SAL 2002 - 5

Data Structure and Algorithms for Fast Automatic Differentiation

I. Tsukanov, M. Hall

Data Structure and Algorithms for Fast Automatic Differentiation

I.Tsukanov[a] * M.Hall[b]

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

[b]DNA Star Inc.
1228 S.Park St.
Madison, WI 53715, U.S.A.

To appear in the International Journal for Numerical Methods in Engineering

Abstract

In this paper we discuss the data structure and algorithms for the direct application of generalized Leibnitz rules to the numerical computation of partial derivatives in forward mode. The proposed data structure provides constant time access to the partial derivatives, which accelerates the automatic differentiation computations. The interaction among elements of the data structure is explained by several numerical examples. The paper contains analysis of the developed data structure and algorithms.

Keywords: multivariable arbitrary-order automatic differentiation, Taylor coefficients, Leibnitz chain rules, data structure

1 Introduction

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

Our automatic differentiation approach is based on specialized data structure that holds (1) a *differential tuple* [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 differentiation rule is applied to the inputted tuples and the resulting tuple is sent to the next node.

Our automatic differentiation technique utilizes the generalized Leibnitz rules, which were derived for many independent variables by Manko [9, 10], Shevchenko [13, 14]. The same formulas were obtained independently by Neidinger in [5]. The beauty of the generalized Leibnitz 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 [7, 6, 3]. It turns out that derivatives can be also computed automatically using Taylor series. Indeed, Taylor coefficients are merely scaled partial derivatives. Automatic differentiation algorithms dealing with Taylor coefficients execute faster because these algorithms do not utilize the binomial coefficients which means fewer multiplication operations are performed.

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

[1]The notion of differential tuple was introduced by Shevchenko in [13], however Neidinger calls the similar data structure a "PYRAMID" [5].

This paper is focused on the development of a data structure and algorithms that allow constant time access to partial derivatives. Since the discussed data structure can be used by both automatic differentiation approaches, we will explain first the data structure for the automatic differentiation technique based on the generalized Leibnitz rules. Then we describe the changes in the data structure and the algorithms to adopt them to deal with Taylor coefficients.

1.1 Outline

The rest of the paper is organized as follows: Section 2 describes the mathematical basis of our automatic differentiation approach and explains the data structure and automatic differentiation algorithms; Section 3 contains analysis of the proposed data structure and algorithms.

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 paper are scalar functions having the same number of independent variables ($\mathbb{R}^n \to \mathbb{R}$).

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:

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

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:

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

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 $\binom{\mu_i}{\alpha_i}$, $i = 1, \ldots, n$. 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 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:

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

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 the computation 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 $\binom{\mu_p-1}{\alpha_p}$. Any p which corresponds to a positive element of the 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 paper, we define the multi-index $(\mu_1, \dots, \mu_p - 1, \dots, \mu_n)$ as $\mu - 1$.

To conclude this Section let us introduce a differentiation rule for differentiation operator $\frac{\partial}{\partial x_k}$, where $0 \le k \le n$. 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:

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

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 [8, 5].

2.2 The Taylor coefficient approach

The Taylor coefficient approach is based on the Taylor series expansion of a function $f(x_1, x_2, \dots x_n)$:

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

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

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

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

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

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:

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

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 [6]. Once Taylor coefficients are computed, they can be converted into partial derivatives:

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

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 [13, 9, 4] 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 use of an array rather than a linked list allows constant time access to the derivatives via indices. Moreover, the consistent use 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 [11]:

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

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 [11, 14]. The drawback of such addressing scheme is that it requires $O(n)$ operations to access 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 simply scaled derivatives, the same data structure can be employed to store the Taylor coefficients.

2.4 The data structure for fast automatic differentiation

To achieve constant time access to the elements of differential tuples we employ a 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:

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

where the sum of he elements of the multi-index μ takes 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 $\mu$[n];
    Allocate $\alpha$[n];
    Initialize the multi-index $\mu = 0$;
    $\mu$[n] = max_order;
    k = IC_size;
    M[tuple_size] = IC_size;
    For i=tuple_size-1 to 0 step -1
    {
        Find p such that $\mu$[p]>0 and $\mu$[p] = $min\{\mu\}$;
        Initialize the multi-index $\alpha = \mu$;
        size = $\prod_{j=1}^{n}(\mu[j]+1)$;
        For j=size-2 to 1 step -1
        {
            Decrement the multi-index $\alpha$ with the preference of $\alpha$[p];
            I[k]=R($\alpha$);
            C[k]=$\prod_{l=1}^{n}\binom{\mu[l]}{\alpha[l]}$;
            k = k - 1;
        }
        S[i] = R($\mu$-1);
        M[i] = k;
        Decrement the multi-index $\mu$;
    }
    tuple_size1 = $\binom{max\_order-1+n}{n}$;
    Allocate D[n, tuple_size1];
    Initialize the multi-index $\mu = 0$;
    For i=1 to tuple_size
    {
        For k=1 to n
        {
            D[k,R($\mu$)] = R($\mu_1$, $\mu_2$,$\ldots$, $\mu_k + 1$,$\ldots$, $\mu_n$);
        }
        Increment the multi-index $\mu$;
    }
    Deallocate $\mu$;
    Deallocate $\alpha$;
}
```

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). The 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 the Leibnitz series, index p is selected to be the minimal non-zero element of μ. Once the value of p is determined, the multi-index α is initialized with values of μ. The inner loop assigns values to the elements of the index array I and the 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.

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. We conclude this Section by

$i = R(\mu)$	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 $\backslash$ 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

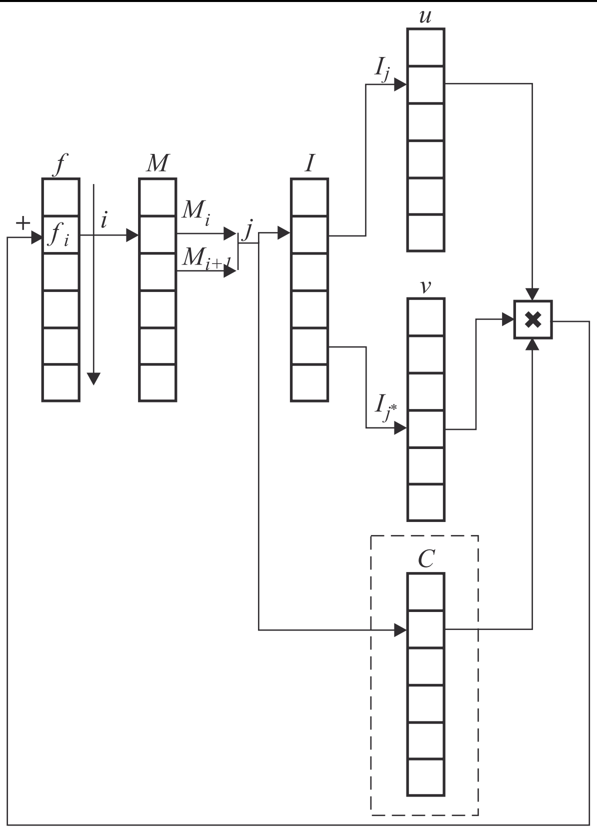


Figure 1: Data structure for automatic differentiation of the product $f = uv$

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 resulting 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 actually 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. 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

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

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:

```
REM Taylor Coefficient Approach
tuple operator * (tuple u, tuple v)
{
   tuple f;
   f.order = min(u.order, v.order);
   f.size = min(u.size, v.size);
   f[0] = u[0] * v[0];
   For i = 1 to f.size-1
   {
      f[i] = u[0]*v[i]+u[i]*v[0];
      MM = M[i]+M[i+1]-1;
      L = M[i+1]-M[i];
      L2 = L/2;
      if( L % 2 == 1 )
      {
         j = M[i]+L2;
         f[i] = f[i] + u[I[j]]*v[I[j]];
      }
      For j = M[i] to M[i]+L2
      {
         f[i] = f[i] + (u[I[j]]*v[I[MM-j]]+u[I[MM-j]]*v[I[j]]);
      }
   }
   return f;
}
```

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

```
REM Differential Tuples Approach
tuple operator / (tuple u, tuple v)
{
   tuple f;
   f.order = min(u.order, v.order);
   f.size = min(u.size, v.size);
   f[0] = u[0] / v[0];
   For i = 1 to f.size-1
   {
      f[i] = f[0]*v[i];
      MM = M[i]+M[i+1]-1;
      L = M[i+1]-M[i];
      L2 = L/2;
      if( L % 2 == 1 )
      {
         j = M[i]+L2;
         f[i] = f[i] + C[j]*v[I[j]]*f[I[j]];
      }
      For j = M[i] to M[i]+L2
```

```
        {
            f[i] = f[i] + C[j]*(v[I[j]]*f[I[MM-j]]+v[I[MM-j]]*f[I[j]]);
        }
        f[i] = (u[i]-f[i])/v[0];
    }
    return f;
}
```

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.

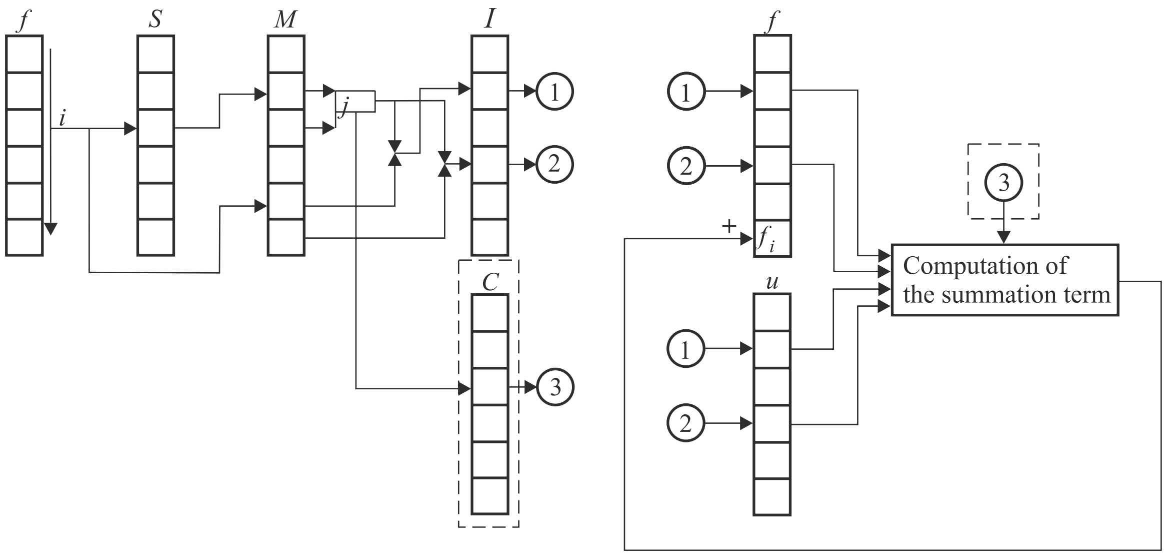


Figure 2: Data structure for automatic differentiation of an elementary function

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 indexes 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). The 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 happens 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] -
```

```
                    u[I[M[i]+L]]*f[I[M[i+1]-L-1];
      If( L % 2 == 1 )
      {
         first = I[M[i]]+L2;
         last = I[M[i+1]-L2-1];
         sum = sum + (s*f[first]*u[last] - u[first]*f[last]);
      }
      For j = M[S[i]] to M[S[i]]+L2
      {
         first = I[j - M[S[i]] + M[i]];
         last = I[M[S[i]] - 1 - j + M[i+1]];
         sum = sum + s*(f[first]*u[last] + u[first]*f[last])-
                         u[first]*f[last] - f[first]*u[last];
      }
      f[i] = f[i] + sum/u[0];
   }
   return f;
}
```

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:

```
tuple dx(tuple u, int k)
{
   tuple f;
   f.order = u.order-1;
   f.size = $\binom{f.order+dimension}{dimension}$;
   For i = 0 to f.size-1
   {
      f[i]=u[D[k,i]];
   }
   return f;
}
```

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`.

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

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 =` $\binom{f.order+dimension}{dimension} = \binom{1+3}{3} = 4$. 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 the 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.

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 [17]) and applying the discussed automatic differentiation algorithms we get a consensus on 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 [14]) may be achieved for large numbers of independent variables and for higher order derivatives. In fact, the algorithms described in [14] 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 that the real speed increase also depends on the order of the derivatives. This fact arises because the implementations of the

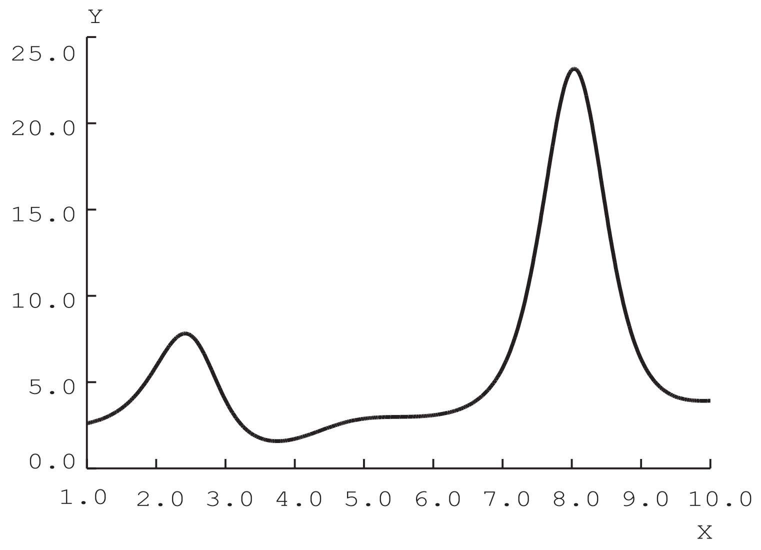


Figure 3: Plot of the function $f = e^{\sin(x+\cos(x+\sqrt{x}))+\ln(x+\frac{1}{2})}$

algorithms given in [14] 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 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.

4 Conclusions

The data structure for multivariable arbitrary order automatic differentiation discussed in this paper offers several advantages. In particular, the proposed addressing scheme provides constant time access to the derivatives and binomial coefficients. The developed data structure eliminates the need of using multi-indices for accessing the derivatives. It is initialized only once, and then this data structure is used for all automatic differentiation computations. Automatic differentiation algorithms, utilizing the precomputed addresses, get direct access to the derivatives and binomial coefficients. This makes it possible to reduce the automatic differentiation algorithms to just two nested loops. One of them goes through the elements of the resulting differential tuple, and the other assembles a single partial derivative combining partial derivatives of the arguments and binomial coefficients according to the proper Leibnitz chain rule. The data structure can be easily adapted to deal with Taylor coefficients instead of the derivatives. In this case, the auxiliary array containing products of the binomial coefficients is not used at all, and it can be deleted from the data

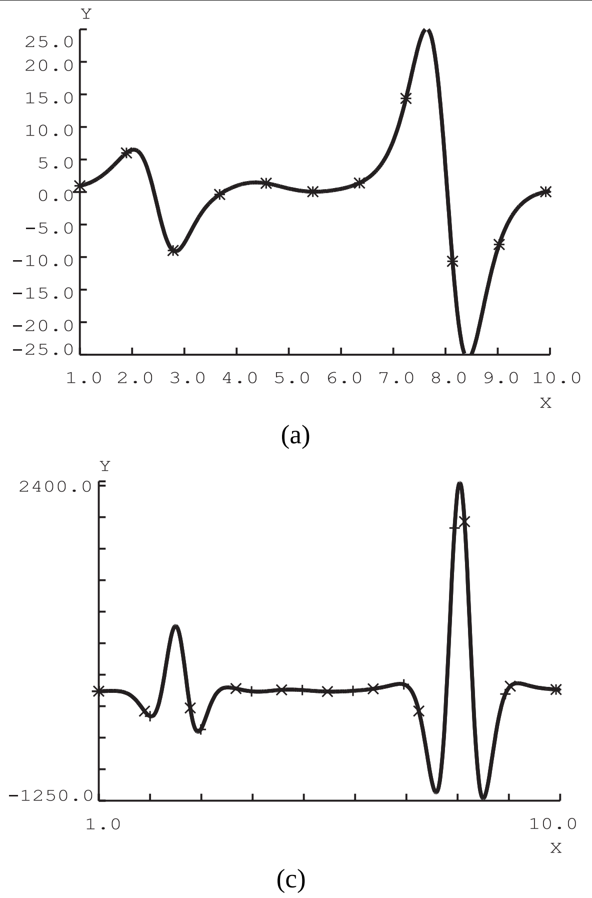
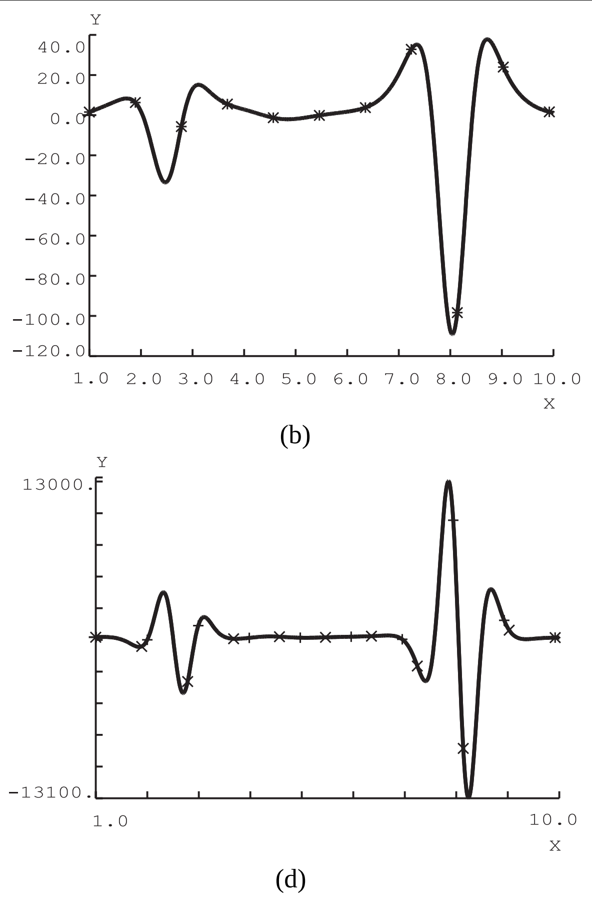


Figure 4: (a)The first derivative of the function (16); (b)the second derivative of the function (16); (c)the fourth derivative of the function (16); (d)the fifth derivative of the function (16). Marker "+" defines the plots of the derivatives computed symbolically and marker "$\times$" designates plots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

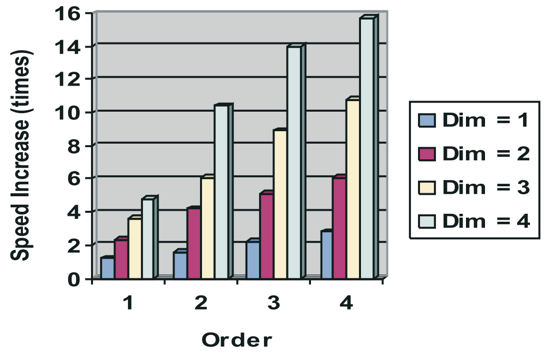
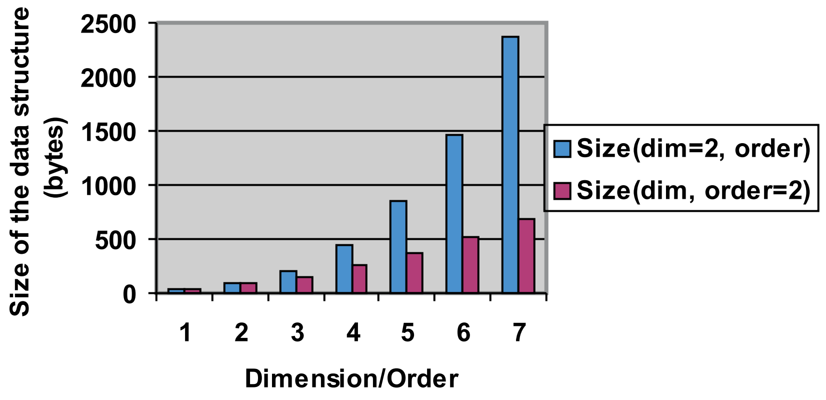
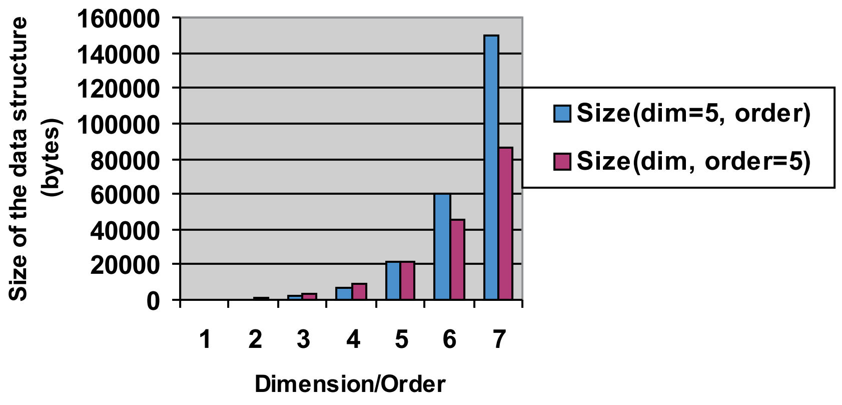


Figure 5: Dependence of the speed increase (relative to the algorithm described in [14]) on number of independent variables and order of derivatives



(a)



(b)

Figure 6: Dependence of the size of the data structure 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)

structure. The automatic differentiation algorithms also need a little change: multiplications by binomial coefficients should be removed. The proposed data structure results in substantial speed increase of the automatic differentiation computations in comparison with algorithms that employ multi-indices for addressing the derivatives and binomial coefficients. The rate of speed increase grows with order of the derivatives and number of independent variables which is illustrated by histogram in Figure 5. For example, in the case of four independent variables and fourth order derivatives the proposed automatic differentiation algorithms are about sixteen times faster than the algorithms described in [14]. The apparent drawback of the proposed addressing scheme is exponential growth of the auxiliary data structure with number of independent variables and order of the derivatives. However, for most practical applications requiring computations of low order partial derivatives with respect to many independent variables the size of auxiliary data structure is relatively small. When the size of the data structure exceeds the physically available memory on a computer, it is possible to combine the proposed addressing scheme with schemes employing multi-indices to access the data structure.

The data structure and automatic differentiation algorithms described in this paper have been implemented in an object-oriented library for automatic differentiation. This library has been developed at the University of Wisconsin-Madison, and it can be downloaded from http://sal-cnc.me.wisc.edu. The library consists of overloaded arithmetic operations and elementary functions that are written in the C++ programming language. The description of all library functions can be found in [15]. A variety of engineering applications can benefit by employing the proposed automatic differentiation algorithms. Among such applications are meshfree methods of engineering analysis [12, 16], robot motion planning, the solution of ordinary differential equations [15].

Acknowledgments

This work was supported in part by the National Science Foundation grants DMI-9900171, 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] I. Tsukanov and M. Hall. Fast Forward Automatic Differentiation Library (FFADLib): A User Manual. SAL Tech. Report 2000-4, Spatial Automation Laboratory, University of Wisconsin-Madison, http://sal-cnc.me.wisc.edu, 2000.

[16] I. Tsukanov and V. Shapiro. The architecture of SAGE – a meshfree system based on RFM. *Engineering with Computers*, 18(4):295–311, 2002.

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