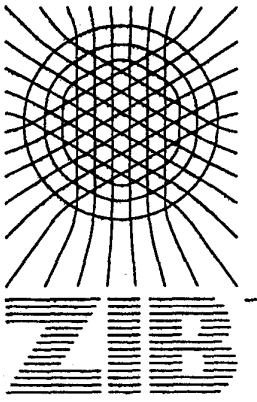
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-1000 Berlin 31



Andreas Griewank*

Sequential Evaluation of Adjoints and Higher Derivative Vectors by Overloading and Reverse Accumulation†

* Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, Illinois 60439

† This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contracts W-31-109-Eng-38.

Preprint SC 91-3 (July 1991)

Contents

Abstract

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions defined by computer programs. It is shown here how vectors of such partial derivatives can be obtained automatically and efficiently if the computer language allows overloading (as is or will be the case for C++, PASCAL-XSC, FORTRAN90, and other modern languages). Here, overloading facilitates the extension of arithmetic operations and univariate functions from real or complex arguments to truncated Taylor-series (or other user-defined types), and it generates instructions for the subsequent evaluation of adjoints. Similar effects can be achieved by precompilation of FORTRAN77 programs. The proposed differentiation algorithm yields gradients and higher derivatives at a small multiple of the run-time and RAM requirement of the original function evaluation program.

Keywords: Automatic Differentiation, Chain Rule, Overloading, Taylor Coefficients, Gradients, Hessians, Reverse Accumulation, Adjoint Equations.

Abbreviated title: Automatic Differentiation by Overloading

1 Introduction

The methodology considered here yields numerical values of first and higher derivatives that are not effected by truncation errors and would thus be exact if the calculations could be carried out in exact arithmetic. This approach is often referred to as *Automatic Differentiation* or *Differentiation Arithmetic* [16]. In contrast to the fully symbolic differentiation performed by symbolic manipulators (e.g. MAPLE, Macsyma, Mathematica), the chain rule is applied here to the numerical values rather than the algebraic expressions of the elementary partial derivatives. This avoids the so-called expression swell, which often limits the use of the fully symbolic approach. The elementary partials themselves are derivatives of a finite number of arithmetic operations and library functions and can therefore be evaluated analytically at any suitable argument. Here we have tacitly assumed that the vector function to be differentiated is *factorable* [2], i.e. the composition of such elementary operations and functions. This is no serious restriction since most functions of practical interest are defined or approximated by sequential programs in a high level computer language.

The term *sequential* in the title alludes to three aspects of this paper. First, we consider only implementations on a single processor machine even though automatic differentiation generates run-time dependency information that can be used for concurrent scheduling on multiprocessor architectures. Second, we will ensure that almost all extra data needed for automatic differentiation are generated and accessed in the same or exactly reversed order. Such *Sequential Access Memory* (SAM) can be allocated on external mass storage devices without unduely slowing the execution speed. Third, we provide for the likely scenario that the user wants to selectively evaluate derivatives of increasing order at the same point. The classical application for this recursive mode are Taylor series methods for ordinary differential equations, where the currently highest derivative feed into the right hand side yields the next higher one by automatic differentiation.

A similar situation arises in constrained optimization, where second derivative information is only required along the tangent space of the feasible set, which is calculated from the first derivatives of the active constraints. If the resulting reduced or projected Hessian is (nearly) singular one may want to examine third derivatives along the null space in order to test for local optimality. In numerical bifurcation this process of examining higher and higher derivatives on subspaces in which the lower derivatives are in some sense degenerate may continue in principle indefinitely. However, in practice such calculations could be performed so far only on simple model problems or based on increasingly inaccurate higher order differences. It is hoped that the techniques described here will facilitate the numerical treatment of higher order singularities for problems of a realistic scale. With regards to the optimization application it should be noted that we obtain the one-sided projection of the Hessian so that one-step quadratic convergence is achievable.

As indicated in the title we concern ourselves primarily with the evaluation of *derivative vectors* of the form

$$\frac{\partial^d}{\partial \alpha^d} F\left(x + \alpha x^{(1)} + \alpha^2 x^{(2)} + \cdots + \alpha^d x^{(d)}\right)\Big|_{\alpha=0} \in \mathbf{R}^m \quad , \tag{1}$$

and

$$\nabla_x \frac{\partial^d}{\partial \alpha^d} \bar{y} \cdot F\left(x + \alpha x^{(1)} + \alpha^2 x^{(2)} + \cdots + \alpha^d x^{(d)}\right)\Big|_{\alpha=0} \in \mathbf{R}^n \quad . \tag{2}$$

Here $F : \mathbf{R^n} \to \mathbf{R^m}$ is a nonlinear mapping, $\bar{y} \in \mathbf{R^m}$ is a row vector, and the column vectors $x^{(i)} \in \mathbf{R^n}$ belong typically to the nullspace of the Jacobian $F'(x)$. By repeated evaluation of such expressions for varying vectors $\bar{y}$ and $x^{(i)}$ one can compose derivative matrices and tensors of arbitrary order and size. The salient fact regarding the evaluation of these derivative vectors is that the increase of run-time and RAM requirement (relative to the cost of evaluating F by itself) grows like the square of the degree d, but it is completely independent of the numbers n and m of independent and dependent variables respectively. However, it should be noted that for vectors of

the form (2) this favorable complexity bound can only be achieved by *reverse accumulation*, which requires SAM proportional to the original run-time is available. A particularly important application is nonlinear least squares, where (2) with $\bar{y} = F^T(x)$ yields the gradient of the sum of squares.

Our theory (though not our current implementation [7]) allows for multivariate Taylor series and thus the direct calculation of Hessians and higher derivative tensors as a whole. However, this aggregate approach reduces the computational complexity only by a constant, while it causes a significant increase in the storage requirement. It is conjectured [6] that the problem of finding the optimal ordering for applying the chain rule to accumulate Jacobian is NP-hard. No such combinatorial optimization problem arises here because the optimal procedure for the evaluation of derivative vectors is *forward or reverse accumulation* in cases (1) and (2) respectively. In Section 2 we will explain these terms, which are equivalent to the notion of *bottom up* and *top down* accumulation used by IRI [10].

The large SAM requirement of the basic reverse mode is definitely a reason for concern but not for despondency. In nuclear engineering and weather modeling [18] the mathematically equivalent method of adjoint sensitivity analysis has been applied to large problems for many years. However, until recently these adjoint codes were written "by hand" after a careful analysis of the underlying mathematical models. It is hoped that further progress in automatic differentiation will eliminate this laborious process and produce adjoint codes of comparable or even better efficency. Moreover, there has been a recent proposal [8] to reduce the SAM requirement such that it is no longer proportional to the run-time of the original evaluation program. The large body of quantitative and qualitative dependency information collected for the purpose of reverse accumulation can also be utilized for the estimation of evaluation errors [11] and the concurrent scheduling of both function and derivative evaluations.

The reader may ask why the concept of overloading figures prominently in the title and indeed our way of looking at automatic differentition. Far from being just a language attribute, overloading allows one to concentrate on the essential aspects of a particular algorithmic task without freezing aspects whose variability may be important on another level. In this spirit we will develop in Section 2 an algorithm for the adjoint evaluation of composite functions with minimal restrictions on the underlying arithmetic in a suitable normed ring with identity. The 'user' may choose between real or complex arithmetics with various levels of precision or for example introduce his own interval arithmatic package. All the while one may simply think of the algorithm discussed in Section 2 as an method for computing gradients of real functions with respect to real arguments. For the evaluation of higher derivatives the reals may be replaced by truncated Taylor series without effecting the program structure. The required properties and most important examples of the algebras and their arithmetics are discussed in the final Section 4. In the central Section 3 we discuss implementation questions and in particular the crucial issue of storage allocation and memory management. Unfortunatly, these questions depend to some extend on the computer language because only C++ allows the overloading of variables declarations and deallocations. Nevertheless we consider these software issues as integral aspects of the algorithmic design, because just specifying and counting the arithmetic operations involved simply does not adequately describe the structure and efficiency of automatic differentian methods.

Most algorithmic devices discussed here have been implemented and tested in our C++ package ADOL-C. However, we do not view this paper as documantation for a particular piece of software because many of the ideas have been utilized much earlier in Rall's PASCAL-SC differentiation package [15] and the FORTRAN precompilers JAKEF [17], GRESS [9], and PADRE2 [13]. It is hoped that a language independent discussion of the major issues will aid the communication between active researchers and spurn new implementations in other computer languages.

2 Reverse Adjoint Evaluation for Composite Functions

Mathematically, we may formulate our task as evaluating the adjoint mapping of a continuously differentiable vector function

$$y = F(x) \quad : \quad X \equiv \mathcal{R}^n \quad \mapsto \quad Y \equiv \mathcal{R}^m \tag{3}$$

Here n, m are positive integers and the *ring of scalars* $\mathcal{R}$ may at first be thought of as the normed field of real numbers. However, as we will discuss in Section 4, $\mathcal{R}$ can be replaced by any normed algebra and in particular rings of truncated Taylor series. The required vector calculus can be extended to this more general setting with little difficulty.

The function F need not be globally defined so that x may be implicitly restricted to some open domain in X on which F is differentiable. Given any such argument x and a linear functional $\bar{y}$ in the dual space $\bar{Y} \equiv \mathcal{L}(Y, \mathcal{R})$ of Y, there exist a unique functional $\bar{x}$ in the dual space $\bar{X} \equiv \mathcal{L}(X, \mathcal{R})$ of X such that for all $v \in X$

$$\bar{x} \cdot v \quad \equiv \quad < \bar{x}, v > \quad = \quad \bar{y} \cdot F'(x) v \tag{4}$$

where $F(x)'$ denotes the Jacobian of F at x. Throughout most of this section we will consider only one particular pair $x, \bar{y}$ and may thus refer to $\bar{y}$ and $\bar{x}$ as the adjoint vectors (or values) of $y = F(x)$ and x respectively. As we will see the extra cost of obtaining $\bar{x}$ for given $\bar{y}$ is in terms of arithmetic operations only a small multiple of the original effort to calculate y from x. However, to achieve this low complexity bound the adjoint calculation may need substantially more storage than the original evaluation of $y = F(x)$.

Since $\bar{y}$ is fixed we may interpret $\bar{x}$ as the gradient

$$\bar{x} \quad = \quad \nabla(\bar{y} \cdot F(x)) \quad = \quad \partial(\bar{y} \cdot F(x)))/\partial x \tag{5}$$

of the scalar function $\bar{y} \cdot F(x)$ at the current argument x. This interpretation of adjoint vectors as gradients will be particularly useful for intermediate quantities. It also shows that gradients can always be evaluated at essentially the same cost as the underlying scalar function.

2.1 The Adjoint Chain Rule

In order to compute y and $\bar{x}$ for given x and $\bar{y}$ we will make the entirely realistic assumption that F is a composite function and then apply an adjoint form of the chain rule. Let us firstly consider the case of a diamond where there y is computed from x via two intermediate vetors u and v such that for continuously differentiable functions G and H and $\hat{F}$ of appropriate dimensions

$$\begin{aligned} u &= G(x) \\ v &= H(x) \\ y &= \hat{F}(u, v) \quad . \end{aligned} \tag{6}$$

Then we obtain by the chain rule the total derivative

$$\bar{x} \quad = \quad \frac{\partial[\bar{y} \cdot \hat{F}(u,v)]}{\partial u} \frac{\partial G(x)}{\partial x} + \frac{\partial[\bar{y} \cdot \hat{F}(u,v)]}{\partial v} \frac{\partial H(x)}{\partial x} \tag{7}$$

$$= \quad \bar{u} \cdot \partial G(x)/\partial x + \bar{v} \cdot \partial H(x)/\partial x \tag{8}$$

where the adjoint values $\bar{u}$ and $\bar{v}$ of u and v are defined by

$$\bar{u} \equiv \bar{y} \cdot \partial \hat{F}(u,v)/\partial u \quad \textit{and} \quad \bar{v} \equiv \bar{y} \cdot \partial \hat{F}(u,v)/\partial v \quad . \tag{9}$$

Starting with $\bar{v} = 0$, $\bar{u} = 0$, and $\bar{x} = 0$ we can thus effect the calculation of $\bar{x}$ by the sequence of assignments

$$\begin{aligned} \bar{v} &+= \bar{y} \cdot \partial \hat{F}(u,v)/\partial v \\ \bar{u} &+= \bar{y} \cdot \partial \hat{F}(u,v)/\partial u \\ \bar{x} &+= \bar{v} \cdot \partial H(x)/\partial x \\ \bar{x} &+= \bar{u} \cdot \partial G(x)/\partial x \end{aligned} \tag{10}$$

where $a+=b$ means increment a by b as in the programming language C. One may derive and execute the *adjoint program* (10) from the *original program* (6) by the simple recipe:

- Select the adjoint $\bar{y}$ of the dependent vector y and initialize the adjoints of all intermediate and independent values to zero.
- Replace each assignment of the form $w = f(z)$ by $\bar{z}+= \bar{w} f(z)'$ and execute these incremental assignments in reverse order.

It should be noticed that in order to execute this *reverse sweep* the current values of all arguments w must have been saved from the preceding *forward sweep*, i.e. the execution of the original program.

2.2 Reverse Accumulation for Sequential Programs

Obviously this recipe can be applied to "programs" involving an arbitrary number of intermediate variables. In fact any program can be broken down recursively into smaller and smaller blocks or procedures until each individual calculation is of suitably elementary nature. From now on we will assume that the overall vector function $y = F(x)$ is evaluated by a sequence of *elementary assignments*

$$\begin{aligned} &FOR \quad t = 1, 2, \ldots, T \\ &\qquad w_t \equiv (v_i)_{i=i_{t-1}+1,\ldots i_t-1,i_t} = f_t(v_j)_{j \in \mathcal{J}_t} \end{aligned} \tag{11}$$

where the variables v_i and v_j belong to $\mathcal{R}$, and the index sets $\mathcal{J}_t$ contain no numbers greater than i_{t-1}. In other words we assume that the scalar quantities v_i are numbered consecutively as they are generated with i_t being the total number of scalar quantities generated by the first t elementary function evaluations. Moreover, with $i_0 \equiv n$ and $I \equiv i_T$ we may assume that the vectors of independent and dependent vectors are given by

$$x = (v_i)_{i=1,\ldots,n} \quad and \quad y = (v_i)_{i=I-m+1,\ldots,I} \,. \tag{12}$$

Usually there are many different ways of breaking $F(x)$ into elementary functions and the sequencing of the elementary assignments can often be altered or even abandoned in favor of concurrent evaluations. However, we will assume here that the user has provided an evaluation program in a high level computer language that can be automatically interpreted in the form (11) during the first evaluation of F at a given argument x. This may be achieved simply by overloading all arithmetic operations and standard univariate functions. Subprograms are no problem and will in effect be inlined, except when the user flags them to be treated as elementary functions thus adding them to the pool of library functions. In the latter case he also has to supply a subprogram for the evaluation of the corresponding adjoint evaluation, which is called in the following adjoint program

$$FOR \quad t = T, T-1, \ldots, 1$$

$$\bar{v}_j \quad += \quad \bar{w}_t \cdot \partial f_t / \partial v_j \quad for\ all\ j \in \mathcal{J}_t \tag{13}$$

where initially

$$(\bar{v}_i)_{i=1,2,\ldots,I-m} = 0 \quad and \quad (\bar{v}_i)_{i=I-m+1,\ldots,I} = \bar{y}$$

Since the scalar increments associated with each elementary function can be combined into one adjoint operation of the form $+= \bar{w}_t f_t'$, one finds that on a serial computer

$$Q\{F\} \quad \equiv \quad \frac{Work\{ += \bar{y} \cdot F'(x)\ ,\ F(x)\ \}}{Work\{\ F(x)\ \}} \quad \leq \quad \max_t Q\{f_t\} \tag{14}$$

where $Work\{\cdot\}$ may be any reasonable measure of computational effort. Thus we see that the penalty factor $Q\{F\}$ for evaluating its adjoint in addition to the composite function F itself is no greater than the maximum of the penalties $Q\{f_t\}$ for anyone of its constituent elementary functions. Provided the latter are drawn from a finite library, the penalty factor is therefore uniformly bounded for all possible compositions, independent of the number of independent and dependent variables. This complexity result dates back at least to Linnainmaa [14] and was subsequently rediscovered by several authors (See e.g. [10] and [5] for other references).

We will refer to the first execution of the loop (11) as the *original forward sweep*. Subsequently one may want to perform more forward sweeps with the components of the independent vector x either altered or extended to elements from a larger scalar ring. After each forward sweep (11) one may perform corresponding *reverse sweeps* (13) for any adjoint vector $\bar{y}$ whose components belong to the appropriate ring. Our only requirement on the elementary functions f_t is that the functionals on the right hand side of (13) can be be easily computed during the reverse sweep. To this end one has to record during the forward sweep for each elementary function the following information

- An operations code and possibly parameters defining the function.
- The names or better addresses of its result and arguments.
- The values of its arguments at the time of evaluation.

As we will discuss in Section 3 there are some variations on this basic storage scheme but none reduces the total storage requirement by more than a constant, except for linear functions and in other special cases.

2.3 Approaches to Limit the Storage Requirement

Since practical calculations tend to involve millions of elementary operations the recording of the execution trace may require a very substantial or even prohibitive amount of storage. To make this problem manageable one may employ the following techniques:

1. Organize the calculation such that most extra data are generated and accessed strictly sequentially, i.e. in either the same or exactly reversed order.
2. Combine scalar variables in vectors and define corresponding elementary functions to avoid storing consecutive addresses and duplicating operations codes.

3. Allow for recursive adjoint evaluations of procedures by reevaluating them during the (outer) reverse sweep to avoid storing their internal intermediate values.

4. Preaccumulate local gradients if only first derivatives are required.

The first point has been fully implemented in our current package, which increases the RAM requirement of the original program roughly by a factor of two. All extra information is recorded on a conceptual *tape* whose content are buffered to and from a mass storage device, e.g. an optical disk. Hence one needs a large Sequential Access Memory (SAM). This should not slow the execution because storage and retrieval are done at the usually higher burst rate. The realization of this very desirable RAM/SAM ratio is based on a live variable analysis discussed in Section 3

To illustrate the second point let us briefly consider the linear case. The adjoint of any linear function $w = f(z) \equiv Aw$ is simply the incremental vector operation $\bar{z} += \bar{w}A$, which is is in fact independent of the argument z. Irrespective of the number of rows and columns in the matrix A the penalty factor is here almost exactly $Q\{f\} = 2$, and the storage requirement between sweeps should be small, especially if the components of z have consecutive indices. However, if the matrix vector product is broken into individual scalar operations each of them will be represented separately on the tape. Even worse, if the same constant matrix A is used in several matrix-vector products all its entries will be copied repeatedly into memory. In principle the inclusion of standard matrix-vector and dot products into the library of elementary functions should provide no difficulty.

The third points closely related to the second except that we are thinking here of nonstandard procedures that involve a significant number of local variables and are quite expensive to evaluate. Purely in terms of arithmetic operations it would then be advantages to store all their internal intermediates during the forward sweep as part of the overall execution trace. However, especially if the same procedure is called many times with different arguments (as may happen for example in a finite element calculation) one can save a lot of memory by storing only the arguments and then repeating their evaluation and the corresponding adjoint operation during the (outer) reverse sweep. This idea was already published by sc Volin and is also the basis of the logarithmic method proposed in [8].

The fourth proposal can be applied to complicated right hand sides or scalar valued subroutines primarily if only first derivatives are required. Actually, in combination with ring arithmetic it can also be used to calculate higher derivative vectors of the form (1) and (2) provided the weight vector $\bar{y}$ and the directions $x^{(i)}$ are already known during the forward sweep. The idea is that whenever a single scalar is calculated as a function of one or more arguments using some unnamed intermediates, then its gradient with respect to these arguments can be calculated by a locally reversed sweep during the overall forward sweep. For the purposes of calculating global first derivatives this local function can be immediately replaced by its linearization, i.e. the gradient. The variables that are eliminated by this preaccumulation need not be unnamed, provided one can be sure that they do not occur anywhere else as an argument. This is clearly true for the local variables in a scalar valued function routine or the temporaries generated during the evaluation of a complicated expression. To the best of our knowledge the precompiler GRESS is the only implementation that immediately linearizes the right hand sides of scalar assignments. In terms of arithmetic operations such preaccumulations are only beneficial if several reverse sweeps follow the original forward evaluation. However, they always lead to a reduction in the storage requirement. This approach may still be worth while if a few scalars are computed from a sizable number of arguments using a large number of intermediates. We will not pursue this idea here because it leads again to a combinatorial optimization problem.

The second point is the reason why we did not simply assume that all elementary functions are scalar valued, which is of course possible from a theoretical point of view. In fact all current implementations of automatic differentiation known to us break the calculation down to real valued elementary functions, and most limit them to binary arithmetic operations and univariate functions

with real arguments. Since this is clearly not satisfactory in the long run we have allowed for vector valued elementary functions. By choosing his elementary functions carefully the user may significantly reduce the storage requirement for his code. If he uses an elementary function f that is not contained in a standard library the user must supply subprograms for the evaluation of $w = f(z)$ and the adjoint operation $\bar{z} += \bar{w} f'(z)$.

3 Implementation Questions and Memory Management

In the previous Section 1 we have assumed that the scalar quantities $v_i \in \mathcal{R}$ are consecutively indexed in the order that they are generated. Using overloading this number is in fact quite easy to determine and we will continue to refer to it as the (unique) *index* of the corresponding scalar variable. If the user wants to overload his original program for the evaluation of derivatives he first has to identify the set all variables that are to be considered as differentiable quantities. Usually this set consists of the independent variables and all quantities that are directly or indirectly computed from them, which can be identified automatically by a precompiler using data flow analysis. Without such a sophisticated tool one may simply overload the operations such that any attempt to assign a differentiable quantity to a constant leads to a compile time error due to type incompatibility.

3.1 Indexing and Taping by Overloading

In a true overloading scheme the user must flag each differentiable variables, say V, by declaring it to be of a composite type, say VARING, which contains a field V.INT of type integer and a field V.VAL of type RING. Here RING is the original type of the variable V, which could for example be equivalent to the single and double precision types FLOAT or DOUBLE in C. The user may also chose any other other normed ring or even just an *inclusion algebra*, in which the basic arithmetic operations and univariate functions are suitably defined. These definitions must be supplied in the form of problem independent and possibly precompiled subprograms. Similarly we must supply VARING valued subroutines that are called wherever the compiler encounters an elementary operation or function involving one or more arguments of type VARING. For example the statement $W = U * V$ in the original evaluation code my generate the following five instructions if all three variables involved are VARINGs :

Overloaded Multiplication

```
INDCOUNT += 1
W.INT = INDCOUNT
W.VAL = U.VAL*V.VAL
("MULT", U.INT, V.INT, W.INT) ⟶TAPE
W.VAL ⟶ BIRTH
```

Here the first two instructions increment the global index counter INDCOUNT and assigns its current value as index to W . The third instruction actually multiplies the the values U.VAL and V.VAL in ring arithmetic. The fourth instruction writes the kind of operation and the indices of the variables into a file (or in-core array), which we will refer to as the TAPE. Similarly, the last instruction announces the birth of W by copying its value field onto a file called BIRTH. In addition one may also write the value of the arguments U.VAL, V.VAL or the values of the corresponding partial derivatives $\partial W.VAL/\partial U.VAL = V.VAL$ and $\partial W.VAL/\partial V.VAL = V.VAL$. In the latter case everything is immediately linearized and one need not store the operations code "MULT". The

other arithmetic operations and all elementary functions that are well defined for ring arguments can be overloaded in much the same way.

When the resulting program is run the counter INDCOUNT will grow up to the total number of scalar quantities generated during the execution, namely $I \geq T$. For a fixed library of elementary functions both I and T are proportional to the run-time as well as the lengths of each file TAPE and BIRTH. The TAPE represents a complete unroling of the original program and can be used to evaluate the function at another argument x unless the original program involves conditional branches that depend on differentiable quantities. In the latter case the resulting vector function is not continuously differentiable, and automatic differentiation will yield the derivatives of a smooth piece including the current argument. In any case, the evaluation of the composite function on the basis of the tape specified above is a bad idea because it would require T storage locations that are accessed in a random fashion. The same applies for the reverse sweep as one has to provide storage for each scalar adjoint $\bar{v}_i$.

3.2 Replacing Variable Indices by their Locations

The number of real storage locations, say R, required by the original evaluation program will usually be much smaller than T, the number of scalar quantities generated. The reason is that many quantities with distinct indices may successively reside in the same storage location, either because they get assigned to the same variable name, or because one variable is destructed (deallocated) and the compiler assigns its address to variables that are constructed (allocated) later. We will refer to these two possibilities for the demise of an intermediate quantity as *death by overwrite* and *death by destruction*, respectively. The second possibility occurs also for unnamed variables that are temporarily generated by the compiler during the evaluation of composite expressions. For example the evaluation of the statement $w = u * sin(v)$ involves a temporary variable that holds the value of $sin(v)$. These temporaries will have the same type as the arguments u and v whether it be RING or VARING. By effectively using the index as an address one forgoes the possibility to reclaim the storage of an intermediate quantity when it has died.

In order to reduce the RAM requirement for the reverse sweep from $\mathcal{O}(T)$ to $\mathcal{O}(R)$ one has to associate with each index i an address $loc(i)$ such that for all $j < i$ and t

$$loc(j) = loc(i) \quad and \quad i_t \geq i \quad \Longrightarrow \quad j \notin \mathcal{J}_t \tag{15}$$

In other words v_j may be replaced by v_i if it appears as an argument only in elementary function evaluations that precede the computation of v_i. One possible choice for the location $loc(i)$ is the actual (or virtual) address under which the value v_i is stored in memory. This seemingly straight forward approach has been successfully implemented in GRESS, but it is somewhat machine dependent and may not be easy to extend from FORTRAN77 to a language with dynamic storage allocation. Therefore we will discuss various strategies of defining $loc(i)$ as a pseudo address, which is allocated on the bases of *death notices*

If at any stage of the original program execution one can for some reason be sure that a particular intermediate quantity v_j will never again occur as an argument, then one may free up its location $loc(j)$ and write its value and on a separate file DEATH. To synchronize the two files one must simultaneously write the index j labeled as a death notice onto TAPE. The freed locations can be held in a linked list or more sophisticated data structure until they are reassigned to subsequent indices. Whenever all freed locations have been reclaimed more memory space must be allocated. In this way the function $loc(i)$ can be inductively constructed during the original program execution, assuming we have good criteria for the detection of death.

At any given time during the execution of the original program it is generally not possible to say

exactly which indices j need to be kept alive because the corresponding quantity v_j occurs as an argument later on. As long as this possibility cannot be excluded the location $loc(j)$ may not be reassigned to another quantity. In languages that allow the overloading of the assignment operator (e.g. C++, PASCAL-XSC, and probably FORTRAN90) death by overwrite can be detected, provided all variables are suitably initialized. Death by destruction can only be recognized in a language like C++ that allows for user-defined destructors, i.e. type-specific subprograms that are called whenever a variable goes out of scope.

The failure to recognize death by destruction will result in a theoretically unnecessary additional growth in the storage requirement for the adjoint evaluation. However, this waste might be of an acceptable size if the user avoids recursive calls, local variables, and composite right hand sides as much as possible. The last measure is aimed at reducing the number of unnamed temporary variables that are constructed and destructed by the compiler.

If want wants to absolutely minimize the storage requirement for the forward and reverse sweep (without redefining or reordering the sequence of elementary assignments) one may utilize the following construction of $loc(i)$. First one completes the original forward sweep without writing any death notices and simply assigning the increasing index i to each variable. This index may then be used as a key to a self-balancing binary search tree during the following symbolic reverse sweep.

To begin with allocate arbitrary but distinct locations $loc(i)$ to the largest m indices i and introduce them into the binary search tree. Then starting from the last elementary assignment reverse through the tape and execute for each t the following instructions:

Reverse Allocation Procedure

1. Check for each argument index $j \in \mathcal{J}_t$ whether the key j has already been entered into the search tree. If not find a free location $loc(j)$ store it under the key j and announce its death on the tape.

2. For each result index $i = i_{t-1} + 1, \ldots, i_t$ retrieve $loc(i)$ under the key i, delete that node, and free the location $loc(i)$.

3. Rewrite the t-th tape entry in terms of the locations instead of the indices.

When the t-th elementary function involves a vector-argument the corresponding locations $loc(i)$ may be allocated and represented as a range. Instead of the binary search tree one could also utilize a hash function or some other dynamic data structure. In any case the determination of such a storagewise optimal location function $loc(i)$ is likely to be several times more expensive than the evaluation of $F(x)$ at a single point. Nevertheless, this one time effort may be worthwhile if the same $loc(i)$ can be used for forward and reverse sweeps at a sequence of points in the domain of F. At the end of this section we will discuss a cheaper way of constructing a nearly optimal location function using the advanced constructs of $C++$. Even if the integers i do already represent locations the reverse allocation procedure may be employed to reduce the storage requirement further. However, if it has been applied once a second application will leave the allocation function essentially unchanged.

After the reverse allocation procedure has been executed the TAPE contains a sequence of birth notices representing elementary function calls and interspersed death notices. The corresponding values of the variables being calculated or "killed" can be found in exactly the right order on the BIRTH and DEATH file respectively. Thus TAPE contains only symbolic information reflecting the program structure, while BIRTH and DEATH store the numerical values at the current arguments. Since all three files have a length of $\mathcal{O}(T)$ they must probably be stored onto disks or other external

devices. However, since they are always accessed strictly sequentially buffering should avoid any significant I/O delays.

The sequence of assignments (11) still yields the same results if all subscripts i, j, and i_t are replaced by $loc(i)$, $loc(j)$, and $loc(i_t)$ respectively. Note that the assignment counter t remains unchanged. The corresponding adjoint assignments (13) remain also correct, provided each is followed up by the reinitialization $\bar{w}_t = 0$, which ensures that the adjoint of the next intermediate quantity with the same location is accumulated from scratch. Thus we see that, once suitable locations have been assigned, they can completely replace the original indices and we may denote them by the same subscripts i and j.

3.3 Sweeping Back and Forth

Since the value of each intermediate quantity is stored twice, namely at its birth and death, keeping both numerical files simultaneously appears to be wasteful. In fact they can be reconstructed from each other quite easily. In order to obtain the DEATH from the BIRTH file one merely has to run through the tape forward, loading the value of v_i from BIRTH at its birth and writing it later onto DEATH at the time of its death. Conversely one may construct BIRTH from DEATH by reversing through TAPE from the end, loading the value of v_i from DEATH at the time of its death during the forward sweep and writing it later onto BIRTH at the time of its birth. In either case the beginning of one file can overwrite the end of the other so that the memory requirement is essentially halved. Rather then performing these conversions separately we can usually combine them with the numerically productive reverse or forward sweeps to be discussed now.

Given the files TAPE and DEATH generated at a particular point x as well as the adjoint vector $\bar{y}$ one can calculate the adjoint vector $\bar{x}$ by performing the following steps.

1. For each number $i \in \{1, \tilde{R}\}$ allocate space for a pair of ring elements $(v_i, \bar{v}_i)$.
2. Seed the adjoints $(\bar{v}_i)_{i=I-m+1,\ldots,I}$ with the components of the adjoint vector $\bar{y}$.
3. Reversing through the TAPE do:
   - If encountering a death notice for v_i move it from DEATH into core.
   - If encountering the record of the t-th element function increment
     $\bar{v}_j \mathrel{+}= \bar{w}_t \cdot \partial f_t / \partial v_j \quad for\ all\ j \in \mathcal{J}_t$
     and set $\bar{w}_t = 0$. Optionally append the components of w_t to BIRTH.
4. Copy the adjoints $\bar{v}_i$ associated with the first n variables into adjoint vector $\bar{x}$.

As we can see from Step 2 the RAM requirement of this reverse sweep is exactly twice that of the original forward sweep, which is still adequately described by the sequence of assignments (11). Thus we have achieved one of our main goals, namely that almost all extra data are generated and accessed strictly sequentially.

The optional reconstruction of BIRTH is not required if one performs a sequence of reverse sweeps for varying adjoints $\bar{y}$. In order to compute the rectangular Jacobian $F'(x) \in \mathcal{R}^{m \times n}$ row by row, one can perform m reverse sweeps with the adjoint vector $\bar{y}$ ranging over the Cartesian bases vectors $e_i \equiv (\delta_{ij})_{j=1}^m$ for $i = 1 \ldots m$. In that case one would not need BIRTH at all and simply keep DEATH from the original program execution. This mode is quite efficient if the Jacobian is fat because the number m of dependent variables is much smaller than the number n of independent variables.

Otherwise one might prefer to perform n forward sweeps in order to obtain the Jacobian column by column.

At first successive forward sweeps do not seem to offer any economy since the numerical values stored on BIRTH depend on the point x and there appears to be nothing else to vary. What happens in fact is that each scalar component of x, say v_i, is appended by a higher order term, say v_i', such that

$$v_i + v_i' \in \mathcal{R}^{(1)} \equiv \mathcal{R} \oplus \mathcal{R}'$$

where $\mathcal{R}'$ is an ideal in the enlarged normed ring $\mathcal{R}^{(1)}$. Moreover, all intermediates obtained by executing the forward loop at the appended point in the extended ring arithmetic are of the form $v_j + v_j'$, where the leading term v_j remains unchanged and can thus be found on the file BIRTH generated by a reverse sweep at the unappended point x. On the other hand the appended parts v_j' depend on the leading terms v_j so that their availability can be used to make the appended forwards sweep cheaper. The algebraic properties that make this recursive updating possible and the special case of truncated Taylor series will be briefly discussed in Section 4. We complete this section with a discussion of some techniques that can so far only be implemented in C++.

3.4 Births and Deaths in C++

Ideally one would like to define the (pseudo-)locations loc(i) such that the ones corresponding to live variables v_i always form a contiguous interval of integers, say $1, 2, \ldots LT$ at "time" t. This can almost be achieved in C++ or another language with explicit constructors and destructors. The AT&T translator *cfront v* 2.0 ensures that named variables are destructed in exactly the reverse order that they are constructed, so that they form in fact a last in first out stack. Hence one could write for the new type VARING a the constructor/destructor pair of the form

$$\begin{array}{ll} VARING() \ \{ & \text{VARING()} \ \{ \\ \quad INT = ++LT; & \quad --LT; \\ \quad VAL = 0 \ \} & \quad DEATH << VAL\} \end{array} \tag{16}$$

Here the unary operators ++ and − increment or decrement the integer LT by 1, and << means output the value on the right to the file on the left. If initialized to zero LT would always represent the current number of live variables were it not for temporaries that the compiler generates during the evaluation of expressions and the return from function calls. Unfortunately these unnamed variables are not necessarily constructed and destructed in a last in first out fashion, which may lead to the overwriting of variables that are still required as arguments. Our current implementation avoids this possibility by delaying the actual destruction of VARINGs until the locations of all potential destructees form a contiguous tail of the integer interval [1,LT]. To this end we characterize the set of destructees at any stage of the computation by ND, the number of its elements, and MD, the smallest location of any one of the potential destructees. Whenever one finds that

$$\mathbf{MD} \ + \ \mathbf{ND} \ = \ \mathbf{LT} \ + 1$$

the destructees must occupy exactly the last ND locations, and can therefore be eliminated by resetting LT to MD-1 and ND to 0. For reasons of storage economy, these multiple deaths are only carried out if in addition to the above condition ND exceeds a certain lower bound. The clean stack allocation has the advantage that vectors of VARINGs are guaranteed to have contiguous locations. Thus they can be represented by integer ranges in the argument lists of multivariate elementary functions, e.g. dot products.

4 Arithmetic and Analysis in Rings

While forward and reverse sweeps can be performed with virtually any definition of addition and multiplication on some set of scalars $\mathcal{R}$, it appears that their results will only relate to each other in a meaningful way if these arithmetic operations obey the algebraic rules in rings at least approximately. Since infinite mathematical structures must be mapped onto finite *screens*, e.g. the floating point numbers, the exact arithmetic is in practice replaced either by a *rounding arithmetic* or an *inclusion arithmetic* [12]. In the first case the algebraic operations between scalars from the screen are modified slightly such that their results belong also to the screen. In the second case one propagates intervals with bounds from the screen such that the final interval is guaranteed to include the hypothetical result of a calculation in exact arithmetic. In either case the distributive law and some ring axioms are no longer satisfied exactly.

The numerical results obtained in either rounding or inclusion arithmetic are often useful, even though the local rounding errors may sometimes build up in a dramatic fashion. Depending on their *stability* two mathematically equivalent algorithms for the evaluation of $F(x)$ may yield substantially different results in inexact arithmetic. During the forward sweep the propagation of error by the $t-th$ elementary function is largely determined by the size and possibly the spectrum of the elementary Jacobian f'_t at the current argument. Since the transposed of these matrices are the Jacobians of the corresponding adjoint operations it seems reasonable to conclude that the numerical conditioning of the reverse sweep is about the same as that of the corresponding forward sweep. Thus a stable algorithm for the evaluation of F will result in a good method for the accumulation of its adjoint and vice versa. There is continuing research and experimentation on how to best define and implement rounding or inclusion arithmetics. By overloading any such system can easily be combined with our method for evaluating adjoints by reverse accumulation.

For simplicity we make from now on the "unrealistic" but even in numerical analysis customary assumption that exact arithmetic can be performed in a ring $\mathcal{R}$ containing infinitely many elements. Somewhat surprisingly, most of differential calculus and analytic function theory can be extended to Banach algebras, i.e. commutative rings that form Banach spaces over the embedded field of complex numbers. Less appears to be known about the properties of linear and nonlinear systems over such "scalars". Nevertheless, the notion of differentiable vector functions and their adjoints are also easily extended, provided domain and range are free modules of the form $\mathcal{R}^p$. When these assumptions apply for all elementary functions the forward and reverse sweeps are meaningfully related and they can be executed without any modifications. It should be noted in particular that the transition from the original to the adjoint "code" does not introduce any division but merely multiplications and additions, which are well defined for arbitrary ring elements. Hence one should think of the reciprocal $v = 1/u$ as just another univariate function that is, like the natural logarithm, undefined at several arguments.

4.1 Differential Calculus on Normed Rings and Modules

In order to develop a differential calculus we require that $\mathcal{R}$ is endowed with a positive modulus $|\cdot|$ such that for any two $u, v \in \mathcal{R}$:

$$|u+v| \leq |u| + |v| \quad , \quad |u*v| \leq |u| \cdot |v| \quad , \tag{17}$$

Moreover, we assume that $\mathcal{R}$ is a real Banach algebra as well as a complete metric space with respect to the distance $d(u,v) \equiv |u-v|$. The last condition requires in particular that $|u| = 0$ if and only if $u = 0$. Just like in real or complex analysis a function function $f : \mathcal{R} \mapsto \mathcal{R}$ is said to be differentiable

at a point u if there exists a derivative value $f'(u) \equiv v$ such that

$$\lim_{|\Delta u| \to 0} |f(u + \Delta u) - f(u) - v * \Delta u| / |\Delta u| \quad = \quad 0 \qquad (18)$$

where Δu ranges over all sufficiently small but nonzero vectors in $\mathcal{R}$. If it exists the derivative value $v = f'(x)$ must be unique provided the reciprocal $1/\Delta u$ is well defined for arbitrarily small Δu. This property will be ensured by the assumptions below.

The set of functions for which k derivative exists and are continuous at all points in some open domain $\mathcal{N} \subset \mathcal{R}$ will be denoted by $\mathcal{C}^k(\mathcal{N})$ as usual. Any $f \in \mathcal{C}^1(\mathcal{N})$ is also differentiable as a mapping from $\mathcal{R}$ viewed as a Euclidean space into itself, but the converse is not true if $\mathcal{R}$ properly contains $\mathbf{R}$. Then the space of linear transformations on $\mathcal{R}$ has a higher dimension than $\mathcal{R}$ itself, so that not all Jacobian matrices can be represented by a single element from $\mathcal{R}$. Thus we see that that exploitation of the ring structure promises substantial economies in terms of storage and operations.

It can be easily checked that differentiation in the ring sense satisfies the usual rules

$$\begin{aligned} (f+g)'(u) &= f'(u) + g'(u), \qquad (19) \\ (f*g)'(u) &= f(u) * g'(u) + f'(u) * g(u), \\ g(f(u))' &= g'(f(u)) * f'(u), \qquad (20) \end{aligned}$$

provided the functions f and g are differentiable at their respective arguments. The only surprise is that the identity function $f(u) = u$ is not differentiable unless the ring has an identity element 1, as we will assume from now on. Excluding furthermore the possibility that an integer multiple of 1 equals zero, we may include real multiples of 1 and arrive at the following basic structure.

Assumption 1 *The ring $\mathcal{R}$ contains a homomorphic image $\mathbf{R}$ of the real numbers as a subring such that the multiplication $*$ in $\mathcal{R}$ makes it a Euclidean space of dimension $r < \infty$. An element $v \in \mathcal{R}$ is called* regular *or* singular *depending upon whether or not there exist a* reciprocal *$w \in \mathcal{R}$ such that $v * w = w * v = 1 \in \mathbf{R}$.*

Because all nonzero reals and in particular the identity element 1 are regular it follows immediately that the set of regular elements is open and that it's closure contains zero, which ensures the uniqueness of derivative values. The restriction that the vector space dimension r of $\mathcal{R}$ be finite is not necessary from an algebraic point of view. For example one could allow $\mathcal{R}$ to be the space of all polynomials, power series or even Laurent series in a certain number of real or complex variables. In theoretical terms this added generality would destroy the equivalence of norms and thus raise questions regarding the convergence of power series. In practical terms it would mean that the data structures needed to represent the elementary scalars $v \in \mathcal{R}$ tend to grow larger and more complex as the evaluation proceeds. In other words we would be subject to the well known phenomenon of *expression swell* in symbolic computations. Instead we prefer to remain in the realm of numeric computations by truncating the Taylor or Laurent series after each elementary function evaluation. Provided the truncation is carried out modulo an ideal of "higher order terms" the required algebraic properties are inherited by the resulting quotient ring.

In terms of computational complexity we will assume that the product $u * v$ of two ring elements u, v is considerably more expensive to compute than their sum $u + v$, unless one of the operands, say u, is is known to be real, in which case we may ommit the $*$ and simply write $u\ v$ or $u \cdot v$ instead. The cost of evaluating special functions and their derivatives for ring arguments is typically almost the same as that of a multiplication.

The product topology on a free module of the form $Z \equiv \mathcal{R}^n$ can be generated by the norm

$$\|(z_i)_{i=1}^n\| \equiv \left[\sum_{i=1}^n |z_i|^2\right]^{1/2} \tag{21}$$

Now suppose that $F : \mathbf{R}^n \mapsto \mathbf{R}^m$ is a vector function whose components have jointly continuous partial derivatives with respect to the components of x. Then it follows as usual that for arbitrary variations $\Delta x \in X$

$$\lim_{|\Delta x| \to 0} \|F(x + \Delta x) - f(x) - F'(x)\,\Delta x\| / \|\Delta x\| = 0 \quad , \tag{22}$$

where the Jacobian matrix $F'(x) \in \mathcal{R}^{n \times n}$ is formed by all first partial derivatives and matrix-vector multiplication is defined in the customary fashion. Since the differentiation rules (19) are also easily generalized to compatible pairs of continuously differentiable vector-functions, we see that Assumption 1 is sufficient to make the reverse accumulation of adjoints described in Section 2 possible and meaningful. Hence the user may define any suitable ring arithmetic via overloading and reverse accumulation will automatically yield appropriate adjoint vectors.

In order to extend the standard set of transcendental functions from R onto the full ring $\mathcal{R}$ one may simply use their power series representations, e.g. set for any $v \in \mathcal{R}$

$$exp(v) \quad \equiv \quad \sum_{k=0}^{\infty} \frac{v^k}{k!} \tag{23}$$

which converges with respect to the modulus $|\cdot|$ for any argument v. In fact, it follows from Taylors theorem for vector-valued paths that this is the only way in which the exponential can be extended to $\mathcal{R}$ as a C^∞ function. The same is true for all other real analytic functions. While the power series representations are of theoretical interest, they do not represent efficient schemes for actually computing function values, since that would involve a large number of multiplications and additions in the ring R. Instead one can exploit the fact that all special functions are quadratures or solutions of linear differential equations, which typically allow their evaluation at ring arguments with a computational effort comparable to that of one or two multiplications.

4.2 The Case of Univariate Taylor Series

For simplicity we consider in the remainder of this paper only the case where the ring R consists of the truncated univariate Taylor series

$$v \equiv v^{(0)} + \alpha v^{(1)} + \alpha^2 v^{(2)} + \cdots + \alpha^d v^{(r-1)} \in \mathcal{R}^{(r)} \quad ,$$

where the coefficients $v^{(i)}$ are real numbers. Therefore we can identify $\mathcal{R}^{(r)}$ as a real vector space with $\mathbf{R}^r$. The addition of elements in $\mathcal{R}^{(r)}$ applies componentwise and the multiplication is defined by the convolution

$$(u * v)^{(k)} = \sum_{i=0}^{k} u^{(i)} \cdot v^{(k-i)} \quad .$$

Multiplications are obviously significantly more expensive than additions, even if they are computed by fast methods (e.g., Fourier transforms [4]). One drawback of the fast methods appears to be that they do not allow the successive calculation of higher and higher order coefficients, as is required in ODE applications [3]. Recurrences for evaluating the exponential, logarithm and other special functions have been given by several authors (e.g., [16]). One can easily check that $\mathcal{R}^{(r)}$, viewed as a subspace of $\mathcal{R}^{(r+1)}$, is invariant with respect to the special functions so that their values on $\mathcal{R}^{(r+1)}$

can be obtained as updates of their values on $\mathcal{R}^{(r)}$ at a cost of order r. For this reason it may make sense to perform repeated forward sweeps, as mentioned at the end of Subsection 3.3.

Finally, we will demonstrate how real derivative vectors of the form (1) and (2) can be calculated using the truncated Taylor series arithmetic discussed above. Suppose we have evaluated a vector function $F(x)$ at some polynomial

$$x = \sum_{i=0}^{r-1} \alpha^i x^{(i)} \in \mathcal{R}^{(r)}$$

by executing the loop (11). Then the coefficients $y^{(d)} \in \mathbf{R}^m$ of the result $y = F(x)$ represent the vectors (1) for $d = 0, \ldots, r-1$ scaled by $1/d!$. Now given a real weight vector $\bar{y} \in \mathbf{R}^m$ one can calculate the adjoint vector

$$\bar{x} = \sum_{i=0}^{r-1} \alpha^i \bar{x}^{(i)} \quad = \quad \bar{y} \cdot F'(x)$$

by executing the reverse loop (13). In order to interpret the coefficients $\bar{x}^{(i)}$ we note that for a real perturbation $\Delta x \in \mathbf{R}^n$ and all $d < r$

$$\bar{y}[y^{(d)}(x + \Delta x) - y^{(d)}(x)] = \bar{x}^d \cdot \Delta x + o(\Delta x)$$

Thus we must have

$$\bar{x}^{(d)} = \nabla_x \langle \bar{y} \cdot y^{(d)}(x) \rangle \quad ,$$

which corresponds to a vector of the form (2), again scaled by $1/d!$. Thus we have demonstrated that the derivative vectors (1) and (2) can indeed be calculated by performing forward and reverse sweeps on rings of truncated series.

5 Summary and Conclusion

In this paper we considered how certain vectors of higher derivatives can be computed automatically from a user–supplied program for evaluating an underlying vector function. Under the assumption that the program is written in a language that allows the overloading of arithmetic operations and elementary functions, we described the generation of a symbolic representation called TAPE, and two numerical files called BIRTH and DEATH during the execution of the overloaded programs. These sequential data sets can be used to subsequently calculate derivative vectors of order d at a cost increase of $\mathcal{O}(d^2)$. In order to reduce the RAM requirement, we suggested to generate or rewrite the tapes in terms of locations rather than indices of the intermediate variables. As a result, the RAM requirement grows only by a modest amount and all other auxilliary data are accessed strictly sequentially.

The mathematical basis of the proposed methodology is the (adjoint) chain rule on normed rings of scalars. By performing the calculation on truncated Taylor series, one can evaluate derivative vectors of arbitrary order using the simple program structure designed originally for first derivatives.

References

[1] R. P. Brent and H. T. Kung: *Fast algorithms for manipulating power series.* ACM Journal, Vol. 25, pp. 581-595 (1978).

[2] R.H.F. Jackson, and G.P. McCormick: *Second order Sensitivity Analysis in Factorable Programming: Theory and Applications*, Mathematical Programming, Vol. 41, No. 1, pp. 1-28 (1988).

[3] G. F. Corliss and Y. F. Chang: *Solving Ordinary Differential Equations using Taylor series.* ACM TOMS, Vol. 8, pp. 114-144.

[4] R.J. Fateman: *Polynomial multiplication, powers and asymptotic analysis.* SIAM J. Computing, Vol. 3, pp. 196-213 (1974).

[5] A. Griewank: *On automatic differentiation* In: M. Iri and K. Tanabe (eds.), "Mathematical Programming: Recent Developments and Applications", Kluwer Academic Publishers, pp. 83-108 (1989).

[6] A. Griewank: *Direct Calculation of Newton Steps without Accumulatiung Jacobians.* In: T. F. Coleman and Yuying Li (eds.), "Large-Scale Numerical Optimization", SIAM, pp. 115-137 (1990) .

[7] A. Griewank, D. Juedes, and J. Srinivasan: *ADOL-C, A package for the automatic differentiation of algorithms written in C/C++*, Preprint MCS-180-1190, Argonne National Laboratory, Argonne, Illinois 60439 (1990).

[8] A. Griewank: *Achieving logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation* Preprint MCS-P228-0491, Argonne National Laboratory (1991).

[9] J.E. Horwedel, B.A. Worley, E.M. Oblow, and F.G. Pin: *GRESS Version 0.0 Users Manual,* ORNL/TM 10835, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37830, U.S.A. (1988).

[10] M. Iri: *Simultaneous Computations of Functions, Partial Derivatives and Estimates of Rounding Errors - Complexity and Practicality.* Japan Journal of Applied Mathematics, Vol. 1, No. 2 pp. 223-252 (1984).

[11] M. Iri, T. Tsuchiya, and M. Hoshi: *Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations.* Journal of Computational and Applied Mathematics, Vol. 24, pp. 365-392 (1988) .

[12] U. Kulisch and W. L. Miranker: *Computer Arithmetic in Theory and Practice.* Academic Press, New York (1980).

[13] K. Kubota and Masao Iri: *Padre2, version 1— User's Manual,* Research Memorandum RMI 90-01, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo (1990).

[14] S. Linnainmaa: *Taylor expansion of the accumulated rounding error.* BIT, Vol. 16, pp. 146-160 (1976).

[15] L.B. Rall: *Differentiation in PASCAL-SC: Type GRADIENT,* ACM TOMS, Vol. 10, pp. 161-184 (1984).

[16] L.B. Rall: *Differentiation Arithmetics.* In: "Computer Arithmetic and Self–validating Numerical Methods", Notes and Reports in Mathematics in Science and Engineering, Vol. 7, pp. 73-90, Academic Press, Boston (1990).

[17] B. Speelpenning: *Compiling fast Partial Derivatives of Functions given by Algorithms.* Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 (1980).

[18] O. Talagrand and P. Courtier: *Variational assimilation of meteorological observations with the adjoint vorticity equation. I: Theory.* Q.J.R. Meteorological Society, Vol. 113, pp. 1311-1328 (1987).