SAL 2000 - 4

Fast Forward Automatic Differentiation Library (FFADLib)

A User Manual

I. Tsukanov, M. Hall

FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY (FFADLIB) *

A User Manual

I.Tsukanov † **M.Hall**

Spatial Automation Laboratory
University of Wisconsin-Madison
1513 University Avenue
Madison WI 53706-1572, U.S.A.

Abstract

In this document we discuss the data structure and algorithms for direct application of recursive chain rules to numerical computations of partial derivatives in forward mode. The proposed data structure providing constant time access to the partial derivatives accelerates the automatic differentiation computations. We implemented the presented algorithms in a software package, which simplifies automatic differentiation of functions represented by a computer program. The library is available for public use and can be downloaded from http://sal-cnc.me.wisc.edu. This manual contains detailed descriptions of all library functions and several examples illustrating the applications and usage of the proposed automatic differentiation software.

† Corresponding author, e-mail: igor@sal-cnc.me.wisc.edu

Contents

1 Introduction

Many engineering applications require computation of the partial derivatives of some functions. For example, the solution algorithm of optimization problems needs the gradient of a goal function; engineering analysis problems use second and sometimes higher order derivatives. Often, once the problem is solved, other important quantities have to be determined. Usually they are defined by expressions containing partial derivatives of the function for which the problem is solved. For instance, structural mechanics problems are formulated and solved for the displacement, but stresses are defined as its partial derivatives.

Unfortunately, numerical methods of computing derivatives based on finite difference schemes are insufficiently accurate, especially for higher-order derivatives. The situation may be fixed by manual coding the partial derivatives of the functions, but this is possible for relatively simple functions only.

Along with the differentiation based on the finite difference scheme, there are two other major differentiation techniques: symbolic and automatic differentiation. The symbolic differentiation methods aim at the derivation of the expressions for the derivatives. However, the symbolic differentiation often results in very large and inefficient formulas for the derivatives. These formulas may contain many common subexpressions that have to be evaluated every time they appear in the expression for the derivative.

Automatic differentiation is a broad field which incorporates the methods for computing the derivatives of the functions represented by computer programs. The automatic differentiation, in contrast with symbolic differentiation, propagates *numerical values* of derivatives rather than symbolic expressions.

Systems like ADIFOR [4], ADIC [5], TAMC [10], DAFOR [2] transform the source code of the program, inserting additional code for the partial derivatives. A different idea is used by the ADOL-C package [11]. It does not require any preprocessing of the source code; rather it overloads the arithmetic operators and elementary functions. Before the differentiation begins, the package records on the "tape" the computational graphs for each function to be differentiated and issues a "tag" — the specific number which designates the position of the computational graph for the particular function on the "tape". Then the package analyzing the "tape" computes numerical values for the derivatives.

This document describes the data structure and automatic differentiation algorithms that are highly optimized for computation of the derivatives in the forward mode — from independent to dependent variables. Our automatic differentiation approach is based on the notion of a *differential tuple* [1] — a specialized data structure that holds values of (1) partial derivatives up to the specified order, (2) binomial coefficients, and (3) index arrays. Differentiation of a composite function is performed in the forward mode: first differential tuples for independent variables are generated, then they are propagated through the computational graph of the function. At each node of the graph, a chain rule is applied to the inputted tuples and the resulting tuple is sent to the next node.

The automatic differentiation algorithms are implemented through overloaded elementary functions (exp, pow, sin, cos, etc.), arithmetic operations (+, -, *, /), and various utility functions including differentiation operators and a function to generate differential tuples corresponding to independent variables. Overloading, in conjunction with our automatic differentiation technique, allows us to make differentiation easy: the partial derivatives of a function are generated as a tuple by simply writing the function in the usual notation (ex. f = pow(cos(x), n)). Moreover we do not need to explicitly analyze the computational graph of the function being differentiated: it is constructed by a compiler and the intermediate differential tuple contains the "history" of the previous computations.

Our automatic differentiation technique utilizes the generalized Leibnitz rules, which were derived for many independent variables by Manko [21, 22], Shevchenko [31, 32]. The same formulas were obtained independently by Neidinger in [14]. The beauty of the generalized Leibnitz chain rules is the fact that these rules give the *exact* representation of derivatives by combining derivatives of the arguments with binomial coefficients.

Recently another approach to automatic differentiation has found broad acceptance [18, 15, 3]. It turns out that the derivatives can be also computed automatically using Taylor series. Indeed, the Taylor coefficients are the scaled partial derivatives. The automatic differentiation algorithms dealing with Taylor coefficients work faster because these algorithms do not utilize the binomial coefficients which means fewer multiplication operations are performed.

1.1 Outline

The rest of the document is organized as follows: Section 2 describes the mathematical basics of our automatic differentiation approach and explains the data structure and automatic differentiation algorithms; in Section 3 we

[1] The notion of differential tuple was introduced by Shevchenko in [31], however Neidinger calls the similar data structure a "PYRAMID" [14].

analyze the proposed data structure and algorithms; Section 4 discusses the usage of the functions from the Fast Forward Automatic Differentiation Library (FFADLib); Section 5 contains examples of the utilization of the library.

Part I
Data Structure and Algorithms for Fast Forward Automatic Differentiation

2 Automatic Differentiation: Rules, Data Structure and Algorithms

In this Section we show the differentiation rules of the arithmetic operations and the elementary functions. Then we discuss the data structure and automatic differentiation algorithms. The functions discussed in the document are scalar functions having the same number of independent variables ($\mathbb{R}^n \rightarrow \mathbb{R}$).

2.1 Automatic Differentiation Rules

The simplest automatic differentiation rule is the rule for the differentiation of sum or difference. Let $f(\mathbf{x}) = u(\mathbf{x}) + v(\mathbf{x})$, since differentiation is a linear operator, any partial derivative of the function f is a sum of partial derivatives of u and v:

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} + \frac{\partial^{|\mu|} v}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}}. \tag{1}$$

In this expression, the multi-index μ defines the order of the partial derivatives and their position in the data structure.

The differentiation rule for a product is more complex. A partial derivative of a product of two functions $f(\mathbf{x}) = u(\mathbf{x})v(\mathbf{x})$ is a linear combination of products of partial derivatives of u and v with binomial coefficients:

$$\begin{aligned} \frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \Bigg(& \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \cdots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} v}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \Bigg), \end{aligned} \tag{2}$$

where $|\mu| = \sum_{i=1}^{n} \mu_i$, $0 < |\mu| \leq m$ (m is the order of the highest derivative), $|\alpha| = \sum_{i=1}^{n} \alpha_i$. The formula (2) utilizes the additional multi-index α whose elements serve as counters of summation loops. In contrast to the differentiation of a sum, the Leibnitz rule (2) combines partial derivatives of different orders. Moreover, each term in (2) is multiplied by n binomial coefficients $\binom{\mu_i}{\alpha_i}$, $i = 1, \dots, n$. From a computational point of view, the differentiation of the product includes both the computation of positions of partial derivatives given by multi-indices α and $\mu - \alpha$, as well as the computation of the products of binomial coefficients $\prod_{i=1}^{n} \binom{\mu_i}{\alpha_i}$.

The rules for differentiation of all elementary functions are significantly different from those previously discussed. As an example we show here the differentiation chain rule of a power function $f(\mathbf{x}) = u^s(\mathbf{x})$ with constant s:

$$\begin{aligned} \frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \Bigg[& s * f * \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} + \\ & \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p-1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \cdots \binom{\mu_p - 1}{\alpha_p} \cdots \binom{\mu_n}{\alpha_n} \\ & \Bigg(s * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} u}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} - \\ & \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} f}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \Bigg) \Bigg] u^{-1}, \end{aligned} \tag{3}$$

where $|\alpha| > 0$. Most importantly, it is recursively defined and the derivatives of f must be computed in a specific order. In particular, lower order derivatives are used for computations of higher order derivatives, so we must propagate derivatives upward in order. This means that multi-index α has to start with zero and grow coordinate-wise up to upper bounds in the summation loops. Another difference is that the highest index of one summation loop is one less than the value of the corresponding element μ_p of the multi-index μ and the pth binomial coefficient is $\binom{\mu_p - 1}{\alpha_p}$. Virtually, any p, which corresponds to the positive element of multi-index μ, may be picked. However, the proper choice of p can reduce the number of elements in the series (3). We will address the issue of the optimal choice of p later in the Section 2.4. Throughout the document, we define the multi-index $(\mu_1, \ldots, \mu_p - 1, \ldots, \mu_n)$ as $\mu - 1$.

In conclusion of this Section let us introduce a differentiation rule for differentiation operator $\frac{\partial}{\partial x_k}$, where $0 \leq k \leq n$. For example, let function f be a first partial derivative of another function u: $f = \frac{\partial u}{\partial x_k}$. The differentiation chain rule for f in this case simply follows from the recursive definition of higher order derivatives:

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \ldots \partial x_n^{\mu_n}} = \frac{\partial^{|\mu|}}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \ldots \partial x_n^{\mu_n}} \left(\frac{\partial u}{\partial x_k} \right) = \frac{\partial^{|\mu+1|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \ldots \partial x_k^{\mu_k + 1} \ldots \partial x_n^{\mu_n}}. \tag{4}$$

This rule shows that all partial derivatives of f are, in fact, partial derivatives of u with one order higher for the chosen independent variable. For derivations of the formulas for the rest of elementary functions, the reader is referred to [19, 14].

2.2 The Taylor Coefficients

Recently another approach to automatic differentiation has gained popularity [18, 15, 3]. This approach is based on the Taylor series expansion of a function $f(x_1, x_2, \ldots x_n)$:

$$f \approx \sum_{\alpha=0}^{\mu} \frac{1}{\alpha_1! \alpha_2! \ldots \alpha_n!} \frac{\partial^{|\alpha|} f(x_1^0, x_2^0, \ldots x_n^0)}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}} (x_1 - x_1^0)^{\alpha_1} (x_2 - x_2^0)^{\alpha_2} \ldots (x_n - x_n^0)^{\alpha_n}, \tag{5}$$

here $(x_1^0, x_2^0, \ldots x_n^0)$ is the point in whose vicinity the function is represented by the Taylor series. This is actually the point where the partial derivatives $\frac{\partial^{|\alpha|} f(x_1^0, x_2^0, \ldots x_n^0)}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}}$ are computed. The product of a partial derivative $\frac{\partial^{|\alpha|} f(x_1^0, x_2^0, \ldots x_n^0)}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}}$ with the coefficient $\frac{1}{\alpha_1! \alpha_2! \ldots \alpha_n!}$ is called Taylor coefficient:

$$F_\alpha = \frac{1}{\alpha_1! \alpha_2! \ldots \alpha_n!} \frac{\partial^{|\alpha|} f(x_1^0, x_2^0, \ldots x_n^0)}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}}. \tag{6}$$

Thus the Taylor series (5) can be rewritten as follows:

$$f \approx \sum_{\alpha=0}^{\mu} F_\alpha (x_1 - x_1^0)^{\alpha_1} (x_2 - x_2^0)^{\alpha_2} \ldots (x_n - x_n^0)^{\alpha_n}. \tag{7}$$

The differentiation rules for the Taylor coefficients are similar to the generalized Leibnitz rules. There is only one difference: the rules for the Taylor coefficients do not employ the binomial coefficients. For example, the Taylor coefficients for the product of two functions u and v can be obtained as follows:

$$F_\mu = \sum_{0 \leq \alpha \leq \mu} U_\alpha V_{\mu - \alpha}, \tag{8}$$

here U and V are the Taylor coefficients of the functions u and v respectively. The differentiation rules for other arithmetical operations and elementary functions can be found in [15]. As soon as the Taylor coefficients are computed they can be converted into partial derivatives:

$$\frac{\partial^{|\alpha|} f(x_1^0, x_2^0, \ldots x_n^0)}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_n^{\alpha_n}} = \alpha_1! \alpha_2! \ldots \alpha_n! F_\alpha. \tag{9}$$

2.3 The Notion of a Differential Tuple

The generalized Leibnitz rules propagate the partial derivatives in forward mode — from independent to dependent variables. This means that all functions taking part in an automatic differentiation process need to communicate with each other via the specialized data structure — a differential tuple — containing the partial derivatives up to specified order. In addition to the derivatives, the differential tuple data structure also includes information about the number of independent variables and the order of the highest derivatives stored.

The important issue that may affect the efficiency of the algorithms is the storage scheme of partial derivatives in a differential tuple. For example, in [1, 2] a storage scheme is proposed which is optimized to deal with sparse differential tuples. For non-sparse differential data several authors [31, 21, 13] suggested to arrange the partial derivatives in an array by increasing order of the derivatives:

$$\left[f(P), \frac{\partial f}{\partial x_1}(P), \ldots, \frac{\partial f}{\partial x_n}(P), \frac{\partial^2 f}{\partial^2 x_1}(P), \frac{\partial^2 f}{\partial x_1 \partial x_2}(P), \ldots, \frac{\partial^2 f}{\partial^2 x_n}(P), \ldots\right]. \tag{10}$$

The employment of an array rather than a linked list allows constant time access to the derivatives using the index of the array's elements. Moreover, the usage of the allocation scheme (10) guarantees that the same spot in the differential tuple is occupied by the partial derivative of the same order (recall that the number of independent variables stays the same for all differential tuples). This allows us to retrieve the partial derivatives by accessing the elements of the differential tuple directly, which can be very critical for some applications. However, the generalized Leibnitz chain rules require the partial derivatives to be addressed through the multi-indices μ, α, and $\mu - \alpha$ (see (2), (3) and (6)). The storage scheme (10) is preferred because it simplifies the mapping of the multi-index μ, which defines the partial derivative $\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \ldots \partial x_n^{\mu_n}}$, to the array index R [26]:

$$R = \sum_{j=1}^{n} \binom{K_j + j}{j}, \tag{11}$$

where

$$K_j = \left(\sum_{i=0}^{j-1} \mu_{n-i}\right) - 1,$$

and n is a number of independent variables (dimension of space). Access to partial derivatives through multi-indices was used in the automatic differentiation algorithms developed in [26, 32]. The apparent drawback of such addressing scheme is that it requires $O(n)$ operations to compute the positions of partial derivatives in the differential tuple.

According to formula (11), the position of the particular partial derivative in the data structure depends not only on its order, but also on the number of independent variables. The assumption that all differential tuples have the same number of independent variables lets us simplify the automatic differentiation algorithms and the data structure. Moreover, a substantial increase in efficiency of the automatic differentiation algorithms may be achieved by utilization of a data structure that stores precomputed products of the binomial coefficients and positions of the partial derivatives in the differential tuple. Since the Taylor coefficients are in fact the scaled derivatives, the same data structure can be employed to store the Taylor coefficients.

2.4 The Data Structure for Fast Forward Automatic Differentiation

To achieve constant time access to the elements of differential tuples we employ special data structure that includes the main array M, the index array I, the coefficient array C, the squaring index array S, and the differentiation array D. The coefficient array C contains products of the binomial coefficients; the index array I contains positions of the partial derivatives in the differential tuple. The elements of the main array M are pointers to the index array. The difference between two of its consecutive elements gives the number of the terms in the series (2). The differentiation array D serves for computation of first partial derivatives of differential tuples with respect to the independent variables. The differentiation rule (4) suggests that all partial derivatives of f can be obtained by placing the partial derivatives of u into proper positions in differential tuple f:

$$f[R(\mu)] = u[R(\mu_1, \mu_2, \ldots, \mu_k + 1, \ldots, \mu_n)], \tag{12}$$

where μ is a multi-index defining the order of the partial derivatives and $R(\mu)$ is the position of the partial derivative given by multi-index μ (11). As we pointed out, formula (11) requires $O(n)$ operations in order to compute the position of a partial derivative in the data structure. However, substantial gain of speed can be achieved through employment of the differentiation array D that consists of n rows and $\binom{max\_order-1+n}{n}$ columns, where n defines the number of independent variables. The kth row of this array corresponds to differentiation with respect to the kth independent variable, and it contains the positions of the partial derivatives in the differential tuple being differentiated:

$$D\left[k, R\left(\mu\right)\right] = R\left(\mu_1, \mu_2, \ldots, \mu_i + 1, \ldots, \mu_n\right), \qquad k = 1, 2, \ldots, n \tag{13}$$

The elements of the multi-index μ take values between 0 and $max\_order - 1$. The following pseudo-code illustrates the initialization procedure of the auxiliary arrays:

```
Initialize( int n, int max_order )
{
    tuple_size = \binom{max_order+n}{n};
    IC_size = ArraySize( n, max_order);
    Allocate M[tuple_size+1];
    Allocate S[tuple_size];
    Allocate I[IC_size];
    Allocate C[IC_size];
    Allocate μ[n];
    Allocate α[n];
    Initialize the multi-index μ = 0;
    μ[n] = max_order;
    k = IC_size;
    M[tuple_size] = IC_size;
    For i=tuple_size-1 to 0 step -1
    {
        Find p such that μ[p]>0 and μ[p] = min{μ};
        Initialize the multi-index α = μ;
        size = ∏_{j=1}^{n} (μ[j] + 1);
        For j=size-2 to 1 step -1
        {
            Decrement the multi-index α with the preference of α[p];
            I[k]=R(α);
            C[k]=∏_{l=1}^{n} \binom{μ[l]}{α[l]};
            k = k - 1;
        }
        S[i] = R(μ-1);
        M[i] = k;
        Decrement the multi-index μ;
    }
    tuple_size1 = \binom{max_order-1+n}{n};
    Allocate D[n, tuple_size1];
    Initialize the multi-index μ = 0;
    For i=1 to tuple_size
    {
        For k=1 to n
        {
            D[k,R(μ)] = R(μ_1, μ_2,..., μ_k + 1,..., μ_n);
        }
        Increment the multi-index μ;
    }
    Deallocate μ;
    Deallocate α;
```

}

The recursive function `ArraySize` computes the number of elements in the arrays I and C for the specified number of independent variables and maximal order of the derivatives. This number is exactly the number of the terms in the Leibnitz series (2). Here is the implementation of the `ArraySize` function:

```
int ArraySize(int maxdim, int maxord)
{
    int i;
    int sum = 0;
    If( maxord == 0 || maxdim == 0)
    {
       return 1;
    }
    For i=0 to maxord
    {
       sum = sum + ArraySize(maxdim-1, maxord-i)*(i+1);
    }
    return sum;
}
```

As soon as size of the data structure is determined, function `Initialize` allocates memory for the auxiliary arrays and multi-indices μ and α. Then multi-index μ is initialized in such way that it corresponds to the last element in differential tuple data structure: all elements of μ have zero value except its nth element whose value is set to the maximal order of the derivatives. Variable `k` is used to go through the arrays I and C, and it is initialized with value of `IC_size` (it points at the last elements of the arrays). Function `Initialize` employs two nested loops for filling the data structure. Both loops decrement their indices from upper bounds to their lower bounds. This results in backward order of initialization of the data structure. As index i of the outer loop decreases, multi-index μ goes element by element through arrays S and M in decreasing order of the partial derivatives and increasing lexically within the same order as shown in Table 1. The element of the squaring index array S standing at ith position receives the value of $R(\mu - 1)$. As we discussed in Section 2.1, the multi-index $\mu - 1$ differs from μ by the element $\mu_p - 1$. In order to minimize the number of terms in Leibnitz series, index p has to point at the minimal non-zero element of μ. Once the value of p is determined, multi-index α is initialized with values of μ. Inner loop assigns values to the elements of the index array I and coefficient array C that correspond to values of multi-index α between 0 and μ: $0 < \alpha < \mu$. In order to exclude the derivatives for $\alpha = \mu$ which are easily accessed, α is decremented at the beginning of the loop. In contrast to the decrementing method for μ, the multi-index α is decremented coordinate-wise, such that α_p is changed last. This scheme allows us to group elements of the data structure by the corresponding value of α_p. As result, the elements of the coefficient array C and the index array I are stored in decreasing α_p order for each multi-index μ. This ordering is used by the squaring index array S that provides access to the terms with α_p strictly less than μ_p as required by (3). We will illustrate the usage of the squaring index array S later, when we discuss the automatic differentiation algorithms of the elementary functions. As soon as α is decremented, the kth element of the index array I takes on the position of the derivative defined by multi-index α: `I[k] = R(`α`)`. The element `C[k]` of the coefficient array is computed as a product of the individual binomial coefficients $\binom{\mu[l]}{\alpha[l]}$, $l = 1, \ldots, n$. At the end of the loop index k is decremented. After termination of the inner loop ith element of the main array M is assigned the value of the index k.

Table 1 presents the values of the elements of the data structure (arrays M, S, I, and C) for three independent variables and second order derivatives ($i = 0, \ldots, 9$). The auxiliary arrays do not include elements corresponding to the first and the last terms in the Leibnitz series because these terms are formed from the derivatives which can be easily accessed. Lines for $i = 10$ and 11 illustrate the difference in incrementing of multi-indices μ and α. Table 2 presents elements of the differential array D computed by formula (13) for three independent variables and second order derivatives. Later in this Section we shall discuss a few examples using the data from Table 1 and Table 2.

$i = R(\mu)$	μ	M	S	j	α	$I = R(\alpha)$	$C = \binom{\mu}{\alpha}$
0	(0,0,0)	0	0				
1	(1,0,0)	0	0				
2	(0,1,0)	0	0				
3	(0,0,1)	0	0				
4	(2,0,0)	0	1	0	(1,0,0)	1	2
5	(1,1,0)	1	2	1	(0,1,0)	2	1
				2	(1,0,0)	1	1
6	(1,0,1)	3	3	3	(0,0,1)	3	1
				4	(1,0,0)	1	1
7	(0,2,0)	5	2	5	(0,1,0)	2	2
8	(0,1,1)	6	3	6	(0,0,1)	3	1
				7	(0,1,0)	2	1
9	(0,0,2)	8	3	8	(0,0,1)	3	2
10	(3,0,0)	9	4	9	(1,0,0)	1	3
				10	(2,0,0)	4	3
11	(2,1,0)	11	4	11	(1,0,0)	1	2
				12	(2,0,0)	4	1
				13	(0,1,0)	2	1
				14	(1,1,0)	5	2

Table 1: Example of the data structure for three independent variables and second order derivatives (lines for $i = 0$ through 9). Lines for $i = 10$ and 11 illustrate the difference in incrementing of multi-indices μ and α

Derivative \ Position	0	1	2	3
$\frac{\partial}{\partial x_1}$	1	4	5	6
$\frac{\partial}{\partial x_2}$	2	5	7	8
$\frac{\partial}{\partial x_3}$	3	6	8	9

Table 2: The elements of the differential array D for three independent variables and second order derivatives

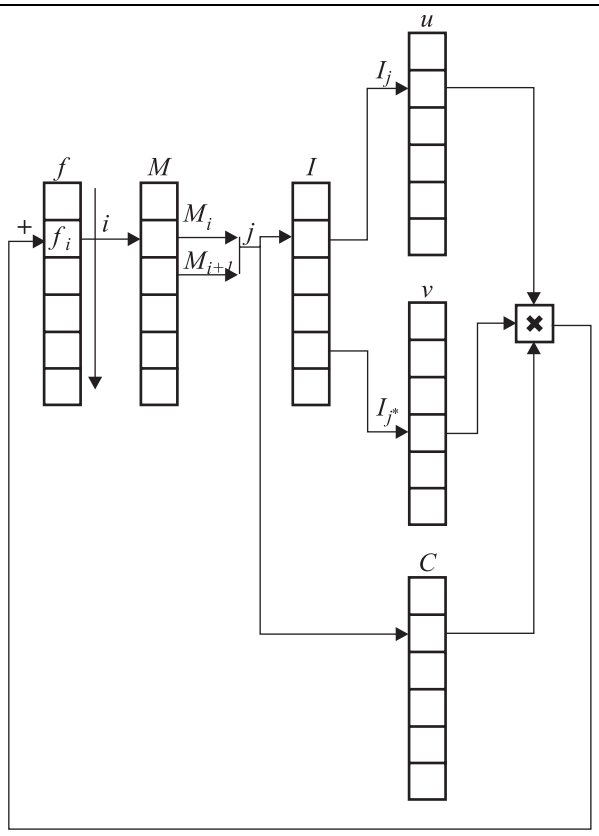


Figure 1: Data structure for automatic differentiation of the product $f = uv$

2.5 Automatic Differentiation Algorithms

Let us show now how the proposed data structure is incorporated in the automatic differentiation algorithms. We begin our discussion with automatic differentiation algorithms for multiplication and division, then we explain the changes in the addressing scheme needed for automatic differentiation of elementary functions, and we conclude this Section describing the data structure for a differential operator.

Figure 1 illustrates the interaction among the elements of the data structure using, as an example, the differentiation of the product $f = uv$, where f, u, and v are differential tuples. As we mentioned earlier, the partial derivatives of the product of two functions, or using our terminology — two differential tuples, are given by the Leibnitz formula (2). The automatic differentiation process utilizes two loops. The first loop goes through the elements of the result differential tuple f (through the partial derivatives of the function f). Defining the parameter i, we choose the partial derivative whose order is defined by the multi-index μ (see (2)). Then from the main array M we retrieve two numbers M_i and M_{i+1}. The difference $M_{i+1} - M_i$ tells us the number of elements in the second (summation) loop. Let us use j as a summation index which changes from M_i to $M_{i+1} - 1$. The index j uniquely corresponds to multi-index α in the Leibnitz chain rule (2). It means therefore that two indices i and j define the particular product of the binomial coefficients stored in the coefficient array C. However, to access the partial derivatives stored in the differential tuples u and v we need the index array I (Figure 1). So, the ith element of the result differential tuple f may be computed as follows:

$$f_i = u_0 v_i + u_i v_0 + \sum_{j=M_i}^{M_{i+1}-1} C_j \cdot u_{I_j} \cdot v_{I_{j^*}}, \tag{14}$$

where $j^* = M_i + M_{i+1} - 1 - j$. The following pseudo-code illustrates the automatic differentiation algorithm of the product:

```
00 REM Differential Tuples Approach.  Not optimized version
01 tuple operator * (tuple u, tuple v)
02 {
03    tuple f;
04    f.order = min(u.order, v.order);
05    f.size = min(u.size, v.size);
06    f[0] = u[0] * v[0];
07    For i = 1 to f.size-1
08    {
09       f[i] = u[0]*v[i]+u[i]*v[0];
10       MM = M[i]+M[i+1]-1;
11       For j = M[i] to M[i+1]-1
12       {
13          f[i] = f[i] + C[j]*u[I[j]]*v[I[MM-j]];
14       }
15    }
16    return f;
17 }
```

This automatic differentiation algorithm does not require its arguments to have the same order. In fact, if orders of the differential tuples u and v are different, the resulting differential tuple has the order of either u or v whichever is less.

Example. Let us show the interaction between elements of the data structure for computation of the partial derivative $\frac{\partial^2 f(x,y,z)}{\partial x \partial y} = \frac{\partial^2}{\partial x \partial y}(uv)$. The Leibnitz rule gives the following expression for the derivative:

$$\frac{\partial^2 f(x,y,z)}{\partial x \partial y} = u\frac{\partial^2 v}{\partial x \partial y} + \frac{\partial u}{\partial x}\frac{\partial v}{\partial y} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \frac{\partial^2 u}{\partial x \partial y}v \tag{15}$$

Now let us watch how the algorithm computes this partial derivative. Using the index `i`, the algorithm goes through the differential tuple `f`, computing the partial derivatives. Table 3 shows the locations of the partial derivatives in the differential tuples `f`, `u` and `v`. The partial derivative $\frac{\partial^2 f}{\partial x \partial y}$ is located at the fifth position in the differential tuple

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]
f	$\frac{\partial f}{\partial x}$	$\frac{\partial f}{\partial y}$	$\frac{\partial f}{\partial z}$	$\frac{\partial^2 f}{\partial x^2}$	$\frac{\partial^2 f}{\partial x \partial y}$	$\frac{\partial^2 f}{\partial x \partial z}$	$\frac{\partial^2 f}{\partial y^2}$	$\frac{\partial^2 f}{\partial y \partial z}$	$\frac{\partial^2 f}{\partial z^2}$
u[0]	u[1]	u[2]	u[3]	u[4]	u[5]	u[6]	u[7]	u[8]	u[9]
u	$\frac{\partial u}{\partial x}$	$\frac{\partial u}{\partial y}$	$\frac{\partial u}{\partial z}$	$\frac{\partial^2 u}{\partial x^2}$	$\frac{\partial^2 u}{\partial x \partial y}$	$\frac{\partial^2 u}{\partial x \partial z}$	$\frac{\partial^2 u}{\partial y^2}$	$\frac{\partial^2 u}{\partial y \partial z}$	$\frac{\partial^2 u}{\partial z^2}$
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
v	$\frac{\partial v}{\partial x}$	$\frac{\partial v}{\partial y}$	$\frac{\partial v}{\partial z}$	$\frac{\partial^2 v}{\partial x^2}$	$\frac{\partial^2 v}{\partial x \partial y}$	$\frac{\partial^2 v}{\partial x \partial z}$	$\frac{\partial^2 v}{\partial y^2}$	$\frac{\partial^2 v}{\partial y \partial z}$	$\frac{\partial^2 v}{\partial z^2}$

Table 3: Position of the partial derivatives in differential tuple for three independent variables and second order derivatives

f. Now let us observe what happens when the loop index `i` in line 07 of the pseudo-code has value of 5. First, the algorithm assigns `f[5] = u[0]*v[5] + u[5]*v[0]` in line 09. From the array M (see Table 1) the algorithm gets values for `M[5]` and `M[5+1]` which are 1 and 3 respectively. These values define the range for index `j` of the inner loop (line 11): the index `j` changes from 1 to 2. Furthermore, the extracted values are used in line 10 to form the value of `MM` variable (in our case `MM=M[5]+M[5+1]-1`=1+3-1=3). For `j=1` the algorithm takes from array C the value of the corresponding binomial coefficient `C[1]=1`, then using index array I algorithm computes the positions of the partial derivatives in the differential tuples `u` and `v` (`u[I[j]] = u[I[1]]= u[2]`; `v[I[MM-j]] = v[I[3-1]]= v[I[2]] = v[1]`), combines the values of the partial derivatives with binomial coefficient and adds to `f[5]`: `f[5] = f[5] + 1*u[2]*v[1]`. For `j=2` line 13 of the pseudo-code looks like this: `f[5] = f[5] + 1*u[1]*v[2]`. At this point element `f[5]` contains the value of $\frac{\partial^2 f}{\partial x \partial y}$. Summarizing what has been done by the algorithm, we conclude that the value of `f[5]` was formed from the following terms: `f[5] = u[0]*v[5] + u[5]*v[0] + u[2]*v[1] + u[1]*v[2]`. Using Table 3 we conclude that the last expression coincides with the expression (15) for the partial derivative $\frac{\partial^2 f}{\partial x \partial y}$.

As soon as the interaction between elements of the data structure is clear, we can speed up the computations using the symmetry of the binomial coefficients and the multi-indices α and $\mu - \alpha$ in the generalized Leibnitz chain rules. We can eliminate half of the multiplications by binomial coefficients by collecting the terms with the same binomial coefficients. The previously discussed pseudo-code appears as follows:

```
00 REM Differential Tuples Approach.  Optimized version
01 tuple operator * (tuple u, tuple v)
02 {
03    tuple f;
04    f.order = min(u.order, v.order);
05    f.size = min(u.size, v.size);
06    f[0] = u[0] * v[0];
07    For i = 1 to f.size-1
08    {
09       f[i] = u[0]*v[i]+u[i]*v[0];
10       MM = M[i]+M[i+1]-1;
11       L = M[i+1]-M[i];
12       L2 = L/2;
13       if( L % 2 == 1 )
14       {
15          j = M[i]+L2;
16          f[i] = f[i] + C[j]*u[I[j]]*v[I[j]];
17       }
18       For j = M[i] to M[i]+L2
19       {
```

```
20           f[i] = f[i] + C[j]*(u[I[j]]*v[I[MM-j]]+u[I[MM-j]]*v[I[j]]);
21       }
22    }
23    return f;
24 }
```

We have discussed how the data structure shown in Figure 1 is employed for the derivative data computations. However, a small change is needed to adopt this data structure to deal with Taylor coefficients: we do not have to use the coefficient array C at all. That is why the coefficient array C is shown in a dashed box. The addressing scheme remains the same and it works the same way as for the derivative data. Because the Taylor coefficients are the scaled partial derivatives, the tuple data structure can be used to store the Taylor coefficients. Here is the pseudo-code for the algorithm implementing the multiplication of two sets of Taylor coefficients:

```
00 REM Taylor Coefficient Approach
01 tuple operator * (tuple u, tuple v)
02 {
03    tuple f;
04    f.order = min(u.order, v.order);
05    f.size = min(u.size, v.size);
06    f[0] = u[0] * v[0];
07    For i = 1 to f.size-1
08    {
09       f[i] = u[0]*v[i]+u[i]*v[0];
10       MM = M[i]+M[i+1]-1;
11       L = M[i+1]-M[i];
12       L2 = L/2;
13       if( L % 2 == 1 )
14       {
15          j = M[i]+L2;
16          f[i] = f[i] + u[I[j]]*v[I[j]];
17       }
18       For j = M[i] to M[i]+L2
19       {
20          f[i] = f[i] + (u[I[j]]*v[I[MM-j]]+u[I[MM-j]]*v[I[j]]);
21       }
22    }
23    return f;
24 }
```

The addressing scheme shown in Figure 1 is also used for automatic differentiation of the quotient of two differential tuples $f = u/v$:

```
00 REM Differential Tuples Approach
01 tuple operator / (tuple u, tuple v)
02 {
03    tuple f;
04    f.order = min(u.order, v.order);
05    f.size = min(u.size, v.size);
06    f[0] = u[0] / v[0];
07    For i = 1 to f.size-1
08    {
09       f[i] = f[0]*v[i];
10       MM = M[i]+M[i+1]-1;
11       L = M[i+1]-M[i];
12       L2 = L/2;
13       if( L % 2 == 1 )
```

```
14        {
15            j = M[i]+L2;
16            f[i] = f[i] + C[j]*v[I[j]]*f[I[j]];
17        }
18        For j = M[i] to M[i]+L2
19        {
20            f[i] = f[i] + C[j]*(v[I[j]]*f[I[MM-j]]+v[I[MM-j]]*f[I[j]]);
21        }
22        f[i] = (u[i]-f[i])/v[0];
23    }
24    return f;
25 }
```

The reader may observe the key distinction between automatic differentiation of the product and the quotient: the automatic differentiation of the quotient requires lower order derivatives to be computed before the computations start for higher derivatives. The partial derivatives of the quotient and many elementary functions are defined recursively by the order of the derivatives.

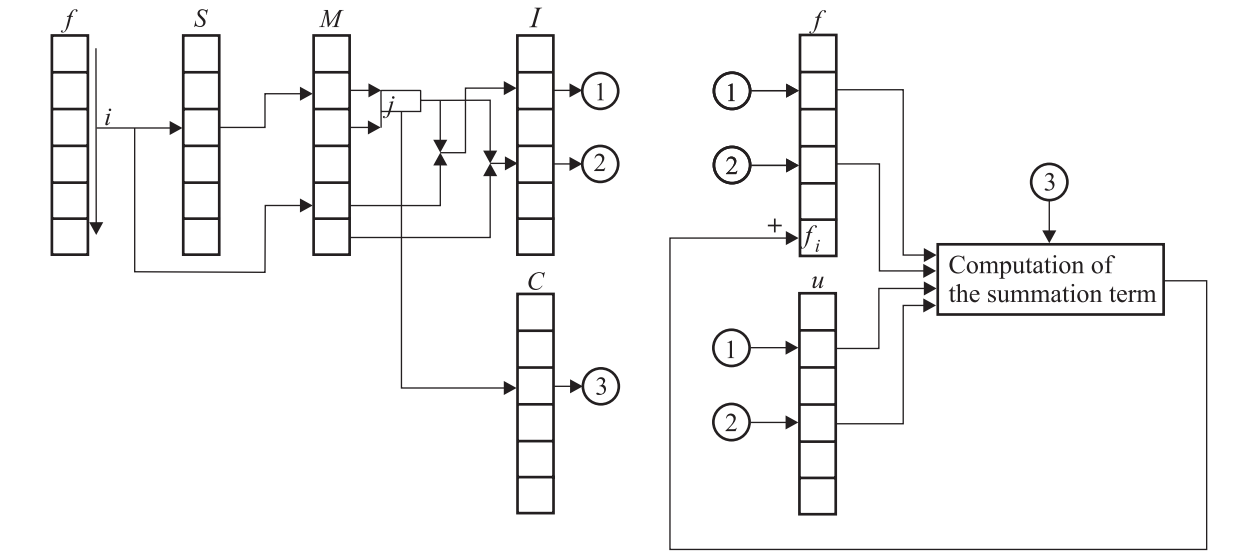


Figure 2: Data structure for automatic differentiation of an elementary function

The discussed addressing scheme is plain and simple, and it makes it possible to access the elements of the differential tuples in constant time. However, it cannot be applied to automatic differentiation algorithms of elementary functions, with the exception of the square root function. This is so because the generalized Leibnitz chain rules for these functions use different sets of the binomial coefficients and upper bounds in the summation loops for which the multi-index $\mu - 1$ is employed. The access to the partial derivatives requires the multi-indices μ, α, and $\mu - \alpha$. We can fix the situation by utilizing the squaring index array S, whose ith element S_i corresponds to the multi-index $\mu - 1$ (recall the index i corresponds to the multi-index μ).

Figure 2 illustrates the data structure for automatic differentiation of elementary functions. The parameter i of the first loop points to the derivatives in resulting differential tuple f and to the elements of the arrays S and M. Elements M_{S_i} and M_{S_i+1} of the main array form the bounds of the second loop, which employs the index j. It points to the elements of the coefficient array C. The parameter j together with elements M_{S_i}, M_i and M_{i+1} of the main array takes part in computation of the positions of the indices in the index array I. Then, using these indices, we retrieve the necessary elements from the differential tuples u and f, and we combine these elements with the binomial coefficient from the coefficient array C into the summation term, whose particular representation depends on the elementary function being differentiated. However, the addressing scheme remains the same for all elementary functions with

the exception of the square root function, which uses the addressing scheme already discussed for the multiplication operation. The following pseudo-code illustrates the automatic differentiation algorithm of the power function $f = u^s$, where $s = const$:

```
00 REM Power function u^s.  Not optimized version
01 tuple pow(tuple u, double s)
02 {
03    tuple f;
04    f[0] = pow(u[0],s);
05    For i = 1 to f.size-1
06    {
07       f[i] = s*f[0]*u[i]/u[0];
08    }
09    For i = f.dimension to f.size-1
10    {
11       L = M[S[i]+1]-M[S[i]];
12       sum = s*f[I[M[i]+L]]*u[I[M[i+1]-L-1] -
13                    u[I[M[i]+L]]*f[I[M[i+1]-L-1];
14       For j = 0 to L-1
15       {
16          first = I[M[i] + j];
17          last = I[M[i+1] - j - 1];
18          sum = sum + C[M[S[i]]+j]*(s*f[first]*u[last]-
19                                u[first]*f[last]);
20       }
21       f[i] = f[i] + sum/u[0];
22    }
23    return f;
24 }
```

Differential tuple	Value	$\frac{\partial}{\partial x}$	$\frac{\partial}{\partial y}$	$\frac{\partial}{\partial z}$	$\frac{\partial^2}{\partial x^2}$	$\frac{\partial^2}{\partial x \partial y}$	$\frac{\partial^2}{\partial x \partial z}$	$\frac{\partial^2}{\partial y^2}$	$\frac{\partial^2}{\partial y \partial z}$	$\frac{\partial^2}{\partial z^2}$
u	10	5	2	0	0	1	0	0	0	0
f=pow(u,2.5)	316.228	395.285	158.114	0	296.464	197.642	0	47.4342	0	0

Table 4: Differential tuple `f` is the result of `pow(u,2.5)`

Example. Let us follow this pseudo-code to explore the interaction between elements of the data structure for computation of the partial derivative $\frac{\partial^2 f(x,y,z)}{\partial x \partial y} = \frac{\partial^2}{\partial x \partial y}\left(u^{2.5}\right)$ (see Table 4). New differential tuple `f` is initialized in line 03. All elements of `f` are zeros. In line 04 the value of zero order derivative is computed: `f[0] = pow(u[0],2.5) = pow(10,2.5) = 316.228`. Lines 05 through 08 compute the first summand of the partial derivatives: $su^{s-1}\frac{\partial^{|\mu|}u}{\partial x^{\mu_1}\partial y^{\mu_2}\partial z^{\mu_3}}$. Note that u to power of $s-1$ is computed as the ratio `f[0]/u[0]`. This loop is performed for all partial derivatives in the differential tuple `f`. For `i=5` we have `f[5] = 2.5 * 316.228 * 1. / 10 = 79.057` The next loop (lines 09 through 22) computes sums of other terms in the Leibnitz series. Let us watch what is happening when the loop index `i` has value 5 (it corresponds to the partial derivative $\frac{\partial^2 f(x,y,z)}{\partial x \partial y}$). In line 11 the algorithm computes the value of the intermediate variable `L` using the arrays S and M: `L = M[S[5]+1]-M[S[5]] = M[2+1]-M[1] = M[3]-M[1] = 0-0 = 0`. Lines 12 and 13 compute the value of `sum`: `sum = s * f[I[M[5]+0]] * u[I[M[6]-0-1]] - u[I[M[5]+0]] * f[I[M[6]-0-1]]`. Retrieving values for elements `M[5]` and `M[6]` we get `sum = s * f[I[1]] * u[I[2]] - u[I[1]] * f[I[2]]`. Now the algorithm extracts the indices from the index array I and composes the value for `sum`: `sum = s * f[2] * u[1] - u[2] * f[1]`. In our case `sum = 2.5 * 158.114 * 5 - 2 * 395.285 = 1185.85`. The loop in

lines 14 through 20 updates the value of `sum`, adding the rest of the terms in Leibnitz series (3). The loop index `j` changes from `0` to `L-1 = 0-1 = -1`. Since the upper bound of the loop index is less than the lower bound, the loop body (lines 16-19) is not executed. In line 21 algorithm updates the value of `f[5]` adding `sum/u[0]`: `f[5] = 79.057 + 1185.85 / 10 = 197.642`. Table 4 presents other elements of the differential tuple `f`.

Combining the terms with the same binomial coefficient we can eliminate half of the executions of the lines 16-19. Here is the optimized version of the automatic differentiation algorithm of the power function $f = u^s$:

```
00 REM Optimized version of the power function u^s.
01 tuple pow(tuple u, double s)
02 {
03    tuple f;
04    f[0] = pow(u[0],s);
05    For i = 1 to f.size-1
06    {
07       f[i] = s*f[0]*u[i]/u[0];
08    }
09    For i = f.dimension to f.size-1
10    {
11       L = M[S[i]+1]-M[S[i]];
11       L2 = L/2;
12       sum = s*f[I[M[i]+L]]*u[I[M[i+1]-L-1] -
13                   u[I[M[i]+L]]*f[I[M[i+1]-L-1];
14       If( L % 2 == 1 )
15       {
16          first = I[M[i]]+L2;
17          last = I[M[i+1]-L2-1;
18          sum = sum + C[M[S[i]]+L2] *
19                (s*f[first]*u[last] - u[first]*f[last]);
20       }
21       For j = M[S[i]] to M[S[i]]+L2
22       {
23          first = I[j - M[S[i]] + M[i]];
24          last = I[M[S[i]] - 1 - j + M[i+1]];
25          sum = sum + C[j]*(s*(f[first]*u[last] + u[first]*f[last])-
26                            u[first]*f[last] - f[first]*u[last]);
27       }
28       f[i] = f[i] + sum/u[0];
29    }
30    return f;
31 }
```

The automatic differentiation algorithm that computes Taylor coefficients instead of derivatives is a small modification of the previous algorithm: multiplication by binomial coefficient in lines 18 and 25 is removed. Thus the automatic differentiation algorithm for Taylor coefficients looks like this:

```
00 REM Power function u^s.  Taylor coefficients approach
01 tuple pow(tuple u, double s)
02 {
03    tuple f;
04    f[0] = pow(u[0],s);
05    For i = 1 to f.size-1
06    {
07       f[i] = s*f[0]*u[i]/u[0];
08    }
09    For i = f.dimension to f.size-1
```

```
10    {
11       L = M[S[i]+1]-M[S[i]];
11       L2 = L/2;
12       sum = s*f[I[M[i]+L]]*u[I[M[i+1]-L-1] -
13                   u[I[M[i]+L]]*f[I[M[i+1]-L-1];
14       If( L % 2 == 1 )
15       {
16          first = I[M[i]]+L2;
17          last = I[M[i+1]-L2-1;
18          sum = sum + (s*f[first]*u[last] - u[first]*f[last]);
19       }
20       For j = M[S[i]] to M[S[i]]+L2
21       {
22          first = I[j - M[S[i]] + M[i]];
23          last = I[M[S[i]] - 1 - j + M[i+1]];
24          sum = sum + s*(f[first]*u[last] + u[first]*f[last])-
25                        u[first]*f[last] - f[first]*u[last];
26       }
27       f[i] = f[i] + sum/u[0];
28    }
29    return f;
30 }
```

Having developed the automatic differentiation algorithms for all elementary functions and arithmetic operations, we can automatically differentiate any composite function, which comprises the elementary functions and arithmetic operations. The automatic differentiation process of such function is performed in the forward mode: first the differential tuples for the independent variables are computed, then each consecutive function in the computational graph takes the differential tuple formed by the previous function. The result of the computations is the differential tuple containing the partial derivatives of the composite function. But for many practical applications it may not be enough just to compute a differential tuple. Some applications may require the differential tuples to be further differentiated. For example, suppose we have computed the differential tuple u, and now we would like to obtain a differential tuple f corresponding to the partial derivative $f = \frac{\partial u}{\partial x_k}$, where $0 \leq k \leq n$ and n is the number of independent variables. As we discussed in Section 2.4, differentiation operation performed on a differential tuple u results in copying its elements into differential tuple f according to formula (12). Employing the differential array D whose elements are given by expression (13), implementation of a differentiation operator on the differential tuples appears as follows:

```
01 tuple dx(tuple u, int k)
02 {
03    tuple f;
04    f.order = u.order-1;
05    f.size = (f.order+dimension dimension);
06    For i = 0 to f.size-1
07    {
08       f[i]=u[D[k,i]];
09    }
10    return f;
11 }
```

This pseudo-code illustrates computation of first order partial derivatives only. However, higher order derivatives of a differential tuple can be derived by sequential application of differentiation operator `dx`.

Example. Let us compute the first partial derivatives of the differential tuple `u` presented in Table 5. In order to compute $\frac{\partial u}{\partial x}$ we call the function `dx` setting `k=1`: `dx(u,1)`. New differential tuple `f` is initialized in line 03. The order of the differential tuple `f` is decreased in line 04 of the algorithm: `f.order = u.order-1 = 2-1 = 1`. The new length of the storage for partial derivatives is computed in line 05: `f.size =` $\binom{f.order+dimension}{dimension} =$

Differential tuple	Value	$\frac{\partial}{\partial x}$	$\frac{\partial}{\partial y}$	$\frac{\partial}{\partial z}$	$\frac{\partial^2}{\partial x^2}$	$\frac{\partial^2}{\partial x \partial y}$	$\frac{\partial^2}{\partial x \partial z}$	$\frac{\partial^2}{\partial y^2}$	$\frac{\partial^2}{\partial y \partial z}$	$\frac{\partial^2}{\partial z^2}$
u	316.228	395.285	158.114	0	296.464	197.642	0	47.4342	0	0

Table 5: Differential tuple u to be differentiated

Position in f (index i)	Position in u (given by D[1,i])	u[D[1,i]]
0	1	395.285
1	4	296.464
2	5	197.642
3	6	0

Table 6: Computation of $\frac{\partial u}{\partial x}$ of differential tuple u

Differential tuple	Value	$\frac{\partial}{\partial x}$	$\frac{\partial}{\partial y}$	$\frac{\partial}{\partial z}$
$\frac{\partial u}{\partial x}$	395.285	296.464	197.642	0
$\frac{\partial u}{\partial y}$	158.114	197.642	47.4342	0
$\frac{\partial u}{\partial z}$	0	0	0	0

Table 7: First partial derivatives of differential tuple u shown in Table 5

$\binom{1+3}{3} = 4$. Lines 06-09 form a loop which goes through all elements of differential tuple `f`. Table 6 shows how the algorithm using the differential array D (see Table 2) for each value of the loop index `i` gets access to the proper elements of the differential tuple `u` and puts their values into differential tuple `f`. Other first partial derivatives can be computed similarly. Table 7 presents the results of the differentiation algorithm: the first partial derivatives of the differential tuple `u` with respect to all independent variables.

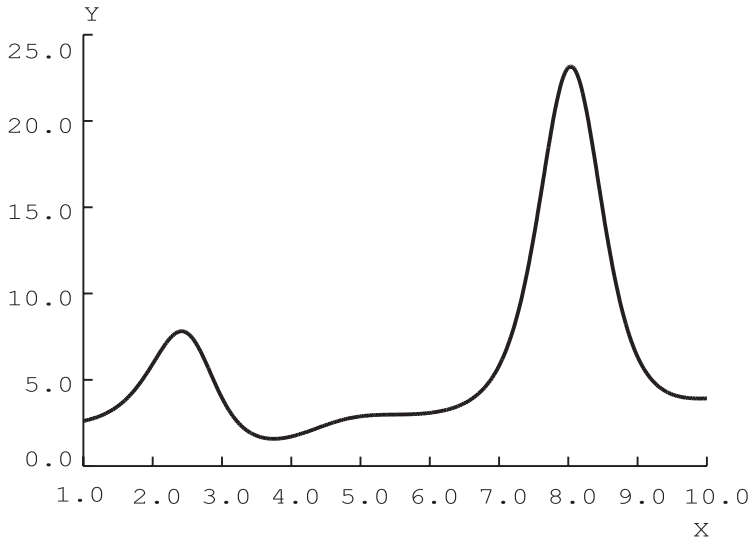


Figure 3: Plot of the function $f = e^{\sin(x+\cos(x+\sqrt{x}))+\ln(x+\frac{1}{2})}$

3 Analysis of the Automatic Differentiation Algorithms

The generalized Leibnitz chain rules and Taylor series computations represent the partial derivatives exactly. In fact, if we differentiate the function

$$f = e^{\sin(x+\cos(x+\sqrt{x}))+\ln(x+\frac{1}{2})}, \tag{16}$$

whose plot is presented in Figure 3, symbolically (in Mathematica [35]) and applying the discussed automatic differentiation algorithms we get good agreement of the derivatives (see plots in Figure 4).

The developed data structure speeds up the automatic differentiation computations substantially providing constant time access to the partial derivatives and precomputed products of the binomial coefficients. It also eliminates the need to multiply the binomial coefficients every time the derivatives are being computed. The histogram shown in Figure 5 illustrates that significant speed increases (relative to the algorithms described in [32]) may be achieved for large numbers of independent variables and for higher order derivatives. In fact, the algorithms described in [32] for each partial derivative defined by multi-index μ perform $n \prod_{i=1}^{n}(\mu_i + 1)$ multiplication operations in order to compute the product of binomial coefficients and $O(n)$ addition operations to compute the positions of the derivatives in the data structure. The proposed data structure eliminates computation of the addresses of the derivatives and decreases the number of multiplications to $\prod_{i=1}^{n}(\mu_i + 1)$. The ratio $\frac{n\prod_{i=1}^{n}(\mu_i+1))}{\prod_{i=1}^{n}(\mu_i+1)} = O(n)$ gives the speed increase, which is proportional to the number of independent variables. However, the histogram shown in Figure 5 illustrates the real speed increase also depends on the order of the derivatives. This fact arises because the implementations of the algorithms given in [32] have supplemental expenses for incrementing the multi-index α (see formulas (2) and (3)).

The Taylor coefficient approach allows us to eliminate another $\prod_{i=1}^{n}(\mu_i + 1)$ multiplication operations. We can estimate the speed increase by comparing multiplication algorithms for differential tuple and Taylor coefficient approach. These two algorithms differ from each other by lines 16 and 20: the algorithm dealing with Taylor coefficients does not multiply the partial derivatives by binomial coefficients. Since lines 16 and 20 of both algorithms are not executed for first partial derivatives at all, the Taylor coefficient approach works in this case with the same speed as

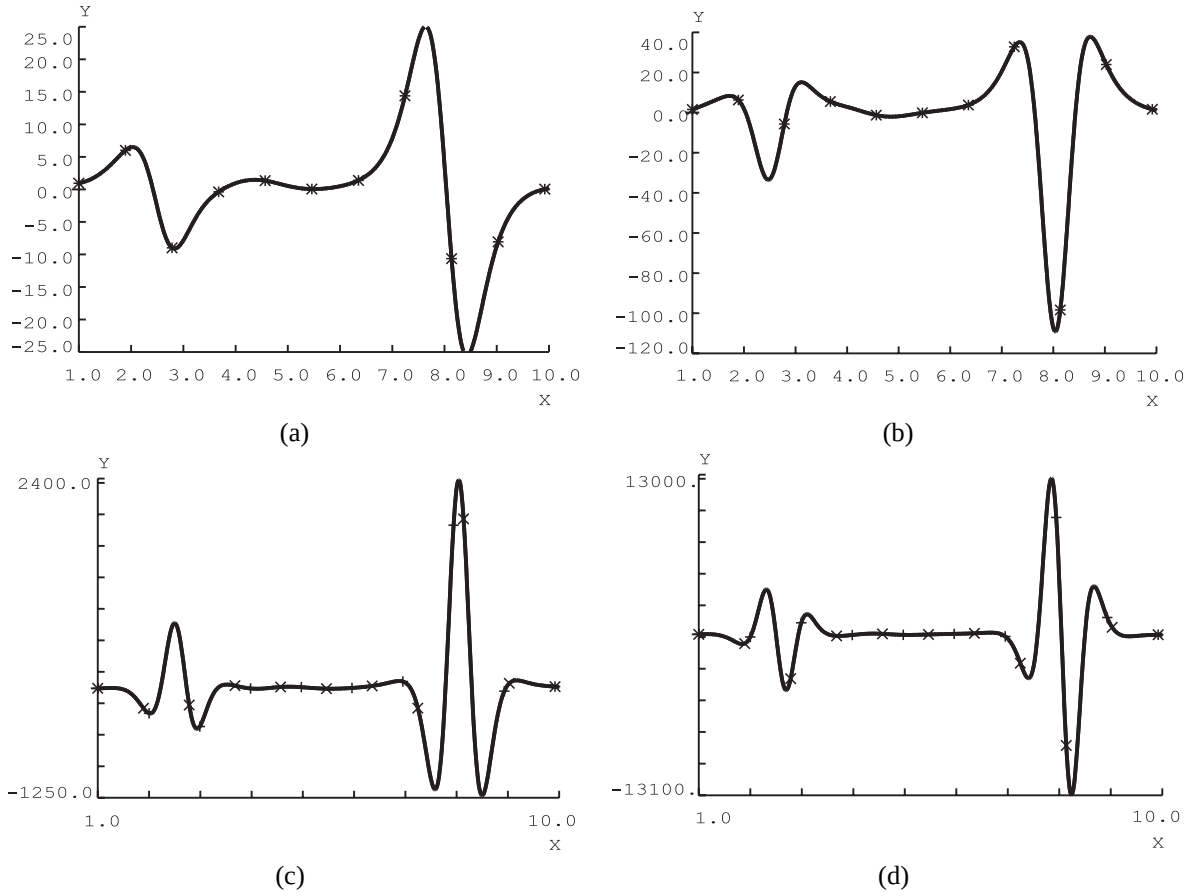


Figure 4: (a)The first derivative of the function (16); (b)the second derivative of the function (16); (c)the fourth derivative of the function (16); (d)the fifth derivative of the function (16). Marker "+" defines the plots of the derivatives computed symbolically and marker "$\times$" designates plots of the derivatives produced by the automatic differentiation library

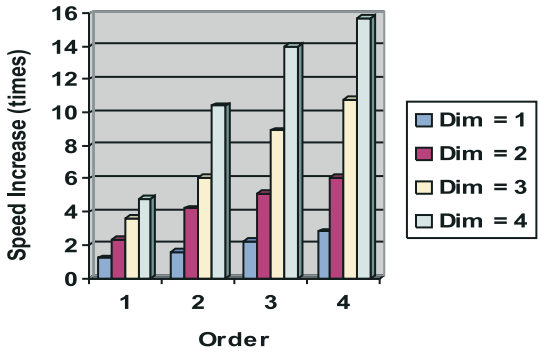


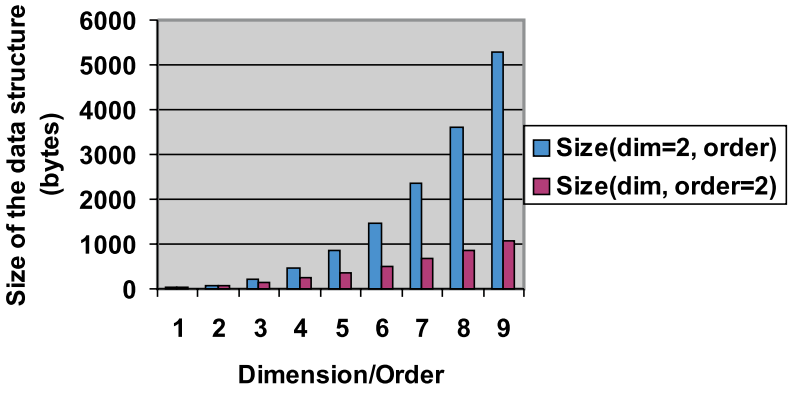
Figure 5: Dependence of the speed increase (relative to the algorithm described in [32]) on number of independent variables and order of derivatives

Dimension \ Order	1	2	3	4	5	6	7	8
1	24	44	72	108	152	204	264	332
2	36	108	260	532	972	1636	2588	3900
3	48	204	644	1652	3680	7404	13788	24156
4	60	332	1292	3972	10420	24372	52212	104292
5	72	492	2272	8132	24540	65420	158500	355620
6	84	684	3652	14908	50860	152220	411564	1024932
7	96	908	5500	25212	95940	318540	950844	2602788
8	108	1164	7884	40092	168348	614076	2005884	5986740
9	120	1452	10872	60732	278928	1108812	3934272	12715572
10	132	1772	14532	88452	441068	1897908	7268988	25292708

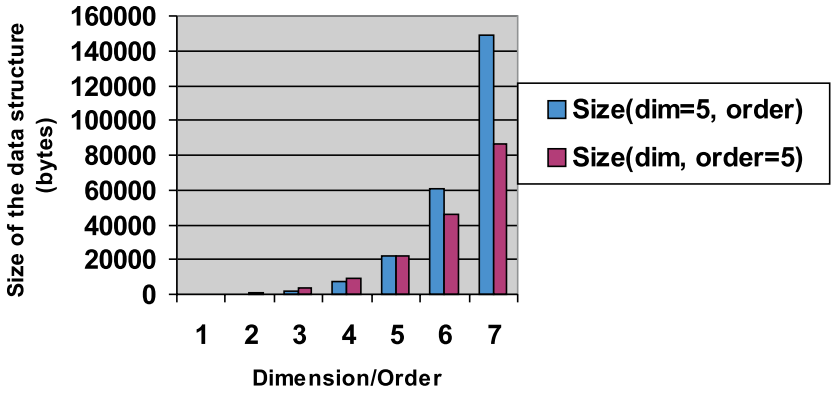
Table 8: Size in bytes of the auxiliary data structure for differential tuples technique (4 bytes per element)

Dimension \ Order	1	2	3	4	5	6	7	8
1	24	40	60	84	112	144	180	220
2	36	92	196	368	632	1016	1552	2276
3	48	168	464	1088	2276	4376	7880	13460
4	60	268	908	2548	6276	14036	29108	56768
5	72	392	1572	5132	14540	37080	87040	190880
6	84	540	2500	9304	29800	85352	223736	545072
7	96	712	3736	15608	55760	177224	513176	1375184
8	108	908	5324	24668	97244	339644	1076732	3147812
9	120	1128	7308	37188	160344	610464	2102988	6660948
10	132	1372	9732	53952	252568	1041048	3872448	13210348

Table 9: Size in bytes of the auxiliary data structure for Taylor coefficient approach (4 bytes per element)



(a)



(b)

Figure 6: Dependence of the size of the data structure on number of independent variables and order of derivatives

the differential tuples technique. For second and higher order derivatives the speed increase can be roughly estimated by the ratio of the number of multiplication operations used by each algorithm in these lines. Since line 20 is executed more times than line 16 our estimation will rely on comparison of line 20 in both algorithms. There are three multiplication operations in line 20 of the algorithm implementing differential tuple technique. The Taylor coefficient approach eliminates one multiplication operation. It gives us a rough value for the speed increase: $\frac{3}{2} = 1.5$.

Unfortunately, we cannot gain the speed for free — Figure 6 illustrates that the auxiliary data structure grows exponentially with order of the derivatives, and it has polynomial growth with the number of independent variables. But for a reasonably small number of independent variables and order of derivatives, the size of the data structure remains relatively small. For example, in the case of five independent variables and seventh order derivatives the data structure, which includes arrays M, I, C, S, and D, needs only 158,500 bytes to be stored. The size in bytes of the data structure for different numbers of independent variables and orders of the derivatives is given in Tables 8 and 9 for differential tuple technique and Taylor coefficient approach respectively.

Part II
Automatic Differentiation Software for Engineering Analysis

4 Fast Forward Automatic Differentiation Library

In this Section we discuss the Fast Forward Automatic Differentiation Library which is developed at University of Wisconsin-Madison and can be downloaded from http://sal-cnc.me.wisc.edu. The library implements a *tuple* class and several utility functions. All automatic differentiation functions are written in the C++ programming language overloading the arithmetic operations and elementary functions in order to deal with differential tuples. Overloading is the powerful tool, which allows us to preserve the usual notation for the mathematical expressions and hide from the user all details of the algorithms. The tuple class uses the data structure discussed in Section 2. The number of independent variables and the maximal order of the partial derivatives are designated as the *static* variables, and therefore they are available for all instances of the class. Moreover, each particular instance of the tuple class includes the storage for the partial derivatives and the order of the highest derivative, which may be different from the maximal order.

4.1 Initialization and Control Functions

Function	**Short Explanation**
`void SetTupleDimensionOrder( int dim, int max_order );`	Initialization of the data structure
`void DeleteAD( void );`	Deallocation of memory
`int GetTupleMaxOrder( void );`	Retrieval of the maximum order of the partial derivatives
`int GetTupleCurrentSize( void );`	Retrieval of the size of a differential tuple
`void IncTupleCurrentOrder( void );`	Increment the current order of the derivatives by 1
`void IncTupleCurrentOrder( int inc );`	Increment by `inc` the current order of the derivatives
`void DecTupleCurrentOrder( void );`	Decrement the current order of the derivatives by 1
`void DecTupleCurrentOrder( int dec );`	Decrement by `dec` the current order of the derivatives

Before using the tuple class, we need to initialize the data structure by calling the function `void SetTupleDimensionOrder( int dim, int max_order)`. It has two parameters — `int dim` — the number of independent variables (dimension of the space) and `int max_order` — the maximal order of the partial derivatives, which will be computed by the program. All differential tuples created later contain the partial derivatives with respect to all independent variables. This means, once the number of independent variables is set, it is impossible to change it without destroying the data structure by the function `void DeleteAD( void )`.

The order of the highest derivatives — or using our terminology, the order of a differential tuple, which is defined by the private variable `current_order` — may vary during the computations, but it cannot exceed the value of `max_order`. Functions `IncTupleCurrentOrder` and `DecTupleCurrentOrder` increment and decrement the value of the private variable `current_order` by one if the functions have no parameters or by the value passed to the functions.

The function `int GetTupleMaxOrder( void )` returns the maximal order of the differential tuples, and the function `int GetTupleCurrentSize( void )` gives the size of the storage for partial derivatives, which corresponds to the order `current_order` of the differential tuples.

4.2 Constructors and Destructor

Function	Short Explanation
`tuple( void );`	Default constructor
`tuple( int order );`	Constructor creating the differential tuple of the given order
`tuple( const tuple &t );`	Copy constructor
`~tuple( void );`	Destructor

A differential tuple may be created by one of the constructors presented in the tuple class. The default constructor `tuple(void)` allocates memory for the partial derivatives up to order defined by the variable `current_order`. If a differential tuple having order different from the one stored in the variable `current_order` needs to be created, the user should use the constructor `tuple( int order )`. However, the order of the resulting tuple may not exceed the value set beforehand by the function `SetTupleDimensionOrder`. These two constructors assign zero values to the all entries of the differential tuple.

The copy constructor `tuple( const tuple &t )` generates a new instance of the tuple class and copies there all data from the differential tuple `t` passed as the argument.

The destructor `~tuple( void )` frees the memory used by the tuple class instance.

4.3 The Auxiliary Member Functions

Function	Short Explanation
`int tuple::GetOrder( void );`	Retrieval of the order of a differential tuple
`int tuple::GetDim( void );`	Retrieval of the number of independent variables
`int tuple::GetSize( void );`	Retrieval of the size of a differential tuple
`void tuple::Print( void );`	Printing of the differential tuple
`void tuple::Print( FILE *fp );`	Printing of the differential tuple into file

The tuple class has a few member functions providing information about the particular differential tuple such as order, size of the data structure, and number of independent variables. The functions `void tuple::Print( void )` and `void tuple::Print(FILE *fp)` print all partial derivatives stored in the differential tuple either to the display or into a file.

4.4 Assignment Operators

Function	Short Explanation
`tuple &operator =( const tuple &t );` `tuple &operator =( double t );`	Assignment operator
`tuple &operator +=( const tuple &t );` `tuple &operator +=( double t );`	Add-and-assign operator
`tuple &operator -=( const tuple &t );` `tuple &operator -=( double t );`	Subtract-and-assign operator
`tuple &operator *=( const tuple &t );` `tuple &operator *=( double t );`	Multiply-and-assign operator
`tuple &operator /=( const tuple &t );` `tuple &operator /=( double t );`	Divide-and-assign operator

The tuple class provides the standard set of assignment operators: do-and-assign as well as the conventional assignment operators. The operator `tuple &operator =( const tuple &t )` adjusts the order of the resulting differential tuple and copies there all data from its argument `t`.

The assignment operators, which have arguments of the type `double` , change neither the order of the resulting differential tuple nor its size. The operator `tuple &operator =( double t )` puts the value of `t` into the zero-th entry of the resulting differential tuple and sets zero values to the other entries representing partial derivatives.

Operators `tuple &operator +=( double t )` and `tuple &operator -=( double t )` change only the zero-th entry of the resulting differential tuple by adding or subtracting the value of their argument `t`. However, the operators `tuple &operator *=( double t )` and `tuple &operator /=( double t )` multiply/divide all elements of the differential tuple by the value of `t`.

The do-and-assign operators having differential tuples as their arguments adjust the order of the resulting differential tuple choosing either the order of the result or the order of the argument, whichever is less. Implementation of the add-and-assign and subtract-and-assign operators is straightforward: they change the resulting differential tuple by adding/subtracting the corresponding elements of the differential tuples. On the other hand, the multiply-and-assign and divide-and-assign operators utilize the automatic differentiation algorithms described in Section 2.

4.5 Arithmetic Operators

Function	Short Explanation
`friend tuple operator +(const tuple &t1, const tuple &t2);`	Addition of two differential tuples
`friend tuple operator +(double t1, const tuple &t2);`	Addition of a constant to a differential tuple
`friend tuple operator +(const tuple &t1, double t2);`	
`friend tuple operator -(const tuple &t1, const tuple &t2);`	Subtraction of a differential tuple from another one
`friend tuple operator -(double t1, const tuple &t2);`	Subtraction of a differential tuple from a constant
`friend tuple operator -(const tuple &t1, double t2);`	Subtraction of a constant from a differential tuple
`friend tuple operator *(const tuple &t1, const tuple &t2);`	Multiplication of two differential tuples
`friend tuple operator *(double t1, const tuple &t2);`	Multiplication of a differential tuple by a constant
`friend tuple operator *(const tuple &t1, double t2);`	
`friend tuple operator /(const tuple &t1, const tuple &t2);`	Fraction of two differential tuples
`friend tuple operator /(double t1, const tuple &t2);`	Division of a constant by a differential tuple
`friend tuple operator /(const tuple &t1, double t2);`	Division of a differential tuple by a constant
`friend tuple operator -(const tuple &t);`	Negation (change of the sign or multiplication by -1)
`friend tuple Add(const tuple* t, ... );`	Multiplace addition

The tuple class offers the arithmetic operators, which can mix differential tuples and constants in the expressions The order of the resulting differential tuple depends on the orders of the arguments of the operator: if one of the arguments is a constant the resulting tuple will have the order of another argument. In the case when both arguments are differential tuples, the order of the resulting differential tuple takes on the value of the minimum order of the arguments.

Among arithmetic operators familiar to the reader, it is worth highlighting the operator `friend tuple Add( const tuple* t, ... )` performing multiplace addition. This operator makes the computations faster by performing the summation of several differential tuples in one loop. Because the multiplace addition operator may take any number of parameters, the last parameter in the list of the arguments has to be NULL.

4.6 Elementary Functions

Function	**Short Explanation**
`friend tuple sqrt( const tuple &t );`	Square root function
`friend tuple square( const tuple &t );`	Square function
`friend tuple pow( const tuple &t, double s );` `friend tuple pow( const tuple &a, const tuple &b );`	Power function
`friend tuple exp( const tuple &t );`	Exponential function
`friend tuple log( const tuple &t );`	Natural logarithm function
`friend tuple sin( const tuple &t );`	Sine function
`friend tuple cos( const tuple &t );`	Cosine function
`friend tuple tan( const tuple &t );`	Tangent function
`friend tuple ctan( const tuple &t );`	Cotangent function
`friend tuple asin( const tuple &t );`	Inverse sine funtion
`friend tuple acos( const tuple &t );`	Inverse cosine function
`friend tuple atan( const tuple &t );`	Inverse tangent function
`friend tuple actan( const tuple &t );`	Inverse cotangent function
`friend tuple sinh( const tuple &t );`	Hyperbolic sine function
`friend tuple cosh( const tuple &t );`	Hyperbolic cosine function
`friend tuple tanh( const tuple &t );`	Hyperbolic tangent function
`friend tuple ctanh( const tuple &t );`	Hyperbolic cotangent function
`friend tuple asinh( const tuple &t );`	Inverse hyperbolic sine function
`friend tuple acosh( const tuple &t );`	Inverse hyperbolic cosine function
`friend tuple atanh( const tuple &t );`	Inverse hyperbolic tangent function
`friend tuple actanh( const tuple &t );`	Inverse hyperbolic cotangent function
`friend tuple abs( const tuple &t );`	Absolute value function
`friend tuple CutNegative( const tuple &t );`	$f = \frac{t+\|t\|}{2}$
`friend tuple CutPositive( const tuple &t );`	$f = \frac{t-\|t\|}{2}$
`friend double sign( const tuple &t );`	Signum function
`friend void Argument( tuple &t, int dim, double value );`	Generation of a differential tuple corresponding to the dim-th independent variable

The tuple class supplies the set of elementary functions, which includes square root function, power function, exponential and logarithmic functions, trigonometric and inverse trigonometric functions, hyperbolic and inverse hyperbolic functions. These functions employ the automatic differentiation algorithm described in Section 2. As we have seen earlier, the automatic differentiation algorithms differ for each elementary function by the block "Computation of the summation term". For each elementary function this block utilizes the corresponding expressions for the summation term that are shown in the Appendix. None of the elementary functions affects the order of resulting differential tuple: it remains the same as the order of the argument of the function.

Among the elementary functions having continuous derivatives there are a few functions, which may not be differentiable at some points. For example the function `friend tuple abs( const tuple &t )`, returning the

absolute value of a differential tuple, is defined as follows:

$$\texttt{abs(t)} = \begin{cases} t & \text{if t.storage[0]} \geq 0, \\ -t & \text{if t.storage[0]} < 0. \end{cases}$$

At the points where the zero-th entry of t is zero the derivatives have discontinuity.

The functions `friend tuple CutNegative( const tuple &t )` and `friend tuple CutPositive( const tuple &t )` given by the following rules:

$$\texttt{CutNegative(t)} = \begin{cases} t & \text{if t.storage[0]} \geq 0; \\ 0 & \text{if t.storage[0]} < 0; \end{cases}$$

$$\texttt{CutPositive(t)} = \begin{cases} t & \text{if t.storage[0]} \leq 0; \\ 0 & \text{if t.storage[0]} > 0; \end{cases}$$

have discontinuous partial derivatives when the zero-th entry of t is zero.

The tuple class also provides the signum function `friend double sign( const tuple &t )` returning one of three values -1, 0 or 1 depending on the sign of the zero-th entry of the argument differential tuple:

$$\texttt{sign(t)} = \begin{cases} 1 & \text{if t.storage[0]} > 0, \\ 0 & \text{if t.storage[0]} = 0, \\ -1 & \text{if t.storage[0]} < 0. \end{cases}$$

Probably, the most important function among the elementary functions is `friend void Argument( tuple &t, int dim, double value )`, which generates a differential tuple for the dim-th independent variable setting the value of `value` to the zero-th entry of the differential tuple `t`.

4.7 Differentiation Operators

Function	**Short Explanation**
`friend tuple dx(const tuple &t, int n);`	Differentiation operator $f = \frac{\partial t}{\partial x_n}$
`friend double dx_value(const tuple &t, int n);`	
`friend tuple dx(const tuple &t, int order, int n);`	Differentiation operator $f = \frac{\partial^{order} t}{\partial x_n^{order}}$
`friend double dx_value(const tuple &t, int order, int n);`	
`friend double dx_direct(const tuple *t, int n);`	Direct access to the elements of a differential tuple
`friend double GetDerivative(const tuple &t, int* miu);`	Retrieval of the partial derivative defined by the array miu
`double tuple::GetDerivative(int* miu);`	
`void tuple::SetDerivative(int* miu, double value);`	Setting the value to the partial derivative defined by the array miu
`double tuple::GetValue(void);`	Getting the value of the zero-th entry of a differential tuple
`void tuple::SetFunctionValue(double value);`	Setting the value to the zero-th entry of a differential tuple
`friend double Grad(const tuple &t, double *gradient);`	Gradient of a differential tuple
`double tuple::Grad(double *gradient);`	
`friend tuple grad2(const tuple &t);`	Square of the gradient of a differential tuple
`friend tuple d1(const tuple &t1, const tuple &t2);`	Operator $D1$

This group of functions provides the access to the partial derivatives stored in a differential tuple. The differential operators `friend tuple dx(const tuple &t, int n)` and `friend tuple dx(const tuple &t, int order, int n)` return the differential tuples corresponding to the partial derivatives of the first and the `order`th orders of the tuple `t` with respect to the nth independent variable:

$$\texttt{dx}(\texttt{t},\texttt{n}) = \frac{\partial t}{\partial x_n};$$

$$\texttt{dx}(\texttt{t},\texttt{order},\texttt{n}) = \frac{\partial^{order} t}{\partial x_n^{order}}.$$

The functions `friend double dx_value(const tuple &t,int n)` and `friend double dx_value( const tuple &t, int order, int n)` do not produce any differential tuples, rather they retrieve just the numerical values of the partial derivatives $\frac{\partial t}{\partial x_n}$ and $\frac{\partial^{order} t}{\partial x_n^{order}}$ from the differential tuple t.

The function `friend double dx_direct(const tuple *t, int n)` provides the direct access to the elements of the differential tuple using the parameter n as the number of the entry, which has to be retrieved. For example the call `dx_direct(&t, 0)` gives the value of the zero-th element of the differential tuple t. Despite the fact that function `dx_direct` is the fastest way to access the elements of a differential tuple, it is worth pointing out that the user should be very cautious using this function because the meaning of its result depends on the number of the independent variables. For example, in the case of two independent variables, the fifth entry of a differential tuple corresponds to the partial derivative $\frac{\partial^2}{\partial x_2^2}$ but for three independent variables the same entry corresponds to $\frac{\partial^2}{\partial x_1 \partial x_2}$.

The differentiation operators `friend double GetDerivative(const tuple &t, int* miu)` and `double tuple::GetDerivative (int* miu)` retrieve the values of the partial derivatives from a differen-

tial tuple using the multi-index array miu to define the order of the derivative. The multi-index array `miu` does not use its zero entry. The orders of the partial derivatives are stored starting from the first entry: the first entry contains the order with respect to the first independent variable, the second — with respect to the second independent variable, etc. This means either the size of the multi-index array has to be one more than the number of independent variables, or its size may be the same as the number of independent variables, but with the pointer to the array needs adjusted as described in [16].

The member function `void tuple::SetDerivative(int* miu, double value)` uses the same addressing scheme in order to set the value of the partial derivative defined by the array `miu`.

The member functions `double tuple::GetValue(void)` and `void tuple::SetFunctionValue( double value)` retrieve/set the value of the zero-th entry of the differential tuple. This entry stores the value of a function — its zero-th order derivative.

The functions `friend double Grad(const tuple &t, double *gradient)` and `double tuple::Grad(double *gradient)` return the absolute value of the gradient and the first partial derivatives of a differential tuple with respect to all independent variables, which are stored in the array `gradient`. The function `friend tuple grad2(const tuple &t)` computes the square of the gradient of the differential tuple t.

The function `friend tuple d1(const tuple &t1, const tuple &t2)` returns the dot product of the gradients of its arguments: $d1(t_1, t_2) = \nabla t_1 \nabla t_2$. Sometimes this function is called operator $D1$ [20].

4.8 The R-functions

Function	**Short Explanation**
`friend tuple operator &( const tuple &t1, const tuple &t2 );`	R_0-conjunction
`friend tuple operator |( const tuple &t1, const tuple &t2 );`	R_0-disjunction
`friend tuple equiv( const tuple &t1, const tuple &t2 );`	R-equivalence
`friend tuple operator !( const tuple &t );`	R-negation

An R-function is a real-valued function whose sign is completely determined by the signs of its arguments [20]. The sign of a function can be considered as its logical property: negative values of a function can be associated with logical FALSE, and positive values can be associated with logical TRUE.

In other words, an R-function works as a Boolean switching function, changing its sign only when its arguments change their signs; it can be regarded as "on" or "off" (or "true" or "false") depending on the values of the input variables. For example, the function xyz can be negative only when the number of its negative arguments is odd. As another example, $\min(x_1, x_2)$ is an R-function whose companion Boolean function is logical "and" ($\wedge$) (logical conjunction), and $\max(x_1, x_2)$ is an R-function whose companion Boolean function is logical "or" ($\vee$) (logical disjunction). This is seen in that function $\min(x_1, x_2)$ takes on positive values only when x_1 AND x_2 are positive; similarly function $\max(x_1, x_2)$ takes on positive values when x_1 OR x_2 are positive. Except the functions $\min(x_1, x_2)$ and $\max(x_1, x_2)$ there are infinitely many R-functions with different differential properties [20, 29]. Here is the most popular system of R-functions:

$$t_1 \wedge_0 t_2 \equiv t_1 + t_2 - \sqrt{t_1^2 + t_2^2}, \tag{17}$$

$$t_1 \vee_0 t_2 \equiv t_1 + t_2 + \sqrt{t_1^2 + t_2^2}, \tag{18}$$

where the symbol $\wedge_0$ defines the R_0-conjunction, and the symbol $\vee_0$ — the R_0-disjunction.

The theory of R-functions was developed in Ukraine by Rvachev and his students [20]. The R-functions are used to construct functions which *implicitly* define some geometrical objects [29]. The term *implicitly* means that the zero set of the function coincides with the boundary of some geometrical object. The reader can find more about R-functions and their properties in [20, 27, 29].

The operators `friend tuple operator &(const tuple &t1, const tuple &t2)` and `friend tuple operator |(const tuple &t1, const tuple &t2)` perform automatic differentiation of the R_0-functions (17) and (18).

The function `friend tuple equiv(const tuple &t1, const tuple &t2)` implements the R_0-equivalence operation:

$$f = \frac{t_1 t_2}{t_1 + t_2}.$$

The R-negation operator `friend tuple operator !(const tuple &t)` produces a differential tuple whose entries have the same absolute values as the elements of the differential tuple t but have the opposite signs.

4.9 The Normalized Geometrical Primitives

Function	**Short Explanation**
`friend tuple line( double x, double y, double x1, double y1, double x2, double y2 );`	Function implicitly defining a line going through the points $(x1, y1)$ and $(x2, y2)$
`friend tuple circle( double x, double y, double xc, double yc, double r );`	Function implicitly defining a circle of radius r with center at the point (xc, yc)
`friend tuple bandx( double x, double y, double yc, double a );`	Function implicitly defining a strip parallel to the X axis
`friend tuple bandy( double x, double y, double xc, double a );`	Function implicitly defining a strip parallel to the Y axis

This group of functions perform automatic differentiation of the functions implicitly defining the following primitive geometrical objects:

- the halfspace whose boundary is the line going through the points $(x1, y1)$ and $(x2, y2)$

$$\texttt{line}(\texttt{x}, \texttt{y}, \texttt{x1}, \texttt{y1}, \texttt{x2}, \texttt{y2}) = \frac{(y - y1)(x2 - x1) - (x - x1)(y2 - y1)}{\sqrt{(x2 - x1)^2 + (y2 - y1)^2}}$$

- the halfspace bounded by the circle of radius r with center at (xc, yc)

$$\texttt{circle}(\texttt{x}, \texttt{y}, \texttt{xc}, \texttt{yc}, \texttt{r}) = \frac{r^2 - (x - xc)^2 - (y - yc)^2}{2r}$$

- the horizontal strip with the center line $y = yc$ and half-width a

$$\texttt{bandx}(\texttt{y}, \texttt{yc}, \texttt{a}) = \frac{a^2 - (y - yc)^2}{2a}$$

- the vertical strip with the center line $x = xc$ and half-width a

$$\texttt{bandy}(\texttt{x}, \texttt{xc}, \texttt{a}) = \frac{a^2 - (x - xc)^2}{2a}.$$

Later, these functions may be combined using R-functions into the implicit equation of complex geometrical object.

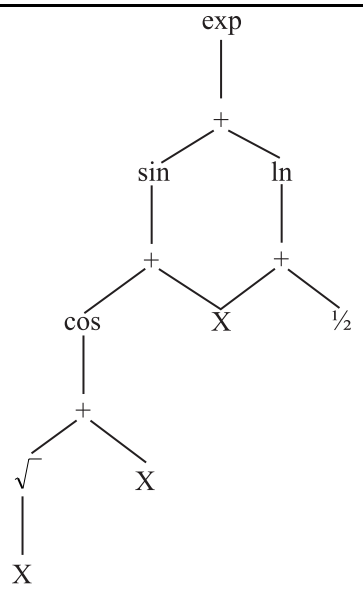


Figure 7: The computational graph of the function $f = e^{\sin(x+\cos(x+\sqrt{x}))+\ln(x+\frac{1}{2})}$

5 Applications of the Fast Forward Automatic Differentiation Library

Example 1. Let us explain the usage of the proposed Fast Forward Automatic Differentiation Library using as an example differentiation of function (16). The following C++ program computes derivatives of the function up to fifth order and stores them into files:

```
#include "tuple.h"
#include <stdio.h>
#include <math.h>

tuple f(double x)
{
    tuple X, result;
    Argument(X, 1, x);
    result = exp(sin(X+cos(X+sqrt(X)))+log(X+0.5));
    return result;
}

int main(void)
{
    FILE *fp_ad_x1, *fp_ad_x2, *fp_ad_x3, *fp_ad_x4, *fp_ad_x5;
    double Xmin, Xmax, Dx, x;
    int i,n;
    SetTupleDimensionOrder(1,5);
    tuple y;
    Xmin = 1.0;
    Xmax = 10.0;
    n = 1000;
    Dx = (Xmax - Xmin)/(n-1);
    fp_ad_x1 = fopen("fx1_ad.dat","w");
    fp_ad_x2 = fopen("fx2_ad.dat","w");
```

```
    fp_ad_x3 = fopen("fx3_ad.dat","w");
    fp_ad_x4 = fopen("fx4_ad.dat","w");
    fp_ad_x5 = fopen("fx5_ad.dat","w");
    for( i=0; i<n; i++ )
    {
        x = Xmin + Dx*i;
        y = f(x);
        fprintf(fp_ad_x1, "%le %le\n", x, dx_value(y,1,1));
        fprintf(fp_ad_x2, "%le %le\n", x, dx_value(y,2,1));
        fprintf(fp_ad_x3, "%le %le\n", x, dx_value(y,3,1));
        fprintf(fp_ad_x4, "%le %le\n", x, dx_value(y,4,1));
        fprintf(fp_ad_x5, "%le %le\n", x, dx_value(y,5,1));
    }
    fclose(fp_ad_x1);
    fclose(fp_ad_x2);
    fclose(fp_ad_x3);
    fclose(fp_ad_x4);
    fclose(fp_ad_x5);
    return 0;
}
```

The function `tuple f(double x)` computes a differential tuple of the function (16) at the point x. First, the function `Argument(X, 1, x)` creates the differential tuple X for the independent variable x. Then at each node of the computational graph (Figure 7) the differential tuple is changed according to the chain rule associated with the particular function. The single line in the program does this work: `result = exp(sin(X + cos(X + sqrt(X))) + log(X + 0.5))`. In the main program the data structure for automatic differentiation is initialized for one independent variable and the highest derivatives of the fifth order by calling the function `SetTupleDimensionOrder(1,5)`. Then in the loop 1000 points ($n = 1000$) are sampled on the segment $[Xmin, Xmax]$ where the derivatives of the function f are computed. For each point the line `y = f(x)` calls the function `tuple f(double x)` and stores the derivatives of the function `f` in the differential tuple `y`. Then differential operator `dx_value` extracts the derivatives from differential tuple `y`. The values of the derivatives along with the value of `x` are put into files.

Example 2. This example explains how functions, which implicitly define the geometrical objects, are constructed and differentiated.

According to the theory of R-functions [20], any set theoretic expression may be converted into a real-valued function such that it is positive at all points where the set theoretic expression takes on "true" values and negative at other points. Constructive Solid Geometry (CSG) represents geometrical objects via union and intersection of the primitive regions (halfspaces). Being a pure set theoretic representation, the CSG model may be represented by a real-valued function which is constructed using the R-functions (17, 18). For instance, the Boolean expression

$$\Omega = \Omega_1 \cap \overline{(\Omega_2 \cap (\Omega_3 \cup \Omega_4) \cap \Omega_5)}, \tag{19}$$

defines the domain shown in Figure 8. The logical functions $\Omega_i = (\omega_i(x, y) \geq 0)$, $i = 1, \ldots, 5$ describe the halfspaces bounding the domain, and the functions $\omega_i(x, y)$, which are:

$$\begin{aligned}
\omega_1(x, y) &= \frac{1 - x^2 - y^2}{2}; \\
\omega_2(x, y) &= \frac{0.25^2 - x^2}{0.5}; \\
\omega_3(x, y) &= -\frac{x + y}{\sqrt{2}}; \\
\omega_4(x, y) &= \frac{x - y}{\sqrt{2}}; \\
\omega_5(x, y) &= y + 0.25;
\end{aligned} \tag{20}$$

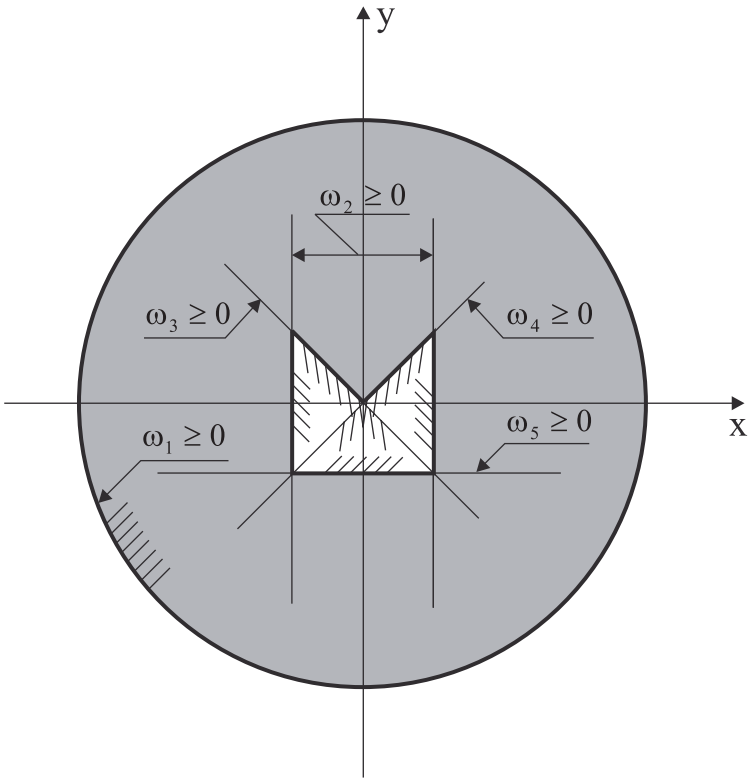


Figure 8: The geometric domain

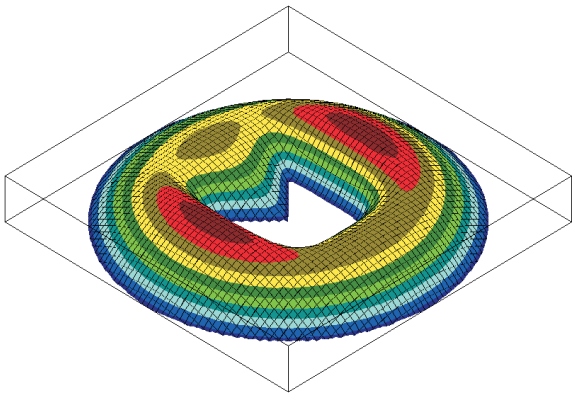
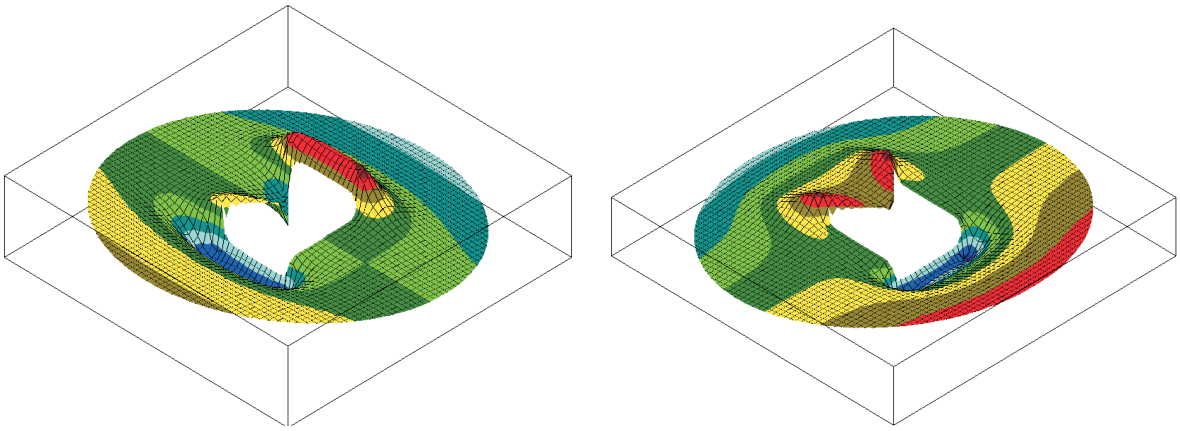


Figure 9: The function implicitly defining the geometric domain shown in Figure 8



(a) (b)

Figure 10: The first partial derivatives of the function shown in Figure 9: (a) $\frac{\partial \omega}{\partial x}$; (b) $\frac{\partial \omega}{\partial y}$

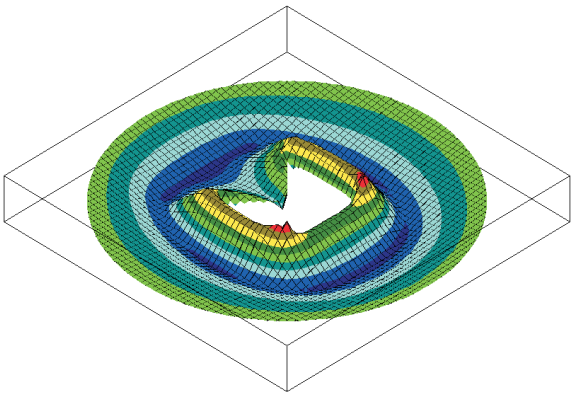


Figure 11: The absolute value of the gradient of the function shown in Figure 9

implicitly represent the boundaries of the halfspaces. To obtain the function implicitly defining the geometric object we substitute in the Boolean expression of the real-valued functions ω_i for the logical functions, and R-operations for the Boolean operations. Having done this for the Boolean expression (19), we get the function

$$\omega(x,y) = \omega_1(x,y) \wedge_0 (-(\omega_2(x,y) \wedge_0 (\omega_3(x,y) \vee_0 \omega_4(x,y)) \wedge_0 \omega_5(x,y))). \tag{21}$$

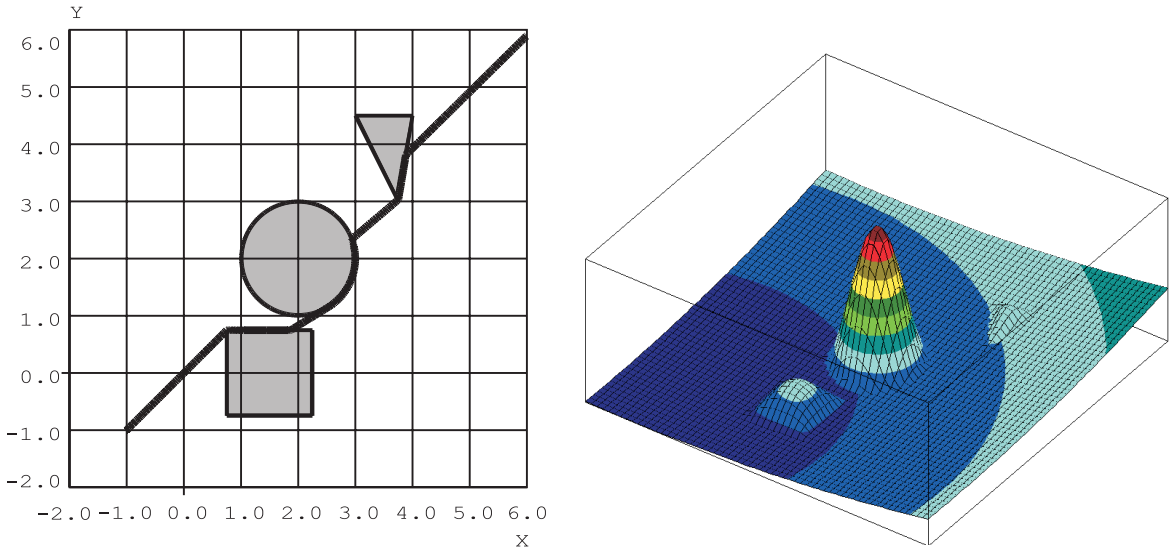
The complement operation in (19) corresponds to the change of the sign in (21). The symbols $\wedge_0$ and $\vee_0$ designate the R_0-conjunction (17) and the R_0-disjunction (18) respectively. Figure 9 shows the plot of the positive portion of the function ω (21).

The following C++ code performs the automatic differentiation of this function:

```
tuple omega(double x, double y)
{
    tuple omega;
    omega = circle(x,y, 0.0,0.0,1.0) &
            ((!(bandy(x,y, 0.0,0.25) &
                (line(x,y, 0.0,0.0, -0.25,0.25) |
                 line(x,y, 0.25,0.25, 0.0,0.0)) &
                line(x,y, -0.25,-0.25, 0.25,-0.25)));
    return omega;
}
```

Plots of the first partial derivatives $\frac{\partial \omega}{\partial x}$ and $\frac{\partial \omega}{\partial y}$ of the function (21) are given in Figure 10. Figure 11 shows the plot of the absolute value of gradient of the ω function.

For the sake of clarity in this example we manually constructed the function (21), which implicitly defines the boundary of the relatively simple geometric domain. However, such functions can be constructed *automatically* for virtually any geometric object using R-functions [28, 29].



(a) (b)

Figure 12: (a) The robot's path among the obstacles; (b) the potential function used for the path planning

Example 3. This example explains the application of the automatic differentiation toolkit to robot motion planning using potential field methods [12]. These methods employ a potential function whose gradient gives the preferable direction of the motion. The potential function U is superposition of the attractive potential U_{att} and the repulsive potential U_{rep} [12]:

$$U = U_{att} + U_{rep}.$$

The attractive potential is usually defined as a parabolic well

$$U_{att} = \frac{1}{2}\varepsilon\left((x - x_{goal})^2 + (y - y_{goal})^2\right), \tag{22}$$

here ε is a positive scaling factor; (x_{goal}, y_{goal}) are coordinates of the goal point. In our example $\varepsilon = \frac{1}{500}$; $x_{goal} = -1$; $y_{goal} = -1$.

The repulsive potential function describes the positions of the obstacles, and it can be constructed using R-functions. In our example we use three obstacles shown in Figure 12(a): triangular domain with vertices at points $(3.0, 4.5)$, $(4.0, 4.5)$, and $(3.75, 3.0)$; a circular disk of unity radius with center at $(2, 2)$; a square with vertices at the points $(2.25, 0.75)$, $(2.25, -0.75)$, $(0.75, -0.75)$, $(0.75, 0.75)$. The triangular domain is described as an intersection of three linear halfspaces:

$$\begin{aligned}\omega_1 =& (4.5 - y) \wedge_0 \frac{(y - 4.5)(3.75 - 3.0) - (x - 3.0)(3.0 - 4.5)}{\sqrt{(3.75 - 3.0)^2 + (3.0 - 4.5)^2}} \wedge_0 \\ & \frac{(y - 3.0)(4.0 - 3.75) - (x - 3.75)(4.5 - 3.0)}{\sqrt{(4.0 - 3.75)^2 + (4.5 - 3.0)^2}}.\end{aligned} \tag{23}$$

The function taking on zero value on the boundary of the square is defined as R_0-conjunction of two strips:

$$\omega_2 = \frac{0.75^2 - (y - 0.0)^2}{1.5} \wedge_0 \frac{0.75^2 - (x - 1.5)^2}{1.5}. \tag{24}$$

And the following function defines the circular disk:

$$\omega_3 = \frac{1.0^2 - (x - 2.0)^2 - (y - 2.0)^2}{2}. \tag{25}$$

The functions (23), (24), and (25) are positive inside the obstacles, negative outside and take on zero value on their boundaries. According to [12] the repulsive potential function has to be zero outside the obstacles, and it should be at least once continuously differentiable. Such repulsive potential function can be easily constructed from the functions (23), (24), (25): we need to "cut" their negative parts and then raise the results at least to the second power (this assures the necessary degree of differentiability). Thus, the repulsive potential function appears as follows:

$$U_{rep} = \left(\frac{\omega_1 + |\omega_1|}{2}\right)^2 + \left(\frac{\omega_2 + |\omega_2|}{2}\right)^2 + \left(\frac{\omega_3 + |\omega_3|}{2}\right)^2 \tag{26}$$

Figure 12(b) presents the plot of the potential function for $x \in [-2, 6]$ and $y \in [-2, 6]$. The following C++ code computes this potential function and its partial derivatives:

```
tuple f(double x, double y)
{
    tuple X, Y, result, rep, att;
// Generation of differential tuples for independent variables
    Argument(X, 1, x);  Argument(Y, 2, y);
// Repulsion potential function
    rep = square(CutNegative( line(x,y, 4.0,4.5, 3.0,4.5)  &
                              line(x,y, 3.0,4.5, 3.75,3.0) &
                              line(x,y, 3.75,3.0, 4.0,4.5) )) +
          square(CutNegative( bandx(y, 0.0, 0.75) & bandy(x, 1.5,0.75) ))+
          square(CutNegative( circle(x,y, 2.0, 2.0, 1.0)) );
// Attractive potential function
    att = (square(Y+1.0) + square(X+1.0))*0.001;
// The potential function is the superposition of the attractive and
// repulsion potential functions
    result = att + rep;
    return result;
}
```

For the path planning the depth-first algorithm may be used, which performs the motion in the direction opposite the gradient of the potential function:

$$x_{i+1} = x_i - \delta \frac{\frac{\partial U}{\partial x}}{\sqrt{\left(\frac{\partial U}{\partial x}\right)^2 + \left(\frac{\partial U}{\partial y}\right)^2}}; \qquad y_{i+1} = y_i - \delta \frac{\frac{\partial U}{\partial y}}{\sqrt{\left(\frac{\partial U}{\partial x}\right)^2 + \left(\frac{\partial U}{\partial y}\right)^2}}, \tag{27}$$

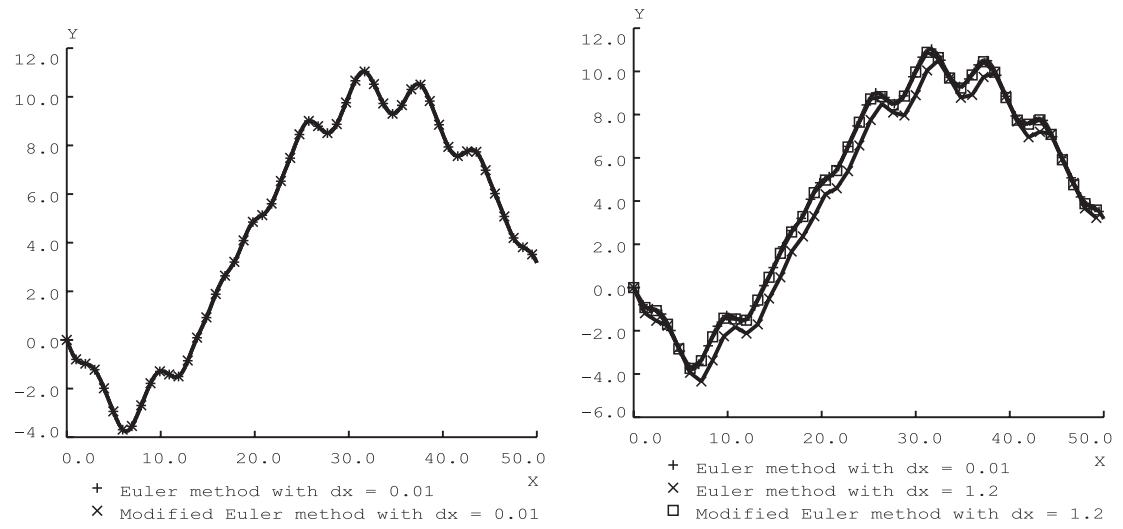
where δ is the size of the robot's step. The algorithm starts at the point (x_0, y_0) (in our example it is the point $(6.1, 6.0)$) and the iterations (27) are performed until the local minimum of the potential function is achieved. The algorithm with variable step size is implemented in the following portion of the C++ code:

```
#include "tuple.h"
#include <stdio.h>
#include <math.h>

void main(void)
{
    double x1,y1,x2,y2,f1,f2, step, eps, g_abs;
    double *grad;
    FILE *fp;
//  Initialization of the data structure
//  2 independent variables; maximal order of the derivatives -- 1
    SetTupleDimensionOrder(2,1);
    tuple z;
//  Allocation of the storage for gradient
    grad = new double[2]; grad--;
    x2 = 6.1;   y2 = 6.0;   // The starting point
    step = .01;             // The step size
    eps  = 1e-5;            // The desired accuracy
//  Open file to track the sequence of points
    fp = fopen("path.dat","w");
//  The depth-first motion planning algorithm
    z = f(x2,y2);
    f2 = z.GetValue();
    do
    {
        f1 = f2; x1 = x2; y1 = y2;
        g_abs = z.Grad(grad);           // Compute the gradient of f
        x2 = x1 - step*grad[1]/g_abs ;  // Next point
        y2 = y1 - step*grad[2]/g_abs;
        z = f(x2,y2);
        f2 = z.GetValue();
        if( fabs(f2) > fabs(f1) )
        {
            step /= 2.0;
        }
        fprintf(fp,"%14.7le %14.7le\n",x2,y2);
    }while( sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)) > eps );
    fclose(fp);
//  Destruction of the automatic differentiation data structure
```

```
    DeleteAD();
    grad++; delete [] grad; grad = NULL;
    return;
}
```

Figure 12(a) shows the robot's path from point $(6.1, 6.0)$ to point $(-1, -1)$ among the obstacles.



(a) (b)

Figure 13: (a) The solutions of the differential equation (32) given by the Euler and modified Euler methods with step size $\Delta x = 0.01$; (b) comparison of the solutions of the differential equation (32) given by the Euler and modified Euler methods with step size $\Delta x = 1.2$ with the solution obtained for the step size $\Delta x = 0.01$

Example 4. Automatic differentiation techniques, as it was shown in [17, 7, 8, 9, 19], could be also used to improve numerical methods of solving ordinary differential equations. Let us, for example, solve the following initial value problem:

$$\begin{aligned} &\frac{dy}{dx} = \varphi(x); \\ &y_{|x=x_0} = y_0. \end{aligned} \tag{28}$$

According to the Euler method [6] the solution of the problem (28) is given by the formula:

$$y(x + \Delta x) = y(x) + \varphi(x)\Delta x. \tag{29}$$

Formula (29) gives the first order approximation of the solution. In fact, this is the Taylor series expansion of the solution in the neighborhood of x. To achieve higher order of the approximation we need to use more terms in the Taylor series:

$$y(x + \Delta x) = y(x) + \sum_{i=1}^{N} \frac{1}{i!} \frac{d^i y}{dx^i} \Delta x^i + O(\Delta x^{N+1}). \tag{30}$$

Since $\frac{dy}{dx} = \varphi(x)$ the higher derivatives of y can be derived by differentiation of the function $\varphi(x)$ [19]: $\frac{d^i y}{dx^i} = \frac{d^{i-1}\varphi(x)}{dx^{i-1}}$. Thus, the solution of the initial value problem can be represented as follows:

$$y(x + \Delta x) = y(x) + \sum_{i=1}^{N} \frac{1}{i!} \frac{d^{i-1}\varphi(x)}{dx^{i-1}} \Delta x^i + O(\Delta x^{N+1}). \tag{31}$$

The utilization of the developed automatic differentiation library makes trivial the computations of the derivatives $\frac{d^{i-1}\varphi(x)}{dx^{i-1}}$: their are computed during the evaluation of the function $\varphi(x)$.

To show the advantage of the modified Euler method let us solve the following initial value problem:

$$\begin{aligned} &\frac{dy}{dx} = \cos(\sin(x) + \sqrt{x+1} + 2); \\ &y_{|x=0} = 0. \end{aligned} \tag{32}$$

Here is a C++ program which solves the problem (32):

```
#include"tuple.h"
#include <stdio.h>
#include <math.h>

tuple fi(double x);
void ode_Euler( double x_start, double y_start, double x_end,
                double dx, tuple f(double),const char* file);
void ode_Euler_modified( double x_start, double y_start, double x_end,
                         double dx, int max_order, tuple f(double),
                         const char* file);
void main()
{
    double dx, y_start, x_start, x_end;

//  Initialization of the data structure for one independent
//  variable and fifth order derivatives
    SetTupleDimensionOrder( 1, 5 );

//  Definition of the initial value
    y_start = x_start = 0.0;
//  End point
    x_end = 50.0;
//  Step of the discretization
    dx = 1.2;
//  Solution of an ODE via Euler method
    ode_Euler( x_start, y_start, x_end, dx, fi,"euler.dat");
//  Solution of an ODE via modified Euler method
    ode_Euler_modified( x_start, y_start, x_end, dx, 5, fi,"euler_m.dat");
    DeleteAD();
    return;
}

// The right site of the differential equation
tuple fi( double x )
{
    tuple result, X;
    Argument(X,1,x);
    result = cos(sin(X)+2.0+sqrt(X+1));
    return result;
}

// Implementation of the Euler method
void ode_Euler( double x_start, double y_start, double x_end,
                double dx, tuple f(double),const char* file)
```

```
{
    int n, i;
    double x, y;
    FILE *fp;
    fp = fopen(file,"w");
    y = y_start;
    x = x_start;
    n = (int)( (x_end-x_start) / dx );
    fprintf(fp,";;Euler method with dx=%f\n",dx);
    fprintf(fp," %f %f\n",x,y);
    for( i = 0; i < n; i++ )
    {
        y += f(x).GetValue() * dx;
        x += dx;
        fprintf(fp," %f %f\n",x,y);
    }
    fclose(fp);
    return;
}

// Implementation of the modified Euler method
void ode_Euler_modified( double x_start, double y_start, double x_end,
                         double dx, int max_order,
                         tuple f(double),const char* file)
{
    int n, i, j;
    double x, y, d, fact;
    tuple fi;
    FILE *fp;
    fp = fopen(file,"w");
    y = y_start;
    x = x_start;
    n = (int)( (x_end-x_start) / dx );
    fprintf(fp,";;Modified Euler method with dx=%f\n",dx);
    fprintf(fp," %f %f\n",x,y);
    for( i = 0; i < n; i++ )
    {
        fi = f(x);
        d  = dx;
        fact = 1;
//  Computation of the increment via evaluation of the Taylor series
        for( j = 1; j<=max_order+1; j++ )
        {
            fact *= j;
            y += dx_value(fi,j-1,1) * d / fact;
            d *= dx;
        }
        x += dx;
        fprintf(fp," %f %f\n",x,y);
    }
    fclose(fp);
    return;
}
```

Functions `ode_Euler` and `ode_Euler_modified` implement Euler and modified Euler methods respectively. Parameters of these functions are coordinates of the starting point (`x_start`, `y_start`), abscissa of the end point `x_end`, discretization step `dx`, the right site of the differential equation (function `f(double)`) and name of a file where the solution is saved to. The function `ode_Euler_modified` has an additional parameter `max_ord` that defines the number of the used terms in Taylor series (30).

Figure 13(a) shows that for small value of the discretization step Δx ($\Delta x = 0.01$) Euler method and the modified Euler method give the same results. However, for bigger values of Δx the modified Euler method is more accurate. Figure 13(b) illustrates the advantage of the modified Euler method: the solution produced by the modified Euler method with $\Delta x = 1.2$ coincides with the solution of the initial value problem obtained using $\Delta x = 0.01$, but the Euler method for $\Delta x = 1.2$ causes the noticeable approximation error.

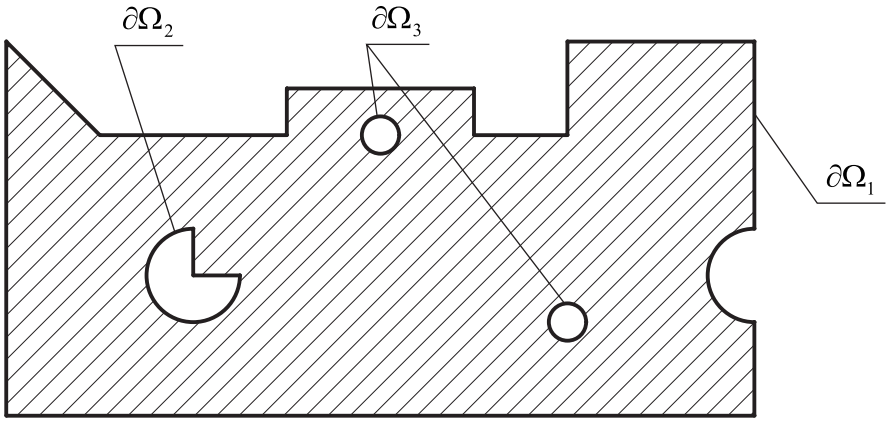
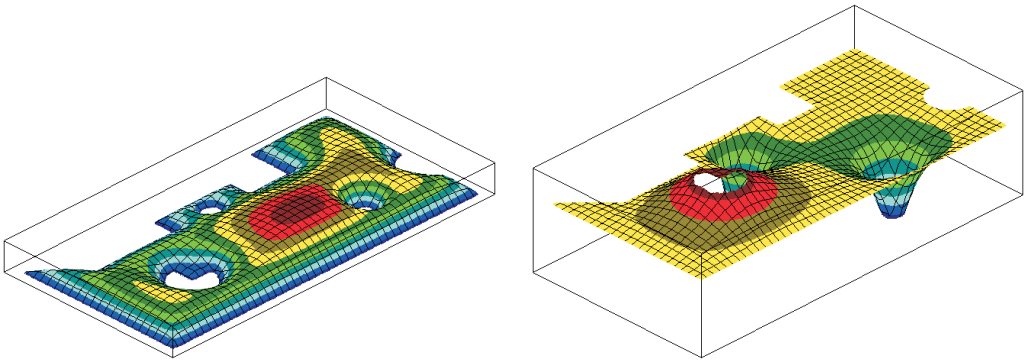


Figure 14: The geometric domain



(a) (b)

Figure 15: (a) The function implicitly defining the boundary of the domain shown in Figure 14; (b) the function transfinitely interpolating the boundary conditions (34)

Example 5. The automatic differentiation techniques are in great demand when the functions to be differentiated are unknown *a priori* and are constructed at run time. Such situation arises, for example, when the R-function meshfree method (RFM) is applied to solve an engineering problem. The method utilizes the distance-like functions taking on zero value on the portions of the boundary of geometric object to construct a solution to the boundary value problem that satisfies the given boundary conditions *exactly* [20, 24]. The RFM approximates the differential equation of the problem using the suitable variational, projection or other methods.

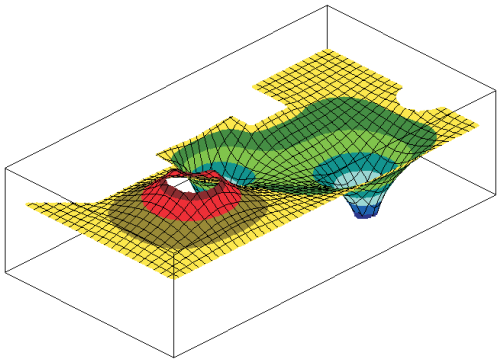


Figure 16: The solution of the boundary value problem

For the sake of simplicity let us solve the Laplace equation

$$\nabla^2 u = 0 \tag{33}$$

with Dirichlet boundary conditions

$$\begin{aligned} u_{|\partial\Omega_1} &= \varphi_1 \equiv 0, \\ u_{|\partial\Omega_2} &= \varphi_2 \equiv 1, \\ u_{|\partial\Omega_3} &= \varphi_3 \equiv -2, \end{aligned} \tag{34}$$

prescribed on the boundaries $\partial\Omega_i$, $i = 1, 2, 3$ of geometric domain shown in Figure 14. According to [20, 24] the *solution structure* satisfying the Dirichlet boundary conditions appears as follows:

$$u = \omega\Phi + \varphi, \tag{35}$$

where the function ω takes on zero value on the boundary of the domain; function φ transfinitely interpolates the functions φ_i given as the boundary conditions [25]. The functions ω and φ are constructed automatically for the given geometry and boundary conditions at run time using techniques described in [29, 25]. Figures 15(a) and (b) present plots of ω and φ for the domain shown in Figure 14 and boundary conditions (34). Function Φ is an unknown function, which makes the function u to be a solution to the differential equation of the problem. Since in many practical situations it is impossible to determine Φ exactly, it can be approximated by a linear combination of the linear independent functions:

$$\Phi = \sum_{i=1}^{k} C_i \chi_i. \tag{36}$$

The functions $\{\chi_i\}_{i=1}^k$ have to be chosen from any complete set of the basis functions. Let us choose the functions $\{\chi_i\}$ to be bicubic B-splines on 60×30 uniform rectangular grid. Substituting the expression (36) into solution structure (35) we obtain:

$$u = \sum_{i=1}^{k} C_i \omega \chi_i + \varphi. \tag{37}$$

The products $\omega\chi_i$ form the new basis whose functions satisfy homogeneous Dirichlet boundary condition exactly. Different sets of the coefficients C_i result in different functions u satisfying the given boundary conditions exactly, but only one set of the coefficients gives the best approximation of the solution to the boundary value problem. This set of the coefficients can be obtained via application, for example, of Ritz method which requires minimization of the following functional:

$$F = \iint_{\Omega} \left(\nabla^2 \omega\Phi\right) \left(\omega\Phi\right) d\Omega - 2 \iint_{\Omega} \left(-\nabla^2 \varphi\right) \left(\omega\Phi\right) d\Omega. \tag{38}$$

As result we obtain a set of the coefficients C_i that give the solution of the boundary value problem (Figure 16).

Functional (38) contains partial derivatives of the basis functions χ_i, of the functions ω and φ. Since the functions ω and φ are unknown a priori they cannot be differentiated symbolically. Only the automatic differentiation techniques allow to compute accurately the partial derivatives of such functions.

The developed automatic differentiation algorithms and software have been successfully used to model temperature field in construction of an internal combustion engine [30] and to solve computational fluid dynamics problems as described in [34].

6 Summary

The automatic differentiation technique described in this document has numerous applications and offers several advantages. In particular, the utilization of the generalized Leibnitz chain rules makes it possible to compute numerical values of the derivatives with the machine precision. Numerous numerical experiments, which we performed in order to test the automatic differentiation algorithms, confirmed their good accuracy.

The Fast Forward Automatic Differentiation Library implements the proposed algorithms. The library offers overloaded elementary functions, arithmetic operators and the utility functions. Since the overloaded elementary functions have the same appearance as the functions from the standard C/C++ library, it does not take much effort to plug the automatic differentiation functions into a program. These functions allow mixing data of the standard type (float, double, integer) and the differential tuples in mathematical expressions. In this case, variables of the standard type are considered to be constants. The user-friendly utility functions provide access to the derivatives stored in the differential tuple offering a variety of addressing schemes. To use the automatic differentiation software on computers with low amounts of memory, the library functions also implement the algorithms described in [32] that do not store the precomputed positions of the derivatives and products of the binomial coefficients. Since both implementations of the automatic differentiation library provide the same user interface, it is easy to switch between implementations: the user simply needs to link either one library or the other without changing the source code.

The developed Fast Forward Automatic Differentiation Library is the essential component of the Semi Analytic Geometry Engine (SAGE) [33]. The SAGE software implements the R-function meshfree method [20, 23, 24, 34] incorporating in one system the geometric engine, automatic differentiation and integration tools. The software prototype, specialized for heat transfer and plate natural vibration problems, is available for public use and can be downloaded at http://sal-cnc.me.wisc.edu. The Fast Forward Automatic Differentiation Library may also find applications in problems requiring the computations of partial derivatives like sensitivity analysis and optimization problems, robot motion planning, root finding algorithms and many others.

Acknowledgments

This work was supported in part by the National Science Foundation grants DMI-9900171, DMI-9522806, CCR-9813507, and the NATO Linkage grant PST.CLG.976192. The authors are grateful to Martin Berz and Malcolm Sabin for reading the earlier draft and providing a number of useful suggestions.

References

[1] M. Berz. Forward algorithms for higher derivatives in many variables with applications to beam physics. In *G. F. Corliss and A. Griewank, editors, Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 147–156, January 6–8 1991.

[2] Martin Berz. The DA precompiler DAFOR. Tech. Report, Lawrence Berkeley National Laboratory, Berkeley, Calif., 1990.

[3] Martin Berz. Calculus and numerics on Levi-Civita fields. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 19–35, Philadelphia, Penn., 1996. SIAM.

[4] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR - Generating Derivative Codes from Fortran Programs. *Scientific Programming*, (1):1–29, 1992.

[5] C. H. Bischof, W. T. Jones, A. Mauer, and J. Samareh-Abolhassani. Experiences with the application of the ADIC automatic differentiation tool to the cscmdo 3-d volume grid generation code. In *34th AIAA Aerospace Sciences Meeting*, AIAA, 1996.

[6] I.N. Bronstein and K.A. Sememdyayev. *Handbook of mathematics*. Verlag Harri Deutsch, 1985.

[7] Stephen L. Campbell and Richard Hollenbeck. Automatic differentiation and implicit differential equations. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 215–227. SIAM, Philadelphia, Penn., 1996.

[8] Y. F. Chang and George F. Corliss. Solving ordinary differential equations using Taylor series. *ACM Trans. Math Software*, 8:114–144, 1982.

[9] Y. F. Chang and George F. Corliss. ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers and Mathematics with Applications*, 28:209–233, 1994.

[10] Ralf Giering and Thomas Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. On Math. Software*, 24(4):437–474, 1998.

[11] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM TOMS*, 22(2)(June):131–167, 1996. Algor. 755.

[12] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[13] L.Michelotti. MXYZPLTK: A C++ Hacker's Implementation of Automatic Differentiation. In *G. F. Corliss and A. Griewank, editors, Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 218–227, January 6–8 1991.

[14] Richard D. Neidinger. An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order. *ACM Transactions on Mathematical Software*, 18(2):159–173, 1992.

[15] Richard D. Neidinger. Computing multivariable taylor series to arbitrary order. In *APL95 Conference Proceedings, APL Quote Quad*, volume 25, pages 134–144, San Antonio, Texas, USA, June 1995.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[17] L. B. Rall and G. F. Corliss. An introduction to automatic differentiation. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation, Procs Second International Workshop on Computational Differentiation*. SIAM, 1996.

[18] Louis B. Rall and George F. Corliss. An introduction to automatic differentiation. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 1–17. SIAM, Philadelphia, Penn., 1996.

[19] V.N. Rokityanska. *Development of Mathematical Principles and Software for Automatic Differentiation Tools for Computational Similarities of Physical and Mechanical Fields*. PhD thesis, Institute for Problems in Machinery of Ukrainian National Academy of Sciences, Kharkov, Ukraine, January 1996.

[20] V. L. Rvachev. *Theory of R-functions and Some Applications*. Naukova Dumka, 1982. In Russian.

[21] V. L. Rvachev, G. P. Manko, and V. V. Fedko. To the problem of software engineering of the computer differentiation of superpositions of the functions. *Dokl AS UkrSSR*, (1):72–74, 1981. In Russian.

[22] V. L. Rvachev, G. P. Manko, and V. V. Fedko. Technology of differentiating functions of many variables on an electronic computer. *Cybernetics*, 19(5):626–629, 1983.

[23] V. L. Rvachev and T. I. Sheiko. R-functions in boundary value problems in mechanics. *Applied Mechanics Reviews*, 48(4):151–188, 1995.

[24] V. L. Rvachev, T. I. Sheiko, V. Shapiro, and I. Tsukanov. On Completeness of RFM Solution Structures. *Computational Mechanics*, 22(1), 2000.

[25] V. L. Rvachev, T. I. Sheiko, V. Shapiro, and I. Tsukanov. Transfinite interpolation over implicitly defined sets. *Computer Aided Geometric Design*, 18(4):195–220, 2001.

[26] V. L. Rvachev and A. N. Shevchenko. *Problem-oriented languages and systems for engineering computations*. Tekhnika, Kiev, 1988. In Russian.

[27] V. Shapiro. Theory of R-functions and applications: A primer. Tech. Report TR91-1219, Computer Science Department, Cornell University, Ithaca, NY, 1991.

[28] V. Shapiro. Real functions for representation of rigid solids. *Computer-Aided Geometric Design*, 11(2):153–175, 1994.

[29] V. Shapiro and I. Tsukanov. Implicit functions with guaranteed differential properties. In *Fifth ACM Symposium on Solid Modeling and Applications*, Ann Arbor, MI, 1999.

[30] V. Shapiro and I. Tsukanov. Meshfree simulation of deforming domains. *Computer-Aided Design*, 31(7):459–471, 1999.

[31] A. N. Shevchenko. DIFOR and its application for automation of programming of boundary value problems. Tech. report 32, Institute for Problems in Machinery of Ukrainian Academy of Sciences, Kharkov, Ukraine, 1977.

[32] A.N. Shevchenko and V.N. Rokityanskaya. Automatic differentiation of functions of many variables. *Cybernetics and System Analysis*, 32(5):709–723, 1996.

[33] I. Tsukanov and V. Shapiro. The architecture of SAGE — a meshfree system based on RFM. *Engineering with Computers*, 2002. Accepted for publication.

[34] I. Tsukanov, V. Shapiro, and S. Zhang. A meshfree method for incompressible fluid dynamics problems. Tech. report 2002-1, Spatial Automation Laboratory, http://sal-cnc.me.wisc.edu, 2001.

[35] S. Wolfram. *The Mathematica. Fourth edition.* Wolfram Media, 1999.

Appendix

Differentiation Rules

Addition: $f = u + v$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} + \frac{\partial^{|\mu|} v}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} \tag{39}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m.$

Additive Inverse: $f = -u$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -\frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} \tag{40}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m.$

Subtraction: $f = u - v$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} - \frac{\partial^{|\mu|} v}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} \tag{41}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m.$

Multiplication: $f = u * v$

$$\begin{aligned} \frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \Bigg(& \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \cdots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} v}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \Bigg), \end{aligned} \tag{42}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Multiplicative Inverse: $f = 1/u$

$$\begin{aligned} \frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -\Bigg(& \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \cdots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} f}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \Bigg) / u, \end{aligned} \tag{43}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, u \neq 0.$

Division: $f = u/v$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} v}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} f}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \right) / v, \tag{44}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, v \neq 0.$

Squaring: $f = u^2$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = 2 \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p-1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} u}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \right), \tag{45}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Square Root: $f = \sqrt{u}$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left[\frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} - 2 \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} f}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \right) \right] / 2f, \tag{46}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, u \geq 0.$

Power Function: $f = u^s$

$$\begin{aligned} \frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \Bigg[& s * f * \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} + \\ & \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p-1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \\ & * \left(s * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} u}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} - \right. \\ & \left. \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|} f}{\partial x_1^{\mu_1-\alpha_1} \partial x_2^{\mu_2-\alpha_2} \dots \partial x_n^{\mu_n-\alpha_n}} \right) \Bigg] / u, \end{aligned} \tag{47}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0.$

Exponential: $f = \exp(u)$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right), \tag{48}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Logarithm: $f = \ln(u)$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} f}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right) / u, \tag{49}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, u > 0.$

Sine and Cosine: $f = \sin(u), g = \cos(u)$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right), \tag{50}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

$$\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = - \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right), \tag{51}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Tangent: $T = \tan(u) = \sin(u)/\cos(u) = f/g$

$$\begin{aligned}\frac{\partial^{|\mu|}T}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg[&\frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} - T * \frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} \\ &- \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}T}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}g}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &+ \left.\frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}T}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/g,\end{aligned} \tag{52}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, u \ne \pi/2 + k\pi.$

Cotangent: $C = \cot(u) = \cos(u)/\sin(u) = g/f$

$$\begin{aligned}\frac{\partial^{|\mu|}C}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg[&\frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} - C * \frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} \\ &- \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}C}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}f}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &+ \left.\frac{\partial^{|\alpha|}f}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}C}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/f,\end{aligned} \tag{53}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, u \ne k\pi.$

Inverse Sine: $f = \arcsin(u), g = \sqrt{1-u^2}$

$$\begin{aligned}\frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg(&\frac{\partial^{|\alpha|}u}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}f}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\Bigg)/g,\end{aligned} \tag{54}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, -1 \le u \le 1.$

$$\begin{aligned}\frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = -\Bigg[&u * \frac{\partial^{|\mu|}u}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} + \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}u}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}u}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &+ \left.\frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}g}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/g,\end{aligned}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, -1 \le u \le 1.$

Inverse Cosine: $f = \arccos(u) = \pi/2 - \arcsin(u)$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -\frac{\partial^{|\mu|} \arcsin(u)}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} \tag{55}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, -1 \leq u \leq 1.$

Inverse Tangent: $f = \arctan(u), g = 1 + u^2$

$$\begin{aligned}\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \Bigg(& \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} f}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \Bigg) / g,\end{aligned} \tag{56}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0.$

$$\begin{aligned}\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = & 2 \Bigg(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \Bigg),\end{aligned}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Inverse Cotangent: $f = arc\cot(u) = \pi/2 - \arctan(u)$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -\frac{\partial^{|\mu|} \arctan(u)}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} \tag{57}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m.$

Hyperbolic Sine and Hyperbolic Cosine: $f = \sinh(u), g = \cosh(u)$

$$\begin{aligned}\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = & \Bigg(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \Bigg),\end{aligned} \tag{58}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

$$\begin{aligned}\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = & \Bigg(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \\ & * \frac{\partial^{|\alpha|} f}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \Bigg),\end{aligned} \tag{59}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i.$

Hyperbolic Tangent: $T = \tanh(u) = \sinh(u)/\cosh(u) = f/g$

$$\begin{aligned}\frac{\partial^{|\mu|}T}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg[&\frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} - T * \frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} \\ &- \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}T}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}g}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &+ \left.\frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}T}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/g,\end{aligned} \tag{60}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, u \neq 0.$

Hyperbolic Cotangent: $C = \coth(u) = \cosh(u)/\sinh(u) = g/f$

$$\begin{aligned}\frac{\partial^{|\mu|}C}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg[&\frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} - C * \frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} \\ &- \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}C}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}f}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &+ \left.\frac{\partial^{|\alpha|}f}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}C}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/f,\end{aligned} \tag{61}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0, u \neq 0.$

Inverse Hyperbolic Sine: $f = arc\sinh(u) = \ln(u + \sqrt{u^2+1}), g = \sqrt{u^2+1}$

$$\begin{aligned}\frac{\partial^{|\mu|}f}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg(&\frac{\partial^{|\alpha|}u}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}f}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\Bigg)/g,\end{aligned} \tag{62}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0.$

$$\begin{aligned}\frac{\partial^{|\mu|}g}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} = \Bigg[&u * \frac{\partial^{|\mu|}u}{\partial x_1^{\mu_1}\partial x_2^{\mu_2}\dots\partial x_n^{\mu_n}} + \sum_{\alpha_1=0}^{\mu_1}\sum_{\alpha_2=0}^{\mu_2}\cdots\sum_{\alpha_p=0}^{\mu_p-1}\cdots\sum_{\alpha_n=0}^{\mu_n}\binom{\mu_1}{\alpha_1}\binom{\mu_2}{\alpha_2}\dots\binom{\mu_p-1}{\alpha_p}\dots\binom{\mu_n}{\alpha_n} \\ &* \left(\frac{\partial^{|\alpha|}u}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}u}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right. \\ &- \left.\frac{\partial^{|\alpha|}g}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}\dots\partial x_n^{\alpha_n}} * \frac{\partial^{|\mu-\alpha|}g}{\partial x_1^{\mu_1-\alpha_1}\partial x_2^{\mu_2-\alpha_2}\dots\partial x_n^{\mu_n-\alpha_n}}\right)\Bigg]/g,\end{aligned}$$

$|\mu| = \sum_{i=1}^n \mu_i, 0 < |\mu| \le m, |\alpha| = \sum_{i=1}^n \alpha_i, |\alpha| > 0.$

Inverse Hyperbolic Cosine: $f = arc\cosh(u) = \ln(u + \sqrt{u^2 - 1}), g = \sqrt{u^2 - 1}$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$\left. * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} f}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right) / g, \tag{63}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, u \geq 1.$

$$\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left[u * \frac{\partial^{|\mu|} u}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} + \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$* \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right.$$
$$\left. \left. - \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} g}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right) \right] / g,$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, u \geq 1.$

Inverse Hyperbolic Tangent: $f = arc\tanh(u) = \frac{\ln(\frac{1+u}{1-u})}{2}, g = 1 - u^2$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$\left. * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} f}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right) / g, \tag{64}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, -1 \leq u \leq 1.$

$$\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -2 \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$\left. * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right),$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, -1 \leq u \leq 1.$

Inverse Hyperbolic Cotangent: $f = arc\coth(u) = \frac{\ln(\frac{u+1}{u-1})}{2}, g = 1 - u^2$

$$\frac{\partial^{|\mu|} f}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = \left(\frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} - \sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$\left. * \frac{\partial^{|\alpha|} g}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} f}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right) / g, \tag{65}$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |\alpha| > 0, |u| > 1.$

$$\frac{\partial^{|\mu|} g}{\partial x_1^{\mu_1} \partial x_2^{\mu_2} \dots \partial x_n^{\mu_n}} = -2 \left(\sum_{\alpha_1=0}^{\mu_1} \sum_{\alpha_2=0}^{\mu_2} \cdots \sum_{\alpha_p=0}^{\mu_p - 1} \cdots \sum_{\alpha_n=0}^{\mu_n} \binom{\mu_1}{\alpha_1} \binom{\mu_2}{\alpha_2} \dots \binom{\mu_p - 1}{\alpha_p} \dots \binom{\mu_n}{\alpha_n} \right.$$
$$\left. * \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \dots \partial x_n^{\alpha_n}} * \frac{\partial^{|\mu - \alpha|} u}{\partial x_1^{\mu_1 - \alpha_1} \partial x_2^{\mu_2 - \alpha_2} \dots \partial x_n^{\mu_n - \alpha_n}} \right),$$

$|\mu| = \sum_{i=1}^{n} \mu_i, 0 < |\mu| \leq m, |\alpha| = \sum_{i=1}^{n} \alpha_i, |u| > 1.$

Copyright Notice

The License Information

Allowed Uses: User may use FFADLib only in accordance with the appropriate Use License. Academic institutions should agree to the Academic Use License for FFADLib, while all others should agree to the Internal Use License for FFADLib.

Use Restrictions: User may not and User may not permit others to (a) decipher, disassemble, decompile, translate, reverse engineer or otherwise derive source code from FFADLib, (b) modify or prepare derivative works of FFADLib, (c) copy FFADLib, except to make a single copy for archival purposes only, (d) rent or lease FFADLib, (e) distribute FFADLib electronically, (f) use FFADLib in any manner that infringes the intellectual property or rights of another party, or (g) transfer FFADLib or any copy thereof to another party.

Warranty Disclaimer: USER ACKNOWLEDGES AND AGREES THAT: (A) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE BOARD OF REGENTS OF THE UNIVERSITY OF WISCONSIN SYSTEM (REGENTS) MAKE ANY REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT THE SUITABILITY OF FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY FOR ANY PURPOSE; (B) FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY IS PROVIDED ON AN ”AS IS, WITH ALL DEFECTS” BASIS WITHOUT EXPRESS OR IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT; (C) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE REGENTS SHALL BE LIABLE FOR ANY DAMAGE OR LOSS OF ANY KIND ARISING OUT OF OR RESULTING FROM USER POSSESSION OR USE OF FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY (INCLUDING DATA LOSS OR CORRUPTION), REGARDLESS OF WHETHER SUCH LIABILITY IS BASED IN TORT, CONTRACT, OR OTHERWISE; AND (D) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE REGENTS HAVE AN OBLIGATION TO PROVIDE DEBUGGING, MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Damages Disclaimer: USER ACKNOWLEDGES AND AGREES THAT IN NO EVENT WILL THE SPATIAL AUTOMATION LABORATORY TEAM OR THE REGENTS BE LIABLE TO USER FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY EVEN IF THE SPATIAL AUTOMATION LABORATORY TEAM OR THE REGENTS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Attribution Requirement: User agrees that any reports, publications, or other disclosure of results obtained with FFADLib will attribute its use by an appropriate citation. The appropriate reference for FFADLib is ”The Fast Forward Automatic Differentiation Library was developed by the Spatial Automation Laboratory Team at the Mechanical Engineering Department of the University of Wisconsin-Madison. All rights, title, and interest in Fast Forward Automatic Differentiation Library are owned by the Spatial Automation Laboratory Team.”

Compliance with Applicable Laws: User agrees to abide by copyright law and all other applicable laws of the United States including, but not limited to, export control laws. User acknowledges that FFADLib in source code form remains a confidential trade secret of the Spatial Automation Laboratory Team and/or its licensors and therefore User agrees not to modify FFADLib or attempt to decipher, decompile, disassemble, translate, or reverse engineer FFADLib, except to the extent applicable laws specifically prohibit such restriction.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restric-

tions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Spatial Automation Laboratory Team, Attention: Professor Vadim Shapiro, 1513 University Avenue, Madison, WI 53706-1572, (608) 262-3591 or vshapiro@engr.wisc.edu.

Internal Use License

This is an Internal Use License for FFADLib. This License is to be signed by RECIPIENT (the "RECIPIENT"), and returned to the Spatial Automation Laboratory Team at the Mechanical Engineering Department of the University of Wisconsin-Madison (the "PROVIDER"). The FFADLib software program was developed by the Spatial Automation Laboratory Team. All rights, title, and interest in FFADLib are owned by the Spatial Automation Laboratory Team. The subject computer program, including source code, executables, and documentation shall be referred to as the "SOFTWARE."

RECIPIENT and PROVIDER agree as follows:

1. **Definitions:**
   - The "Object Code" of the SOFTWARE means the SOFTWARE assembled or compiled in magnetic or electronic binary form on software media, which are readable and usable by machines, but not generally readable by humans without reverse assembly, reverse compiling, or reverse engineering.
   - The "Source Code" of the SOFTWARE means the SOFTWARE written in programming languages, such as C and FORTRAN, including all comments and procedural code, such as job control language statements, in a form intelligible to trained programmers and capable of being translated into Object Code for operation on computer equipment through assembly or compiling, and accompanied by documentation, including flow charts, schematics, statements of principles of operations, and architecture standards, describing the data flows, data structures, and control logic of the SOFTWARE in sufficient detail to enable a trained programmer through study of such documentation to maintain and/or modify the SOFTWARE without undue experimentation.
   - A "Derivative Work" means a work that is based on one or more preexisting works, such as a revision, enhancement, modification, translation, abridgment, condensation, expansion, or any other form in which such preexisting works may be recast, transformed, or adapted, and that, if prepared without authorization of the owner of the copyright in such preexisting work, would constitute a copyright infringement. For purposes hereof, a Derivative Work shall also include any compilation that incorporates such a preexisting work. Unless otherwise provided in this License, all references to the SOFTWARE include any Derivative Works provided by PROVIDER or authorized to be made by RECIPIENT hereunder.
   - "Support Materials" means documentation that describes the function and use of the SOFTWARE in sufficient detail to permit use of the SOFTWARE.
2. **Copying of SOFTWARE and Support Materials.** PROVIDER grants RECIPIENT a non-exclusive, non-transferable use license to copy and distribute internally the SOFTWARE and related Support Materials in support of RECIPIENTs use of the SOFTWARE. RECIPIENT agrees to include all copyright, trademark, and other proprietary notices of PROVIDER in each copy of the SOFTWARE as they appear in the version provided to RECIPIENT by PROVIDER. RECIPIENT agrees to maintain records of the number of copies of the SOFTWARE that RECIPIENT makes, uses, or possesses.
3. **Use of Object Code.** PROVIDER grants RECIPIENT a royalty-free, non-exclusive, non-transferable use license in and to the SOFTWARE, in Object Code form only, to:

   A Install the SOFTWARE at RECIPIENTs offices listed below;

   B Use and execute the SOFTWARE for research or other internal purposes only;

   C In support of RECIPIENTs authorized use of the SOFTWARE, physically transfer the SOFTWARE from one (1) computer to another; store the SOFTWAREs machine-readable instructions or data on a temporary basis in main memory, extended memory, or expanded memory of such computer system as necessary for such use; and transmit such instructions or data through computers and associated devices.

4. **Back-up Copies.** RECIPIENT may make up to two (2) copies of the SOFTWARE in Object Code form for nonproductive backup purposes only.

5. **Term of License.** The term of this License shall be one (1) year from the date of this License. However, PROVIDER may terminate RECIPIENTs License without cause at any time. All copies of the SOFTWARE, or Derivative Works thereof, shall be destroyed by the RECIPIENT upon termination of this License.

6. **Proprietary Protection.** PROVIDER shall have sole and exclusive ownership of all right, title, and interest in and to the SOFTWARE and Support Materials, all copies thereof, and all modifications and enhancements thereto (including ownership of all copyrights and other intellectual property rights pertaining thereto). Any modifications or Derivative Works based on the SOFTWARE shall be considered a part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request. This License does not provide RECIPIENT with title or ownership of the SOFTWARE, but only a right of internal use.

7. **Limitations on Use, Etc.** RECIPIENT may not use, copy, modify, or distribute the SOFTWARE (electronically or otherwise) or any copy, adaptation, transcription, or merged portion thereof, except as expressly authorized in this License. RECIPIENTs license may not be transferred, leased, assigned, or sublicensed without PROVIDER's prior express authorization

8. **Data.** RECIPIENT acknowledges that data conversion is subject to the likelihood of human and machine errors, omissions, delays, and losses, including inadvertent loss of data or damage to media, that may give rise to loss or damage. PROVIDER shall not be liable for any such errors, omissions, delays, or losses, whatsoever. RECIPIENT is also responsible for complying with all local, state, and federal laws pertaining to the use and disclosure of any data.

9. **Warranty Disclaimer.** USER ACKNOWLEDGES AND AGREES THAT: (A) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE BOARD OF REGENTS OF THE UNIVERSITY OF WISCONSIN SYSTEM (REGENTS) MAKE ANY REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT THE SUITABILITY OF FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY FOR ANY PURPOSE; (B) FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY IS PROVIDED ON AN "AS IS, WITH ALL DEFECTS" BASIS WITHOUT EXPRESS OR IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT; (C) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE REGENTS SHALL BE LIABLE FOR ANY DAMAGE OR LOSS OF ANY KIND ARISING OUT OF OR RESULTING FROM USER POSSESSION OR USE OF FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY (INCLUDING DATA LOSS OR CORRUPTION), REGARDLESS OF WHETHER SUCH LIABILITY IS BASED IN TORT, CONTRACT, OR OTHERWISE; AND (D) NEITHER THE SPATIAL AUTOMATION LABORATORY TEAM NOR THE REGENTS HAVE AN OBLIGATION TO PROVIDE DEBUGGING, MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

10. **Damages Disclaimer.** USER ACKNOWLEDGES AND AGREES THAT IN NO EVENT WILL THE SPATIAL AUTOMATION LABORATORY TEAM OR THE REGENTS BE LIABLE TO USER FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE FAST FORWARD AUTOMATIC DIFFERENTIATION LIBRARY EVEN IF THE SPATIAL AUTOMATION LABORATORY TEAM OR THE REGENTS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

11. **Compliance with Applicable Laws.** RECIPIENT agrees to abide by copyright law and all other applicable laws of the United States including, but not limited to, export control laws.

12. **U.S. Government Rights Restrictions.** Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Spatial Automation Laboratory Team, Attention: Professor Vadim Shapiro, 1513 University Avenue, Madison, WI 53706-1572, (608) 262-3591 or vshapiro@engr.wisc.edu

13. **Governing Law.** This License shall be governed by and construed and enforced in accordance with the laws of the State of Wisconsin as it applies to a contract made and performed in such state, except to the extent such laws are in conflict with federal law.

14. **Modifications and Waivers.** This License may not be modified except by a writing signed by authorized representatives of both parties. A waiver by either party of its rights hereunder shall not be binding unless contained in a writing signed by an authorized representative of the party waiving its rights. The nonenforcement or waiver of any provision on one (1) occasion shall not constitute a waiver of such provision on any other occasions unless expressly so agreed in writing. It is agreed that no use of trade or other regular practice or method of dealing between the parties hereto shall be used to modify, interpret, supplement, or alter in any manner the terms of this License.

Academic Use License

This is an Academic Object Code Use License for FFADLib. This license is between you (the ”RECIPIENT”), and the Spatial Automation Laboratory Team at the Mechanical Engineering Department of the University of Wisconsin-Madison (the ”PROVIDER”). The FFADLib software program was developed by the Spatial Automation. All rights, title, and interest in FFADLib are owned by the Spatial Automation Laboratory Team. The subject computer program, including executables and supporting documentation, shall be referred to as the ”SOFTWARE”.

RECIPIENT and PROVIDER agree as follows:

1. A non-exclusive, non-transferable academic use license is granted to the RECIPIENT to install and use the SOFTWARE on any appropriate computer systems located at the RECIPIENT institution to which the RECIPIENT has authorized access. Use of the SOFTWARE is restricted to the RECIPIENT and collaborators at RECIPIENT institution who have agreed to accept the terms of this license.

2. The PROVIDER shall retain ownership of all materials (including magnetic tape, unless provided by the RECIPIENT) and SOFTWARE delivered to the RECIPIENT. Any modifications or derivative works based on the SOFTWARE shall be considered part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request.

3. The RECIPIENT may make a reasonable number of copies of the SOFTWARE for the purpose of backup and maintenance of the SOFTWARE, or for development of derivative works based on the SOFTWARE. The RECIPIENT agrees to include all copyright or trademark notices on any copies of the SOFTWARE or derivatives thereof. All copies of the SOFTWARE, or derivatives thereof, shall be destroyed by the RECIPIENT upon termination of this license.

4. The RECIPIENT shall use the SOFTWARE for research, educational, or other non-commercial purposes only. The RECIPIENT acknowledges that this license grants no rights whatsoever for commercial use of the SOFTWARE or in any commercial version(s) of the SOFTWARE. The RECIPIENT is strictly prohibited from deciphering, disassembling, decompiling, translating, reverse engineering or otherwise deriving source code from the SOFTWARE, except to the extent applicable laws specifically prohibit such restriction.

5. The RECIPIENT shall not disclose in any form either the delivered SOFTWARE or any modifications or derivative works based on the SOFTWARE to any third party without prior express authorization from the PROVIDER.

6. If the RECIPIENT receives a request to furnish all or any portion of the SOFTWARE to any third party, RECIPIENT shall not fulfill such a request, and further agrees to refer the request to the PROVIDER.

7. The RECIPIENT agrees that the SOFTWARE is furnished on an ”as is, with all defects” basis, without maintenance, debugging, support or improvement, and that neither the PROVIDER nor the Board of Regents of the University of Wisconsin System warrant the SOFTWARE or any of its results and are in no way liable for any use that the RECIPIENT makes of the SOFTWARE.

8. The RECIPIENT agrees that any reports, publications, or other disclosure of results obtained with the SOFTWARE will acknowledge its use by an appropriate citation. The appropriate reference for the SOFTWARE is ”The Fast Forward Automatic Differentiation Library was developed by the Spatial Automation Laboratory Team at the Mechanical Engineering Department of the University of Wisconsin-Madison. All rights, title, and interest in Fast Forward Automatic Differentiation Library are owned by the Spatial Automation Laboratory Team.”

9. The term of this license shall not be limited in time. However, PROVIDER may terminate RECIPIENT license without cause at any time.

10. Source code for the SOFTWARE is available upon request and at the sole discretion of the PROVIDER.

11. This license shall be construed and governed in accordance with the laws of the State of Wisconsin.