ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

# On Automatic Differentiation [1]

by

*Andreas Griewank*

Mathematics and Computer Science Division

Preprint ANL/MCS-P10-1088

November 1988

[1]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.

# On Automatic Differentiation

Andreas Griewank

Mathematics and Computer Science Division

Argonne National Laboratory
Argonne, IL 60439, U.S.A.

**Abstract**

In comparison to symbolic differentiation and numerical differencing, the chain rule based technique of automatic differentiation is shown to evaluate partial derivatives accurately and cheaply. In particular it is demonstrated that the reverse mode of automatic differentiation yields any gradient vector at no more than five times the cost of evaluating the underlying scalar function. After developing the basic mathematics we describe several software implementations and briefly discuss the ramifications for optimization.

**Key words:** gradient evaluation, automatic differentiation, symbolic differentiation, reverse accumulation, analytic derivatives.

## 1 Introduction

In 1982 Phil Wolfe [31] made the following observation regarding the ratio between the cost of evaluating a gradient with $n$ components and the cost of evaluating the underlying scalar function.

> *If care is taken in handling quantities which are common to the function and derivatives, the ratio is usually around 1.5, not n+1.* [31]

The main purpose of this article is to demonstrate that Phil Wolfe's observation is in fact a theorem (with the average 1.5 replaced by an upper bound 5) and that *care* can be taken automatically. This remarkable result is achieved by one variant of *automatic differentiation* [25], which simply implements the chain rule in a suitable fashion. The same approach can be used to compute second and higher derivatives. At least since the fifties these techniques have been developed by computational scientists in various fields, and several software implementations are now available. Although a theorem confirming Wolfe's assertion for rationals was published in 1983 by Baur and Strassen [2], the optimization community took little notice of these developments. This can be partly explained by a lack of clarity in the customary terminology.

Automatic differentiation is often confused with symbolic differentiation or even with the approximation of derivatives by divided differences. For algebraically rather simple functions, the explicit derivative expressions obtained by symbolic differentiation may be readable to an experienced user and thus provide an extremely useful extension of research with pencil and paper. However, for functions of any complexity in more than three variables, the analytic expressions for gradient or Hessian tend to take up several pages and are unlikely to facilitate any insights.

In this article we will concentrate on the goal of obtaining numerical derivative values at given arguments. The need for efficient and accurate derivative evaluations arises in particular during the iterative solution of nonlinear problems and the subsequent sensitivity analysis. Following several other authors, notably Iri [15], we will argue that **for these numerical purposes the reverse mode of automatic differentiation is far superior to symbolic differentiation or divided difference approximations**. The latter technique is always less accurate and about as costly as the forward form of automatic differentiation.

The paper is organized as follows. The remainder of this Section we briefly discuss the historical development and applications of automatic differentiation. In Section 2 we utilize two simple example functions to illustrate the characteristic properties of various techniques for evaluating gradients. In Section 3 we develop the two modes of automatic differentiation for the general case and conclude that the cost of evaluating gradients in the reverse mode is additive with respect to function composition. As a corollary we obtain Wolfe's assertion with 1.5 replaced by the uniform bound 5. Section 4 describes several implementations of automatic differentiation that require the user to do little more than provide a subroutine for the evaluation of the underlying function. In the final Section 5 we briefly discuss the implications of automatic differentiation on the design and selection of optimization algorithms.

The literature relating to automatic differentiation is extensive and very diverse. The main stream of research and implementation has been concerned with the automatic evaluation of gradients ( or more generally truncated Taylor series ) in the forward mode. This effort goes back at least to Beda et al [3] in the Soviet Union and Wengert [30] in the United States. Numerous other references are contained in the paper by Kedem [21], the books by Rall [25] and Kagiwada et al [19], and the recent report by Fischer [11]. In general the researchers in this main stream were unaware of the reverse mode or continued to consider it as a somewhat obscure approach of a rather theoretical nature.

Mathematically the reverse mode is closely related to adjoint differential equations. Nuclear engineers have long used *adjoint sensitivity analysis* [4], [5] to evaluate the partial derivatives of certain system responses (e.g. the reactor temperature) with respect to thousands of design parameters. This approach yields all sensitivities simultaneously at a cost comparable to only a few reactor simulations. In contrast, thousands of these lengthy calculations would be needed to approximate all sensitivities by divided differences. For a recent survey on the software and applications in this field see the paper by Worley [32]. Similarly, in atmospheric and oceanographic research, adjoints of the governing partial differential equations have been used to obtain the gradients of residual norms with respect to initial conditions and other unknown quantities [29]. Here the residuals represent discrepancies between observed and predicted conditions in the atmosphere or ocean. Even though these

3D calculations may involve millions of variables, the gradient of the sum of squares can be obtained at essentially the same cost as an evaluation of the residual vector itself. In order to avoid any storage and manipulation of matrices the gradient is then utilized in a conjugate gradient like minimization routine.

Apparently the first general purpose implementation of the reverse mode was the precompiler JAKE due to Speelpenning. In his unpublished thesis [28] Speelpenning showed that Wolfe's assertion is true, but did not state it formally. His original intention was to optimize the gradient code generated in the forward mode by sharing common expressions. During this attempt he realized that the optimal gradient code can be obtained directly without any optimization by (what we call here) the reverse mode of automatic differentiation. Several other papers proposing the reverse or *top down* mode are referenced in the survey [17]. This excellent article discusses also the closely related issue of estimating evaluation errors. Now let us examine various techniques for evaluating gradients on a couple of simple problems.

## 2 Comparisons on two Examples

The use of a cubic equation of state [24] yields the Helmholtz energy of a mixed fluid in a unit volume at the absolute temperature T as

$$f(x) = RT\sum_{i=1}^{n} x_i \log \frac{x_i}{1-b^Tx} - \frac{x^TAx}{\sqrt{8}b^Tx}\log\frac{1+(1+\sqrt{2})b^Tx}{1+(1-\sqrt{2})b^Tx} \quad ,$$

where $R$ is the universal gas constant and

$$0 \le x, b \in \mathbf{R}^n \quad , \quad A = A^T \in \mathbf{R}^{n\times n}.$$

During the simulation of an oil reservoir this function and its gradient have to be evaluated at thousands of points in space and time. Typically the number of fluid components n is restricted to less than 20, but we will include larger values in our comparative timings.

### 2.1 MACSYMAl Results on the Helmholtz Energy

First let us examine the results of symbolic differentiation with MACSYMA, version 309, distributed by Symbolics Inc. After entering $f(x)$ and computing its gradient with the *diff* command one may translate the symbolic representations into FORTRAN using the *fortran* command. On the following page we list the resulting code for the evaluation of $f(x)$ and the first component of its gradient when $n = 5$. Actually the original code had to be modified, mainly because it contained more than the maximum of 19 continuation lines allowed in FORTRAN 77. Due to our familiarity with the function we could break the expression for the first gradient component $g(1)$ in the middle, but in general that would be a rather challenging task. Even after this problem and some type conflicts in the original code were overcome the results are clearly unimpressive. Just imagine this code segment

```
RUTU=DSQRT(2.D0)
    F=0.0013564*(-(x(5)+x(4)+x(3)+x(2)+x(1))*DLOG(-b(5)*x(5)-b(4)*x(
   1  4)-b(3)*x(3)-b(2)*x(2)-b(1)*x(1)+1)+x(5)*DLOG(x(5))+x(4)*DLOG
   2  (x(4))+x(3)*DLOG(x(3))+x(2)*DLOG(x(2))+x(1)*DLOG(x(1)))-(x(5
   3  )*(x(5)*a(5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)+x(1)*a(
   4  5,1))+x(4)*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)*a(4,2
   5  )+x(1)*a(4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)*a(3,3)+x
   6  (2)*a(3,2)+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4)+a(2,
   7  3)*x(3)+x(2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a(1,4)*
   8  x(4)+a(1,3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))*DLOG(((RUTU+1
   9  )*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)/(
   :  (1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*
   ;  x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x( 1))
    g(1)=b(1)*(x(5)*(x(5)*a(5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)
   1  +x1(1)*a(5,1))+x(4)*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)
   2  *a(4,2)+x(1)*a(4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)
   3  *a(3,3)+x(2)*a(3,2)+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4
   4  )+a(2,3)*x(3)+x(2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a
   5  (1,4)*x(4)+a(1,3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))*DLOG(((RUTU
   6  +1)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1)
   7  )+1)/((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)
   8  +b(1)*x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b
   9  (1)*x(1))**2-(x(5)*a(5,1)+a(1,5)*x(5)+x(4)*a(4,1)+a(1,4)*x
   7  (4)+x(3)*a(3,1)+a(1,3)*x(3)+x(2)*a(2,1)+a(1,2)*x(2)+2*x(1)
   7  *a(1,1))*DLOG(((RUTU+1)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b
   7  (2)*x(2)+b(1)*x(1))+1)/((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b
   1  (3)*x(3)+b(2)*x(2)+b(1)*x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3
   6  )*x(3)+b(2)*x(2)+b(1)*x(1))
    g(1)=g(1)+0.0013625*(-DLOG(-b(5)*x(5)-b(4
   7  )*x(4)-b(3)*x(3)-b(2)*x(2)-b(1)*x(1)+1)+DLOG(x(1))+b(1)*(x(
   7  5)+x(4)+x(3)+x(2)+x(1))/(-b(5)*x(5)-b(4)*x(4)-b(3)*x(3)-b(
   7  2)*x(2)-b(1)*x(1)+1)+1)-((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+
   7  b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)*((RUTU+1)*b(1)/((1-RUTU
   7  )*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))
   7  +1)-(1-RUTU)*b(1)*((RUTU+1)*(b(5)*x(5)+b(4)*x(4)+b(3
   7  )*x(3)+b(2)*x(2)+b(1)*x(1))+1)/((1-RUTU)*(b(5)*x(5)+b(4
   7  )*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)**2)*(x(5)*(x(5)*a
   7  (5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)+x(1)*a(5,1))+x(4
   7  )*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)*a(4,2)+x(1)*a(
   7  4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)*a(3,3)+x(2)*a(3,2
   7  )+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4)+a(2,3)*x(3)+x
   7  (2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a(1,4)*x(4)+a(1,
   7  3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))/((b(5)*x(5)+b(4)*x(4)+b(
   7  3)*x(3)+b(2)*x(2)+b(1)*x(1))*((RUTU+1)*(b(5)*x(5)+b(4)*
   7  x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1))
```

had been inserted into a subroutine and subsequently the programmer made a trivial editing error. Then it would be quite difficult to determine by inspection whether the segment had been corrupted and nearly impossible to correct it. In other words the code is not only inefficient but unmaintainable.

While some aspects of MACSYMA's FORTRAN interface are annoying, they are by no means the root of our problems. The main culprit is the wrong-headed idea of generating separate expressions for the function and each gradient component, directly in terms of the independent variables. By definition this approach eliminates any possibility of utilizing common expressions during the evaluation. **Instead one should write a program for evaluating the function efficiently and then generate an extended program that evaluates the function and gradient simultaneously.** As we will see later the extended program can be generated automatically.

Everything may be done *by hand* on our second example

$$f(x) \;\equiv\; \prod_{i=1}^{n} x_i \;=\; x_1 \cdot x_2 \cdots x_{n-1} \cdot x_n$$

which was already used by Speelpenning [28]. Obviously the $i{-}th$ component of the gradient $\nabla f(x)$ is given by

$$\partial f / \partial x_i \;=\; \prod_{j \neq i} x_j \;=\; x_1 \cdots x_{j-1} \cdot x_{j+1} \cdots x_n$$

If calculated in this form each gradient component involves $n-1$ multiplications and is thus almost as expensive to evaluate as the function $f$ itself. Since symbolic differentiators generate separate algebraic expressions for each component of $\nabla f(x)$ they require exactly $n$ times as many arithmetic operations for evaluating function and gradient jointly as for evaluating the function by itself. Formally we may write $q\{f\} = n$, where

$$q\{f\} \equiv work\{f, \nabla f\} / work\{f\} \quad .$$

Since the work ratio $q\{f\}$ is even slightly larger for divided differences this may at first seem a fair price to pay. However, according to Wolfe's assertion we should be able to do a lot better, namely to bound $q\{f\}$ by a constant independent of $n$.

## 2.2 Automatic Differentiation of the Product Example

In order to obtain the gradient cheaply one could use the identity

$$\partial f(x) / \partial x_i = f(x) / x_i \quad if \quad x_i \neq 0 \quad .$$

Unfortunately, this 'solution' suggests that the efficient evaluation of gradients involves some special cancellations, which have to be detected by human inspection and require numerical exception handling when certain denominators are zero or small. Fortunately, for this example and other cases, **the gradient can be evaluated efficiently without any human intervention or numerical instabilities**.

In order to discuss the alternative methods we have to base the evaluation of the function and its gradient on sequential programs rather than a set of explicit expressions. Using an informal programming language we can evaluate $y = f(x)$ by the following code.

**Evaluation of Product**

```
$x_{n+1} = x_1$
$For\ i = n+2, n+3 \ldots 2n$
    $x_i = x_{i-n}\ x_{i-1}$
$y = x_{2n}$
```

Here and throughout the paper we will allocate all scalar quantities in a single memory vector $\langle x_i \rangle_{i=1\ldots m}$, starting with the independent variables $\langle x_i \rangle_{i=1\ldots n}$ and ending with a single dependent variable $x_m$. The issue of the storage requirements for actual computer implementations will be discussed in Section 4.

Since the intermediate quantities $x_{n+i}, i = 1 \ldots n$ are smooth functions they possess gradients $\nabla x_{n+i}, i = 1 \ldots n$ with respect to the independent variables $x_1, x_2, \ldots, x_n$. In particular we have $\nabla x_{2n} = g \equiv \nabla f$ and $\nabla x_{n+1} = e_1$. Evaluating the intermediate gradients by the chain rule we obtain the following expanded program.

**Forward Differentiation of Product**

```
$x_{n+1} = x_1$
$\nabla x_{n+1} = e_1$
$For\ \ i = n+2, n+3 \ldots 2n$
    $x_i = x_{i-n}\ x_{i-1}$
    $\nabla x_i = x_{i-1}\ e_{i-n} + x_{i-n}\ \nabla x_{i-1}$
$y = x_{2n}$
$g = \nabla x_{2n}$
```

This program evaluates both function and gradient simultaneously. It can be generated in a 'mechanical' fashion and is only about twice as long as the original program because each assignment to an intermediate quantity is simply augmented by the calculation of its gradient. This *forward* approach has been developed and advocated by several authors (See e.g. [3], [30], [21], and [25]). Various software implementations will be discussed in Section 4.

A simple count reveals that the calculation of our example gradient by the program above involves $\frac{1}{2}n^2$ nontrivial multiplications, so that $q \simeq n/2$. In general we must expect that the forward mode of automatic differentiation increases the number of arithmetic operations by the factor $n$, because each evaluation of an intermediate scalar quantity $x_i$ is accompanied by the calculation of the corresponding gradient vector $\nabla x_i$. Apparently Speelpenning was the first to notice that, instead of the gradient vector, only another scalar, say $\bar{x}_i$, needs to be associated with each quantity $x_i, i = 1 \cdots 2n$. In case of the product example one may define $\bar{x}_{n+i}$ as the product of all $x_j$ with $i < j \leq n$ and then set

$$\partial f / \partial x_i = \bar{x}_i \equiv x_{n+i-1}\ \bar{x}_{n+i}.$$

This calculation is performed by the following extended program.

**Reverse Differentiation of Product**

```
$x_{n+1} = x_1$
$For\ i = n+2, n+3, \ldots, 2n$
        $x_i = x_{i-n}\ x_{i-1}$                {Forward Sweep}
$y = x_{2n}$
$\bar{x}_{2n} = 1$
$For\ i = 2n, 2n-1, \ldots, n+2$
        $\bar{x}_{i-1} = \bar{x}_i\ x_{i-n}$            {Reverse Sweep}
        $\bar{x}_{i-n} = \bar{x}_i\ x_{i-1}$
$\bar{x}_1 = \bar{x}_{n+1}$
$g = \langle \bar{x}_i \rangle_{i=1\ldots n}$
```

This algorithm requires $3n - 3$ multiplications in order to compute the function and its gradient, so that now $q \simeq 3$. That is 50% more than the number of arithmetic operations required by the method based on cancellations, but now there is no need for any branching when one of the variables is small. The amazing fact is that this apparently tricky algorithm for the gradient of a product can be obtained by a general, straight-forward transformation from the original function evaluation program.

## 2.3 Experimental Comparison on Helmholtz Energy

Before discussing the details of this transformation in the following sections, let us list some empirically observed values for the work ratio $q\{f\}$ on our first example. The numbers in Table 1 represent the ratio between the execution times of an extended program that evaluates $f(x)$ and $\nabla f(x)$ jointly and of the original program that evaluates only $f(x)$ at a given argument. The entries in the first column represent the work ratio for divided differences, namely $n+1$ with $n$ being the number of variables. The three numbers in the second column were obtained as follows. The Helmholtz energy function $f(x)$ was entered into the algebraic manipulation package MAPLE [6] and then differentiated symbolically using the *grad* command. On a Sun 3/140 with 16 megabytes real memory, the symbolic generation of the gradient always took several minutes, and when $n$ was set to 30 the differentiation failed after 15 minutes due to a lack of memory space. The time for this process was not included in the listed work ratios, which reflect only the times needed to substitute the indeterminates $x_i$ by real arguments in the expressions for $f(x)$ and $\nabla f(x)$. For example when $n = 20$ the substitution took 7.13 and 160 seconds CPU time respectively.

The results in the third and forth column were obtained on an IBM XT using the programming language PASCAL-SC [22]. Like other modern languages this extension of

| **Div. Diff.** | **Symbolic** | **Forward** | **Reverse 1** | **Reverse 2** |
|---|---|---|---|---|
| 6 | 2.0 | 1.5 | 1.00 | 6.80 |
| 11 | 9.8 | 2.1 | 1.66 | 4.66 |
| 21 | 22 | 3.8 | 1.94 | 3.46 |
| 31 | - | 5.2 | 2.04 | 3.95 |
| 41 | - | 7.6 | 2.67 | 3.65 |
| 51 | - | - | 2.88 | 3.82 |
| 61 | - | - | - | 3.76 |
| 71 | - | - | - | 3.80 |
| 81 | - | - | - | 3.83 |
| FORTRAN | MAPLE | PASCAL-SC | PASCAL-SC | JAKEF |

Table 1: Observed work ratios on Helmholtz energy for $n = 5, 10, 20, \ldots, 80$.

standard PASCAL allows the transformation of a program for the evaluation of $f(x)$ into one that evaluates $f(x)$ and $\nabla f(x)$ by a process called *operator overloading.* This approach was first implemented by Rall [26],[27] in the forward mode of automatic differentiation. We have implemented the same approach in the reverse mode as described in Section 4. Again the entries in the table do not include the compilation times for the original and extended programs but represent the ratios of the respective execution times. The fifth column was obtained in almost the same way, except that the original program was written in FORTAN and then extended to the gradient routine by the precompiler JAKEF [14] (an update of Speelpennings original version JAKE [28]). The resulting pair of FORTRAN programs was run on the Sun 3 so that the execution times were naturally much smaller than those of the PASCAL-SC programs on the IBM XT. Nevertheless the comparison between runtime ratios provides some meaningful information.

As in the case of divided differences, the observed work ratios grow linearly with the number of variables $n$, for both symbolic differentiation and the forward mode of automatic differentiation. However, in the latter case the proportionality factor is only about .2 compared to 1.0 in case of the popular divided differences. The reverse mode of automatic differentiation in PASCAL-SC is always faster than the corresponding forward scheme, and the work ratio seems indeed uniformly bounded in n. The same is true for the FORTRAN version of reverse accumulation, though there the ratios are initially somewhat larger. Due to the limitation to 512K core memory, the forward and reverse implementation in PASCAL-SC can handle the Helmholtz energy function only up to 40 and 50 variables respectively. MAPLE exhausts the many times larger memory on the Sun much earlier. On the basis of our experience with MACSYMA and MAPLE we conclude that symbolic manipulators cannot be considered suitable tools for our purposes. Finally we note that a carefully handcoded routine for evaluating a suitable representations of the first four derivative tensors requires only about 1.5 times the computing time of evaluating the Helmholtz energy by itself. Thus we see that when runtime really counts, some mental effort may still be worthwhile.

# 3 Automatic Differentiation of Composite Functions

## 3.1 Composite Functions and their Computational Graph

Throughout this section we consider a function $y = f(x) : \mathbf{R}^n$ that is defined by a given sequential program of the following form.

**Original Program**

```
For i= $n+1, n+2, \ldots, m$
        $x_i = f_i \langle x_j \rangle_{j \in \mathcal{J}_i}$
  $y$  $= x_m$
```

Here the *elementary functions* $f_i$ depend on the already computed quantities $x_j$ with $j$ belonging to the index sets

$$\mathcal{J}_i \subset \{1, 2, \ldots, i-1\} \quad for \quad i = n+1, n+2, \ldots, m$$

In other words $f$ is the composition of $m - n$ elementary or *library* functions $f_i$, whose gradients

$$\nabla f_i = \langle \partial f_i / \partial x_j \rangle_{j \in \mathcal{J}_i}$$

are assumed to be computable at all arguments of interest.

For example, this is clearly the case when all $f_i$ represent either elementary arithmetic operations, i.e. + , - , * and / or nonlinear system functions of a single argument, e.g. logarithms, exponentials and trigonometric functions. Almost all scalar functions of practical interest can be represented in this *factorable* form, which has been used extensively by McCormick et al. [18]. Rather than restricting ourselves to unary and binary elementary functions we allow for any number of arguments $n_i \equiv |\mathcal{J}_i| < i$, where $|\cdot|$ denotes cardinality. In particular we may trivially interpret any function $f(x)$ as a composition of itself so that in the program above $f_{n+1} = f$ and $m = n+1, n_m = n$. More importantly, this general framework allows for user defined subroutines.

Sometimes it is very helpful to visualize the original program as a *computational graph* with the vertex set $\{x_i\}_{1 \leq i \leq m}$. An arc runs from $x_j$ to $x_i$ exactly if $j$ belongs to $\mathcal{J}_i$. With each arc one may associate the value of the corresponding partial derivative $\partial f_i / \partial x_j$. Because of the restriction on $\mathcal{J}_i$ one obtains an acyclic graph, whose minimal elements are the independent variables. Usually there are several linear orderings of the $x_i$ that are compatible with the partial ordering induced by the directed graph. Whenever two elementary functions do not directly or through intermediaries depend on each others result, they can be evaluated in either order or even concurrently on a parallel machine. This aspect has been examined in [9], but will not be pursued any further here. Also, in contrast to the analysis in [15], we will not use the graph structure for our complexity bounds.

$$\begin{tikzcd}
 & & & x_8 = f_8\langle x_7, x_4\rangle \\
 & & x_7 = f_7\langle x_6, x_3\rangle \arrow[ur, "\frac{\partial f_8}{\partial x_7}"] & \\
 & x_6 = f_6\langle x_5, x_2\rangle \arrow[ur, "\frac{\partial f_7}{\partial x_6}"] & & \\
x_5 = f_5\langle x_1\rangle \arrow[ur, "\frac{\partial f_6}{\partial x_5}"] & & & \\
x_1 \arrow[u, "\frac{\partial f_5}{\partial x_1}"'] & x_2 \arrow[uu, "\frac{\partial f_6}{\partial x_2}"'] & x_3 \arrow[uuu, "\frac{\partial f_7}{\partial x_3}"'] & x_4 \arrow[uuuu, "\frac{\partial f_8}{\partial x_4}"']
\end{tikzcd}$$

Figure 1: Graph for Product where $f_5\langle x_1\rangle = x_1$ and $f_i\langle x_j, x_k\rangle = x_j * x_k$ for $i = 6, 7, 8$.

For any reasonable measure of computational work on a serial machine we may assume that

$$work\{f\} = \sum_{i=n+1}^{m} work\{f_i\} \quad .$$

In defining $work\{f\}$ one may account for the number of certain arithmetic operations as well as fetches and stores from and to memory. Now let us develop the extended programs for evaluating the gradient $\nabla f$ jointly with $f$.

## 3.2 Automatic Differentiation with Forward Accumulation

Again denoting by $\nabla x_i$ the gradient of $x_i$ with respect to the independent variables $\langle x_j\rangle_{j=1\ldots n}$ we derive from the original program by the chain rule:

**Forward Extension**

```
$For\ i = 1, 2 \ldots n$
        $\nabla x_i = e_i$
$For\ i = n+1, n+2, \ldots m$
        $x_i = f_i\langle x_j\rangle_{j\in\mathcal{J}_i}$
        $\nabla x_i = \sum_{j\in\mathcal{J}_i} \frac{\partial f_i}{\partial x_j} \nabla x_j$
  $y = x_m$
  $g = \nabla x_m$
```

where $e_i$ denotes the $i-th$ Cartesian basis vector in $\mathbf{R}^n$.

Due to the assumed additivity of the work measure we find that

$$work\{f, \nabla f\} = \sum_{i=n+1}^{m} [work\{f_i, \nabla f_i\} + n\, n_i(mults + adds)] \quad ,$$

where the extra $n\, n_i$ arithmetic operations are needed to form $\nabla x_i$ as a linear combination of the $n_i$ gradient vectors $\nabla x_j$ with $j \in \mathcal{J}_i$. Here we have neglected the fact that for $j$ just above $n$, the gradient vectors $\nabla x_j$ will be sparse so that some arithmetic operations operations could theoretically be avoided. However, the added complexity of a suitable sparse implementation is unlikely to be justified by the savings, except in very special cases. Another possible alternative is to run through the basic loop $n$ times, each time only evaluating the partial derivatives $\partial x_i/\partial x_j$ with respect to one particular independent variable $x_j$. This implementation of forward accumulation is considerably less economical in terms of computational effort but requires only about twice as much storage as the original program. We will not consider this space saver solution in the remainder of the paper.

Now suppose that the evaluation of any library function $f_i$ requires at most $c\, n_i$ arithmetic operations, where $c$ is a common positive constant. Then it follows from the last equation that the work ratio defined above satisfies $q\{f\} \geq 1 + n/c$. This linear growth in the number of variables was clearly observed on the Helmholtz example and is not acceptable for large problems.

## 3.3 Automatic Differentiation with Reverse Accumulation

In order to obtain a method with a uniformly bounded work ratio we associate with each intermediate variable $x_i$ the scalar derivative

$$\bar{x}_i \equiv \partial x_m / \partial x_i$$

rather than the gradient vector $\nabla x_i$. By definition we have $\bar{x}_m = 1$ and for $i = 1 \ldots n$

$$\partial f(x)/\partial x_i = \bar{x}_i \quad .$$

As a consequence of the chain rule it can be shown (see e.g. [20]) that these *adjoint* quantities satisfy the relation

$$\bar{x}_j = \sum_{i \in \mathcal{I}_j} \frac{\partial f_i}{\partial x_j} \bar{x}_i \quad ,$$

where $\mathcal{I}_j \equiv \{i \leq m : j \in \mathcal{J}_i\}$. Thus we see that $\bar{x}_j$ can be computed once all $\bar{x}_i$ with $i > j$ are known. In terms of the program structure it is slightly more convenient to increment all $\bar{x}_j$ with $j \in \mathcal{J}_i$ for a known $i$ by the appropriate contribution $\bar{x}_i\, \partial f_i/\partial x_j$. This mathematically equivalent looping leads to the following extended program.

**Reverse Extension**

```
For   $i = n+1, n+2, \ldots\ldots, m$
          $x_i = f_i\langle x_j\rangle_{j\in\mathcal{J}_i}$                    {Forward Sweep}
          $\bar{x}_i = 0$
$y$        $= x_m$
$\bar{x}_m$       $= \gamma$
$\langle\bar{x}_i\rangle_{i=1}^n$ $= \bar{g}$
For   i $= m, m-1, \ldots, n+1$
          $\bar{x}_j = \bar{x}_j + \frac{\partial f_i}{\partial x_j}\bar{x}_i$   for all $j \in \mathcal{J}_i$        {Reverse Sweep }
$g$        $= \langle\bar{x}_i\rangle_{i=1}^n$
```

When the initial vector $\bar{g}$ is set to zero and $\gamma$ equals one, then the resulting vector $g$ is simply the gradient $\nabla f$. Otherwise we obtain for exactly the same computational effort the more general result

$$g = \bar{g} + \gamma\ \nabla f(x) \quad .$$

In other words the above program can increment a certain multiple of the gradient $\nabla f$ to a given vector $\bar{g}$ of the same length. This is exactly the operation we have to perform for each elementary function in the reverse extension. Hence we have additivity of the computational work in that

$$work\{f,\ \bar{g} + \gamma\ \nabla f\} = \sum_{i=n+1}^{m} work\,\{f_i,\ \bar{g}_i + \gamma_i\ \nabla f_i\}$$

for arbitrary scalars $\gamma_i$ and vectors $\bar{g}_i$ of length $n_i$. After division by the last equation of Subsection 3.1 one finds by elementary arguments that

$$Q\{f\} \equiv \frac{work\,\{f,\ \bar{g} + \gamma\ \nabla f\}}{work\{f\}} \leq \max_{n<i\leq m} Q\{f_i\} \quad ,$$

Note that $Q\{f\}$ is slightly larger than the work ratio $q\{f\}$ defined in Subsection 2.1. This means that the work ratio for $f$ is bounded above by the worst ratio for any of the library functions $f_i$, which is clearly independent of the total number of variables $n$. In other words **the set of functions $f$ for which the work ratio $Q\{f\}$ does not exceed a certain bound $\bar{Q}$ is closed with respect to composition.** This rather surprising result holds for a wide range of work functionals, provided memory space is unlimited and free. However as was mentioned above, memory access, i.e. fetches and stores, may be included as costs.

Now suppose the $f_i$ are restricted to the elementary arithmetic operations and standard univariate functions on a modern mainframe. For sine and cosine the work ratio lies just above two, and for all other system functions it is close to 1, because their derivatives come practically free once the function itself has been evaluated. Assuming that an addition is cheaper than a multiplication and a division costs at least 50% more than a multiplication, one finds that the largest work ratio is attained for the multiplication function $f_i(x_1, x_2) \equiv x_1 * x_2$. Therefore we may use the upper bound

$$\bar{Q} \equiv Q\{x_1 * x_2\} = \frac{3\ mults + 2\ adds + 5\ fetches + 3\ stores}{1\ mult + 2\ fetches + 1\ store} \leq 5 \quad .$$

Thus we can conclude that under quite realistic assumptions **the evaluation of a gradient requires never more than five times the effort of evaluating the underlying function by itself**. Obviously the bound of 5 is somewhat pessimistic and one might expect to incur an even smaller penalty for evaluating the gradient in practice. This was found to be true in our experiments on the Helmholtz example. On the other hand the extended program may involve communications overhead, e.g. extra subroutine calls, that is not included in our work measure.

While the reverse mode is clearly superior to the forward mode in terms of computational effort, it may require a lot more storage than the latter. As coded in Subsection 3.2 the forward extension associates with each scalar variable of the original program a gradient vector of length $n$. Hence the storage requirement grows by the predictable factor $n + 1$. This is true even if some variables are repeatedly updated during the function evaluation. In that case the associated vectors can also be overwritten by the gradient of the latest value of the variable. For example in the product program of Subsection 2.2 one would normally not allocate $n$ extra storage locations for the partial products $x_{n+i} = x_1 \ldots x_i$ but instead store them successively in the same place. Similarly all gradients $\nabla x_{n+i}$ in the corresponding extended program could be stored in a common $n$-vector.

In sharp contrast the reverse accumulation in Subsection 2.2 relies on all $n-1$ partial products $x_{n+i}$ being still available after the final function value $x_{2n}$ has been computed. Nevertheless, for this problem both modes require essentially the same storage, and on the Helmholtz energy function reverse accumulation uses slightly less space than forward accumulation. However, the difference in the memory requirement of the two methods can be much more dramatic.

## 3.4 Relations to Adjoints of Initial Value Problems

Suppose the evaluation of $f(x)$ involves the numerical solution of an initial value problem

$$y'(t) = F[y(t), t, x] \quad for \quad 0 \le t \le 1 \quad with \quad y(0) = y_0(x) \quad ,$$

where $y$ has $r$ components and $y_0$ is a smooth function of $x \in \mathcal{R}^n$. For a scheme with fixed step size h the result $y_h(1)$ will be a differentiable function of $x$. Provided $f$ depends in turn smoothly on the final values $y(1)$, the whole evaluation procedure fits (for each fixed mesh) into our framework. For simplicity let us assume that $f(x) = w^T y(1)$ with some fixed weighting vector $w \in \mathbf{R}^r$. During the numerical integration of the initial value problem with a p-stage scheme, one only has to store $p$ vectors of length $r$. In the forward mode the associated gradients would increase the storage requirement for this part of the program to $n\,p\,r$ locations. In the reverse mode we have to keep track of all $r/h$ intermediate values, which represent a discrete approximation of the solution function $y(t)\ for\ 0 \le t \le 1$.

Interestingly enough this is exactly the information one needs to calculate the gradient of $\nabla f(x)$ by solving the so called adjoint differential equation [23],

$$z'(t) = -F_y^T[y(t), t, x]z(t) \quad with \quad z(1) = w,$$

where $F_y$ denotes the Jacobian of the right hand side with respect to $y$. Since the boundary conditions are terminal and the sign on the right hand side is reversed, this linear system has exactly the same stability and stiffness properties as the original initial value problem. The desired gradient is given by

$$\nabla f(x)^T = z(0)^T \frac{\partial y_0}{\partial x} + \int_0^1 z(t)^T F_x[y(t), t, x] dt \quad ,$$

where $F_x$ denotes the Jacobian of the right hand side with respect to $x$. Thus we see that in the limiting continuous case, the evaluation of the gradient involves a definite integration based on the solution of an additional ODE with the same dimensions as the original initial value problem. Consequently the work ratio for appropriate discretizations should be close to 2 and certainly below 5.

**In fact we may interpret reverse accumulation simply as a discrete analog of the classical adjoint equations from the calculus of variations and control theory** [10]. Obviously the vector $y$ need not be finite dimensional, and one can adopt the theoretical arguments and numerical techniques to more general evolution equations in Hilbert spaces.

In terms of consistency it is probably preferable to discretize only the forward integration and then to apply reverse accumulation without explicitly referring to the adjoint differential equation at all. On the other hand separate discretizations of the original and adjoint equation allow the usage of standard software, with automatic differentiation only being used to obtain the Jacobian of the right hand side [19]. With the benefit of hindsight one could also construct an 'optimal' spline representation of $y(t)$ in order to economize on storage, especially if the integrator is adaptive and involves many tentative evaluations. Apparently nobody has studied the relative merits and computational performance of these various options.

When the differential equation is solved using an adaptive grid the actually computed function is only piecewise differentiable. As for any program that includes branching depending on values of variables, **automatic differentiation will generally yield the derivative of the smooth piece containing the current argument**. Obviously this is the best one can achieve, whereas divided differences may yield completely meaningless results if taken across a crack of the actually computed function. In transforming the original program to the extended routine with automatic differentiation, all control statements are left unaltered. In effect this means that the form of the loop in the original program may become dependent on the current argument. As pointed out by Kedem [21] errors may arise when reals are tested for equality. For example the conditional assignment

$$if \quad x \neq 0 \quad then \quad y = (1 - \cos x)/x \quad else \quad y = 0$$

would lead to the derivative $\partial y / \partial x$ at $x = 0$ being automatically evaluated as 0 rather than the correct value 1/2. Obviously the original programming leaves something to be desired in this particular example. In our implementation of the reverse mode in PASCAL-SC tests for equality involving real variables lead to warning messages.

## 3.5 Estimation of the Evaluation Error

The adjoint quantities $\bar{x}_i$ can be utilized to obtain good estimates of the total error in evaluating $f(x)$. Suppose one knows that the actually computed intermediate values $\tilde{x}_i$ satisfy for each $i > n$

$$|\tilde{x}_i - f_i\langle \tilde{x}_j\rangle_{j\in\mathcal{J}_i}| \leq \delta x_i \quad .$$

Moreover, let us assume that the discrepancies between the actual inputs $\langle \tilde{x}_i\rangle_{i=1\ldots n}$ and their ideal values $\langle x_i\rangle_{i=1\ldots n}$ are bounded by data tolerances $\langle \delta x_i\rangle_{i=1\ldots n}$. Then one can expect that the actually computed final value $\tilde{x}_m$ satisfies

$$|\tilde{x}_m - f(x)| \leq \sum_{i=1}^{m} |\bar{x}_i|\, \delta x_i \quad .$$

As shown by induction in [1] this inequality must hold if all functions $f_i$ are linear and the adjoint values $\bar{x}_i$ are exact. Even though these two assumptions are rather unrealistic the right hand side above was found in [17] to provide a usually somewhat pessimistic upper bound on the total error. In that paper the local error bounds $\delta x_i$ were obtained from the machine precision of the computer in question. However, other sources of local error (such as discretizations, the approximation of a transcendental function by rationals or the uncertainty of certain problem parameters) could be accounted for as well.

Since the local evaluation errors are rarely correlated and usually unbiased, it makes sense to consider them as stochastically independent random variables with zero mean and standard deviations $\delta x_i$. This assumption implies that the standard deviation of $\tilde{x}_m - f(x)$ is simply the $l_2$-norm of the m-vector $\langle \bar{x}_i\ \delta x_i\rangle_{i=1\ldots m}$ rather than the $l_1$ norm occuring on the right hand side above. Iri et al. found that this error estimate was somewhat tighter on their test problems. Either choice is certainly far superior to the ad hoc guesses that users currently have to make in order to specify tolerances for stopping criteria in iterative methods. Therefore these error estimates could be incorporated into optimization codes, to provide optimal solution accuracy without inconveniencing the user.

## 3.6 Extension to Higher Derivatives

In the forward mode the Hessian $\nabla^2 x_m$ of $x_m = f(x)$ can be obtained by updating for $i = n+1\ldots m$

$$\nabla^2 x_i = \sum_{j\in\mathcal{J}_i}\left[\frac{\partial f_i}{\partial x_j}\nabla^2 x_j + \sum_{k\in\mathcal{J}_i}\nabla x_j \frac{\partial^2 f_i}{\partial x_j \partial x_k}(\nabla x_k)^T\right]$$

starting with $\nabla x_i = e_i$ and $\nabla^2 x_i = 0$ for $i = 1\ldots n$. Similar chain rules of differentiation apply for third and higher derivative tensors. While the inclusion of these recursive relations into the original program provides in principle little difficulty, the resulting computational effort is at least of order $(m-n)n^p$, where p is the degree of the derivative tensor. In particular the evaluation the Hessian matrix in forward mode will usually be roughly $n^2$ times as expensive as the function itself.

Applying the complexity bound for the reverse mode separately to each component of the gradient one finds that

$$work\{\nabla^2 f\} \leq \sum_{i=1}^{n} work\left\{\nabla\left(\frac{\partial f}{\partial x_i}\right)\right\} \leq \bar{Q} \sum_{i=1}^{n} work\left\{\frac{\partial f}{\partial x_i}\right\} \quad .$$

After division by $work\{f\}$ we obtain in agreement with the results in [17] and [11]

$$\frac{work\{\nabla^2 f\}}{work\{f\}} \leq \bar{Q}\ \frac{\sum_{i=1}^{n} work\{\partial f/\partial x_i\}}{work\{\nabla f\}} \cdot \frac{work\{\nabla f\}}{work\{f\}} \leq n\bar{Q}^2 \quad .$$

In terms of powers of $n$ this bound is unfortunately optimal, as one can see on the simple example

$$f(x) = .5[x^T x + (a^T x)^2]\ ,\ \nabla f(x) = x + (a^T x)a\ ,\ \nabla^2 f(x) = I + aa^T \quad .$$

Here the function and gradient involve both $2n$ multiplication, whereas the accumulation of the Hessian requires certainly $.5n^2$ multiplications.

Fortunately, it is often sufficient to calculate derivative vectors of the form

$$\begin{aligned}\nabla^{1+p} f(x) v_1 v_2 \ldots v_p &= \nabla\left[\nabla^p f(x) v_1 v_2 \ldots v_p\right] \\ &= v_p^T\left(\nabla\left[\nabla^p f(x) v_1 v_2 \ldots v_{p-1}\right]\right)\end{aligned}$$

where the $n$-vectors $v_j, j = 1 \ldots p$ are given directions. For example Hessian-vector products of the form $\nabla^2 f(x)\ v_1$ can be used in the conjugate gradient method (See e.g. [8] and [20]). Second and third derivatives of the form $\nabla^2 f(x) v_1 v_2$ and $\nabla^3 f(x) v_1 v_2 v_3$ characterize the quadratic and cubic turning points [12] of bifurcation theory. Moreover, the gradients of these scalars involve terms of the form $\nabla^3 f(x) \tilde{v}_1 \tilde{v}_2$ and $\nabla^4 f(x) \tilde{v}_1 \tilde{v}_2 \tilde{v}_3$, which need be evaluated during the calculation of the turning points by Newton's method. Selected second derivatives of the Lagrangian occur in the gradient of smooth exact penalty functions [7] for constrained optimization.

According to the second equation above, the desired vector of $p + 1 - st$ derivatives is the gradient of the dot product between $v_p$ and an analogous vector of $p - th$ derivatives. Hence it may be computed recursively using $p + 1$ sweeps of reverse gradient accumulation. This shows that evaluating the left hand side above should only be about $5^{1+p}$ times as costly as evaluating the scalar function $f$ itself. Thus we have exponential growth in the order of the derivative $p$ but still no dependence on the number of variables $n$.

## 4 Computer Implementations of Automatic Differentiation

So far we have not really justified the adjective *automatic* because all program transformations were carried out *by hand*. Moreover, we can certainly not expect that the scalar function $f(x)$ is supplied by the user in form of the Original Program in Section 3.1. Also, our specification of the reverse mode via the extended program in Subsection 3.3 is not complete, because the required partial derivatives may be evaluated either during the forward or the reverse sweep. Either variant has been implemented and yields certain advantages.

## 4.1 Immediate versus Delayed Differentiation

The first variant might be called immediate differentiation with reverse accumulation. Provided only first derivatives are required, every elementary function is linearized at its current arguments during the forward sweep, and only the computational graph with the nodes $x_i$ and the arc values $\partial f_i / \partial x_j$ needs to be stored in a suitable fashion. Even the nodal values $x_i$ are no longer required after the forward sweep, and they may be overwritten by the corresponding adjoint values $\bar{x}_i$ during the reverse sweep. User defined subroutines that return their gradient together with the function value are easily incorporated.

Similarly, if there are segments of code that produce only one or two scalar values for the subsequent calculations, the corresponding gradients can be preaccumulated in a local reverse sweep. In other words, these scalars may be interpreted as $super-elementary$ functions of the variables that enter into the segment, and their gradients can be computed during the forward sweep. This applies in particular to single assignment statements with complicated right hand sides, e.g.

$$x_3 = (x_1 + 3x_2)^2 + \sin^2 x_1 \exp(.2x_2) \quad .$$

Here the the representation of $x_3$ as a factorable function of $x_1$ and $x_2$ involves six unary functions and three binary arithmetic operations. Thus we have originally $12 = 6 + 2 * 3$ partial derivatives as arc values. Preaccumulation of the partial derivatives $\partial x_3 / \partial x_1$ and $\partial x_3 / \partial x_2$ would cut that number to 2. Another example is the product considered in Section 2, which might occur as a super-elementary function in a larger program. Preaccumulating its gradient would essentially halve the number of arcs, whose origins, destinations and values have to be stored until the global reverse sweep.

Except in the simple cases mentioned above, the detection of suitable super-elements or *funnels* [28] requires some combinatorial analysis of the computational graph. If the same function is evaluated over and over such a potentially very large preprocessing effort may well be justified. However, it probably will only be economical when the graph is essentially static, i.e. the control flow of the original program is largely independent of the variable values. As far as we know this kind of combinatorial optimization on the graph has not yet been implemented.

A major disadvantage of immediate differentiation is the impossibility of obtaining higher directional derivatives after the forward sweep has been completed. To this end one has to construct a complete representation of the computational graph at the current argument, rather than just its linearization. In other words one has to store the type and data dependence of each elementary function in a suitable symbol table. In a way this doubles up the structural information that is already contained in the program.

## 4.2 FORTRAN Precompiler

There are at least three such implementations, namely JAKEF [14], GRESS [13], and PADRE2 [17]. All three precompilers require the user to supply a source code for the evaluation of $f(x)$ in some dialect of FORTRAN. The dependent and independent variables must be nominated through explicit declarations or a naming convention. The source code is then fed to the precompiler, which analyses its arithmetic assignment statements very much like a normal compiler. As we have mentioned before the control statements remain unaltered. All calculations involving real variables are broken down into elementary arithmetic operations and univariate system functions, e.g. exponentials or trigonometric functions. For each of these elementary functions $f_i$ the precompiler has built in expressions of the one or two partial derivatives $\partial f_i/\partial x_j$.

Using this 'knowledge' the precompiler can construct an extended FORTRAN program that evaluates the partial derivatives simultaneously with each elementary function. In the forward mode of GRESS, these local partial derivatives are used immediately to calculate the full gradient $\nabla x_i$ of the intermediate value $x_i$ with respect to the independent variables nominated by the user. In the case of JAKEF and the reverse mode of GRESS, the local partials are stored as arc values with a suitable encoding of their origin and destination, i.e. the $j-th$ and $i-th$ node respectively. PADRE2 delays the differentiation by storing instead a symbol identifying the elementary function and the current argument, so that its first and possibly higher derivatives can be evaluated during the reverse sweep. To effect the reverse sweep the precompilers insert a call to a standard accumulation subroutine at the end of the program.

The resulting extended FORTRAN programs rely on runtime support packages containing various standard subroutines and possibly also problem specific scratch files. The user then compiles and links the whole suite to obtain an executable code for evaluating the function, its gradient, and in the case of PADRE2 also second derivatives or error estimates. As an example the next page displays the FORTRAN subroutine PROD that evaluates the product of $n$ independent variables followed by the subroutine PRODJ obtained by precompiling PROD with JAKEF. The in-line comments on the right were added later and would naturally result in compilation errors.

Apart from the five subroutines called in the extension PRODJ there are two other subroutines in the runtime support library of JAKEF. Its total length is less than 150 lines of FORTRAN. When calling PRODJ the user has to provide the integer work arrays IFS and the real work array RFS with a sufficiently large common length LFS. The precompiler cannot provide a lower bound on LFS, because the storage requirement is usually a function of the number of variables and other problem parameters. This difficulty occurs in all reverse implementations, whereas the storage requirement in the forward mode is predictable.

Even though we have had no opportunity to test it, the recently released package GRESS, developed at Oak Ridge National Laboratory, appears to be the most versatile and user friendly precompiler for automatic differentiation that is currently available. It operates in the forward or reverse mode and allows for user defined functions as well as implicit relationships. PADRE2 is the only precompiler capable of producing second derivatives and error estimates, but as yet it is only documented in Japanese. JAKEF is quite efficient but does not allow user defined subroutines.

**FORTRAN subroutine for evaluating product**

```
      SUBROUTINE PROD(N,X,F)
      INTEGER N,I
      DOUBLE PRECISION F,GRAD
      DOUBLE PRECISION X(N)
CONSTRUCT D(F)/D(X) IN GRAD(N)                    {Nominate the dependent
      F = 1.D0                                      /independent variables }
      DO 10 I = 1,N
         F = F*X(I)
10    CONTINUE
      RETURN
      END
```

**Extended FORTRAN program generated by JAKEF**

```
      SUBROUTINE PRODJ(N,X,F,GRAD,YGRAD,LYGRAD,RFS,IFS,LFS)
      INTEGER LFS,IFS(LFS)                        { Lot's of extra storage }
      DOUBLE PRECISION RFS(LFS),TGRA(543)
      INTEGER N,I,LQ00,LQ01,LYGRAD,IGRAD,RGRAD,IX
      DOUBLE PRECISION X(N),F,GRAD(N),YGRAD(LYGRAD)
      IX = 544
      CALL DPINIT(IX+N,LYGRAD)                    { Initialization Routine }
      CALL DMIT0(1,RFS,IFS,LFS)                   {Storage of zero arc for
      F = 1.D0                                          constant assignment}
      LQ00 = 1
      LQ01 = N
      DO 90001 I = LQ00,LQ01                      {Loop logically unaltered}
      CALL DMIT2(1,X(I),IX+I,F,1,RFS,IFS,LFS)     {Storage of two arcs for
      F = F*X(I)                                             multiplication}
90001 CONTINUE
90000 CONTINUE
      RGRAD = 0
      CALL DPGRAD(YGRAD,LYGRAD,1,RGRAD,IGRAD,RFS,IFS,LFS)  {Accumulation
      CALL DPCOPY(GRAD,IGRAD,1,YGRAD(IX+1),N)                of gradient}
      RETURN
      END
```

## 4.3 Operator Overloading

The use of a precompiler means in effect that the original program is compiled twice, with a rather cryptic extended source code being generated as a by product. Hence one may ask, whether it is not possible to saddle the main compiler with the task of issuing the instructions that have to be executed in order to evaluate certain derivatives. This is in fact possible by a facility called *operator overloading*, which is available in most modern computer languages, including hopefully FORTRAN 8X. The key idea here is that the programmer can define new types of variables, whose occurence as arguments of an elementary function triggers the compiler to issue additional instructions. The source code itself remains essentially unchanged.

Apparently the first implementation of this kind is due to Kedem [21]. Since FORTRAN itself does not support overloading, he used the general purpose precompiler AUGMENT, which allowed the user to write the original program in a Taylor made extension of FORTAN. The resulting source code was then precompiled into standard FORTRAN by AUGMENT. Since most of its facilities are more conveniently available in modern computer languages, AUGMENT is no longer supported by its authors or anybody else. Kedem's extension of FORTRAN enabled the user to compute gradients or truncated Taylor series in the forward mode of automatic differentiation.

A few years later Rall [26] achieved a much cleaner implementation of the forward mode in the language PASCAL-SC, an extension of PASCAL for PC Compatibles distributed by Teubner and Wiley [22]. The transformation process is extremely simple. Suppose we have a standard PASCAL code for the evaluation of a function in the variables $X[1..N]$ of type REAL. Then the $X[I]$ and all real variables that depend on them are redeclared to be of the new type GRADIENT, which is completely problem independent. Each variable $XJ$ of type GRADIENT is a record consisting of a scalar part $XJ.F$ and a vector part $XJ.D[1..N]$. At each stage of the calculation the vector part represents the gradient of the scalar part with respect to the independent variables $X[1..N]$. The vector part of the independent variable $X[I]$ is initialized as the i-th Cartesian basis vector. Whenever an argument of type GRADIENT occurs in an elementary arithmetic operation, say the assignment $Z := X * Y$, the compiler looks for an appropriate overloading of the usual elementary operation on REALs. Therefore Rall supplied small, problem independent operator declarations for every possible combination of arguments, e.g. GRADIENT*GRADIENT, REAL*GRADIENT, and GRADIENT*REAL. In the last case for example, both the scalar and vector part of the first variable are multiplies by the second variable, which is of type REAL. Unfortunately PASCAL-SC does not allow the overloading of standard functions, so that the definition of $SIN(X)$ cannot be extended to arguments $X$ of type GRADIENT. Instead one has to introduce a new function $GSIN(X)$ that evaluates and differentiates the sine for arguments of type GRADIENT. This and some other limitations of PASCAL-SC require minor modifications of the program body. Any such changes could be avoided in a more powerful programming language such as C++.

The reverse mode of automatic differentiation can be implemented in a very similar way. Instead of GRADIENT we define a new type VAREAL that represents a record consisting of one REAL value and two pointers to other VAREALs. In contrast to the length of the vector part in GRADIENT, the size of each record of type VAREAL does not depend on the total number of independent variables. At execution time the extended program generates a doubly linked list of such records to represent the linearization of the computational graph at the current argument. Since they have to manipulate this data structure the overloaded operators for arguments of type VAREAL are logically more complicated than those for arguments of type GRADIENT in Rall's implementation. However, according to columns 2 and 3 of Table 1 in Subsection 2.3 the reverse mode is always faster than the forward mode, even when the number of variables and hence the difference in the number of arithmetic operations is small. This may partly be due to the lack of a mathematical coprocessor or floating point accelerator on the IBM PC in use. On systems with such devices the generation and manipulation of the doubly linked list might be relatively more expensive and thus shift the balance a bit in favor of the forward mode. Possibly for the same reason, it was found that recreating the list during each of several function evaluations is no more expensive than reusing the pointers from the first evaluations during subsequent calls. Overloading as such has no bearing on the execution time, because the type dependent decision which declaration of an operator applies at a particular occurence in the code is already made during the compilation. Again using the product example, we have listed on the next page the original evaluation program in PASCAL-SC and its modification for reverse differentiation via operator overloading. The program on the left simply reads in the nine variable values and prints out their product. The program on the right does exactly the same and then prints out the nine components of the gradient at the given argument.

The central sections of both codes are almost identical, except that the one on the right needs the conversion function VARY in assigning real values to variables of the new type VAREAL. Conversely the function EVAL extracts the real value from a VAREAL, which is needed in particular for output operations. The type VAREAL, the functions VARY and EVAL, the gradient accumulation procedure ACCUMULATE, the multiplication operator * between VAREALs, and the two pointer variables TAIL and SPARE are all defined in the problem independent header file VHEAD.SRC occuring in the compiler directive $INCLUDE right at the top. The explicit initialization of TAIL and SPARE, and the two conversion functions could be avoided in a programming language like C++, where the assignment operator can also be overloaded. Here, any oversight in making the required modifications will result in compile or run time errors. If the independent variables are declared as VAREALs and program executes normally, then the gradient values should be correct.

Compared to precompilation overloading probably requires more user sophistication but on the other hand it clearly offers more flexibility. Provided all subprograms are compiled together, either mode of automatic differentiation in PASCAL-SC can deal with user defined functions and even recursive procedure calls. This does not require any extension or modification of the header file. Higher derivatives and some optimization of the computational graph can also be implemented by overloading. The forward evaluation of general and structured Hessians in the advanced language ADA is discussed by Dixon and Mohseninia in [8]. When the currently proposed standard for FORTRAN 8X is actually implemented one of the major objections to operator overloading will be removed.

```
PROGRAM PROD(INPUT,OUTPUT);

VAR    X : ARRAY[1..9] OF REAL;
       Y,T : REAL;
       I,N : INTEGER;
BEGIN


  N := 5;
  Y := 1;
  FOR I := 1 TO N DO
  BEGIN
    READ(T);
    X[I] := T;
    Y := Y*X[I]
  END;

  WRITELN(Y);




END.
```

**Program for Product Example**

```
PROGRAM PROD(INPUT,OUTPUT);
$INCLUDE VHEAD.SRC
VAR    X : ARRAY[1..9] OF VAREAL;
       Y : VAREAL; T : REAL;
       I,N : INTEGER;
BEGIN
  TAIL := NIL; SPARE := NIL;

  N := 5;
  Y := VARY(1);
  FOR I := 1 TO N DO
  BEGIN
    READ(T);
    X[I] := VARY(T);
    Y := Y*X[I]
  END;

  WRITELN(EVAL(Y));

  ACCUMULATE(Y);
  FOR I := 1 TO N DO
    WRITELN(EVAL(X[I]));

END.
```

**Extension with Reverse Differentiation**

## 5 Conclusions and Discussion

Like several previous authors we conclude that in theory and practice the gradients of all functions defined by computer programs can be evaluated cheaply and automatically. This observation suggests the reexamination of the many arguments in the optimization literature, that rely at least implicitly on the seemingly reasonable assumption, that gradients codes are often hard to to come by and run typically much slower than the corresponding function routine.

Since truly derivative-free algorithms rarely have worked for more than a handful of variables, many researchers recommend the approximation of gradients by central or one-sided differences. Whenever this classical technique can be applied at all, we must have a reasonably accurate evaluation algorithm, in which case automatic differentiation provides a far superior alternative. Provided there is enough storage, reverse accumulation yields truncation error free gradient values at less than $5/n$ times the computing time of divided differences. This technique has been successfully implemented on problems in nuclear engineering and oceanography with thousands or even millions of variables. Should the function evaluation be so lengthy that the storage of all intermediate results is impossible, then one can still employ the forward mode to achieve better accuracy at essentially the same cost as divided differences.

Many line search procedures avoid the evaluation of the gradient at trial points before these have been accepted as the next main iterate. This strategy could still make sense, since we found that the gradient may well be four or five times more expensive to evaluate then the function. Also, the cubic interpolation made possible by the value of the directional derivative at the trial point destroys the simplicity of usual quadratic interpolation. Moreover the improved accuracy of the cubic interpolants rarely leads to a significant reduction in the overall number of evaluations or iterations. On the other hand, keeping two evaluation routines ( one without and one with the gradient) and calling them successively at all main iterates does not seem that economical either.

Penalty functions have long been used to convert constrained optimization problems into unconstrained problems. If one wants the penalty functions to be exact, i.e. attain local minima right at the solutions of the constrained problem, then there are basically two choices. Either the penalty function is nonsmooth or it depends explicitly on the gradients of the objective and constraint functions [7]. In the latter case the resulting gradient and Hessian depend on second and third derivatives of the original problem functions respectively. Since this additional level of differentiation was thought to be unacceptable, nonsmooth penalty functions have generally been preferred. However, automatic differentiation can produce the restricted second derivative terms in the gradient of smooth exact penalty functions at a reasonable cost, namely a fixed multiple of evaluating the objective and constraint functions. Therefore a suitable implementation of unconstrained BFGS could be both user friendly and efficient, especially since the troublesome Maratos effect of nonsmooth penalty functions cannot occur here.

The combination of automatic differentiation with the variable metric method BFGS recommended above may seem a strange mixture. Indeed, some researchers in automatic differentiation feel that the development of quasi-Newton methods was an emergency measure, which is outdated now that we can obtain the Hessian automatically. This seems to us a rather premature assessment. As we have seen in Subsection 3.6 the evaluation of a Hessian-vector product by either mode of automatic differentiation may be up to $5n$ times as expensive as that of the gradient. Thus we must expect that sometimes an exact or inexact Newton method based on automatic differentiation of the gradient will be less efficient than the corresponding finite difference version. In view of the trouble with negative curvature one may then prefer the simple and usually quite efficient BFGS method with line-search.

In any event automatic differentiation should allow the design of an optimization package that requires the user only to supply source code for the evaluation of the objective and constraint functions. The generation of the corresponding gradient codes, the detection of sparsity, and the determination of the maximal achievable solution accuracy, could all be done automatically. Ideally, the selection of a suitable linear equation solver for the computation of steps on large structured problems could also be left to the package.

In nonlinear least squares it is usually assumed that the calculating the gradient of the residual norm requires the evaluation of the full Jacobian. Hence, the argument goes, we might as well fully utilize this derivative information by employing a Gauss-Newton like procedure. However, as is the case for certain inverse problems [29], the Jacobian matrix may be huge and dense, whereas reverse accumulation always yields the gradient cheaply. Then nonlinear conjugate gradients or a variable metric method with limited memory is clearly the only choice. On the other hand, there are many problems, where the Jacobian is of moderate size and costs little more than the residual vector to evaluate.

Throughout this paper we have restricted our attention to a scalar valued function $f(x)$ in $n$ variables. Naturally all results and techniques can be separately applied to the $m$ components of a vector valued function $F(x)$. However, this approach may be far from optimal if the component functions are closely related, i.e. have many common expressions. Also, if $m$ is significantly larger than $n$ the forward mode of automatic differentiation is likely to be cheaper. Currently there appears to be no clearly superior strategy for the evaluation of derivative matrices (rather than vectors).

Even though the underlying mathematics are straight forward much remains to done in the field of Automatic Differentiation. With regards to general purpose differentiation software for various machine architectures, the problems are mainly of a computer science nature. However, some combinatorial analysis of the graph structure might be beneficial for the optimal evaluation of derivative matrices and the local preaccumulation of gradients, which was briefly mentioned in Subsection 4.1. Also, as in the case of evolution equations discussed in Subsection 3.4, there are probably other problem classes in which the reverse sweep has a natural interpretation and can be implemented in various ways. Finally, automatic differentiation could and should be integrated into numerical packages for special purposes, such as optimization, stiff differential equation, boundary value problems, optimal control, and path-following with bifurcation analysis. This process would be a lot simpler and more widely acceptable, if the next FORTRAN standard were to allow user-defined types with function and operator overloading.

## 6 Acknowledgements

In preparing this paper the author was aided by the incisive comments of Jorge Moré, Steven Wright, and Rob Womersley.

## References

[1] F.L. Bauer (1974). "Computational Graphs and Rounding Errors", *SINUM*, Vol.11, No.1, pp.87-96 .

[2] W. Baur and V. Strassen (1983). "The Complexity of Partial Derivatives", *Theoretical Computer Science*, Vol. 22, pp.317-330.

[3] L.M. Beda et al (1959). "Programs for Automatic Differentiation for the Machine BESM", Inst. Precise Mechanics and Computation Techniques, Academy of Science, Moscow.

[4] D.G. Cacuci (1981). "Sensitivity Theory for Nonlinear Systems. I. Nonlinear Functional Analysis Approach", Journal of Mathematical Physics, Vol.22, No.12, pp.2794-2802.

[5] D.G. Cacuci (1981). "Sensitivity Theory for Nonlinear Systems. II. Extension to Additional Classes of Responses", Journal of Mathematical Physics, Vol.22, No.12, pp.2803-2812.

[6] B.W.Char, K.O.Geddes, G.H.Gonnet, M.B.Monegan, and S.M.Watt (1988). "MAPLE Reference Manual, Fifth Edition", Symbolic Computation Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

[7] G. Di Pillo and L. Grippo (1986). "An Exact Penalty Method with Global Convergence Properties for Nonlinear Programming Problems", *SIAM J. Control Optim.* Vol.23, pp.72-84.

[8] L.C.W.Dixon and M.Mohseninia (1987). "The Use of the Extended Operations Set of ADA with Automatic Differentiation and the Truncated Newton Method", *Technical Report No.176*, The Hatfield Polytechnic, Hatfield, U.K.

[9] L.C.W.Dixon (1987). "Automatic Differentiation and Parallel Processing in Optimisation", *Technical Report No.180*, The Hatfield Polytechnic, Hatfield, U.K.

[10] Iu.G.Evtushenko (1982) . "Metody resheniia ekstremal'nykh zadach ikh primenenie v sistemakh optimizatsii", Nauka Publishers, Moskow

[11] H. Fischer (1987). "Automatic Differentiation: How to compute the Hessian matrix", *Technical Report #104A*, Technische Universität München, Institut für Angewandte Mathematik und Statistik.

[12] A. Griewank and G.W. Reddien (1988). "Computation of Cusp Singularities for Operator Equations and their Discretizations", *Technical Memorandum* ANL/MCS-TM-115, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL 60439. To appear in the special issue on *Continuation Techniques and Bifurcation Problems* of the *Journal of Computational and Applied Mathematics.*

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

[14] K.E. Hillstrom (1985). "Users Guide for JAKEF", *Technical Memorandum ANL/MCS-TM-16*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL 60439.

[15] M. Iri (1984). "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.

[16] M. Iri, T. Tsuchiya, and M. Hoshi (1985). "Automatic Computation of Partial Derivatives and Rounding Error Estimates with Application to Large-Scale Systems of Nonlinear Equations" (in Japanese), *Journal of the Information Processing Society of Japan*, Vol. 27, No.4, pp.389-396.

[17] M. Iri, and K. Kubota (1987). "Methods of Fast Automatic Differentiation and Applications", *Research memorandum RMI 87-0*, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo.

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

[19] H. Kagiwada, R. Kalaba, N.Rosakhoo, and Karl Spingarn (1986). "Numerical Derivatives and Nonlinear Analysis", Vol. 31 of **Mathematical Concepts and Methods in Science and Engineering** Edt. A.Miele, Plenum Press, New York and London

[20] K.V. Kim, Iu.E. Nesterov, V.A. Skokov, and B.V. Cherkasskii (1984). "An efficient Algorithm for Computing Derivatives and extremal Problems" English translation of "Effektivnyi algoritm vychisleniia proizvodnykh i ekstremal'nye zaduchi", *Ekonomika i matematicheskie metody*, Vol.20, No.2, pp.309-318.

[21] G. Kedem (1980). "Automatic Differentiation of Computer Programs", *ACM TOMS*, Vol.6, No.2, pp.150-165.

[22] U. Kulisch et al (1987). *PASCAL-SC, A PASCAL Extension for Scientific Computation, Information Manual and Floppy Disk*, B.G. Teubner, Stuttgart, and John Wiley & Sons, New York.

[23] G. Leitmann (1981). "The Calculus of Variations and Optimal Control" Vol.20 of **Mathematical Concepts and Methods in Science and Engineering** Edt. A.Miele, Plenum Press, New York and London

[24] D.Y. Peng and D.B. Robinson (1976). "A new two-constant Equation of State", *Ind. Eng. Chem. Fundamentals*, Vol.15, pp.59-64.

[25] L.B. Rall (1981). "Automatic Differentiation - Techniques and Applications", *Springer Lecture Notes in Computer Science*, Vol.120 .

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

[27] L.B. Rall (1987). "Optimal Implementation of Differentiation Arithmetic", in *Computer Arithmetic, Scientific Computation and Programming Languages*, ed. U. Kulisch, Teubner, Stuttgart.

[28] B.Speelpenning (1980). "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.

[29] W.C. Thacker and R.B. Long (1988). "Fitting Dynamics to Data", *Journal of Geophysical Research*, Vol.93, No.C2, pp.1227-1240.

[30] R.E. Wengert (1964). "A simple Automatic Derivative Evaluation Program". *Com. ACM*, Vol. 7,pp.463-464 .

[31] P.Wolfe (1982) ."Checking the Calculation of Gradients", *ACM TOMS*, Vol.6, No.4, pp. 337-343.

[32] B.A. Worley et al (1989). "Deterministic Sensitivity, and Uncertainty Analysis in Large Scale Computer Models", *Proceedings of 10th Annual DOE low level Waste Management Conference* in Denver, Aug.30 -Sept.1, 1988.