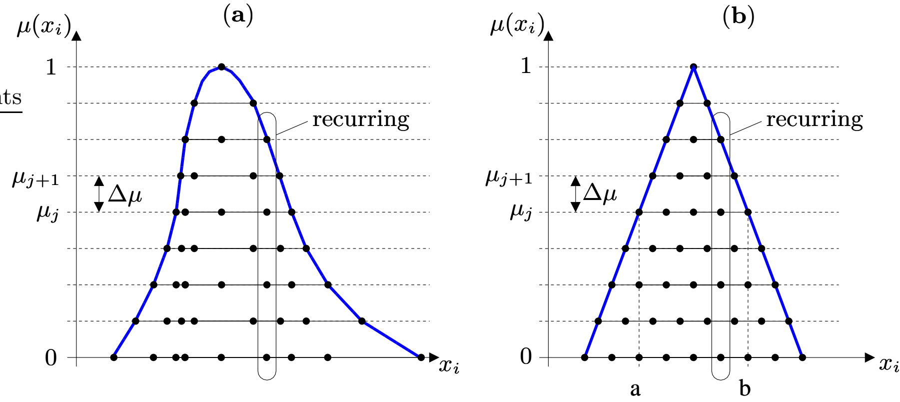


Figure 5: Decomposition, new formulation: (**a**) arbitrary, (**b**) symmetric triangular shape

Table 1: Number of permutations

original	*new*
$\sum_{k=1}^{m+1} k^n$	$m^n + (m+1)^n$

Table 2: Number of permutations for various m, n

# of α-cuts	*# of parameters*	*original*	*new*	*factor*
$m+1=10$	$n=2$	385	181	2.1
$m+1=100$	$n=2$	338350	19801	17.1
$m+1=10$	$n=3$	3025	1729	1.7
$m+1=100$	$n=3$	26 Million	2.0 Million	12.9
$m+1=10$	$n=5$	220825	159049	1.4
$m+1=100$	$n=5$	172 Billion	20 Billion	8.8
$m+1=10$	$n=10$	15 Billion	13 Billion	1.1
$m+1=100$	$n=10$	$1 \cdot 10^{21}$	$2 \cdot 10^{20}$	5.0

3.3.2 Convergence

The performance of the new scheme implementing the removal of recurring permutations depends on the shape of the membership functions of the input parameters. In the following, we examine three common cases: symmetric triangular, non-symmetric triangular, and quasi-Gaussian membership functions.

Symmetric triangular fuzzy numbers. Since the new formulation generates exactly the same discretization points for symmetric triangular fuzzy numbers as the original formulation, but recurring points are not evaluated, we can expect the according new algorithm to be computationally more efficient regardless of the evaluated model function. In Fig. 8, we show the number of function evaluations compared to the error of the resulting fuzzy number for three test functions (definitions see Table 3) considering symmetric triangular fuzzy numbers. For the functions f_1 and f_2, the input parameters have been taken to $\tilde{p}_1 = \langle 2.5, 2.5, 2.5\rangle_{\mathrm{TFN}}$ and $\tilde{p}_2 = \langle 3, 2, 2\rangle_{\mathrm{TFN}}$, and for the function f_3, the input parameters are $\tilde{p}_1 = \langle 0, 3, 3\rangle_{\mathrm{TFN}}$ and $\tilde{p}_2 = \langle 0, 2, 2\rangle_{\mathrm{TFN}}$ (the notation is explained in Appendix A.1; the input parameters and the results are illustrated in the Figure 6 & 7, top row). One can see that for the two-dimensional case, the new algorithm `gtrmrecur` requires significantly less function evaluations to achieve the same accuracy as the original method `gtrm`.

Non-symmetric triangular and quasi-Gaussian fuzzy numbers. As can be seen from Fig. 5, the new scheme generates a less regular distribution of the inner points for non-symmetric and/or non-linear membership functions. Thus, the benefits of a smaller number of evaluated permutations are partly lost due to the disadvantage of less regularly distributed inner points. However, experiments show that the new algorithm usually outperforms the original algorithm already for a relatively small number of α-cuts. Two examples are given below. For an illustration of the input parameters and the results, see the Figures 6 & 7, middle and bottom row.

- *Non-symmetric triangular membership functions:* Input parameters: $\tilde{p}_1 = \langle 1, 1, 4\rangle_{\mathrm{TFN}}$ and $\tilde{p}_2 = \langle 4.5, 3.5, 0.5\rangle_{\mathrm{TFN}}$ for the functions f_1 and f_2, and $\tilde{p}_1 = \langle -2, 1, 5\rangle_{\mathrm{TFN}}$ and $\tilde{p}_2 = \langle 0.25, 2.25, 1.75\rangle_{\mathrm{TFN}}$ for the function f_3. Convergence history see Fig. 9).
- *Quasi-Gaussian membership functions:* Input parameters: $\tilde{p}_1 = \langle 2.5, 2.5/3, 3\rangle_{\mathrm{QGFN}}$ and $\tilde{p}_2 = \langle 3, 2/3, 3\rangle_{\mathrm{QGFN}}$ for the functions f_1 and f_2, and $\tilde{p}_1 = \langle 0, 1, 3\rangle_{\mathrm{QGFN}}$ and $\tilde{p}_2 = \langle 0, 2/3, 3\rangle_{\mathrm{QGFN}}$ for the function f_3. Convergence history see Fig. 10).

Table 3: Test functions

function		*characteristics*
$f_1(x_1, x_2) =$	$\cos(\pi x_1) x_2$	Non-monotonic with respect to x_1 within the examined range.
$f_2(x_1, x_2) =$	$\left[(x_1-2)^4 + (x_2-2)^2 + 0.2\right]^{-1}$	One local extremum.
$f_3(x_1, x_2) =$	$3(1-x_1)^2 \exp[-x_1^2 - (x_2+1)^2]$ $-10(\frac{x_1}{5} - x_1^3 - x_2^5)\exp[-x_1^2 - x_2^2]$ $-\frac{1}{3}\exp[-(x_1+1)^2 - x_2^2]$	Several local extrema.

PSfrag replacements
PSfrag replacements
PSfrag replacements

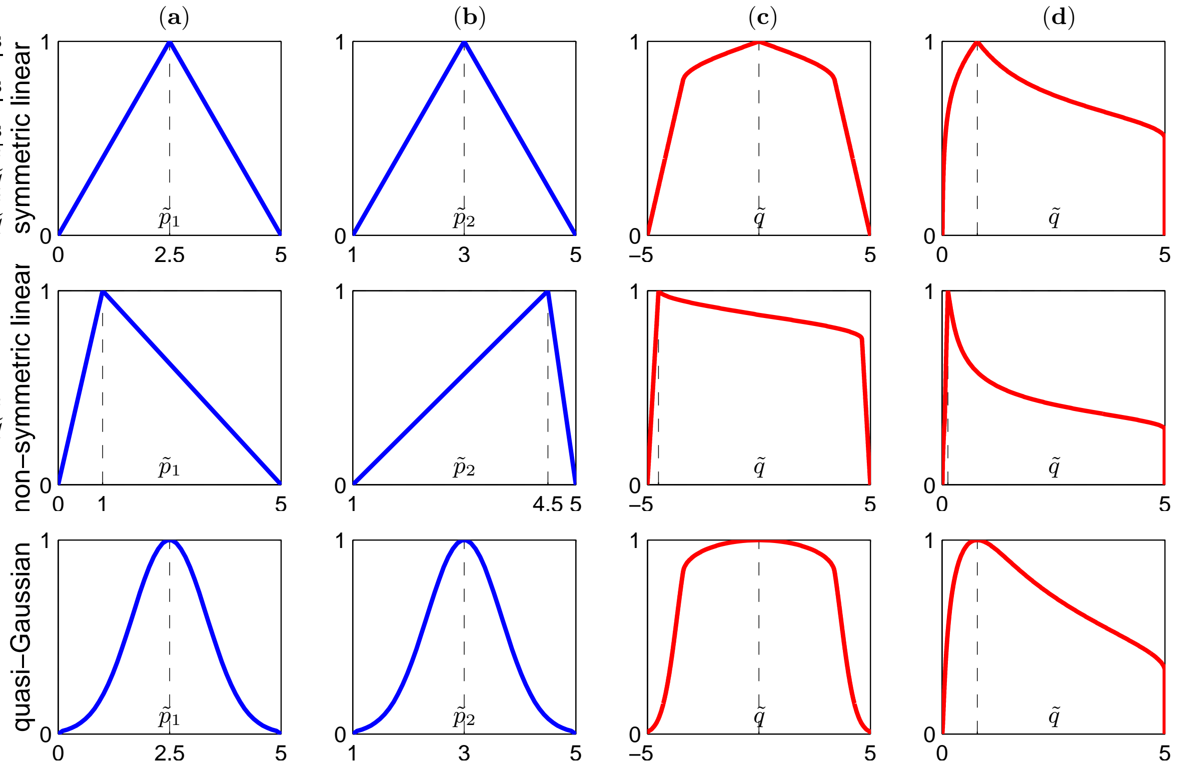


Figure 6: Fuzzy input parameters (**a**,**b**) and output parameter of f_1 (**c**) and f_2 (**d**)

PSfrag replacements
PSfrag replacements
PSfrag replacements

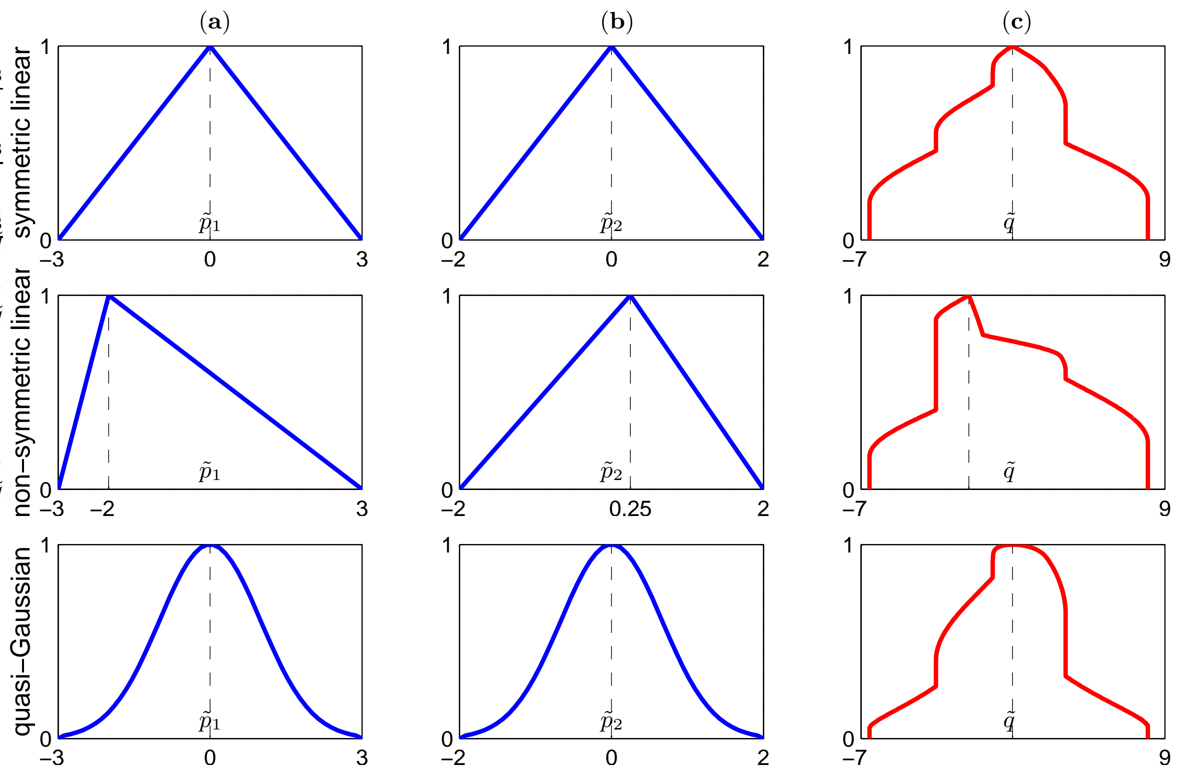


Figure 7: Fuzzy input parameters (**a**,**b**) and output parameter (**c**) of f_3

PSfrag replacements

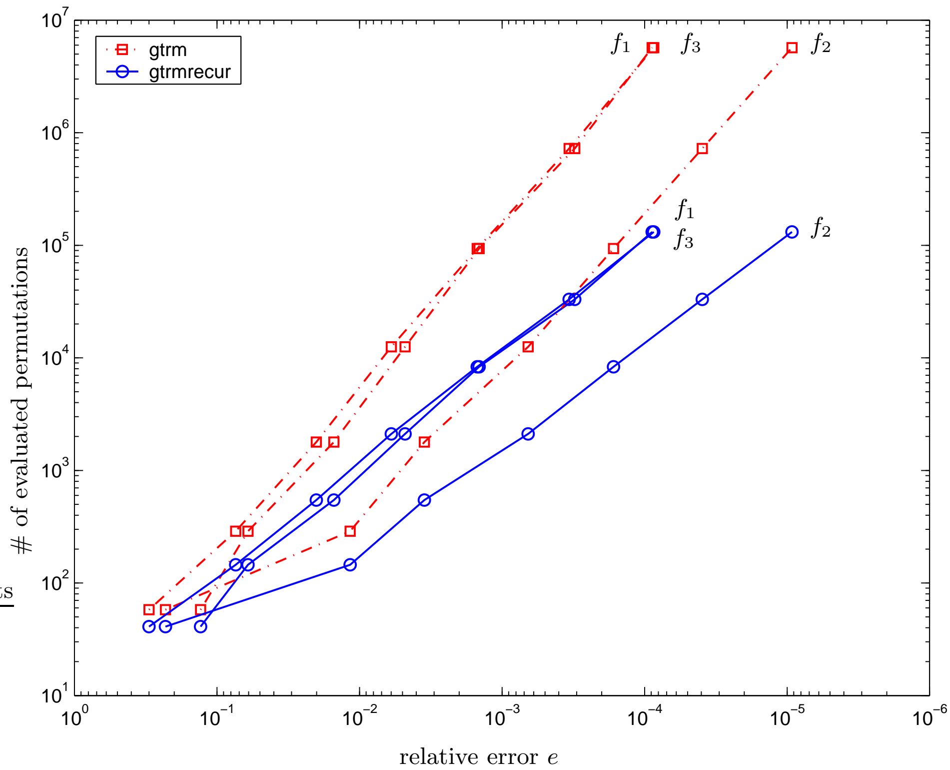


Figure 8: Number of permutations vs. error, symmetric triangular input parameters

PSfrag replacements

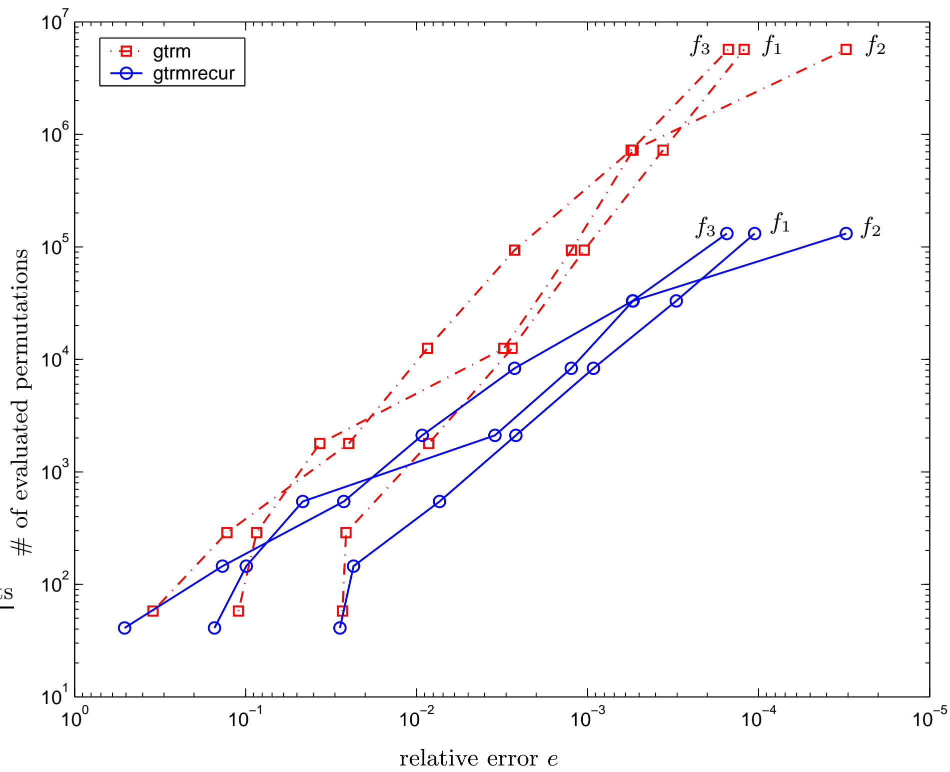


Figure 9: Number of permutations vs. error, non-symmetric triangular input parameters

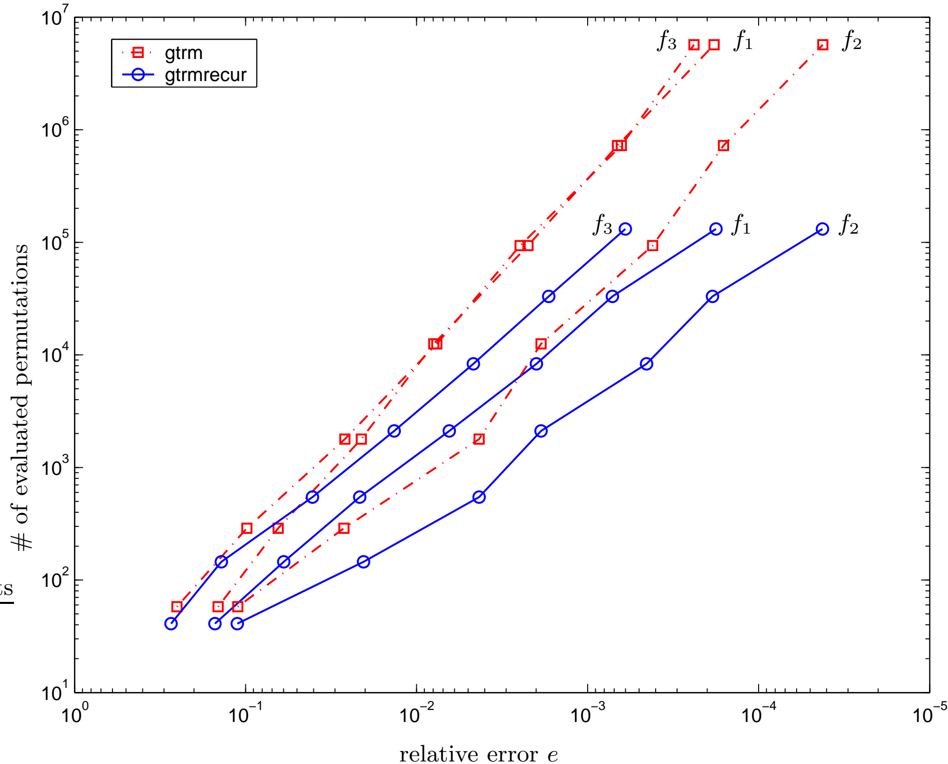


Figure 10: Number of permutations vs. error, Gaussian input parameters

3.3.3 Implementation

We have concluded above that the new formulation can achieve better accuracies with a lower number of function evaluations. The next question is how to efficiently implement the new scheme. An elegant way of avoiding the recurring permutations is described in the following.

First of all, we split each discretization of the n fuzzy parameters $\tilde{p}_i$ into two separate parts (I_i) and (II_i), as illustrated in Fig. 11. This offers the advantage that when generating the permutations, we can treat the parts separately, and we only need to consider the lowest α-cut of each part to obtain all of the permutations, since the permutations of the upper α-cuts are automatically included (this can be observed in Fig. 11). In addition to that, we construct the array containing the permutations by numbering the points from the inside of the fuzzy number towards the outside so we can very easily extract all permutations of any desired membership level as an entire block.

By following the above procedure, we obtain two full n-ary arrays containing all permutations. The evaluation of the arrays is performed element-wise, as usual. The final step, the recomposition of the resulting fuzzy number, can be done by computing, for both result arrays, the minima and maxima for all entries that belong to the same membership level (which is easy to determine due to the convenient point ordering). An implementation in MATLAB is shown in Fig. 12.

PSfrag replacements

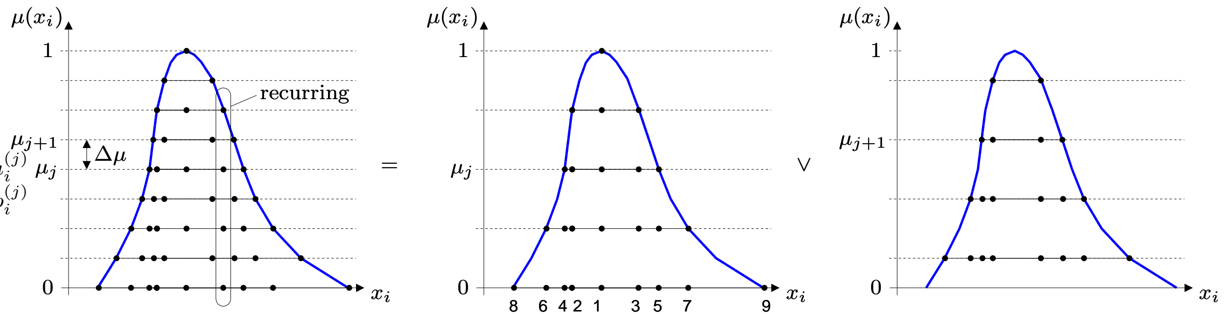


Figure 11: Splitting the discretization

```
function fzr = gtrmrecur(func, varargin)

n = nargin - 1;
m = size(varargin{1},1) - 1;
indices.subs = {};
indices.type = '()';
fzr = zeros(m+1,2);
for k = 1:2
  t = (floor((m+2-k)/2)+rem(m+2-k,2))*2-2+k;
  for i = 1:n
    c = reshape(varargin{i}(end+1-k:-2:1,:)',t+2-k,1);
    c = c(3-k:end);
    repvec = t*ones(1,n);
    repvec(i) = 1;
    x{i} = repmat(shiftdim(c,1-i),repvec);
  end
  s = k;
  z = feval(func,x{:});
  for j = m+2-k:-2:1
    indices.subs(1:n) = {1:s};
    w = reshape(subsref(z,indices),s^n,1);
    fzr(j,1) = min(w);
    fzr(j,2) = max(w);
    s = s + 2;
  end
end
for j = m:-1:1
  fzr(j,1) = min(fzr(j+1,1),fzr(j,1));
  fzr(j,2) = max(fzr(j+1,2),fzr(j,2));
end
```

Figure 12: MATLAB implementation, recurring points removed

3.3.4 Computation time

In this section, we present computation time results for the new algorithm `gtrmrecur` in comparison to the fast algorithm `gtrm` of Section 3.2 (see Fig. 13). Since the performance gains regarding triangular and quasi-Gaussian fuzzy input parameters are very similar, we only show plots for quasi-Gaussian fuzzy numbers here.

In addition to the algorithm `gtrmrecur`, Fig. 13 shows the performance of the function `gtrmrecuropt`. Instead of computing the minima and maxima separately for all permutations of each level of membership, this function uses a compiled subroutine (MEX function) to compute the minima and maxima in an optimized manner such that when moving from one membership level to the next lower one, only the additional permutations are taken into consideration. This is a necessary step to avoid computing the minima and maxima for recurring permutations as well. With the arrival of JIT (just in time) compilation technology in MATLAB 6.5, it is also possible to avoid a MEX file completely with only minor performance losses.

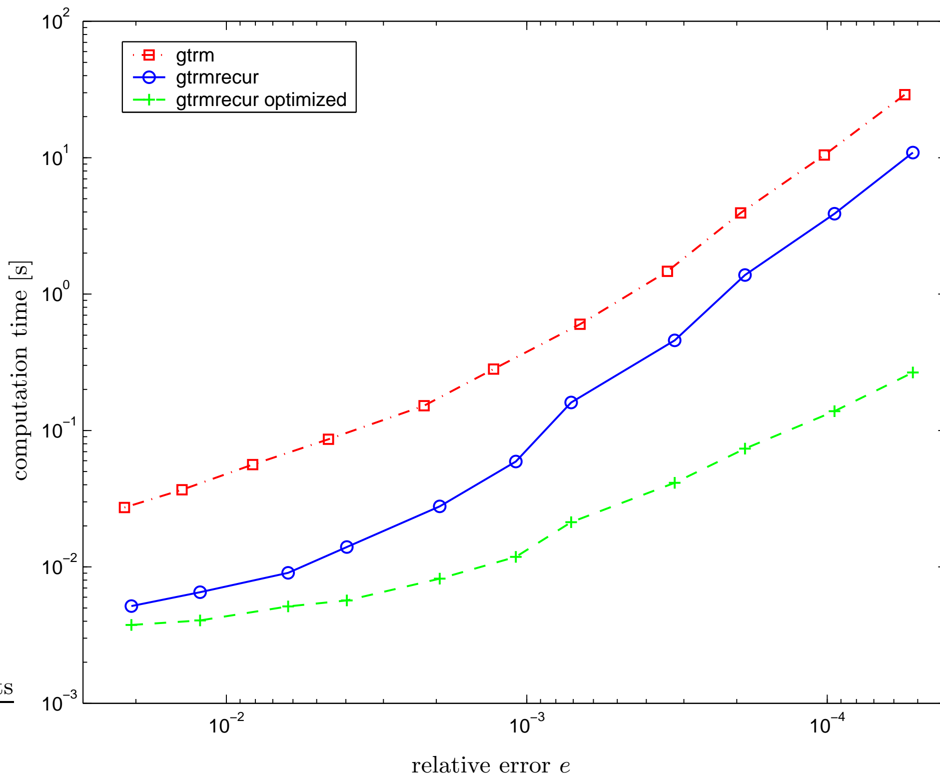


Figure 13: Computation time vs. error, two quasi-Gaussian input parameters

To conclude this section, let us have a look at the limitations of the new algorithm. From Table 2, we can see that with increasing number n of fuzzy input parameters, the reduction factor for the number of permutations becomes smaller, especially for crude discretizations with a small number of α-cuts. As an example, we consider 10 fuzzy parameters discretized with 10 α-cuts. The number of permutations decreases only from about 15 Billion to 13 Billion compared to the original formulation. A higher number of α-cuts would result in a larger difference (e.g. about one-fifth for 100 α-cuts), but this fine discretization resolution

is still impracticable due to the extremely large number of generated permutations ($2 \cdot 10^{20}$). To illustrate this, Fig. 14 shows convergence plots for the output of a fuzzy-parameterized model with 6 fuzzy input parameters of quasi-Gaussian shape. `gtrmrecuropt` is only about 50 % faster for 10 α-cuts.

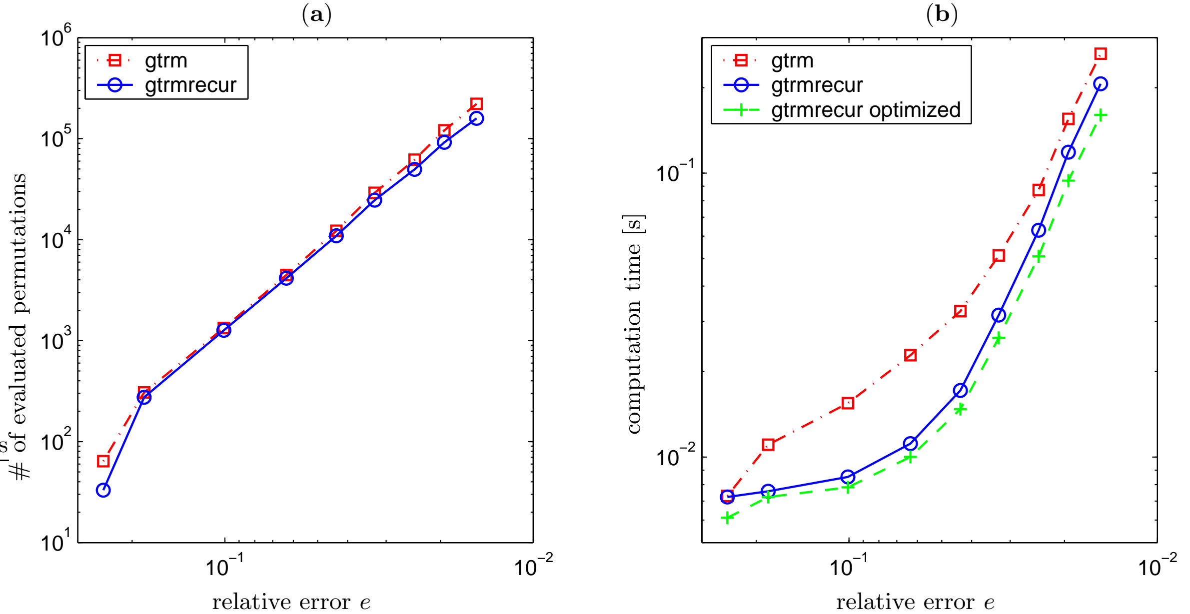


Figure 14: Number of permutations (**a**) and computation time (**b**) vs. error, six quasi-Gaussian input parameters

3.4 Decomposition of model functions

The transformation method is defined in a way so that the function evaluation is performed globally for each permutation. An implementation, however, does not necessarily have to consider each permutation separately. By using intermediate results suitably, the overall number of elementary operations can often be reduced. We can obtain these intermediate results by decomposing the original function into nested sub-functions that are evaluated individually. The key to a reduction in computational cost is to find sub-functions of the least possible dimensionality.

To measure the efficiency of decomposition compared to global evaluation, we will compute the arithmetic complexity more accurately by counting the number of arithmetic operations rather than the mere number of permutations. Although nonlinear operations, such as exponentials and trigonometric functions take much longer than elementary operations (e.g. additions or multiplications), we will count each operation with the same cost factor of one for the sake of simplicity.

The decomposition scheme we will propose in this section can be applied to all variations of the transformation method (reduced, general), including the general transformation method considering recurring combinations, as described in Section 3.3.

Let us first illustrate the idea on the example function f_2 from Table 3 for the general transformation method:

$$f_2(x_1, x_2) = \left[(x_1 - 2)^4 + (x_2 - 2)^2 + 0.2\right]^{-1}. \tag{19}$$

If we would like to compute f_2 using fuzzy numbers decomposed into 20 α-levels, we would have to consider

$$n_{\text{perm}} = \sum_{k=1}^{20} k^2 = 2870 \tag{20}$$

permutations according to Table 1. Counting the total number of operations o required to execute one function evaluation in the two-dimensional space, we count $o = 7$. Multiplying o by the number of permutations gives us the total number of arithmetic operations to

$$n_{\text{ops}} = o \cdot n_{\text{perm}} = 20090. \tag{21}$$

Instead of evaluating the function f_2 directly, we could decompose the function first to reduce the number of arithmetic operations executed in the two-dimensional space:

$$f_2(x_1, x_2) = [v_1(x_1) + v_2(x_2)]^{-1} \quad \text{with} \tag{22}$$

$$y_1 = v_1(x_1) = (x_1 - 2)^4 \tag{23}$$

$$y_2 = v_2(x_2) = (x_2 - 2)^2 + 0.2. \tag{24}$$

Obviously, we have reduced the number of operations in the two-dimensional space to $o_2 = 2$ (one addition and one division in Eq. 22). In addition, we have two functions in one fuzzy variable, with a combined number of $o_1 = 5$ operations. The total number of elementary operations is therefore

$$n_{\text{ops}} = 2 \sum_{k=1}^{20} k^2 + 5 \sum_{k=1}^{20} k^1 = 6790. \tag{25}$$

We have thus reduced the number of elementary operations by a factor of about three compared to Eq. 21.

We now illustrate the procedure in detail for the above function in two variables using the multi-dimensional array notation introduced in Section 3.2.

Evaluating Eq. 23 gives

$$\begin{aligned} \widehat{X}_{1,1}^{(j)} &= \left[c_{1,1}^{(j)}, \ldots, c_{m-j+1,1}^{(j)}\right]_{1,1} \\ {}^k\hat{y}_1^{(j)} &= f_1\left({}^k\hat{x}_{1,1}^{(j)}\right) \quad \forall \;\; k = 1, 2, \ldots, m - j + 1 \\ \widehat{Y}_{1,1}^{(j)} &= \left[f_1(c_{1,1}^{(j)}), \ldots, f_1(c_{m-j+1,1}^{(j)})\right]_{1,1}. \end{aligned} \tag{26}$$

Similarly, for Eq. 24, we get

$$\begin{aligned} \widehat{X}_{1,2}^{(j)} &= \left[c_{1,2}^{(j)}, \ldots, c_{m-j+1,2}^{(j)}\right]_{1,2} \\ {}^k\hat{y}_2^{(j)} &= f_2\left({}^k\hat{x}_{1,2}^{(j)}\right) \quad \forall \;\; k = 1, 2, \ldots, m - j + 1 \\ \widehat{Y}_{1,2}^{(j)} &= \left[f_2(c_{1,2}^{(j)}), \ldots, f_2(c_{m-j+1,2}^{(j)})\right]_{1,2}. \end{aligned} \tag{27}$$

We now have two arrays $\widehat{Y}_{1,1}^{(j)}$ and $\widehat{Y}_{1,2}^{(j)}$ containing the intermediate results of the one-dimensional sub-problems. Since we now have to continue the calculation in two dimensions, we have to expand the intermediate result array to two dimensions. In the short notation, we can do this by increasing the dimension parameter of the multi-dimensional arrays Y by one:

$$\begin{aligned}\widehat{Y}_{2,1}^{(j)} &= \left[f_1(c_{1,1}^{(j)}), \ldots, f_1(c_{m-j+1,1}^{(j)})\right]_{2,1} \\ \widehat{Y}_{2,2}^{(j)} &= \left[f_2(c_{1,2}^{(j)}), \ldots, f_2(c_{m-j+1,2}^{(j)})\right]_{2,2}\end{aligned} \tag{28}$$

Finally, evaluating Eq. 22 using the transformed arrays $\hat{Y}_{2,1}^{(j)}$ and $\hat{Y}_{2,2}^{(j)}$ gives the array $\hat{Z}$ with its elements

$$^{k}\hat{z}^{(j)} = F\left({}^{k}\hat{y}_{2,1}^{(j)}, {}^{k}\hat{y}_{2,2}^{(j)}\right) \;\forall\; k = (k_1, k_2),\; k_1, k_2 = 1, 2, \ldots, m - j + 1. \tag{29}$$

For model functions in two variables, the above scheme can be quite easily implemented. However, once we consider an arbitrary number of fuzzy parameters with functions that are decomposed into a deeply nested structure of sub-functions of varying degree, the implementation becomes a rather non-trivial task. The next section describes our approach. Thereafter, the benefits of decomposing the model function are analyzed with respect to algorithmic complexity and computation time. We do not need to perform error or convergence analyses, since the achieved results are identical to the ones obtained for global function evaluation without decomposition from a non-algorithmic point of view.

3.4.1 Implementation

Our reference implementation in MATLAB uses an object-oriented approach with operator overloading to dynamically traverse the model function, and apply the decomposition according to the built-in rules of arithmetic operator precedence. This may not always lead to the optimal decomposition of the function, but usually achieves a very good result. One can always manually define a better decomposition by either rearranging the function suitably or adding parentheses to the expression when the standard selection of operator precedence (e.g. left-to-right precedence for $+$) is not satisfactory. For example, our implementation would by default decompose equation f_2 from Table 3 to

$$\begin{aligned}f_2(x_1, x_2) &= [v_1(x_1) + v_2(x_2) + 0.2]^{-1} \quad \text{with} \\ v_1(x_1) &= (x_1 - 2)^4 \\ v_2(x_2) &= (x_2 - 2)^2,\end{aligned} \tag{30}$$

since by applying standard arithmetic operator precedence, one obtains a parse tree according to Fig. 15(**a**). Adding a pair of parentheses such that

$$f_2(x_1, x_2) = \left[(x_1 - 2)^4 + \left((x_2 - 2)^2 + 0.2\right)\right]^{-1}. \tag{31}$$

would improve the parse tree as shown in Fig. 15(**b**), and result in a decomposition according to Eq. 22-24.

Finally, it is important to mention that the entire evaluation of the decomposed model must be performed using the transformed sub-arrays $\widehat{X}$ – one may not re-compose intermediate results into fuzzy numbers, since this would lead to the well-known dependency effect [19, 14, 1] of interval analysis if there are multiple occurrences of the variables.

PSfrag replacements

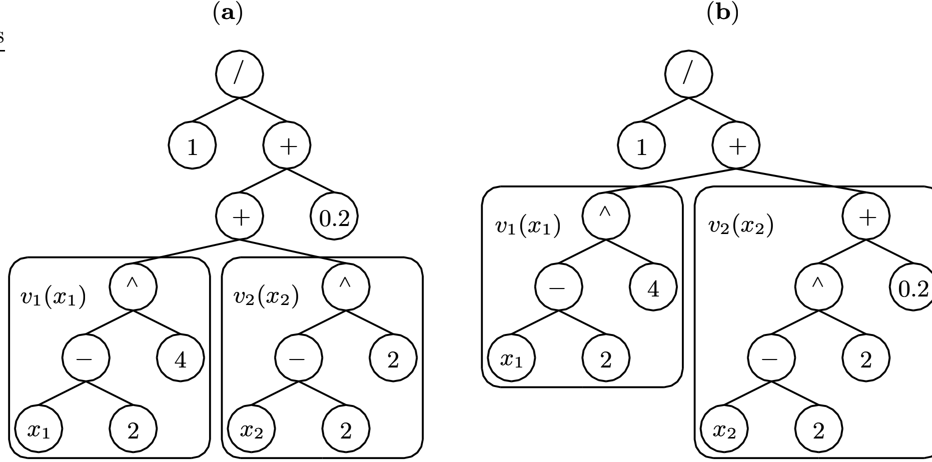


Figure 15: function parse tree, standard (**a**) and with additional parentheses (**b**)

3.4.2 Algorithmic complexity

Instead of just considering the total number of permutations, the complexity is now more accurately determined by the number of elementary arithmetic operations. The general and the reduced transformation method treat each permutation separately. With a decomposition of the model function, the substructure of the function is considered and parts of the evaluation can be performed in a lower-dimensional space. However, at least one arithmetic operation will occur in the maximum dimension. Table 4 shows the algorithmic complexities of different variations of the transformation method. o denotes the number of arithmetic operations in the model function. o_l, $l = 1, 2, \ldots, n$ denote the arithmetic operations in l-dimensional space.

Let us consider a function f_4 of four variables to demonstrate the computation of the arithmetic complexity:

$$f_4(x_1, x_2, x_3, x_4) = \frac{2x_1x_2 - (x_3 + x_4)^2}{5x_1 - (3x_2 + 3)}. \tag{32}$$

We determine the number of elementary arithmetic operators o to 10 (2 additions, 2 subtrac-

Table 4: Number of arithmetic operations n_{ops}. $o_n \geq 1$, $o = \sum_{l=1}^{n} o_l$.

General transformation method		*Reduced transformation method*	
w/out	*with decomposition*	*w/out*	*with decomposition*
$o \sum_{k=1}^{m+1} k^n$	$\sum_{l=1}^{n} \left(o_l \sum_{k=1}^{m+1} k^l \right)$	$o(m+1)2^n$	$\sum_{l=1}^{n} \left(o_l (m+1) 2^l \right)$

General considering recurring combinations	
w/out	*with decomposition*
$o(m^n + (m+1)^n)$	$\sum_{l=1}^{n} \left(o_l (m^l + (m+1)^l) \right)$

tions, 4 multiplications, 1 power operation, and 1 division). The decomposition is obtained to

Function	# ops.
$f_4(x_1,x_2,x_3,x_4) = (v_4(x_1,x_2) - v_6(x_3,x_4))/v_5(x_1,x_2)$	2
$v_6(x_3,x_4) = (x_3 + x_4)^2$	2
$v_5(x_1,x_2) = v_1(x_1) - v_2(x_2)$	1
$v_4(x_1,x_2) = v_3(x_1)x_2$	1
$v_3(x_1) = 2x_1$	1
$v_2(x_2) = 3x_2 + 3$	2
$v_1(x_1) = 5x_1$	1
Total # of arithmetic operators $o =$	10

(33)

with $o_1 = 4, o_2 = 4, o_3 = 0, o_4 = 2$. Evaluating this function with $n = 4$ fuzzy parameters decomposed into $(m+1) = 10$ α-cuts by applying the general transformation method considering recurring combinations, we calculate the number of total elementary arithmetic operations with decomposition to

$$n_{\text{ops}} = 4(9+10) + 4(9^2+10^2) + 2(9^4+10^4) = 33922, \tag{34}$$

and without decomposition to

$$n_{\text{ops}} = 10(9^4+10^4) = 165610. \tag{35}$$

In this case, the decomposition reduces n_{ops} by a factor of 4.9.

3.4.3 Computation time

Obviously, the achievable computation time savings are related to the reduction of elementary operations n_{ops}. Unfortunately, some of the potential benefits are lost due to the required dynamic resizing and rescaling of the multi-dimensional arrays that must occur on-the-fly. Furthermore, a more sophisticated data structure is needed to hold the meta-data, such as the current dimension of a sub-array and its variable dependencies.

We show the time savings achieved with our reference implementation in Matlab for the three test functions of Table 3 in Fig. 16. We have measured only the time savings for the function `gtrmrecuropt` representing the fastest of the variations of the general transformation method that decomposition can be applied to. For f_2, for example, we achieve a speed-up of 1.8 for sufficiently large amounts of data, which is an efficiency of about 60% compared to the theoretically determined reduction of arithmetic operations by a factor of about three. For the function f_4 from Eq. 32, the speed-up is a factor of about 2.3 (see Figure 17). If the function includes expensive elementary functions that can be executed in a lower-dimensional space, the time savings are most noticeable (such as the cos function in f_1). In this case, the resizing of the arrays takes much less time compared to the additionally required evaluations of the cos function without decomposition.

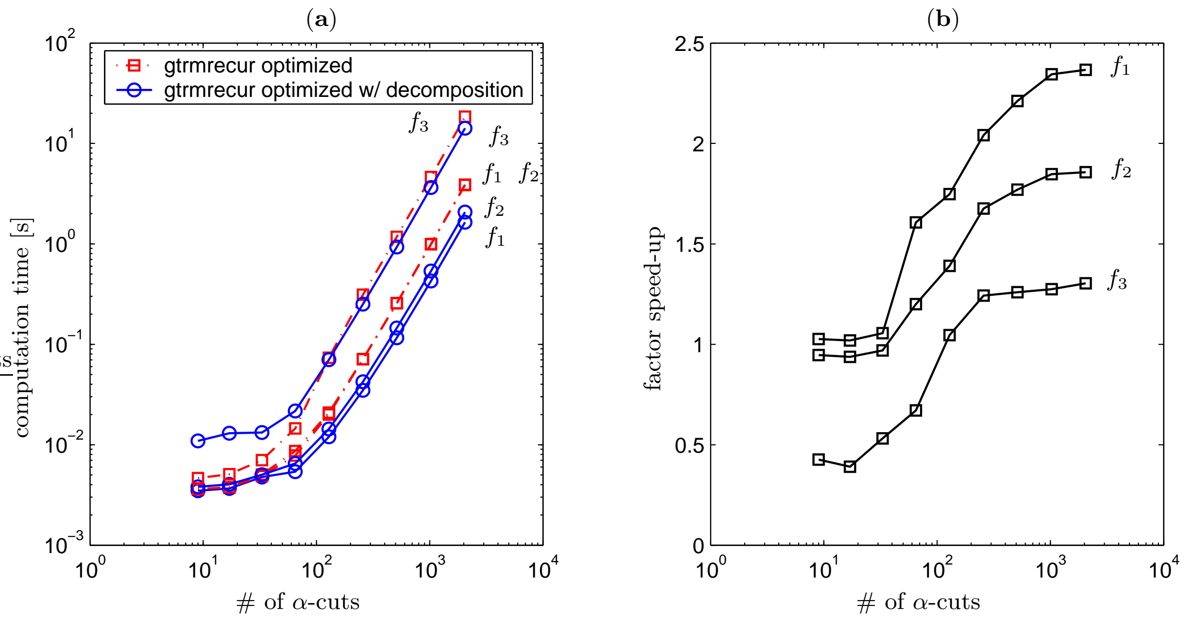


Figure 16: Functions of 2 variables: computation time (**a**) and factor of speed-up (**b**)

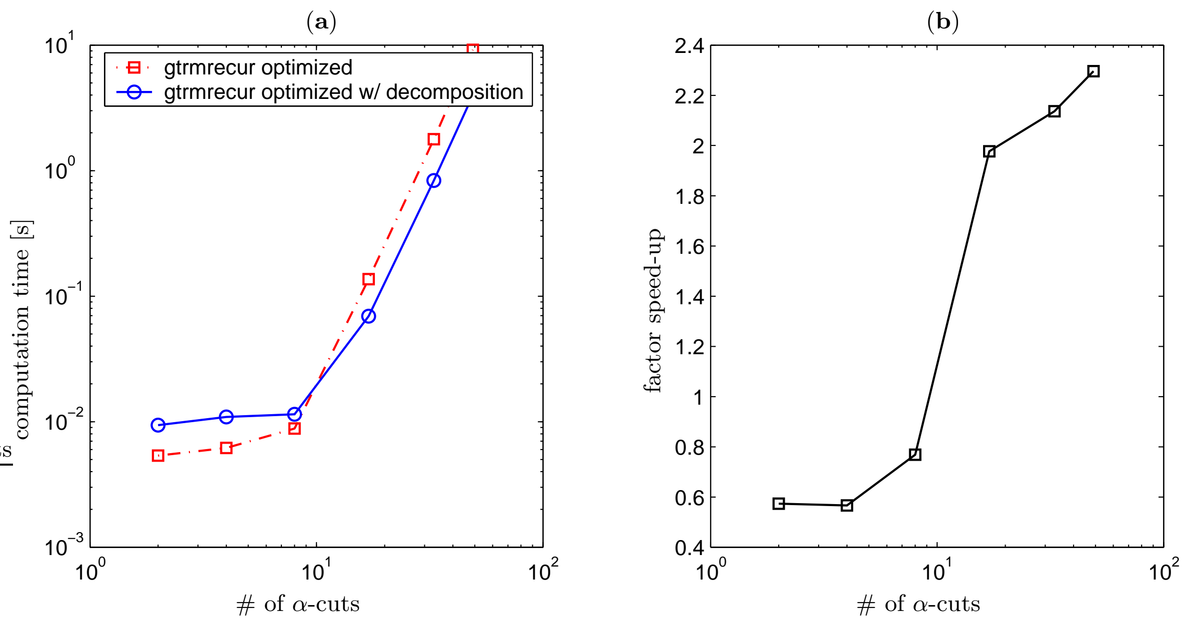


Figure 17: Function f_4 of 4 variables: computation time (**a**) and factor of speed-up (**b**)

3.5 Applying interval arithmetic techniques

So far, we have shown that an efficient implementation can improve the performance of the transformation method by a significant factor. For high accuracies (i.e. a high number of α-cuts), the time savings can well be two orders of magnitude or more. However, one inherent problem of the transformation method has not been tackled yet: The order of the general transformation method still remains, despite all improvements, $\mathcal{O}((m+1)^n)$ for $(m+1)$ α-cuts and n uncertain parameters. This serious limitation makes the general transformation method unsuited for all higher-dimensional problems. In this section, we will combine the transformation method with well-known interval arithmetic techniques to reduce the complexity in certain cases while maintaining the most important property of the method, the avoidance of overestimation. In the first subsection, we will examine single occurrences of variables, which can be accurately handled by interval arithmetic in case of continuous functions. The second subsection describes a monotonicity test based on interval arithmetic and automatic differentiation that can be safely used to greatly reduce the computational complexity for partly monotonic problems.

3.5.1 Complexity reduction by treating single occurrences of variables with interval arithmetic

While standard arithmetic for fuzzy numbers [17] relies entirely on interval arithmetic, the transformation method is based entirely on function evaluations of crisp permutations. Standard fuzzy arithmetic is of linear complexity, but cannot avoid the overestimation effect (also known as the dependency problem) of interval arithmetic. The overestimation effect occurs since each variable occurring more than once is treated as a different variable in each occurrence (see Appendix A.3).

The question that now arises is whether functions that contain several variables more than once, but others only once, can be treated without having to sample the entire Cartesian product space. As shown by Wood et al. [23], it can. They show how to reduce the complexity of Dong and Wong's FWA algorithm [7] from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{n-p})$ with p denoting the number of the n uncertain parameters that occur only once in the model function. We can apply similar techniques to the transformation method to achieve a reduction from $\mathcal{O}((m+1)^n)$ to $\mathcal{O}((m+1)^{\max(n-p,1)})$. In the following, we describe the implementation of our approach and show the numerical results in terms of computation time vs. error.

Implementation. The difficulty that we have to face is that we do not wish to execute the entire computation in interval arithmetic, but only the necessary parts. An essential element of the implementation to achieve this is the creation of three different data types, namely

1. simple fuzzy arrays,
2. constrained fuzzy arrays, and
3. mixed fuzzy arrays.

We perform standard interval arithmetic on the simple fuzzy arrays separately for each α-level. Simple fuzzy arrays are associated with variables that occur only once in the expression.

For operations on variables that occur more than once, we generate permutations stored in a multi-dimensional array according to the scheme of the transformation method with the slight modifications described in Section 3.3. If a simple array has to be combined with a constrained fuzzy array, we create a mixed fuzzy array that enhances the constrained array by an additional dimension to hold the interval bounds. We then perform interval arithmetic for all of the contained permutations.

Obviously, we need to parse the model function prior to the computation to recognize the number of occurrences of each variable. For the computation itself, we proceed in the same manner as described in Section 3.4, however, we can now re-compose intermediate constrained and mixed fuzzy arrays to simple fuzzy arrays as soon as we know that its depending variables have no further occurrences.

The following example will illustrate the procedure. Let

$$y = f_5(x_1, x_2, x_3, x_4) = (x_3(x_1 + x_2)^2 + x_2)/\exp(x_4 + x_1) \tag{36}$$

be the model function. We first parse the function, and find that x_1, x_2 occur more than once, and x_3, x_4 occur only once in the expression. We therefore initialize x_1, x_2 as constrained fuzzy arrays (containing discretized points of the α-cuts in one dimension), and x_3, x_4 as simple fuzzy arrays (containing the left and right bounds of each α-cut). Next, we perform the actual computation in a decomposed manner, according to standard operator precedence, by executing the following steps:

1. $x_1 + x_2 \mapsto u_1$: Expand the constrained fuzzy arrays x_1, x_2 to contain permutations in two dimensions according to the specified number of α-cuts. Perform element-wise addition and store the result in the constrained fuzzy array u_1. Note that u_1 depends on x_1 and x_2.
2. $u_1^2 \mapsto u_2$: Take the square of each permutation in u_1 and store the result in u_2.
3. $x_3 * u_2 \mapsto u_3$: (a): Convert the constrained fuzzy array u_2 to a mixed fuzzy array such that it contains all n_{perm} permutations $p_i, i = 1\ldots, n_{\text{perm}}$, but as degenerate intervals $[p_i, p_i]$. (b): For each α-cut, perform interval arithmetic to obtain the left and the right bound of $x_3 * u_2$ for each interval-valued permutation. Store the result in u_3. Note that u_3 still depends on x_1 and x_2. We do not register x_3 as dependency, since this variable occurs only once in the expression.
4. $u_3 + x_2 \mapsto u_4$: (a): Similar to step 3 (b), perform interval addition of x_2 for each interval-valued permutation in u_3 of each α-cut. (b): Recognize that x_2 does no longer occur in the remaining part of the expression. Therefore, remove the dimension with respect to x_2 from the multi-dimensional array by computing the minima and maxima of the according interval-valued permutations. Store the result in u_4. Note that u_4 now only depends on x_1.
5. $x_4 + x_1 \mapsto u_5$: Convert the constrained fuzzy array x_1 to a mixed fuzzy array and add x_4 by performing interval arithmetic. Store the interval-valued permutations in u_5. Note that u_5 depends on x_1.
6. $\exp(u_5) \mapsto u_6$: For each interval-valued permutation, compute the exponential and store the result in u_6.

7. $u_4/u_6 \mapsto y^*$ (a) Perform interval division for each of the permutations. (b) Since there are no further occurrences of x_1, and x_1 was the last remaining dependency of u_4 and u_6, re-compose the mixed fuzzy array to the simple fuzzy array y^*. y^* contains now a lower approximation of the fuzzy output variable y that can be computed arbitrarily accurate with increasing number of α-cuts.

We can conclude the following: Although the considered function is 4-dimensional, the maximum dimension of the sampled space is just two. Due to the decomposition scheme, we have also avoided to execute the entire computation in two dimensions, but perform each arithmetic operation only in the least required dimensionality. The result y^* is guaranteed to be an underestimation of y, and is at least of identical accuracy as the result obtained by sampling the entire four dimensions, but usually even better, since nearly exact bounds (apart from rounding errors) are used for intermediate interval-valued computations.

Computation time. We have performed extensive numerical tests on the algorithm described in this section, some of the results are presented. Again, we are considering the test functions f_1 to f_4. Figure 18 shows the computation time over the relative error of the fuzzy result. For comparison, we include plots for the implementations `gtrm` and `gtrmrecuropt` (with decomposition), representing the fastest algorithm without interval arithmetic. We observe the following behavior for each of the functions:

- f_1 (Fig. 18 (**a**)): Since each of the variables occurs only once, and the involved operators and elementary functions are continuous, the entire computation is performed in standard interval arithmetic. The complexity is linear, down from quadratic complexity.
- f_2 (Fig. 18 (**b**)): Although the function exhibits an internal extremum, the entire computation can be performed in standard interval arithmetic by applying Eq. 63. The complexity is linear, down from quadratic complexity.
- f_3 (Fig. 18 (**c**)): Both variables occur more than once. The dimensionality of the problem can therefore not be reduced by applying interval arithmetic, only at the cost of overestimation. The complexity remains quadratic.
- f_4 (Fig. 18 (**d**)): The variables x_3 and x_4 occur only once in the expression, the complexity can therefore be reduced from $\mathcal{O}((m+1)^4)$ to $\mathcal{O}((m+1)^2)$.

We can conclude the following: If one or more variables occur only once in the expression, we can take full advantage of this and reduce the complexity accordingly. At the same time, we do not have to compromise the time savings achieved with the methods described in the previous sections for constrained fuzzy numbers. The successive traversal of the function also permits to reduce the complexity of the problem during the execution phase of the computation. This is a very important additional feature compared to the method described in [23]. Consider, for example, the function

$$f(x_1, x_2, x_3) = (x_1 \cos(x_1) - x_3)/(x_2 \sin(x_2)). \tag{37}$$

Although both x_1 and x_2 occur more than once, the problem is only one-dimensional, since only the sub-functions $(x_1 \cos(x_1) \mapsto u_1)$ and $(x_2 \sin(x_2) \mapsto u_2)$ are evaluated using constrained fuzzy arrays. After that, u_1 and u_2 are reduced to simple fuzzy arrays, since both x_1 and x_2 do not occur in the remaining part of the expression.

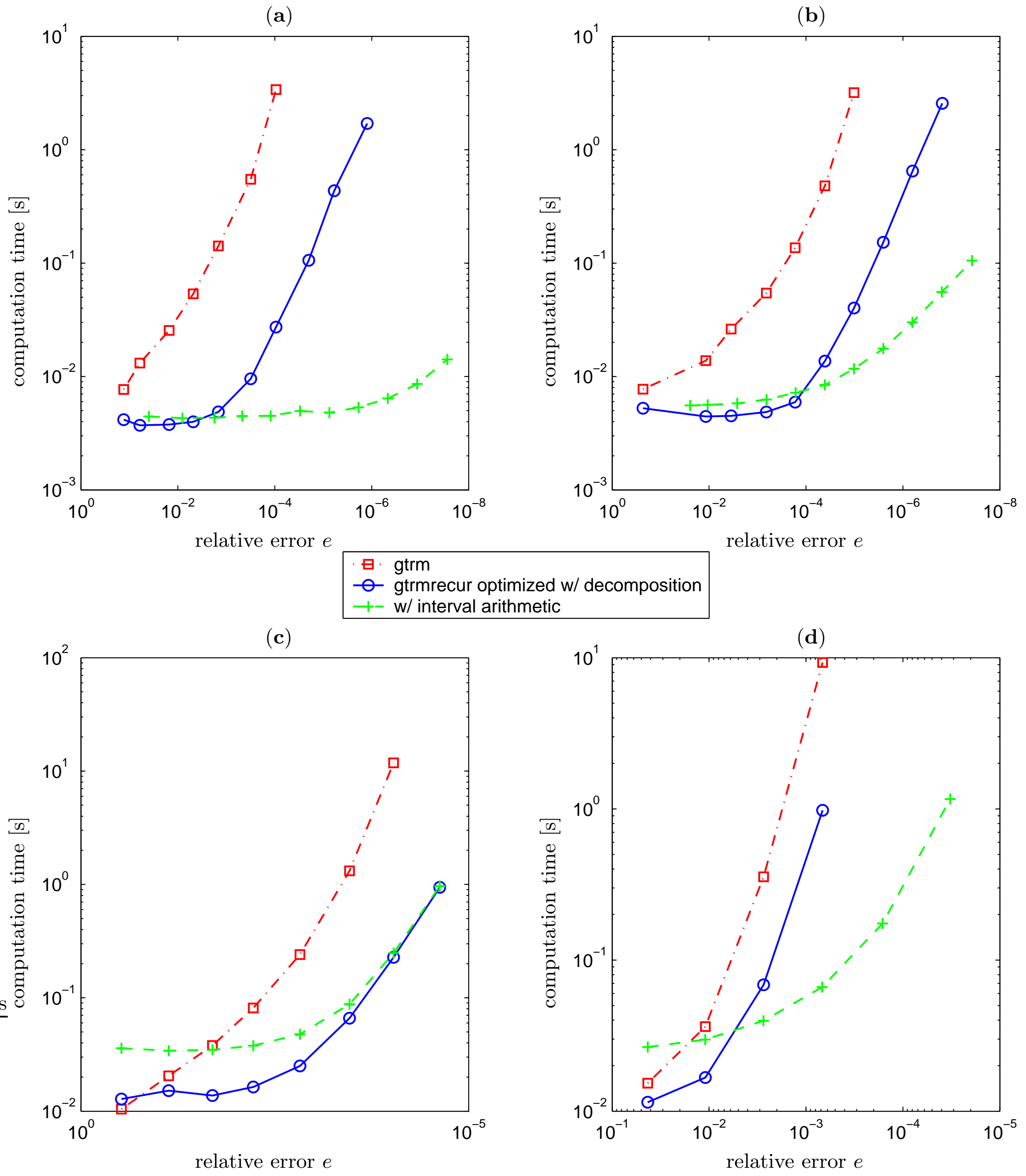


Figure 18: Computation time results with interval arithmetic

3.5.2 Testing for monotonicity using automatic differentiation

Note that all of the techniques presented in the previous sections did not take the actual values of the fuzzy parameters into account. In fact, the actual range of the parameters can play an important role in further reducing the computational effort.

For many real-world applications, the range of uncertainty of model input parameters will not be as large as the academic examples provided in this paper so far, but instead will be a small percentage of the peak value. Thus, it is quite likely that the model does not exhibit internal extrema, at least with respect to some of the fuzzy input parameters. While monotonic functions can be handled without overestimation by the reduced transformation method [14], the vertex method [7], or similar algorithms [23, 20], the case of only partly monotonic expressions is somewhat more difficult to handle. Particularly, it is often difficult or at least cumbersome to determine whether a model is monotonic with respect to particular input parameters or not. In the following, we present an approach based on automatic differentiation and interval arithmetic to detect monotonicity in a safe manner. This approach is successfully and commonly used in global optimization based on interval analysis [13, 16, 15]. A brief review of automatic differentiation along with other common methods of computational differentiation is given in Appendix A.4.

The following general theorem holds: A function $f(x_1, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$ is monotonic with respect to a variable x_i within a specified interval box $(I_1, \ldots, I_n), I_i \subset \mathcal{I}$ if the set image of its corresponding partial derivative $J_i = \partial f / \partial x_i (I_1, \ldots, I_n)$ does not change sign:

$$\begin{gathered} J_i \geq 0 \text{ or } J_i \leq 0, \text{ with} \\ J_i = \frac{\partial f}{\partial x_i}(I_1, \ldots, I_n) = \left\{ y = \frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) \mid x_1 \in I_1, \ldots, x_n \in I_n \right\}. \end{gathered} \tag{38}$$

This theorem still holds for the natural inclusion function $[\partial f / \partial x_i]$ of the partial derivative $\partial f / \partial x_i$, i.e. if we replace each occurrence of a real variable x_i by its interval counterpart, since

$$\partial f / \partial x_i (I_1, \ldots, I_n) \subset [\partial f / \partial x_i](I_1, \ldots, I_n). \tag{39}$$

Simply speaking, if the natural inclusion function of the partial derivative does not change sign, we have proven monotonicity.

Unfortunately, there is one drawback to this approach: By using the natural inclusion function of interval arithmetic, we accept the well-known dependency effect leading to an overestimation of the actual range of the derivative for multiple occurrences of variables in the objective function. This basically means that our monotonicity test based on interval arithmetic may fail even though the function is monotonic. We can improve the accuracy of the test by splitting the interval box, however, this will again result in an exponential growth of the complexity. We can sometimes also improve accuracy by using other inclusion functions based on Taylor expansions [16], which promise a reduced overestimation compared to natural inclusion functions especially for "smaller" intervals.

Nevertheless, despite the mentioned drawback, the test based on interval arithmetic will never suggest monotonicity when it is not actually present. This is a very important advantage compared to just computing the derivative at sampled points (it is also computationally much more efficient).

As an example, consider once again function f_2 from Table 3. We can compute its gradient vector to:

$$\nabla f_2(x_1, x_2) = \begin{pmatrix} \frac{-4(x_1-2)^3}{\left((x_1-2)^4+(x_2-2)^2+0.2\right)^2} \\ \frac{-2\,x_2+4}{\left((x_1-2)^4+(x_2-2)^2+0.2\right)^2} \end{pmatrix}.$$

For our original example using the uncertain parameters $\tilde{p}_1 = \langle 2.5, 2.5, 2.5\rangle_{\text{TFN}}$ and $\tilde{p}_2 = \langle 3, 2, 2\rangle_{\text{TFN}}$, we know that the function f_2 exhibits one local extremum. However, if we reduce the spread of the fuzzy numbers, e.g. if we assume $\tilde{p}_1 = \langle 2.5, 0.25, 0.25\rangle_{\text{TFN}}$ and $\tilde{p}_2 = \langle 3.0, 0.5, 0.5\rangle_{\text{TFN}}$, we can prove that the function is monotonic with respect to both parameters within the considered range. Obviously, we only need to test the interval of the α-cut at membership level $\mu = 0$, since all other α-cuts are subsets of it:

$$\begin{aligned}
\frac{\partial f_2}{\partial x_1}([2.25, 2.75], [2.5, 3.5]) &= \frac{-4\left([2.25, 2.75] - 2\right)^3}{\left(([2.25, 2.75] - 2)^4 + ([2.5, 3.5] - 2)^2 + 0.2\right)^2} \\
&= [-8.191, -0.008] \\
\frac{\partial f_2}{\partial x_2}([2.25, 2.75], [2.5, 3.5]) &= \frac{-2\,[2.5, 3.5] + 4}{\left(([2.25, 2.75] - 2)^4 + ([2.5, 3.5] - 2)^2 + 0.2\right)^2} \\
&= [-14.561, -0.131]
\end{aligned}$$

Both resulting intervals are < 0, i.e. do not contain the zero. Thus, we have shown that f is monotonic decreasing with respect to both parameters for the interval in question.

In the following, we describe our implementation of the described monotonicity test, and examine the computation time for the example functions from Table 3.

Implementation. Several packages provide the means for automatic differentiation in MATLAB, such as [21, 5, 22, 3]. Due to the straight-forward way operator overloading can be implemented in MATLAB, we decided to provide our own routines for the forward mode of automatic differentiation, which suffice our specific needs. As an additional feature, our routines have been designed to provide automatic differentiation not only for models of real and interval variables, but also for models of constrained fuzzy variables, which permits us to perform a kind of fuzzy sensitivity analysis based on automatic differentiation. A future paper will cover this topic.

Computation time. As numerical examples, we consider the functions f_2 and f_3 from Table 3. For low accuracies, the overall computational effort is higher because of the additional monotonicity check performed prior to the actual function evaluation with fuzzy arithmetic. If no monotonicity is detected, we can of course not expect performance gains, as illustrated in Figure 19 **(a,b)**. If monotonicity is present, however, such as for the fuzzy input parameters f_2: $\tilde{p}_1 = \langle 2.5, 0.25, 0.25\rangle$, $\tilde{p}_2 = \langle 3, 0.5, 0.5\rangle$, and f_3: $\tilde{p}_1 = \langle 0, 0.2, 0.2\rangle$, $\tilde{p}_2 = \langle 0, 0.2, 0.2\rangle$, the complexity is reduced significantly, as shown in Figure 19 **(c,d)**.

To conclude this section, we would like to emphasize that with increasing number of uncertain parameters, it becomes more important to take advantage of monotonicity properties

due to the exponential growth of complexity. For relatively small ranges of uncertainty, a monotonicity test based on automatic differentiation provides an efficient and reliable way to automatically determine partial (i.e. monotonicity with respect to some of the fuzzy input parameters) or complete monotonicity. Once monotonicity has been detected, our algorithms based on the transformation method can take full advantage of it. The arithmetic complexity becomes $\mathcal{O}((m+1)^{\max(n-p-q,1)})$, with n denoting the number of uncertain parameters, q the number of parameters q_i for which monotonicity is detected, and p the number of parameters p_j, $i \neq j$, occurring only once in the model function.

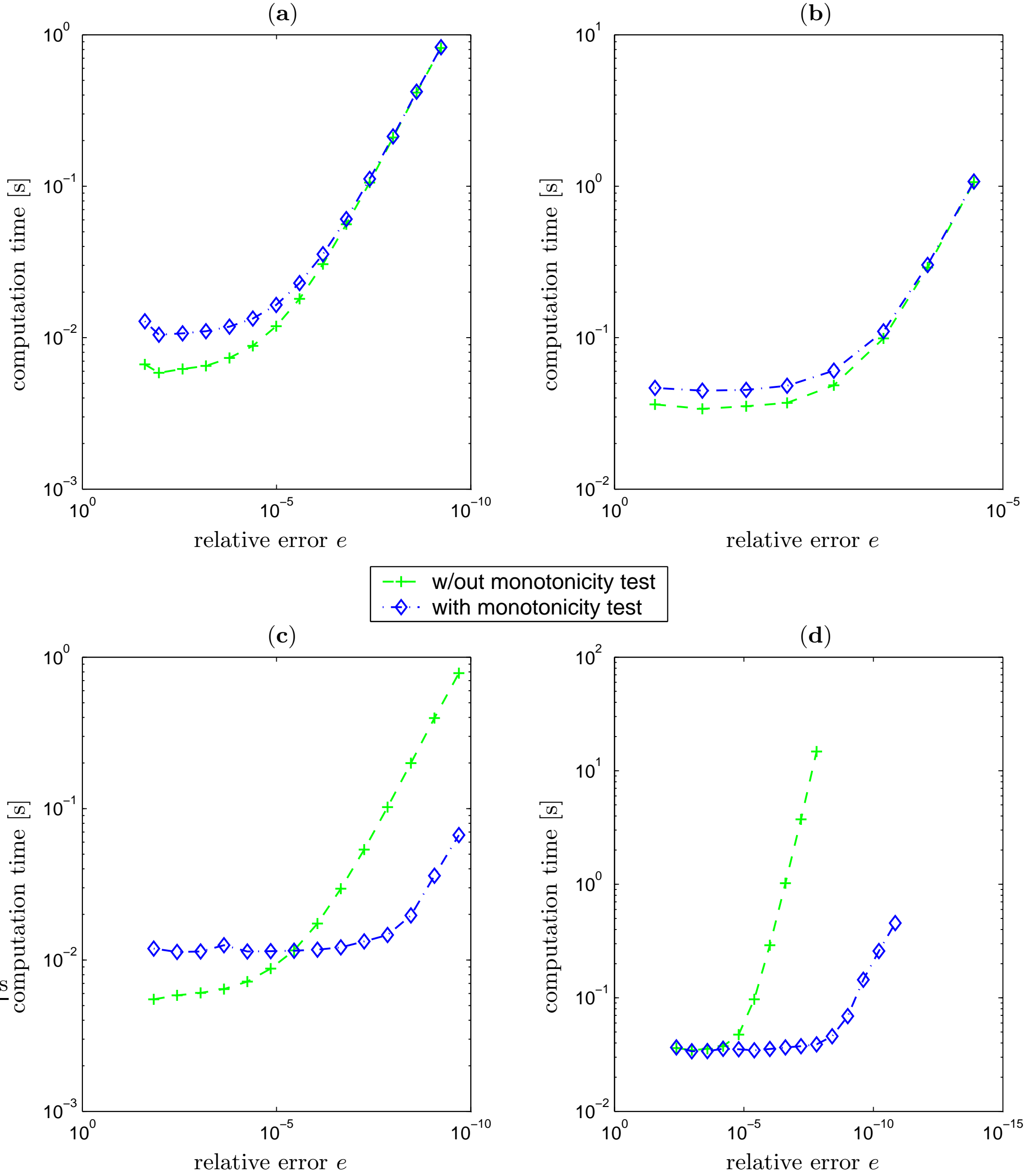


Figure 19: Computation time results for f_2 (**a,c**) and f_3 (**b,d**) with monotonicity check

4 Comparison to related work

Navara and Žabokrtský describe their approach to implement constrained fuzzy arithmetic to evaluate fuzzy-valued functions in [20]. The approach can be summarized as follows: The analytic expression is split into one ore more subexpressions which have mutually disjoint sets of variables. Each of the subexpressions is classified into one of four classes:

- *Simple fuzzy expressions*, which do not contain any fuzzy variable more than once.
- *Monotonic equality-constrained fuzzy expressions*, which may contain each variable several times, but are monotonic with respect to all fuzzy variables for the considered range of values.
- *Vertex equality-constrained fuzzy expressions*, which may contain each variable several times, but do not exhibit any internal extrema for the considered range of values.
- *Equality-constrained fuzzy expressions*, which are non-linear functions containing internal extrema for the considered range of values.

The evaluation of the sub-functions is performed by using an appropriate method, according to the classification, to compute a non-overestimating result. Thus, standard fuzzy arithmetic is applied to simple fuzzy expressions. Monotonic-equality-constrained fuzzy expressions can be computed by applying a basic theorem of interval arithmetic (the interval-valued result of a monotonic function can be easily obtained, see Appendix A.3). Vertex equality-constrained fuzzy expressions can be evaluated using the so-called *vertex algorithm*, which is actually identical to the Dong and Shah's vertex method [6] or the reduced transformation method [14] in case of fuzzy numbers. An iterative search algorithm or a symbolic method to locate internal extrema has to be used for equality-constrained fuzzy expressions. Finally, the sub-results are combined by standard fuzzy arithmetic to obtain the final result.

In the appendix of [20], an implementation of the *vertex algorithm* in Matlab is presented. In the following, we compare this implementation to our implementation of the transformation method and outline important differences with respect to the criteria of applicability/flexibility, accuracy, and computational complexity.

- *Applicability/flexibility*

  Navara's vertex algorithm: Operates on trapezoidal fuzzy intervals and triangular fuzzy numbers. The model function may contain an arbitrary number of the four elementary arithmetic operators $+, -, *, /$. The number of uncertain parameters is limited to six.

  Our implementation: Operates on any type of fuzzy number, but not on fuzzy intervals. Implemented are triangular and Gaussian-shaped fuzzy numbers, other types may be added easily due to the object-oriented design of our approach. The model function may contain an arbitrary number of elementary operators as well as elementary functions, such as $\exp, \cos, \mathrm{sqrt}, \mathrm{sqr}$. The number of uncertain parameters is theoretically unlimited, but limited to around six to eight uncertain parameters that occur more than once in an irreducible subexpression (depending on the main memory of the used computer). The number of uncertain parameters that occur just once in the expression

is unlimited. The number of uncertain parameters can be increased further for each parameter that the monotonicity test of the previous section is successfully applied for.

- *Accuracy*

  Navara's vertex algorithm: Arbitrarily accurate results can only be obtained for functions that do not contain internal extrema within the considered interval ranges, which is stated in [20].
  Our implementation: Arbitrarily accurate results can be achieved for all classes of functions, including functions containing internal extrema, by increasing the number of α-cuts.
  A comparison of the obtained results using Navara's vertex algorithm routine `'cfa'` with 10 and 100 α-cuts to the correct result as obtained with our implementation based on the general transformation method is shown in Fig. 20 for function f_2 of Table 3 using the triangular fuzzy input parameters $\tilde{p}_1 = \langle 2.5, 2.5, 2.5\rangle_{\text{TFN}}$ and $\tilde{p}_2 = \langle 3, 2, 2\rangle_{\text{TFN}}$. The vertex algorithm is, as expected, not applicable to function f_2 due to the internal extrema of the function within the considered range.

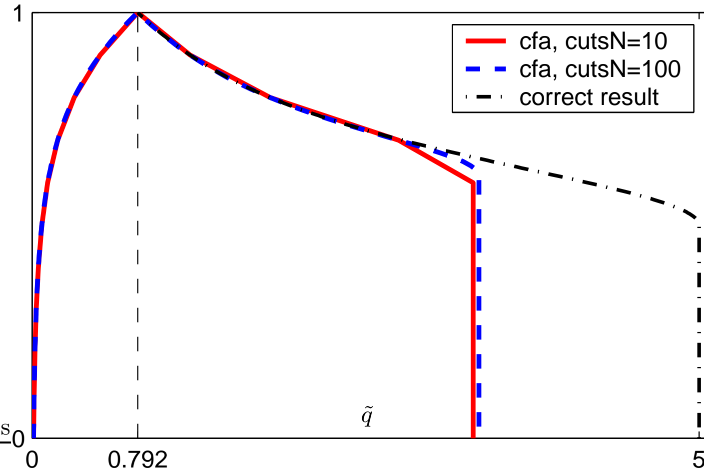


Figure 20: Vertex algorithm results for f_2

- *Computational complexity*

  Navara's vertex algorithm: For fuzzy numbers, the method is identical to the reduced transformation method, and the computational complexity is $\mathcal{O}((m+1)2^n)$, with $(m+1)$ denoting the number of α-cuts, and n denoting the number of fuzzy parameters (also see Table 4). For fuzzy operands with non-singleton cores, i.e. trapezoidal fuzzy numbers, an iterative search for internal extrema within the central core of the fuzzy number must be performed. This may represent a bottleneck of the implementation [20] depending on the applied search method.
  Our implementation: With the techniques described in Section 3.5, we achieve a complexity of $\mathcal{O}((m+1)^{\max(n-p,1)})$ for any type of analytic function, with p denoting the number of the n fuzzy parameters that occur just once in the expression. The complexity is further reduced automatically if monotonicity is detected with respect to q

input parameters, i.e. $\mathcal{O}((m+1)^{\max(n-p-q,1)})$. If we restrict the considered expressions to functions that do not exhibit any internal extrema for the considered range of values, we get $\mathcal{O}((m+1)2^{\max(n-p-q,1)})$ by applying the reduced transformation method. This complexity corresponds to the one obtained with the improved Level Interval Algorithm (LIA) [23] if $q=0$.

Conclusions. [20] describes three important techniques to efficiently implement constrained fuzzy arithmetic: (1) decomposition of the expression into subexpressions, (2) classification of the subexpressions, and (3) application of appropriate methods to compute the subexpressions with the lowest possible computational effort. Unfortunately, the provided system in MATLAB consist only of the vertex algorithm and does not put the suggested ideas into practice.
Our implementation successfully realizes decomposition, as well as a reduction of the dimensionality of the problem (see Section 3.5). Furthermore, by using three different classes of fuzzy data types combined with operator overloading techniques, we automatically apply standard fuzzy arithmetic whenever possible to lower the computational effort for simple fuzzy subexpressions. We can often take advantage of partial or complete monotonicity due to our monotonicity test based on automatic differentiation.

A Appendix

A.1 Parametric representation of fuzzy numbers

Dubois and Prade [8] describe a good parametric representation of fuzzy numbers that can be obtained by using two reference (or shape) functions $f\colon \mathbb{R}_0^+ \to [0,1]$ with the properties

1. f is monotonic decreasing
2. $f(0)=1$
3. $\forall\, x>0: f(x)<1$
4. $\forall\, x<1: f(x)>0$
5. $f(1)=0$ or $[\ \forall\, x: f(x)>0$ and $\lim_{x\to\infty} f(x)=0\ ]$.

A fuzzy number $\tilde{p}$ is called LR fuzzy number if two shape functions L (left) and R (right) with three parameters $\overline{m}\in\mathbb{R}$, $\alpha\in\mathbb{R}^+$, and $\beta\in\mathbb{R}^+$ exist such that

$$\forall\, x: \mu_{\tilde{p}}(x) = \begin{cases} L\left(\frac{\overline{m}-x}{\alpha}\right) & \text{if} \quad x<\overline{m} \\ 1 & \text{if} \quad x=\overline{m} \\ R\left(\frac{x-\overline{m}}{\beta}\right) & \text{if} \quad x>\overline{m} \end{cases} . \tag{40}$$

$\overline{m}$ is called the peak value (or modal value), α and β are called the left-hand spread and the right-hand spread. A short notation of an LR fuzzy number $\tilde{p}$ is then given by

$$\tilde{p} = \langle \overline{m}, \alpha, \beta \rangle_{\mathrm{LR}}. \tag{41}$$

Special cases of LR fuzzy numbers are

- *Triangular fuzzy numbers (TFN).* If $\tilde{p}$ is of triangular form, the shape functions are given to
$$L(u) = R(u) = \max(0, 1-u), \tag{42}$$
and the TFN can be denoted by the triplet
$$\tilde{p} = \langle \overline{m}, \alpha, \beta \rangle_{\text{TFN}}. \tag{43}$$
This triplet of type LR should not be confused with the definition of triangular fuzzy numbers according to Kaufmann and Gupta [17]. For $\alpha = \beta$, the TFN is symmetric, otherwise semi-symmetric (since $L(u) = R(u)$).

- *Gaussian fuzzy numbers (GFN).* Since uncertain parameters often carry an empirically determined uncertainty range, Gaussian shape functions are often considered a practical assumption. The Gaussian (or normal) density function [4, 11] of probability theory is
$$f(x) = C \ \exp\left(-\frac{1}{2}\frac{(\overline{m}-x)^2}{\sigma^2}\right) \tag{44}$$
with the constant scaling factor $C = 1/(\sigma\sqrt{2\pi})$ to satisfy the important property of a probability density function
$$\int_{-\infty}^{\infty} f(x)dx = 1. \tag{45}$$
The membership function of a GFN, however, can usually not satisfy this equality to unity, since we set $C = 1$ instead to meet the definition requirements of a fuzzy number, which demands that $\mu_{\tilde{p}}(\overline{m}) = 1$. In LR notation, the shape functions therefore become
$$L(u) = R(u) = \exp\left(-\frac{1}{2}u^2\right). \tag{46}$$
The left- and right-hand spread α and β of the so-defined membership function are equal to the distance σ of the points of inflection to the peak value $\overline{m}$. The GFN $\tilde{p}$ can be denoted
$$\tilde{p} = \langle \overline{m}, \sigma, \sigma \rangle_{\text{GFN}} = \langle \overline{m}, \sigma \rangle_{\text{GFN}}. \tag{47}$$
Note: In probability theory, σ represents the standard deviation of a random variable with the probability density function $f(x)$. Within the context of GFN's, σ is often still referred to as the standard deviation although the term no longer applies in a strict sense, since the definition of the variance and the standard deviation depend on the property Eq. 45.

- *Quasi-Gaussian fuzzy numbers (QGFN)* [14]. Gaussian fuzzy numbers do not have a compact (bounded) support. This is disadvantageous for the numerical treatment because a decomposition of the GFN in intervals results in an unbounded interval for membership level $\mu_{\tilde{p}}(x) = 0$. To avoid this difficulty, one can bound the support by letting
$$L(u) = R(u) = \begin{cases} \exp\left(-\frac{1}{2}u^2\right) & \text{if} \quad |\overline{m} - x| \leq r\sigma \\ 0 & \text{if} \quad |\overline{m} - x| > r\sigma, \end{cases} \tag{48}$$

with $r \in \mathbb{R}^+$, and usually $r \geq 3$. To uniquely define QGFN's, we use the parameter triplet

$$\tilde{p} = \langle \overline{m}, \sigma, r \rangle_{\text{QGFN}}. \tag{49}$$

Note: Unlike Gaussian fuzzy numbers, the membership functions of a QGFN is not continuous but only piecewise continuous (example see Fig. 21).

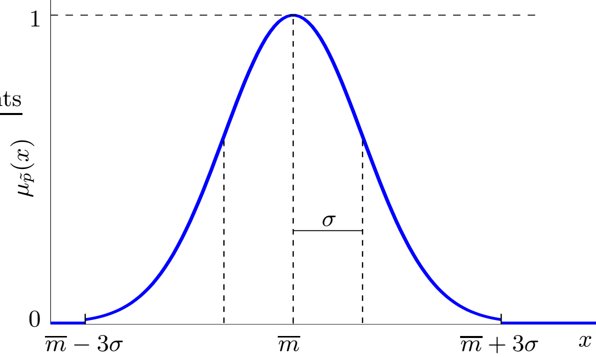


Figure 21: Quasi-Gaussian fuzzy number with $r = 3$.

A.2 Measuring the error of a fuzzy number

The output of the general transformation method is a discrete fuzzy number (i.e. it is obtained as a set of intervals). Furthermore, it is only an approximation of the exact solution. Therefore, the question arises on how to measure the error of the result.

In publications by Giachetti and Young [9, 10], error measures were given by comparing the approximation and the actual result separately for the left and the right segment of the fuzzy numbers for a given α-level $\alpha \in [0, 1]$. If we denote $\tilde{p}$ as the actual result and $\tilde{p}_*$ as the approximation with $[a^{(\alpha)}, b^{(\alpha)}]$ and $[a_*^{(\alpha)}, b_*^{(\alpha)}]$, respectively, as the intervals at a given α-level, we get the absolute error ϵ to

$$\epsilon_{\text{left}}^{(\alpha)} = |a^{(\alpha)} - a_*^{(\alpha)}| \quad \text{and} \quad \epsilon_{\text{right}}^{(\alpha)} = |b^{(\alpha)} - b_*^{(\alpha)}|. \tag{50}$$

This measure can also be interpreted as distance to the left $\Delta_{\text{left}}^{(\alpha)} = \epsilon_{\text{left}}^{(\alpha)}$ and distance to the right $\Delta_{right}^{(\alpha)} = \epsilon_{\text{right}}^{(\alpha)}$ of two intervals of confidence [17]. Adding both measures gives the distance $\Delta^{(\alpha)} = \Delta_{\text{left}}^{(\alpha)} + \Delta_{\text{right}}^{(\alpha)}$. By integrating over all membership levels $\alpha \in [0, 1]$, one can obtain the distance between two fuzzy numbers. In our case, we interpret the distance (not normalized as in [17]) as the absolute error

$$\epsilon = \int_{\alpha=0}^{1} \Delta^{(\alpha)} \, \mathrm{d}\alpha \tag{51}$$

$$= \int_{\alpha=0}^{1} \left(|a^{(\alpha)} - a_*^{(\alpha)}| + |b^{(\alpha)} - b_*^{(\alpha)}| \right) \, \mathrm{d}\alpha. \tag{52}$$

A more meaningful measure is the relative error, since the magnitude of the error has to be related to the magnitude and spread of the fuzzy number. We divide by the area of the fuzzy number to obtain the relative error e

$$e = \epsilon / \text{area}(\tilde{p}) \tag{53}$$

with

$$\text{area}(\tilde{p}) = \int_{\alpha=0}^{1} (b^{(\alpha)} - a^{(\alpha)}) \, \mathrm{d}\alpha. \tag{54}$$

$\tilde{p}$ must not the be crisp, since the area of a crisp number is zero.

The error e has the following properties:

1. $e \geq 0$
2. $(\tilde{p}_* = \tilde{p}) \rightarrow e = 0$
3. If $\tilde{p}_*$ is a crisp number with the real value p, and the peak value of $\tilde{p}$ is equal to p, $e = 1$.

For fuzzy numbers in discrete representation, we can obtain a continuous membership function through piecewise linear interpolation. To compute the error measures, we can use the trapezoidal integration rule.

A.3 Interval analysis basics

In the following, a short summary of some basic definitions of interval arithmetic and interval analysis are given. For a detailed introduction to interval analysis, see [19, 13, 16].

Arithmetic of closed intervals
Let I be a continuous subset of $\mathbb{R}$ with closed bounds. In standard set notation, I can be written as

$$I = \{x \in \mathbb{R} \mid -\infty < a \leq x \leq b < \infty\}. \tag{55}$$

I is then called real interval or interval number, and is denoted by the ordered pair of real numbers $[a, b]$. The set of closed real intervals is denoted by $\mathcal{I}$ in the following. Degenerate or punctual intervals of the form $[a, a]$ are equivalent to real numbers.

The set operations and symbols intersection ($\cap$), union ($\cup$), Cartesian product ($\times$), inclusion ($\subset$), etc. maintain the usual properties of set theory when applied to intervals.

To perform arithmetic operations with two intervals $I = [a, b]$ and $J = [c, d]$, the arithmetic of numbers can be extended. In set notation, one obtains for the operator $\diamond$ being one of the elementary operators $+, -, *, /$:

$$I \diamond J = \{x \diamond y \mid x \in [a, b], y \in [c, d]\}. \tag{56}$$

The elementary operations can also be expressed as operations on their bounds:

$$\begin{aligned}
[a,b]+[c,d] &= [a+b, c+d], \\
[a,b]-[c,d] &= [a-d, b-c], \\
\alpha[a,b] &= [\alpha a, \alpha b] \quad \text{if} \quad a \geq 0, \\
&= [\alpha b, \alpha a] \quad \text{if} \quad a < 0, \\
[a,b]*[c,d] &= [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}], \\
1/[a,b] &= [1/b, 1/a] \quad \text{if} \quad 0 \notin [a,b].
\end{aligned} \tag{57}$$

The dependency effect

When performing an interval computation where a given variable occurs more than once, the resulting interval is widened. This can be illustrated with the following examples. Let $I = [2,4]$ be a closed interval. Then, with the above arithmetic rules, we obtain

$$I - I = [2,4] - [2,4] = [-2,2] \quad \text{and} \tag{58}$$

$$I/I = [2,4] * [1/4, 1/2] = [1/2, 2]. \tag{59}$$

The actual sharp result, however, should be $[0,0]$ for Eq. 58 and $[1,1]$ for Eq. 59. This overestimation, or “pessimism”, occurs since instead of

$$I \diamond I = \{x \diamond x \mid x \in [a,b]\}, \tag{60}$$

the interval arithmetic computation gives

$$I \diamond I = \{x \diamond y \mid x \in [a,b], y \in [a,b]\}. \tag{61}$$

In the case of the n-th positive integer power of an interval $I = [a,b]$, the dependency effect can be avoided. Instead of computing

$$I^n = \prod_{i=1}^{n} [a,b], \tag{62}$$

one can directly define I^n by

$$\begin{aligned}
I^n &= [a^n, b^n] && \text{if } a \geq 0 \text{ or if } a \leq 0 \leq b \text{ and } n \text{ is odd}, \\
&= [b^n, a^n] && \text{if } b \leq 0, \\
&= [0, \max\{a^n, b^n\}] && \text{if } a \leq 0 \leq b \text{ and } n \text{ is odd}
\end{aligned} \tag{63}$$

to obtain the actual sharp interval bounds.

While addition and multiplication are associative and commutative in interval arithmetic, multiplication is not distributive with respect to addition. This is caused by the dependency effect. Instead, only a weaker form of distributivity called sub-distributivity proves valid:

$$I * (J + K) \subset I * J + I * K. \tag{64}$$

Elementary interval functions

The elementary mathematical functions of real numbers, e.g. exp, sqrt, sin, cos, log, can be extended to interval arithmetic and expressed in terms of bounds. This is especially straightforward for monotonic functions, where only the bounds have to be considered to obtain the sharp interval bounds, for example:

$$\exp([a,b]) = [\exp(a), \exp(b)]. \tag{65}$$

For non-monotonic functions, the internal extrema have to be considered to obtain the correct bounds. This does not pose a problem, since algorithms can be written that rapidly check for internal extrema. For an algorithm of the sin function, see [16].

Inclusion functions

Consider the real-valued function

$$f: \begin{array}{ll} I_1 \times \ldots \times I_n & \to J \\ (x_1, \ldots, x_n) & \mapsto y. \end{array} \quad \text{with } I_1, \ldots, I_n \in \mathcal{I}, J \subset \mathbb{R}$$

Its image set is then given by

$$J = f(I_1, \ldots, I_n) = \{y = f(x_1, \ldots, x_n) \mid x_1 \in I_1, \ldots, x_n \in I_n\}. \tag{67}$$

The interval function $[f] : \mathcal{I}^n \to \mathcal{I}$ is an inclusion function of f if

$$\forall\, I_1, \ldots I_n \in \mathcal{I} : \; f(I_1, \ldots, I_n) \subset [f](I_1, \ldots, I_n). \tag{68}$$

Natural inclusion functions

To compute the range of an interval-valued function, one can extend a real-valued function f of real variables $x_1, \ldots, x_n$ to an interval function $[f]$ of interval variables $I_1, \ldots, I_n$ by replacing each real variable and operator by its interval counterparts. In other words, $[f](I_1, \ldots, I_n)$ has the same formal expression as $f(x_1, \ldots, x_n)$, but considers intervals instead of numbers. This extended function $[f]$ is called the natural inclusion function of f with the important property of an inclusion function (see Eq. 68).

Usually, the natural inclusion function gives a pessimistic range of the actual image set of the real-valued function. There is one particularly important exception. If f involves only continuous operators and continuous elementary functions and if each of the n variables occurs only once in the expression, then

$$[f](I_1, \ldots, I_n) = f(I_1, \ldots, I_n). \tag{69}$$

To illustrate this, consider the following examples:

$$f_1(x_1, x_2) = (x_1 + x_2)/(x_1 * x_2), \tag{70}$$

$$f_2(x_1, x_2) = 1/x_1 + 1/x_2, \tag{71}$$

$$f_3(x) = (\text{sign}(x))^2. \tag{72}$$

Evaluating their natural inclusion functions for $I_1 = [1, 2]$, $I_2 = [3, 4]$ and $I = [-2, 2]$ gives:

$$[f_1](I_1, I_2) = (I_1 + I_2)/(I_1 * I_2) = [1/2, 2] \supset f_1(I_1, I_2), \tag{73}$$

$$[f_2](I_1, I_2) = 1/I_1 + 1/I_2 = [3/4, 4/3] = f_2(I_1, I_2), \tag{74}$$

$$[f_3](I) = (\text{sign}(I))^2 = [0, 1] \supset f_3(I). \tag{75}$$

Interpretation of the results: Eq. 73 overestimates the actual range due to multiple occurrences of the variables. Eq. 74 gives the exact interval bounds of the image set of f_2 for the specified intervals, since each variable occurs at most once and the involved operators are continuous. Note that f_2 and f_1 are the same function in different forms. In Eq. 75, the variable I occurs only once. However, the sign function is not continuous and the result $[0, 1]$ pessimistic, since $f_3([-2, 2]) = 1$ (see [16]).

A.4 Computation of derivatives

Computing derivatives of a differentiable function $f(x_1, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$ can be performed in three different ways. Each of the methods has distinct advantages/disadvantages, which are briefly mentioned.

A.4.1 Symbolic differentiation and evaluation

Either by hand or with computer algebra systems, we can create a symbolic representation of the gradient vector $\mathbf{g}$ of a multi-variate expression by systematic application of the chain rule of differential calculus.

$$\mathbf{g} = \nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^T . \tag{76}$$

This gradient vector can then be evaluated for any point $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$ in a second step:

$$\mathbf{g}(\mathbf{x}^{(0)}) = \nabla f(\mathbf{x}^{(0)}) \tag{77}$$

$$= \left(\frac{\partial f}{\partial x_1}(x_1^{(0)}, \ldots, x_n^{(0)}), \ldots, \frac{\partial f}{\partial x_n}(x_1^{(0)}, \ldots, x_n^{(0)}) \right)^T . \tag{78}$$

Advantage: very accurate. Disadvantage: symbolic representation must be stored in memory.

A.4.2 Difference quotients

In this widely used numerical approach, the derivative is replaced by a finite difference quotient. Most commonly used are the forward, the backward, and the central difference quotient. For example, by using central differences, we can write for each component i of Eq. 78

$$\frac{\partial f}{\partial x_i}(x_1^{(0)}, \ldots, x_n^{(0)}) = \tag{79}$$

$$\frac{f(x_1^{(0)}, \ldots, x_i^{(0)} + h, \ldots, x_n^{(0)}) - f(x_1^{(0)}, \ldots, x_i^{(0)} - h, \ldots, x_n^{(0)})}{2h} + \mathcal{O}(h^2),$$

and thus obtain an approximation of the gradient vector at any point $\mathbf{x}^{(0)}$ with two function evaluations only and without the need to explicitly compute any derivative.

Advantage: no computation of derivatives is required. Disadvantage: only limited accuracy (h cannot be chosen arbitrarily small due to the obvious problem with the cancellation error

when computing the difference of two close values with limited precision arithmetic. Griewank writes in [12] that only about 1/2 to 2/3 of the significant digits will be correct even if h is optimally chosen).

A.4.3 Algorithmic (automatic) differentiation (AD)

Similar to symbolic differentiation, AD consecutively applies the chain rule to compute accurate derivatives. However, the chain rule is not applied to symbolic expressions, but directly to actual numerical values. In the following, we describe the so-called *forward mode*, which is the most common variant of AD. More advanced topics of AD include the *reverse mode*, methods to compute second- and higher order derivatives, algorithms to exploit sparsity patterns of Jacobian matrices, and treatment of non-differentiability (see [12]).

The computational evaluation of an expression can be interpreted as a sequence of binary (e.g. $+, -, *$) and unary (e.g. $\sin, \exp$) operations and subsequent assignments to intermediate variables (this is illustrated later in a brief example). To obtain the derivative, we can apply the chain rule to each of these elementary operations along with the regular evaluation of the expression.

According to the chain rule, the derivative with respect to a variable x of a binary operation $f(a(x), b(x)) = a(x) \diamond b(x)$ is given by

$$\frac{df}{dx} = \frac{da}{dx}\frac{\partial f}{\partial a} + \frac{db}{dx}\frac{\partial f}{\partial b}. \tag{80}$$

A unary operation $f(a(x))$ yields

$$\frac{df}{dx} = \frac{da}{dx}\frac{\partial f}{\partial a}. \tag{81}$$

With these two simple rules, we can compute the gradient vector or Jacobian matrix of any differentiable function that is comprised of an arbitrary number of such elementary operations. We can easily extend the procedure to the multi-variate case, since the differentiation with respect to a specific variable allows to treat the remaining ones as constants.

To illustrate the procedure, let us consider the following example for a function $f : \mathbb{R} \to \mathbb{R}$ of one variable x at $x = 2$:

$$y = f(x) = \frac{\exp(x) + x^2}{x}$$

Before computing the derivative at $x = 2$, let us take a look at the *evaluation trace* [12] of the function. “An evaluation trace is a record of a particular run of a particular program, with particular specified values for the input variables, showing the sequence of floating point values calculated by a [...] processor and the operations that computed them.” (Griewank).

v_0	$= x$		$= 2.0000$
v_1	$= \exp(v_0)$	$= \exp(2.0000)$	$= 7.3891$
v_2	$= \mathrm{sqr}(v_0)$	$= \mathrm{sqr}(2.0000)$	$= 4.0000$
v_3	$= v_1 + v_2$	$= 7.3891 + 4.0000$	$= 11.3891$
v_4	$= v_3/v_0$	$= 11.3981/2.0000$	$= 5.6945$
y	$= v_4$		$= 5.5945$

We now show how the computation of the derivative $\dot{y} = \frac{df}{dx}$ is performed using the forward mode of AD. The chain rule is applied to each line of the evaluation trace:

v_0	$= x$		$= 2.0000$
$\dot{v}_0$	$= \frac{x}{dx}$		$= 1.0000$
v_1	$= \exp(v_0)$	$= \exp(2.0000)$	$= 7.3891$
$\dot{v}_1$	$= \dot{v}_0 * v_1$	$= 1.0000 * 7.3891$	$= 7.3891$
v_2	$= \mathrm{sqr}(v_0)$	$= \mathrm{sqr}(2.0000)$	$= 4.0000$
$\dot{v}_2$	$= \dot{v}_0 * 2 * v_0$	$= 1.0000 * 2.0000 * 2.0000$	$= 4.0000$
v_3	$= v_1 + v_2$	$= 7.3891 + 4.0000$	$= 11.3891$
$\dot{v}_3$	$= \dot{v}_1 + \dot{v}_2$	$= 7.3891 + 4.0000$	$= 11.3891$
v_4	$= v_3/v_1$	$= 11.3981/2.0000$	$= 5.6945$
$\dot{v}_4$	$= (\dot{v}_3 - v_4 * \dot{v}_0)/v_0$	$= (11.3891 - 5.6945 * 1.0000)/2.0000$	$= 2.8473$
y	$= v_4$		$= 5.6945$
$\dot{y}$	$= \dot{v}_4$		$= 2.8473$

Implementation of the forward mode can be done either through operator overloading, or through code transformation.

Advantages of AD compared to symbolic differentiation: the symbolic expressions of the gradient vector ∇f need not to be stored. Common subexpressions can be automatically used repeatedly for evaluating ∇f. Compared to difference quotients, AD offers higher accuracy.

References

[1] G. Alefeld and G. Mayer. Interval analysis: theory and applications. *J. Comput. Appl. Math.*, 121(1-2):421–464, 2000.

[2] A. M. Anile, S. Deodato, and G. Privitera. Implementing fuzzy arithmetic. *Fuzzy Sets and Systems*, 72:239–250, 1995.

[3] C. Bischof, B. Lang, and A. Vehreschild. Automatic differentiation for Matlab programs. *Preprint RWTH-CS-02-09, Aachen University, Germany*, 2002.

[4] I. F. Blake. *An introduction to applied probability theory.* John Wiley & Sons, New York-Chichester-Brisbane, 1979.

[5] T. Coleman and A. Verma. ADMAT: An automatic differentiation toolbox for Matlab. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, Philadelphia, PA, 1998. SIAM.

[6] W. Dong and H. C. Shah. Vertex method for computing functions of fuzzy variables. *Fuzzy Sets and Systems*, 24:65–78, 1987.

[7] W. M. Dong and F. S. Wong. Fuzzy weighted averages and implementation of the extension principle. *Fuzzy Sets and Systems*, 21:183–199, 1987.

[8] D. Dubois and H. Prade. *Fuzzy Sets and Systems, Theory and Applications*, volume 144 of *Mathematics in Science and Engineering.* Academic Press, Inc., New York, 1980.

[9] R. E. Giachetti and R. E. Young. Analysis of the error in the standard approximation used for multiplication of triangular and trapezoidal fuzzy numbers and the development of a new approximation. *Fuzzy Sets and Systems*, 91(1):1–13, 1997.

[10] R. E. Giachetti and R. E. Young. A parametric representation of fuzzy numbers and their arithmetic operators. *Fuzzy Sets and Systems*, 91(2):185–202, 1997.

[11] B. V. Gnedenko. *Theory of Probability*. Taylor & Francis, London, Great Britain, 1998.

[12] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, PA, 2000.

[13] E. H. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, NY, 1992.

[14] M. Hanss. The transformation method for the simulation and analysis of systems with uncertain parameters. *Fuzzy Sets and Systems*, 130(3):277–289, 2002.

[15] J. Heeks. *Charakterisierung unsicherer Systeme mit intervallarithmetischen Methoden*, volume 919 of *Reihe 8*. VDI Verlag, Düsseldorf, Germany, 2002.

[16] L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis*. Springer, London, Great Britain, 2001.

[17] A. Kaufmann and M. M. Gupta. *Introduction to Fuzzy Arithmetic*. Van Nostrand Reinhold Co., New York, 1991.

[18] G. J. Klir. Fuzzy arithmetic with requisite constraints. *Fuzzy Sets and Systems*, 91:165–175, 1997.

[19] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[20] M. Navara and Z. Žabokrtský. How to make constrained fuzzy arithmetic efficient. *Soft Computing*, 6:412–417, 2001.

[21] L. C. Rich and D. R. Hill. Automatic differentiation in Matlab. *Applied Numerical Mathematics*, 9:33–43, 1992.

[22] S. M. Rump. *Developments in Reliable Computing*, chapter INTLAB – INTerval LABoratory, pages 77–104. Kluwer, Dortrecht, Netherlands, 1999.

[23] K. L. Wood, K. N. Otto, and E. K. Antonsson. Engineering design calculations with fuzzy parameters. *Fuzzy Sets and Systems*, 52:1–20, 1992.

[24] H. Q. Yang, H. Yao, and J. D. Jones. Calculating functions of fuzzy numbers. *Fuzzy Sets and Systems*, 55:273–283, 1993.

[25] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

Andreas Klimke

Institute of Applied Analysis and Numerical Simulation
University of Stuttgart
Pfaffenwaldring 57
70569 Stuttgart, Germany

E-Mail: `klimke@ians.uni-stuttgart.de`

Erschienene Preprints ab Nummer 2003/001

Komplette Liste: `http://preprints.ians.uni-stuttgart.de`

2003/001 *Lamichhane, B. P., Wohlmuth, B. I.*: Mortar Finite Elements for Interface Problems.

2003/002 *Dryja, M., Gantner, A., Widlund, O. B., Wohlmuth, B. I.*: Multilevel Additive Schwarz Preconditioner For Nonconforming Mortar Finite Element Methods.

2003/003 *Klimke, A., Hanss, M.*: On the Reliability of the Influence Measure in the Transformation Method of Fuzzy Arithmetic.

2003/004 *Klimke, A.*: RANDEXPR: A Random Symbolic Expression Generator.

2003/005 *Klimke, A.*: How to Access Matlab from Java.

2003/006 *Merkle, T.*: Phase separation in solid mixtures under elastic loadings with application to solder materials.

2003/007 *Lamichhane, B. P., Wohlmuth, B. I.*: Second Order Lagrange Multiplier Spaces for Mortar Finite Elements in 3D.

2003/008 *Fritz, A., Hüeber, S., Wohlmuth, B. I.*: A comparison of mortar and Nitsche techniques for linear elasticity.

2003/009 *Klimke, A.*: An Efficient Implementation of the Transformation Method of Fuzzy Arithmetic