PCOMP: A FORTRAN Code for Automatic Differentiation

M. Dobmann, M. Liepelt, K. Schittkowski [1]

Abstract: Automatic differentiation is an interesting and important tool for all numerical algorithms that require derivatives, e.g. in nonlinear programming, optimal control, parameter estimation, differential equations. The basic idea is to avoid not only numerical approximations, which are expensive with respect to CPU time and contain round-off errors, but also *hand-coded* differentiation. The paper introduces the forward and backward accumulation methods and describes the numerical implementation of a computer code with the name PCOMP. The main intention of the approach used is to provide a flexible and portable FORTRAN code for practical applications. The underlying language is described in the form of a formal grammar and is a subset of FORTRAN with a few extensions. Besides a parser that generates an intermediate code and that can be executed independently from the evaluation routines, there are other subroutines for the direct computation of function and gradient values, which can be called directly from a user program. On the other hand it is possible to generate FORTRAN code for function and gradient evaluation that can be compiled and linked separately. [2]

[1]Mathematisches Institut, Universität Bayreuth, D - 95440 Bayreuth

[2]This work was supported by MBB, Unternehmensbereich Flugzeuge, Munich.

1. Introduction

Let $f(x)$ be a nonlinear differentiable function with real values defined for all $x \in \mathsf{R}^n$. By automatic differentiation we understand the numerical computation of a derivative value $\nabla f(x)$ of f at a given point x without truncation errors and without hand-coded formulas.

Hand-coded differentiation is time-consuming and always full of human errors, at least when more complicated functions are involved. To avoid these difficulties, software systems are available to generate derivative formulas symbolically. MACSYMA is probably the best known system, distributed by Symbolics Inc. However the software is quite extensive and the execution is time-consuming. Griewank (1989) reports that the evaluation of the Helmholtz energy function with $n = 30$ by another algebraic manipulation system MAPLE, Char et al. (1988), failed after 15 minutes CPU time on a SUN 3/140 with 16 MB memory due to lack of memory space.

Numerical differentiation requires at least n additional function evaluations for one gradient calculation and induces truncation errors. Although very easy to implement, the numerical errors are often not tolerable, e.g. when the derivatives are used within another numerical approximation scheme. A typical example is the differentiation of solutions of differential equations in an optimal control problem with respect to control variables.

Automatic differentiation overcomes the drawbacks mentioned and is a very useful tool in all practical applications that require derivatives. The resulting code can be used for the evaluation of nonlinear function values by interpreting symbolic function input without extra compilation and linking. Whenever needed, gradients can be evaluated exactly at run time.

Symbolic function input and automatic differentiation is used particularly within interactive nonlinear optimization systems like NLPSOLVER (Idnani (1987)), PADMOS (Kredler e.al. (1990)), SYSTRA (Kelevedzhiev and Kirov (1989)), or EMP (Schittkowski (1987a)). Another typical application is the possibility of including the techniques in mechanical optimal design systems based on FE-techniques like MBB-LAGRANGE, cf. Kneppe (1990). In these cases, the whole system is far too complex to link additional codes to the system whenever user-provided nonlinear functions are to be processed. Whereas the main system functions are built in (e.g. bounds on stresses, displacements, frequencies), it is often desirable to have the additional option of defining arbitrary problem-dependent constraints or objective functions.

There exists meanwhile a large variety of different computer codes for automatic differentiation, cf. Juedes (1991) for a review. They differ in the underlying design strategy, domain of application, mathematical method, implementation and numerical performance. The code PCOMP to be introduced in this paper, is another member of this increasing family of computer implementations. Whereas some general-purpose systems were developed to differentiate more or less arbitrary code given in higher programming languages, e.g. GRADIENT (Kedem (1980)), JAKEF (Hillstrom (1985)) or ADOL-C (Griewank, Juedes, Srinivasan (1991)), PCOMP is a tool with a somewhat restricted language related to FORTRAN, but with emphasis on code flexibility and speed.

PCOMP proceeds from a subset of FORTRAN to define constants and expressions, and possesses additional constructs to define arrays, sum and product expressions and function sets over arbitrary index sets. This allows, for example, the declaration of constraints in certain nodes in a finite element system. Since arbitrary external functions can be linked to PCOMP, it is even possible to use symbolic expressions for externals, e.g. terms like `sigma(i)` for describing the stress in a node `i`. Additional conditional statements `if`, `else` and `endif` may control the execution.

The program PCOMP consists of three FORTRAN modules that can be implemented independently from each other. One module scans and parses the input of data and functions, respectively, and generates an intermediate code. This code can be used either to compute function and gradient values directly in the form of subroutines, or to generate FORTRAN codes for function and gradient evaluation. Thus PCOMP can be used in a very flexible way covering a large variety of possible applications, particularly since all modules are written in standard FORTRAN 77.

Basically there are two ways to implement automatic differentiation, called forward and backward accumulation respectively. Both are used in PCOMP and outlined in this paper briefly. A particular advantage of gradient calculations in reverse accumulation mode is the limitation of relative numerical effort by a constant that is independent of the dimension, i.e. the number of variables.

A more general treatment of automatic differentiation is found in the books of Rall (1981) and Kagiwada et al. (1986). A review of further literature and a more extensive discussion of symbolic and automatic differentiation is given in Griewank (1989). An up-to-date summary of related papers is published in Griewank and Corliss (1991).

In Section 2 of this paper, we describe the basic mathematical ideas, including function evaluation and the forward and reverse accumulation methods for gradient calculation. The subsequent section contains a description of the input format for data and functions that is required to execute PCOMP. All allowed operations of the proposed language are defined. Examples and numerical results of test runs are presented in Section 4. Program organization and use of the FORTRAN subroutines is outlined in Section 5, which is particularly important for those who want to implement PCOMP within their own software environment. Section 6 then shows how user-provided external functions can be linked to PCOMP and called from the underlying program. Some appendices contain a listing of the formal grammar, a list of all error messages, and a FORTRAN code for an example of Section 4 generated by PCOMP.

2. Function and Gradient Evaluation

First we have to investigate the question of how a nonlinear function is evaluated. The idea is to break a given expression into elementary operations that can be evaluated either by internal compiler operations directly or by external function calls. For a given function f the existence of a sequence f_i of elementary functions is assumed, where each individual function $\{f_i\}$ is real-valued and defined on R^{n_i}, $1 \leq n_i \leq m-1$ for $i = n+1, ..., m$. We define now the situation more formally by a pseudo-program:

Definition: Let f be a real-valued function defined on the R^n. Then some real valued functions f_i defined on R^{n_i}, $i = n+1, ..., m$, are called a sequence of elementary functions for f, $m \geq n$, if there exists an index set J_i with $J_i \subset \{1, ..., i-1\}$, $|J_i| = n_i$ for each function f_i, $i = n+1, ..., m$, such that any function value of f for a given vector $x = (x_1, ..., x_n)^T$ can be evaluated according to the following program:

```
For  $i = n+1, ..., m$ let
     $x_i = f_i(x_k, k \in J_i)$
Let  $f(x) = x_m$
```

The proposed way of evaluating function values is implemented in any compiler or interpreter of a higher programming language, if we omit possible code optimization considerations. In computer science terminology, we would say that a postfix expression is built in the form of a stack, which is then evaluated recursively. Thus the elementary functions can be obtained very easily and the corresponding technique is found in any introductory computer science textbook.

Note that for every function $f(x)$ there exists at least one trivial sequence of elementary functions, e.g. by letting $m := n+1$ and $f_{n+1}(x) := f(x)$. For practical use, however, we assume that the functions f_i are basic machine operations, intrinsic or external functions, where the relative evaluation effort is limited by a constant independently of n. Under this condition, suitable bounds for the work ratio can be proved. The algorithm can be implemented efficiently by using stack operations, which reduce the storage as far as possible, i.e. we do not need to store all intermediate variables $x_{n+1}, ..., x_m$.

Of course we assume now that the given function $f(x)$ is differentiable with respect to any $x \in \mathsf{R}^n$ and that a sequence of smooth elementary functions $\{f_i\}$ is given. In this paper, we are only interested in automatic evaluation of gradients and neglect the possible extension of the algorithms to higher order derivatives.

By investigation of the above program for evaluating a function value $f(x)$, we realize immediately that in a very straightforward way the gradient $\nabla f(x)$ can be evaluated simultaneously. If we know how the derivatives of the elementary functions can be obtained, the only thing we have to change is the inclusion of another program line for the gradient update by exploiting the chain rule. In a natural way we denote the resulting approach as *forward accumulation*.

Forward Accumulation Algorithm: Let f be a differentiable function and $\{f_i\}$ be a sequence of elementary functions for evaluating f with corresponding index sets $J_i, i = n+1, ..., m$. Then the gradient $\nabla f(x)$ for a given $x \in \mathsf{R}^n$ is determined by the following program:

```
For  $i = 1, ..., n$ let
     $\nabla x_i = e_i$
For  $i = n+1, ..., m$ let
     $x_i = f_i(x_k, k \in J_i)$,
     $\nabla x_i = \sum_{j \in J_i} \frac{\partial f_i(x_k, k \in J_i)}{\partial x_j} \nabla x_j$
Let  $f(x) = x_m$,
     $\nabla f(x) = \nabla x_m$
```

Here e_i denotes the i-th axis vector in R^n, $i = 1, ..., n$. Again the evaluation of gradients can be performed by suitable stack operations, reducing the required storage.

The complexity of the forward accumulation algorithm is bounded by a constant times n, the number of variables. In other words, the numerical work is the same order of magnitude as for numerical differentiation.

To improve the efficiency of the gradient evaluation, another method can be considered, which is based on the following analytic evaluation. First let

$$y_i := \partial x_m / \partial x_i \ , \ i = 1, ..., m$$

where $\{x_i\}$ is a sequence of original variables x_1, ... ,x_n and of auxiliary variables x_{n+1}, ... ,x_m determined by the elementary function set $\{f_i\}$ and the evaluation procedure of $f(x)$ with respect to a given x. Then we have $y_m = 1$ and

$$\partial f(x) / \partial x_i = y_i \ , \ i = 1, ..., n.$$

By defining inverse index sets

$$K_j := \{i : j \in J_i, 1 \le i \le m\}$$

it can be shown by elementary analysis that

$$y_j = \sum_{i \in K_j} \frac{\partial f_i(x_k, k \in J_i)}{\partial x_j} \, y_i$$

for $j = n+1, ..., m$; see e.g. Kim et al. (1984). Now we are able to compute y_j as soon as we know all $y_i, i = j+1, ..., m$. Also we have to know all intermediate variables $x_i, i = n+1, ..., m$, before starting the loop in reverse form. Therefore we call the resulting method the reverse accumulation algorithm, which can be summarized as follows:

Reverse Accumulation Algorithm: Let f be a differentiable function and $\{f_i\}$ be a sequence of elementary functions for evaluating f with corresponding index sets $J_i, i = n+1, ...m$. Then the gradient $\nabla f(x)$ for a given $x \in \mathsf{R}^n$ is determined by the following program:

```
For  $i = n+1, ..., m$ let
     $x_i = f_i(x_k, k \in J_i)$,
     $y_i = 0$
Let  $f(x) = x_m$,
     $y_i = 0 \; , \; i = 1, ..., n$,
     $y_m = 1$
For  $i = m, m-1, ..., n+1$ let
     $y_j = y_j + \frac{\partial f_i(x_k, k \in J_i)}{\partial x_j} y_i$ for all $j \in J_i$
Let  $f(x) = x_m$,
     $\nabla f(x) = (y_1, ..., y_n)^T$
```

The interesting result is the observation that the relative computational work, i.e. the quotient of the computational effort to evaluate $f(x)$ plus $\nabla f(x)$ and the computational effort to evaluate $f(x)$ alone, is bounded by the maximum work ratio of all elementary functions $f_i(x)$ included in the evaluation list of $f(x)$; see e.g. Griewank (1989) for detailed assumptions needed to prove this statement. If we assume that every elementary function f_i and the corresponding gradient ∇f_i can be evaluated indepently of n, we get the following result:

The evaluation of a gradient by reverse accumulation never requires more than five times the effort of evaluating the underlying function by itself (Griewank (1989)).

The drawback of reverse accumulation, however, is the necessity to store all intermediate data x_i, $i = n+1, ..., m$, which could become unacceptable depending on the storage organization of an actual implementation. Thus PCOMP uses both algorithms. For interpreting the functions and evaluating gradients at run time, the forward accumulation mode is preferred to avoid difficulties with limited memory. Also for the practical applications in mind, calculation time is less important. On the other hand, the alternate possibility to generate FORTRAN codes for function and gradient evaluation uses reverse accumulation.

3. Input Format

The symbolic input of nonlinear functions is only possible if certain syntax rules are satisfied. Here, the allowed language is a subset of FORTRAN with a few extensions. In particular the declaration and executable statements must satisfy the usual FORTRAN input format, i.e. must start at column 7 or later. A statement line is read in until column 72. Comments beginning with `C` at the first column, may be included in a program text wherever needed. Statements may be continued on subsequent lines by including a continuation mark in the 6th column. Either capital or small letters are allowed.

In contrast to FORTRAN, however, most variables are declared implicitly by their assignment statements. Variables and functions must be declared separately only if they are used for automatic differentiation. PCOMP possesses eight special constructs to identify program blocks.

- `* PARAMETER`
  Declaration of constant integer parameters to be used throughout the program, particularly for dimensioning index sets.

- `* SET OF INDICES`
  Definition of index sets that can be used to declare data, variables and functions or to define `sum` or `prod` statements.

- `* REAL CONSTANT`
  Definition of real data, either without index or with one- or two-dimensional index. An index may be a variable or a constant number in an index set. Also arithmetic expressions may be included.

- `* INTEGER CONSTANT`
  Definition of integer data, either without index or with one- or two-dimensional index. An index may be a variable or a constant number in an index set. Also arithmetic integer expressions may be included.

- `* TABLE <identifier>`
  Assignment of constant real numbers to one- or two-dimensional array elements. In subsequent lines, one has to specify one or two indices followed by one real value per line in a free format.

- `* VARIABLE`
  Declaration of variables either with or without index, with respect to which automatic differentiation is to be performed.

* `FUNCTION <identifier>`
  Declaration of functions either with or without index, for which function and gradient values are to be evaluated. The subsequent statements must assign a numerical value to the function identifier.

* `END`
  End of the program.

The order of the above program blocks is obligatory, but they may be repeated whenever desirable. Data must be defined before their usage in a subsequent block. All lines after the final `END` statement are ignored by PCOMP. The statements within the program blocks are very similar to usual FORTRAN notation and must satisfy the following guidelines:

Constant data: For defining real numbers either in analytical expressions or within the special constant data definition block, the usual FORTRAN convention can be used. In particular the `F`-, `E`- or `D`-format is allowed.

Identifier names: Names of identifiers, e.g. variables and functions, index sets and constant data have to follow the FORTRAN syntax rules. The identifier names must begin with a letter and the number of characters must not exceed 6.

Index sets: Index sets are required for the `sum` and `prod` expressions and for defining indexed data, variables and functions. They can be defined in different ways:

1. Range of indices, e.g.
```
ind1 = 1..27
```
2. Set of indices, e.g.
```
ind2 = 3,1,17,27,20
```
3. Computed index sets, e.g.
```
ind3 = 5*i + 100 , i=1..n
```
4. Parameterized index sets, e.g.
```
ind4 = n..m
```

Assignment statements: As in FORTRAN, assignment statements are used to assign a numerical value to an identifier, which may be either the name of the nonlinear function that is to be defined, or of an auxiliary variable that is used in subsequent expressions, e.g.
```
r1 = x1*x4 + x2*x4 + x3*x2 - 11
r2 = x1 + 10*x2 - x3 + x4 + x2*x4*(x3 - x1)
f = r1**2 + r2**2
```

Analytical expressions: An analytical expression is, as in FORTRAN, any allowed combination of constant data, identifiers, elementary or intrinsic arithmetic operations and the special `sum` and `prod` statements. Elementary operations are

```
+ , - , * , / , **
```

and the allowed intrinsic functions are

```
abs, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh,
asinh, acosh, atanh, exp, log, log10, sqrt
```

Alternatively, the corresponding double precision FORTRAN names possessing an initial `d` can be used as well. Brackets are allowed to combine groups of operations. Possible expressions are e.g.

```
5*dexp(-z(i))
```

or

```
log(1 + sqrt(c1*f1)**2)
```

`sum` and `prod` expressions: Sums and products over predetermined index sets are formulated by `sum` and `prod` expressions, where the corresponding index and the index set must be specified, e.g. in the form

```
f = 100*prod(x(i)**a(i), i in inda)
```

In the above example, `x(i)` might be a variable vector defined by an index set, and `a(i)` an array of constant data.

Control statements: To control the execution of a program, the conditional statements

```
if <condition> then
     <statements>
endif
```

or

```
if <condition> then
     <statements>
else
     <statements>
endif
```

can be inserted into a program. Conditions are defined as in FORTRAN by the comparative operators `eq`, `ne`, `le`, `lt`, `ge`, `gt`, which can be combined using brackets and the logical operators `and`, `or` and `not`, e.g.

```
y1 = (x3 - x2)*(x3 - x2)*(x6 - x2)
if ((y1.lt.eps) .and.  (y1.gt.-eps)) then
     y1 = eps
endif
```

Whenever indices are used within arithmetic expressions, it is possible to insert polynomial expressions of indices from a given set. However functions must be treated in a particular way. Since the design goal is to generate short, efficient FORTRAN codes, indexed function names can be used only in exactly the same way as defined. In other words, if a set of functions is declared e.g. by

```
* FUNCTION f(i), i in index
```

then only accesses to `f(i)` are allowed, not to `f(1)` or `f(j)`, for example. In other words, PCOMP does not extend the `sum` and `prod` statements to a sequence of single expressions.

Because of the internal structure of the reverse algorithm, it is not allowed to have any variable names on the left and right hand side of an arithmetic expression at the same time, if the expression depends on any variables. Thus a statement like

```
s = s + 2*x1
```

is forbidden in the reverse mode, i.e. when generating FORTRAN code. To overcome the problem, one could use two statements of the form

```
y = s
s = y + 2*x1
```

However we have to take care of this difficulty only in case of the reverse mode when generating FORTRAN code. If we interprete the symbolic input by forward accummulation, we use the first expression as it stands.

On the other hand it is allowed to pass variable values from one function block to the other. However the user must be aware of a possible failure, if in the calling program the evaluation of a gradient value in the first block is skipped.

One should be very careful when using the conditional statement `if`. Possible traps that prevent a correct differentiation are reported in Fischer (1991), and are to be illustrated by an example. Consider the function $f(x) = x^2$ for $n = 1$. A syntactically correct formulation would be:

```
if (x.eq.1) then
     f = 1
else
     f = x**2
endif
```

In this case PCOMP would try to differentiate both branches of the conditional statement. If x is equal to 1, the derivative value of f is 0; otherwise it is $2x$. Obviously we get a wrong answer for $x = 1$. This is a basic drawback for all automatic differentiation algorithms of the type we are considering.

More examples in the form of complete PCOMP programs are listed in Section 4. The complete formal grammar of the language is found in Appendix A, which should be examined whenever the syntax is not clear from the examples presented. Syntax errors are reported by the parser and are identified by an error number. A list of all possible error messages is presented in Appendix B.

4. Examples and Numerical Experiments

The following examples illustrate typical applications of the PCOMP language. Most of them describe mathematical optimization problems. Since we want to get some information on the performance of the algorithms and techniques used, we also include some test data in the form of absolute and relative calculation times. The following abbreviations are used in the subsequent tables:

SYM	-	symbolic function interpretation, evaluation of gradients by forward accumulation
GEN	-	generated FORTRAN code for function and gradient evaluation by reverse accumulation
TF	-	calculation time for function evaluation
TG	-	calculation time for gradient evaluation
TNG	-	calculation time for numerical gradient approximation by forward differences including one function evaluation
WR	-	work ratio for automatic differentiation
WRN	-	work ratio for numerical differentiation

All tests were performed on a PC 486-DX (33 Mhz) running under MS-DOS, to get more precise measures of calculation time without multi-user or multi-tasking influences. All programs were compiled by the LAHEY V5.01 compiler. The presented calculation times are mean values of successive runs and are measured in hundredths of a second. The number of identical test runs needed to get measurable execution times is adapted to the problem size. A work ratio is the quotient of function plus gradient evaluation and function evaluation; more precisely

$$\begin{aligned} WR &= TG/TF \\ WRN &= TNG/TF \end{aligned}$$

Note here that the automatic evaluation of a gradient implies also a calculation of the corresponding function value, and that the numerical approximation of a gradient requires $n+1$ function calls.

Example 1:

- Problem TP32 of Hock and Schittkowski (1981):

$$\begin{aligned} f(x) &= (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2 \\ g_1(x) &= 6x_2 + 4x_3 - x_1^3 - 3 \\ g_2(x) &= 1 - x_1 - x_2 - x_3 \end{aligned}$$

  The optimization problem consists of minimizing $f(x)$ subject to the constraints $g_1(x) \geq 0$ and $g_2(x) \geq 0$, and we may imagine that an algorithm is to be applied that requires gradients of all problem functions.

- Variables:

$$(x_1, x_2, x_3) = (0.1, 0.7, 0.2)$$

- PCOMP program:

```
c     TP32

*     VARIABLE
      x1, x2, x3

*     FUNCTION g1
      g1 = 1.0 - x1 - x2 - x3

*     FUNCTION g2
      g2 = 6.0*x2 + 4.0*x3 - x1**3 - 3.0

*     FUNCTION f
      f = (x1 + 3.0*x2 + x3)**2 + 4.0*(x1 - x2)**2

*     END
```

- Results:

$n = 3$	TF	TG	TNG	WR	WRN
SYM	0.0403	0.0725	0.1645	1.80	4.08
GEN	0.0026	0.0091	0.0139	3.51	5.35

- Conclusion: Obviously the compiled FORTRAN code requires much less calculation time. Although the initial program is precompiled, the resulting intermediate code must still be interpreted, e.g. by performing look-ups in operator or symbol tables.

Example 2:

- Exponential data fitting:

$$\begin{aligned} h(x,t) &= x_1 x_2 x_3 \quad (\frac{x_4 - x_2}{z_1} e^{-x_2(t-\tau)} + \frac{x_4 - x_3}{z_2} e^{-x_3(t-\tau)} \\ &\quad + \frac{x_4 - x_5}{z_3} e^{-x_5(t-\tau)} + \frac{x_4 - x_6}{z_4} e^{-x_6(t-\tau)}) \end{aligned}$$

where

$$\begin{aligned} z_1 &= (x_3 - x_2)(x_5 - x_2)(x_6 - x_2) \\ z_2 &= (x_2 - x_3)(x_5 - x_3)(x_6 - x_3) \\ z_3 &= (x_2 - x_5)(x_3 - x_5)(x_6 - x_5) \\ z_4 &= (x_2 - x_6)(x_3 - x_6)(x_5 - x_6) \end{aligned}$$

To avoid division by zero, we replace any of the z_i-s by a small value, as soon as the z_i-value is below that tolerance.

The model function is quite typical for a broad class of practical application problems, which are denoted as nonlinear data fitting, parameter estimation or least squares problems. Given a set of experimental data $\{t_i\}$ and $\{y_i\}$, $i = 1, ..., m$, one has to determine parameters x_1,..., x_n, so that the distance of the model function and the experimental data is minimized in the L_2-norm. More precisely we want to minimize the expression

$$\sum_{i=1}^{m} (h(x, t_i) - y_i)^2$$

over all $x \in \mathsf{R}^n$.

In the subsequent PCOMP program, we have $n = 7$ and $m = 14$, where the data sets $\{t_i\}$ and $\{y_i\}$ are given in the form of constants. The last variable plays the role of a lag time and is called τ. Since typical nonlinear least squares codes require the calculation of m individual function values instead of the sum of squares, the PCOMP code is designed to compute m function values of the form $f_i(x) = h(x, t_i) - y_i$, $i = 1, ..., m$.

- Variables:

$$(x_1, ..., x_6, \tau) = (1.0, 3.4148, 1.33561, 0.3411, 1.0278, 0.05123, 0.2)$$

- PCOMP program:

```
c      Exponential parameter estimation

*      PARAMETER
       m = 14

*      SET OF INDICES
       indobs = 1..m
```

```
*     REAL CONSTANT
      eps = 1.D-12
      t(i) = 0.1*i, i in indobs
      t(1) = 0.0

*     TABLE y(i), i in indobs
      1    0.238
      2    0.578
      3    0.612
      4    0.650
      5    0.661
      6    0.658
      7    0.652
      8    0.649
      9    0.647
      10   0.645
      11   0.644
      12   0.644
      13   0.643
      14   0.644

*     VARIABLE
      x1, x2, x3, x4, x5 , x6, tau

*     FUNCTION f(i), i in indobs
      x42 = x4 - x2
      x32 = x3 - x2
      x52 = x5 - x2
      x62 = x6 - x2
      x43 = x4 - x3
      x53 = x5 - x3
      x63 = x6 - x3
      x45 = x4 - x5
      x65 = x6 - x5
      x46 = x4 - x6

      z1 = x32*x52*x62
      if (abs(z1).lt.eps) then
         z1 = eps
      endif
      z2 = -x32*x53*x63
      if (abs(z2).lt.eps) then
         z2 = eps
      endif
      z3 = x52*x53*x65
      if (abs(z3).lt.eps) then
         z3 = eps
      endif
      z4 = -x62*x63*x65
      if (abs(z4).lt.eps) then
         z4 = eps
      endif

      f(i) = x1*x2*x3*
     /       (x42/z1*dexp(-x2*(t(i)-tau))
     /      + x43/z2*dexp(-x3*(t(i)-tau))
     /      + x45/z3*dexp(-x5*(t(i)-tau))
     /      + x46/z4*dexp(-x6*(t(i)-tau))) - y(i)

*     END
```

- Results:

$n = 7$	TF	TG	TNG	WR	WRN
SYM	1.63	5.22	12.94	3.20	7.93
GEN	0.16	1.01	1.35	6.13	8.20

- Conclusion: The differences in calculation times between the compiled FORTRAN code and the interpreted one are as significant as those of Example 1. The work ratio for the generated code is larger than 3, since the source code contains additional assignment statements to pass intermediate values.

Example 3:

- Problem TP295 of Schittkowski (1987b):

$$f(x) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

The optimization problem consists of minimizing $f(x)$ with different dimensions n. It is a generalization of the well-known *banana function* of Rosenbrock (1969) and used here to test the effect of varying dimensions.

- Variables:

$$(x_1, ..., x_n) = (-1.2, 1.0, -1.2, 1.0, ...)$$

- PCOMP program:

```
c     TP295 (n=10)

*     SET OF INDICES
      indn = 1..10
      indnm1 = 1..9

*     VARIABLE
      x(i), i in indn

*     FUNCTION f
      f = sum(100*(x(i+1) - x(i)**2)**2 + (1 - x(i))**2, i in indnm1)

*     END
```

- Results:

	Symbolic Interpretation					Generated and Compiled Code				
n	TF	TG	TNG	WR	WRN	TF	TG	TNG	WR	WRN
10	0.11	0.38	1.18	3.49	10.83	0.01	0.04	0.08	5.47	12.13
20	0.21	1.30	4.45	6.14	21.00	0.01	0.08	0.30	5.80	22.08
30	0.31	2.76	9.74	8.78	31.00	0.02	0.12	0.66	5.91	32.21
40	0.42	4.76	17.21	11.34	41.00	0.03	0.16	1.16	6.01	42.51
50	0.52	7.31	26.78	13.96	51.15	0.03	0.21	1.80	6.06	52.60
60	0.63	10.38	38.34	16.49	60.92	0.04	0.25	2.57	6.16	62.62
70	0.73	14.12	51.92	19.29	70.93	0.05	0.30	3.49	6.19	72.73
80	0.84	18.34	67.62	21.86	80.59	0.06	0.34	4.56	6.17	82.78
90	0.96	23.27	87.73	24.13	91.00	0.06	0.38	5.85	6.08	92.74
100	1.07	28.30	107.37	26.56	100.76	0.07	0.44	7.15	6.33	103.02

- Conclusion: The numerical test results verify the theoretical conclusions of Section 2. The reverse accumulation algorithm has a bounded work ratio, where the work ratios for forward accumulation and numerical approximation are proportional to n. The results are illustrated in Figure 1. Numerical differentiation of generated code is still somewhat faster than forward accumulation of interpreted code.

Example 4:

- Helmholtz energy function (Griewank (1989)):

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

  The above function was used by Griewank (1989) to test and illustrate symbolic versus automatic differentiation on the one hand and the reverse and forward accumulation algorithms on the other. In our tests we let A be the Hilbert-matrix, i.e. $a_{i,j} = \frac{1}{i+j-1}$, $i, j = 1, ..., n$, where $a_{i,j}$ denotes an element of A, and we set $b_i = 0.00001$ for $i = 1, ..., n$, where b_i is the i-th element of the vector b.

- Variables:

$$(x_1, ..., x_n) = (2.0, 2.0, ...)$$

- PCOMP program:

```
c     Helmholtz energy function (n=10)

*     SET OF INDICES
      index = 1..10

*     REAL CONSTANT
      r = 8.314
      t = 273.0
      c1 = 1.0 + dsqrt(2.0)
      c2 = 1.0 - dsqrt(2.0)
      c3 = dsqrt(8.0)
      a(i,j)=1/(i+j-1), i in index, j in index
      b(i)=0.00001, i in index

*     VARIABLE
      x(I), i in index

*     FUNCTION f
      bx = sum(b(i)*x(i), i in index)
      xax = sum(x(i)*sum(a(i,j)*x(j), j in index), i in index)
      f = r*t*sum(x(i)*dlog(x(i)/(1 - bx)), i in index)
     /  - xax*dlog((1 + c1*bx)/(1 + c2*bx))/(c3*bx)

*     END
```

- Results:

	Symbolic Interpretation					Generated and Compiled Code				
n	TF	TG	TNG	WR	WRN	TF	TG	TNG	WR	WRN
5	0.21	0.47	1.26	2.24	5.97	0.02	0.07	0.15	3.08	6.19
10	0.60	1.96	6.62	3.26	11.00	0.06	0.21	0.67	3.53	11.20
20	2.01	10.72	42.32	5.32	21.01	0.19	0.67	3.95	3.57	21.13
30	4.28	31.20	132.26	7.29	30.93	0.39	1.42	12.26	3.60	31.09
40	7.48	69.50	306.53	9.29	40.97	0.64	2.42	26.12	3.81	41.09
50	11.71	130.70	597.21	11.16	51.00	1.05	3.68	53.94	3.49	51.16
60	17.10	219.70	1042.60	12.85	60.97	1.53	5.21	93.64	3.41	61.25
70	22.30	342.80	1582.10	15.37	70.95	2.03	7.70	146.93	3.78	72.25
80	29.25	505.00	2363.25	17.27	80.78	2.67	11.00	216.87	4.12	81.19

- Conclusion: The results repeat the observations made for Example 3; see also Figure 2. Obviously the reverse accumulation algorithm has a bounded work ratio, where the work ratios for forward accumulation and numerical approximation are proportional to n. The conclusions are very similar to made by Griewank (1989), if we neglect differences based on other hardware configuration and, in particular, different data organizations.

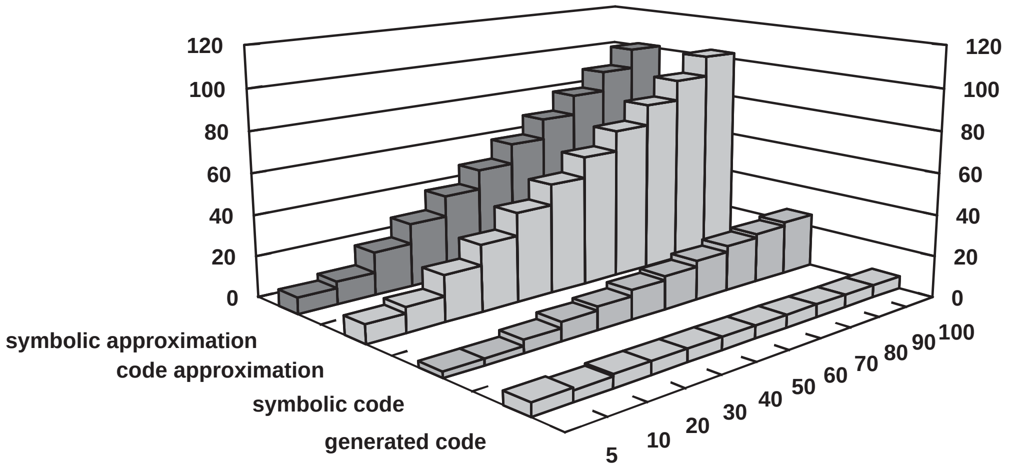

Figure 1: Work ratios for test function TP295

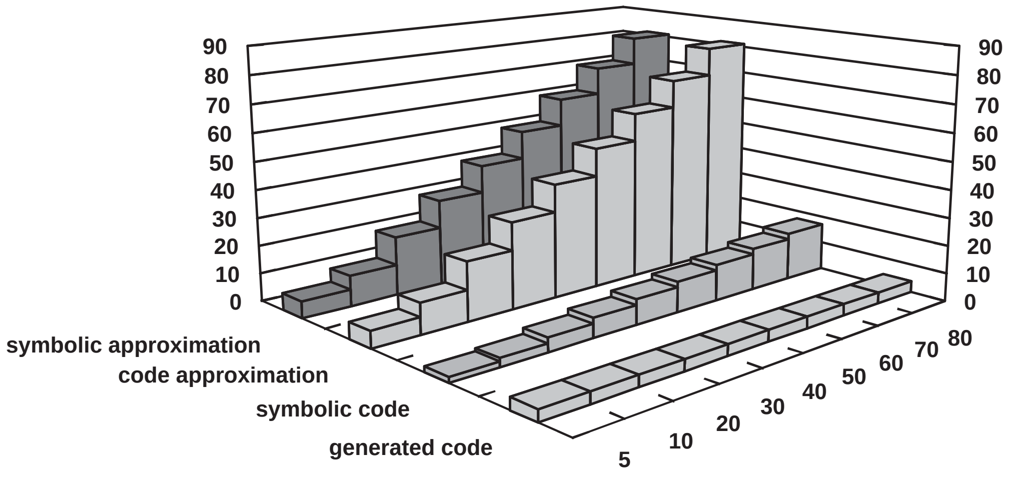

Figure 2: Work ratios for Helmholtz equation

5. Program Organization

The PCOMP system consists of three modules that can be executed completely independently from each other. There are also some auxiliary routines, in particular an error routine called SYMERR to make error messages readable, and a routine with the name SYMPRP to read intermediate code generated by the parser. All routines are implemented in FORTRAN77 and tested on the following systems: VAX/VMS, HP-UX, MS-DOS (WATFOR, MS-FORTRAN 5.0, LAHEY 5.01).

(1) **Parser:**
The source code is analysed and compiled into an intermediate code, which can then be processed by the other routines. The subroutine to be executed has the name SYMINP. The syntax of the code is described in the form of a formal grammar; see Appendix A. The parser was generated in C by the *yacc*-compiler-compiler of UNIX and then transformed into FORTRAN by hand. The following files are needed to link the parser:

PCOMP_P1.FOR	-	parser routines
PCOMP_P2.FOR	-	parser routines
PCOMP_EV.FOR	-	numerical evaluation of analytical expressions used
	-	in index or constant declarations
PCOMP_EX.FOR	-	external functions provided by the user
PCOMP_ER.FOR	-	error messages

To give an example, we list a possible implementation that was used to generate intermediate code for our numerical tests.

```
      parameter (lrsym=15000, lisym=15000)
      double precision rsym(lrsym)
      integer isym(lisym), larsym, laisym, ierr, lrow
      open(2,file='pcomp.fun', status='UNKNOWN')
      open(3,file='pcomp.sym', status='UNKNOWN')
      call SYMINP(2,3,rsym,lrsym,isym,lisym,larsym,laisym,ierr,lrow)
      if (ierr.gt.0) goto 900
      goto 9999
  900 call SYMERR(ierr,lrow)
 9999 continue
      close(2)
      close(3)
      stop
      end
```

(2) **Function and Gradient Evaluation:**
Proceeding from an intermediate code generated by SYMINP, function and gradient values are evaluated in the form of subroutines called SYMFUN and SYMGRA. They can be linked to any user program as required by the underlying application. Gradients are computed by forward accumulation despite the drawbacks outlined in the previous sections, to reduce the size of internal working arrays. The following program files are available and must be linked to the code provided by the user:

PCOMP_S.FOR	-	function and gradient evaluation
PCOMP_EV.FOR	-	evaluation of expressions from given postfix notation
PCOMP_EX.FOR	-	external functions provided by the user
PCOMP_ER.FOR	-	error messages

In the next example, we illustrate a possible implementation of the routines for evaluating function and gradient values. We assume that the symbol file *pcomp.sym* contains the intermediate code of one function with two variables.

```
      implicit double precision(a-h,o-z)
      parameter (nmax=100, mmax=50, lrsym=30000, lisym=10000)
      dimension x(nmax), f(mmax), df(mmax,nmax), rsym(lrsym),
     /          isym(lisym)
      logical act(mmax)
      open(3,file='pcomp.sym',status='UNKNOWN')
      n=2
      m=1
      x(1)=1.0
      x(2)=-1.2
      act(1)=.true.
      call SYMPRP(3,rsym,lrsym,isym,lisym,larsym,laisym,ierr)
      if (ierr.gt.0) goto 900
      call SYMFUN(x,n,f,m,act,rsym,lrsym,isym,lisym,ierr)
      if (ierr.gt.0) goto 900
      call SYMGRA(x,n,f,m,df,mmax,act,rsym,lrsym,isym,lisym,ierr)
      if (ierr.gt.0) goto 900
      write(*,*) f(1),df(1,1),df(1,2)
      goto 9999
  900 call SYMERR(ierr,0)
 9999 continue
      close(3)
      stop
      end
```

(3) **Generation of FORTRAN Code:**
Proceeding from an intermediate code generated by SYMINP, FORTRAN subroutines for function and gradient evaluation are generated. They can be compiled and linked separately from the PCOMP system. Gradients are computed by reverse accumulation. There is only one file to be linked to the user code, and the error routine as before:

PCOMP_G.FOR	-	generate FORTRAN code
PCOMP_ER.FOR	-	error messages

```
      parameter (lrsym=15000, lisym=15000)
      double precision rsym(lrsym)
      integer isym(lisym), larsym, laisym, ierr, lrow
      open(3,file='pcomp.sym', status='UNKNOWN')
      open(4,file='pcomp.for', status='UNKNOWN')
      call SYMPRP(3,rsym,lrsym,isym,lisym,larsym,laisym,ierr)
      if (ierr.gt.0) goto 900
      call SYMFOR(4,rsym,lrsym,isym,lisym,ierr)
      if (ierr.gt.0) goto 900
      goto 9999
  900 call SYMERR(ierr,0)
 9999 continue
      close(3)
      close(4)
      stop
      end
```

Documentation of all subroutines within the files mentioned is given by Dobmann (1993) and Liepelt (1990) together with some additional information on the data structures. In the remainder of this section, we describe only the use of subroutines that can be called from a user program.

Subroutine SYMINP:

- **Purpose:**
  The subroutine compiles symbolically defined nonlinear functions and generates an intermediate code.

- **Calling sequence:**
  SYMINP (INPUT,SYMFIL,WA,LWA,IWA,LIWA,UWA,UIWA,IERR,LNUM)

- **Parameters:**

INPUT	-	When calling SYMINP, the integer value of INPUT is the number of the file that contains the program text.
SYMFIL	-	An integer identifying the output file number to which the intermediate code is to be written.
WA(LWA)	-	Double precision working array of length LWA used internally to store and process data. When leaving SYMINP, WA contains the generated intermediate code in its first UWA positions.
LWA	-	Length of the working array WA. LWA must be sufficiently large depending on the code size.
IWA(LIWA)	-	Integer working array of length LIWA. On return, IWA contains the integer part of the intermediate code in its first UIWA positions.
LIWA	-	Length of the working array IWA. LIWA must be sufficiently large depending on the code size.
UWA,UIWA	-	Storage actually needed for the intermediate code in the form of integers.
IERR	-	On return, IERR shows the termination reason of SYMINP: IERR $= 0$: Successful termination. IERR > 0 : There is a syntax error in the input file. Call SYMERR for more information.
LNUM	-	In case of unsuccessful termination, LNUM contains the line number where the error was detected.

Subroutine SYMERR:

- **Purpose:**
  Proceeding from an error code IERR (> 0) and, if available from a SYMINP call, a line number, SYMERR generates an output message on the standard device.
- **Calling sequence:**
  SYMERR (LNUM,IERR)
- **Parameters:**

LNUM	-	When calling SYMERR after a SYMINP execution, LNUM has to contain the corresponding line number value as determined by SYMINP.
IERR	-	The numerical value of the termination reason is to be inserted when calling SYMERR.

Subroutine SYMPRP:

- **Purpose:**
  The subroutine reads intermediate code from a file generated by a SYMINP call and fills two working arrays with the code for further processing within subroutines SYMFUN, SYMGRA and SYMFOR.
- **Calling sequence:**
  SYMPRP (SYMFIL,WA,LWA,IWA,LIWA,UWA,UIWA,IERR)
- **Parameters:**

SYMFIL	-	An integer identifying the input file number, which contains the intermediate code generated by SYMINP.
WA(LWA)	-	Double precision working array of length LWA that contains the intermediate code in its first UWA positions when leaving SYMPRP.
LWA	-	Length of the working array WA. LWA must be at least UWA as determined by SYMINP.
IWA(LIWA)	-	Integer working array of length LIWA. On return, IWA contains the integer part of the intermediate code in its first UIWA positions.
LIWA	-	Length of the working array IWA. LIWA must be at least UIWA as determined by SYMINP.
UWA,UIWA	-	Storage actually needed for the intermediate code in WA and IWA.
IERR	-	On return, IERR shows the termination reason of SYMPRP: IERR = 0 : Successful termination. IERR > 0 : There is an error in the input file. Call SYMERR for more information.

Subroutine SYMFUN:

- **Purpose:**
  The intermediate code is passed from a SYMINP call to SYMFUN in form of a real and an integer working array. Given any variable vector x, the subroutine computes the corresponding function values $f_i(x)$. The functions that are to be evaluated by SYMFUN must be specified by a logical array.

- **Calling sequence:**
  SYMFUN (X,N,F,M,ACTIVE,WA,LWA,IWA,LIWA,IERR)

- **Parameters:**

X(N)	-	Double precision array of length N that contains the variable values for which functions are to be evaluated.
N	-	Dimension, i.e. number of variables.
F(M)	-	Double precision array of length M to pass the function values computed by SYMFUN, to the user program.
M	-	Total number of functions on the input file.
ACTIVE(M)	-	Logical array of length M that determines the functions to be evaluated. ACTIVE must be set by the user when calling SYMFUN: ACTIVE(J) = .TRUE. : Compute function value $g_j(x)$. ACTIVE(J) = .FALSE. : Do not compute function value $g_j(x)$.
WA(LWA)	-	Double precision working array of length LWA that contains the intermediate code in its first UWA positions.
LWA	-	Length of the working array WA. LWA must be at least UWA as determined by SYMINP.
IWA(LIWA)	-	Integer working array of length LIWA. IWA contains the integer part of the intermediate code in its first UIWA positions.
LIWA	-	Length of the working array IWA. LIWA must be at least UIWA as determined by SYMINP.
IERR	-	On return, IERR shows the termination reason of SYMFUN: IERR = 0 : Successful termination. IERR > 0 : There is an error in the input file. Call SYMERR for more information.

Subroutine SYMGRA:

- **Purpose:**
  The intermediate code is passed from a SYMINP call to SYMGRA in the form

of a real and an integer working array. Given a variable vector x, the subroutine computes the corresponding function and gradient values $f_i(x)$ and $\nabla f_i(x)$. The functions and gradients that are to be evaluated by SYMGRA must be specified by a logical array.

- **Calling sequence:**
  SYMGRA (X,N,F,M,DF,MMAX,ACTIVE,WA,LWA,IWA,LIWA,IERR)

- **Parameters:**

X(N)	-	Double precision array of length N that contains the variable values for which functions and gradients are to be evaluated.
N	-	Dimension, i.e. number of variables.
F(M)	-	Double precision array of length M to pass the function values computed by SYMGRA to the user program.
M	-	Total number of functions on the input file.
DF(MMAX,N)	-	Two-dimensional double precision array to take over the gradients computed by SYMGRA. The row dimension must be MMAX in the driving routine.
MMAX	-	Row dimension of DF. MMAX must not be smaller than M.
ACTIVE(M)	-	Logical array of length M that determines the functions and gradients to be evaluated. ACTIVE must be set by the user when calling SYMGRA: ACTIVE(J) = .TRUE. : Evaluate function $g_j(x)$. ACTIVE(J) = .FALSE. : Do not evaluate function $g_j(x)$.
WA(LWA)	-	Double precision working array of length LWA that contains the intermediate code in its first UWA positions.
LWA	-	Length of the working array WA. LWA must be at least UWA as determined by SYMINP.
IWA(LIWA)	-	Integer working array of length LIWA. IWA contains the integer part of the intermediate code in its first UIWA positions.
LIWA	-	Length of the working array IWA. LIWA must be at least UIWA as determined by SYMINP.
IERR	-	On return, IERR shows the termination reason of SYMGRA: IERR = 0 : Successful termination. IERR > 0 : There is an error in the input file. Call SYMERR for more information.

Subroutine SYMFOR:

- **Purpose:**
  The intermediate code is passed from a SYMINP call to SYMFOR in the form of a real and an integer working array. Then SYMFOR generates two subroutines for function and gradient evaluation on a given output file. The calling sequences of the generated subroutines are

  XFUN (X,N,F,M,ACTIVE,IERR)

  and

  XGRA (X,N,F,M,DF,MMAX,ACTIVE,IERR),

  where the meaning of the parameters is the same as for SYMFUN and SYMGRA, respectively.

- **Calling sequence:**
  SYMFOR (XFIL,WA,LWA,IWA,LIWA,IERR)

- **Parameters:**

XFIL	-	An integer containing the number of the file on which the codes are to be written.
WA(LWA)	-	Double precision working array of length LWA that contains the intermediate code in its first UWA positions when calling SYMFOR. Additional storage is required by SYMFOR.
LWA	-	Length of the working array WA.
IWA(LIWA)	-	Integer working array of length LIWA. IWA contains the integer part of the intermediate code in its first UIWA positions when calling SYMFOR, and is needed for additional working space.
LIWA	-	Length of the working array IWA.
IERR	-	On return, IERR shows the termination reason of SYMFOR: IERR $= 0$: Successful termination. IERR > 0 : There is an error in the input file. Call SYMERR for more information.

6. Inclusion of External Functions

For the practical use of PCOMP, it is extremely important to have the possibility of defining an interface between PCOMP and the user system. In the frame of a mechanical structural optimization system, for example, one might wish to include expressions of the form `sigma(i)` in the PCOMP program to identify the stress at a node *i*. Other examples are the evaluation of inner products or the input of data from a file or a user program.

To include external functions in PCOMP, the following alterations are required:

- Change the number of external functions MAXEXT in subroutines YYPAR, EVAL, REVCDE, FORCDE, FORDF and KEYWD.
- Insert the function names used in the source code, in the array EXTNAM. The subroutine to be altered is KEYWD.
- Define the number of additional integer parameters of the functions to be defined, in the array EXTTYP. The array is found in subroutines YYPAR, EVAL, REVCDE, FORCDE and FORDF.
- Implement subroutines that evaluate function and gradient values and insert their calling sequences in subroutines EXTFUN and EXTGRA.

A user can change the module system provided by the authors. The interface functions EXTFUN and EXTGRA are executed within PCOMP in the following way:

Subroutine EXTFUN:

- **Purpose:**
  Calling user-provided subroutines to evaluate function values that correspond to symbolic names in a source program.
- **Calling sequence:**
  EXTFUN (EXT,X,N,F,EXTPAR)
- **Parameters:**

EXT	-	Integer value to identify the EXT-th external function value to be computed. The order coincides with the order of the symbolic names in the array EXTNAM.

X(N)	-	Double precision array of length N that contains the variable values for which an external function is to be evaluated.
N	-	Dimension, i.e. number of variables.
F	-	Double precision variable to take over the value of the function on return.
EXTPAR(2)	-	Integer array of length two containing up to two actual parameters when calling EXTFUN.

Subroutine EXTGRA:

- **Purpose:**
  Calling user-provided subroutines to evaluate gradient values that correspond to symbolically defined functions in a source program.

- **Calling sequence:**
  EXTGRA (EXT,X,N,DF,EXTPAR)

- **Parameters:**

EXT	-	Integer value to identify the EXT-th external function for which the gradient is to be computed. The order coincides with the order of the symbolic names in the array EXTNAM.
X(N)	-	Double precision array of length N that contains the variable values for which an external gradient is to be evaluated.
N	-	Dimension, i.e. number of variables.
DF(N)	-	Double precision array of length N to take over the gradient value of function EXT on return.
EXTPAR(2)	-	Integer array of length two containing up to two actual parameters when calling EXTFUN.

Example 5:

- Helmholtz energy function (Griewank (1989)):

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

We consider again the Helmholtz energy function that was also programmed in the PCOMP language in Section 4, Example 4. By investigating that code in detail, we observe immediately that some operations could be performed much faster 'in core', in particular inner products. Moreover certain intermediate data could be passed to the gradient evaluation, if we assume that a function evaluation always preceeds a gradient evaluation. The following PCOMP program contains six external functions, where there is still an inner product in the PCOMP code that could be eliminated as well. It is left for demonstration purposes.

- Variables:

$$(x_1, ..., x_n) = (2.0, 2.0, ...)$$

- PCOMP program:

```
c     Helmholtz function with externals (n=10)

*     PARAMETER
      n = 10

*     SET OF INDICES
      index = 1..n

*     REAL CONSTANT
      r = 8.314
      t = 273.0
      c1 = 1.0 + dsqrt(2.0)
      c2 = 1.0 - dsqrt(2.0)
      c3 = dsqrt(8.0)

*     VARIABLE
      x(i), i in index

*     FUNCTION f
      xax = sum(x(i)*ax(i), i in index)
      f = r*t*(xlogx - dlog(1 - bx)*x1) -
     /    xax*dlog((1 + c1*bx)/(1 + c2*bx))/(c3*bx)

*     END
```

- External subroutines:

```
      subroutine EXTFUN (ext,x,n,f,extpar)
      integer ext, n, extpar(2)
      double precision x(n), f
      goto (1,2,3,4) ext
    1 call AX(x, n, f, extpar(1))
      return
```

```
2 call BX(x, n, f)
  return
3 call XLOGX(x, n, f)
  return
4 call X1(x, n, f)
  return
  end

  subroutine EXTGRA (ext, x, n, df, extpar)
  integer ext, n, extpar(2)
  double precision x(n), df(n)
  goto (1,2,3,4) ext
1 call DAX(x, n, df, extpar(1))
  return
2 call DBX(x ,n ,df)
  return
3 call DXLOGX(x ,n ,df)
  return
4 call DX1(x, n, df)
  return
  end

  subroutine AX (x, n, f, i)
  double precision x(n), f, a, b
  f=0.0
  do 1 j=1,n
1 f=f + 1.0/dble(i+j-1)*x(j)
  return
  end

  subroutine DAX (x, n, df, i)
  double precision x(n), df(n), a, b
  do 1 j=1,n
1 df(j)=1.0/dble(i+j-1)
  return
  end

  subroutine BX (x, n, f)
  double precision x(n), f, a, b
  f=0.0
  do 1 j=1,n
1 f=f + 0.00001*x(j)
  return
  end
```

```
        subroutine DBX (x, n, df)
        double precision x(n), df(n), a, b
        do 1 j=1,n
      1 df(j)=0.00001
        return
        end

        subroutine XLOGX (x, n, f)
        double precision x(n), f, logx
        common /extlog/ logx(100)
        f=0.0
        do 1 j=1,n
        logx(j)=dlog(x(j))
      1 f=f + x(j)*logx(j)
        return
        end

        subroutine DXLOGX (x, n, df)
        double precision x(n), df(n), logx
        common /extlog/ logx(100)
        do 1 j=1,n
      1 df(j)=logx(j) + 1.0
        return
        end

        subroutine X1 (x, n, f)
        double precision x(n), f
        f=0.0
        do 1 j=1,n
      1 f=f + x(j)
        return
        end

        subroutine DX1 (x, n, df)
        double precision x(n), df(n)
        do 1 j=1,n
      1 df(j)=1.0
        return
        end
```

- Results:

	Symbolic Interpretation					Generated and Compiled Code				
n	TF	TG	TNG	WR	WRN	TF	TG	TNG	WR	WRN
5	0.07	0.16	0.48	2.13	6.36	0.03	0.08	0.19	2.52	6.30
10	0.13	0.38	1.46	2.88	11.10	0.07	0.17	0.79	2.37	11.26
20	0.31	0.90	6.46	2.96	21.12	0.21	0.48	4.39	2.32	21.19
30	0.55	1.76	17.22	3.17	31.08	0.42	0.96	13.12	2.27	31.14
40	0.88	2.91	36.14	3.30	40.93	0.71	1.60	29.29	2.25	41.12
50	1.29	4.32	65.55	3.36	50.89	1.08	2.43	55.31	2.25	51.10
60	1.76	5.95	107.40	3.38	60.94	1.53	3.44	93.61	2.25	61.11
70	2.32	7.95	164.50	3.43	70.88	2.05	4.61	146.34	2.25	71.21
80	2.95	10.27	239.12	3.47	80.93	2.65	5.93	215.65	2.23	81.23
90	3.70	13.20	334.30	3.57	90.35	3.33	7.24	303.21	2.17	91.06
100	4.48	15.40	451.68	3.44	100.78	4.08	9.00	412.52	2.20	101.06

- Conclusion: The results show the possible benefits very clearly. By extracting general-purpose routines in the form of externals, the work ratios for automatic differentiation increase much slower than those for numerical differentiation. Also the absolute time differences between interpreted and generated code decrease.

Acknowledgements

The authors would like to express their gratitude to the referees of the paper. They made substantial proposals for improvements of the PCOMP code, and tested the code very thoroughly.

APPENDIX A: Formal Grammar

The syntax of PCOMP is based on a formal language that is listed below. The input format was chosen according to the requirements of the *yacc*-compiler-compiler of UNIX. The C code generated by *yacc* was translated into FORTRAN by hand.

```
%{
#include <ctype.h>
#include <stdio.h>
%}

%token RANGE, RELOP, AND, OR, NOT, INUM, RNUM, ID, SUM, PROD, IN
%token IF, THEN, ELSE, ENDIF, STANDARD, EXTERN
%token PARAM, INDEX, REAL, INT, TABLE, VAR, FUNC, END, GOTO, LABEL
%token CONTINUE
%left OR
%left AND
%left NOT
%nonassoc RELOP
%left '+' '-'
%left '*' '/'
%left UMINUS
%right '^'
%%
module : declaration_blocks end_module {};
declaration_blocks : declaration_blocks declaration_block
                   |  ;
declaration_block : param_head param_declarations
                  | index_head index_declarations
                  | real_head real_declarations
                  | integer_head integer_declarations
                  | table_head table_declarations
                  | variable_head variable_declarations
                  | function_head stmts {};
param_head : PARAM '\n';
param_declarations : param_declarations param_declaration
                   |  ;
param_declaration : ID '=' INUM '\n' {};
index_head : INDEX '\n';
index_declarations : index_declarations index_declaration
                   |  ;
index_declaration : ID '=' index_delimiter RANGE index_delimiter '\n' {}
```

```
                  | ID '=' INUM ',' INUM {} opt_inum '\n'
                  | ID '=' ind_expr ',' ID '=' index_delimiter
                    RANGE index_delimiter '\n' {}
index_delimiter : ID {}
                | INUM
opt_inum : opt_inum ',' INUM {}
         |  ;
ind_expr : ind_expr '+' ind_expr {}
         | ind_expr '-' ind_expr {}
         | ind_expr '*' ind_expr {}
         | ind_expr '/' ind_expr {}
         | '(' ind_expr ')'
         | '-' ind_expr %prec UMINUS {}
         | INUM {}
         | ID {};
real_head : REAL '\n';
real_declarations : real_declarations real_declaration
                  |  ;
real_declaration : ID '=' expr '\n' {}
                 | ID '(' ID ')' '=' expr ',' ID IN ID '\n' {}
                 | ID '(' INUM ')' '=' expr '\n' {}
                 | ID '(' ID ',' ID ')' '=' expr ','
                   ID IN ID ',' ID IN ID '\n' {}
                 | ID '(' INUM ',' INUM ')' '=' expr '\n' {};
integer_head : INT '\n';
integer_declarations : integer_declarations integer_declaration
                     |  ;
integer_declaration : ID '=' expr '\n' {}
                    | ID '(' ID ')' '=' expr ',' ID IN ID '\n' {}
                    | ID '(' INUM ')' '=' expr '\n' {}
                    | ID '(' ID ',' ID ')' '=' expr ','
                      ID IN ID ',' ID IN ID '\n' {};
                    | ID '(' INUM ',' INUM ')' '=' expr '\n' {};
table_head : TABLE ID '(' ID ')' ',' ID IN ID '\n' {}
           | TABLE ID '(' ID ',' ID ')' ',' ID IN ID ',' ID IN ID '\n' {};
table_declarations : table_declarations table_declaration
                   |  ;
table_declaration : INUM RNUM '\n' {}
                  | INUM '-' RNUM '\n' {}
                  | INUM INUM RNUM '\n' {}
                  | INUM INUM '-' RNUM '\n' {};
variable_head : VAR '\n';
variable_declarations : variable_declarations variable_declaration
                      |  ;
```

```
variable_declaration : ID '(' ID ')' ',' ID IN ID '\n' {}
                     | ID opt_id '\n' {};
opt_id : opt_id ',' ID {}
       | {};
function_head : FUNC ID '\n' {}
              | FUNC ID '(' ID ')' ',' ID IN ID '\n' {};
stmts : stmts stmt
      |  ;
stmt : ID '=' expr '\n' {}
     | ID '(' ID ')' '=' expr '\n' {}
     | IF {} '(' logic_expr ')' {} THEN '\n'
       stmts {} opt_else_if opt_else ENDIF '\n' {}
     | LABEL CONTINUE '\n' {}
     | GOTO INUM '\n' {};
opt_else_if : opt_else_if ELSE IF '(' logic_expr ')' {}
              THEN '\n' stmts {}
            |  ;
opt_else : ELSE '\n' stmts
     |  {};
expr : expr '+' expr {}
     | expr '-' expr {}
     | expr '*' expr {}
     | expr '/' expr {}
     | expr '^' expr {}
     | '(' expr ')'
     | '-' expr %prec UMINUS {}
     | number
     | identifier
     | standard_function
     | extern_function
     | SUM {} '(' expr ',' ID IN ID ')' {}
     | PROD {} '(' expr ',' ID IN ID ')' {};
logic_expr : logic_expr AND logic_expr {}
           | logic_expr OR logic_expr {}
           | NOT logic_expr {}
           | '(' logic_expr ')'
           | expr RELOP expr {};
number : RNUM {}
       | INUM {};
identifier : ID {}
           | ID '(' ind_expr ')' {}
           | ID '(' ind_expr ',' ind_expr ')' {};
standard_function : STANDARD {}
                  | STANDARD '(' expr ')' {}
                  | STANDARD '(' expr ',' expr ')' {};
extern_function : EXTERN {}
                | EXTERN '(' ind_expr ')' {}
                | EXTERN '(' ind_expr ',' ind_expr ')' {};
end_module : END '\n';
%%
```

APPENDIX B: Error Messages

PCOMP reports error messages in the form of integer values of the variable IERR and, whenever possible, also line numbers LNUM. The meaning of the messages is listed in the following table. Note that the corresponding text is displayed if the error routine SYMERR is called with the parameters LNUM and IERR.

1	-	file not found - no compilation
2	-	file too long - no compilation
3	-	identifier expected
4	-	multiple definition of identifier
5	-	comma expected
6	-	left bracket expected
7	-	identifier not declared
8	-	data types do not fit together
9	-	division by zero
10	-	constant expected
11	-	operator expected
12	-	unexpected end of file
13	-	range operator '..' expected
14	-	right bracket ')' expected
15	-	'THEN' expected
16	-	'ELSE' expected
17	-	'ENDIF' expected
18	-	'THEN' without corresponding 'IF'
19	-	'ELSE' without corresponding 'IF'
20	-	'ENDIF' without corresponding 'IF'
21	-	assignment operator '=' expected
22	-	wrong format for integer number
23	-	wrong format for real number
24	-	formula too complicated
25	-	error in arithmetic expression
26	-	internal compiler error
27	-	identifier not valid
28	-	unknown type identifier
29	-	wrong input sign
30	-	stack overflow of parser

31	-	syntax error
32	-	available memory exceeded
33	-	index or index set not allowed
34	-	error during dynamic storage allocation
35	-	wrong number of indices
36	-	wrong number of arguments
43	-	number of variables different from declaration
44	-	number of functions different from declaration
45	-	END - sign not allowed
46	-	FORTRAN code exceeds line
47	-	feature not yet supported
48	-	bad input format
49	-	length of working array IWA too small
50	-	length of working array WA too small
51	-	ATANH: domain error
52	-	LOG: domain error
53	-	SQRT: domain error
54	-	ASIN: domain error
55	-	ACOS: domain error
56	-	ACOSH: domain error

APPENDIX C: Generated FORTRAN Code for Example 3

To give an example of the structure of a FORTRAN code generated automatically by PCOMP in reverse accumulation mode, the following lines list the code of Example 3 of Section 4, i.e. of problem function TP295, with 20 variables.

```
      SUBROUTINE XFUN (X,N,F,M,ACTIVE,IERR)
      INTEGER N,M
      DOUBLE PRECISION X(N),F(M)
      LOGICAL ACTIVE(M)
      INTEGER IERR
C
      DOUBLE PRECISION XAUX(21:211)
      INTEGER IO,IXO
      INTEGER I,OFS
C
      INTEGER VINDEX(39)
      INTEGER VICONS(6)
      DOUBLE PRECISION VRCONS(1)
      DATA (VINDEX(I), I=1,39)
     1     /1,2,3,4,5,
     2      6,7,8,9,10,
     3      11,12,13,14,15,
     4      16,17,18,19,20,
     5      1,2,3,4,5,
     6      6,7,8,9,10,
     7      11,12,13,14,15,
     8      16,17,18,19/
      DATA (VICONS(I), I=1,6)
     1     /20,20,1,19,100,
     2      2/
      DATA (VRCONS(I), I=1,1)
     1     /.00000000000000000D+00/
C
      IF (N .NE. 20) THEN
      IERR=43
      RETURN
      ENDIF
      IF (M .NE. 1) THEN
      IERR=44
      RETURN
      ENDIF
      OFS=0
      IF (ACTIVE(1)) THEN
      XAUX(21)=0.0D0
      DO 14 I0=0,18
      IX0=VINDEX(21+I0)
      XAUX(22+OFS)=DBLE(100)
      XAUX(23+OFS)=X(IX0)**(2)
      XAUX(24+OFS)=X(1+IX0)-XAUX(23+OFS)
      XAUX(25+OFS)=XAUX(24+OFS)**(2)
      XAUX(26+OFS)=XAUX(22+OFS)*XAUX(25+OFS)
      XAUX(27+OFS)=DBLE(1)
      XAUX(28+OFS)=XAUX(27+OFS)-X(IX0)
      XAUX(29+OFS)=XAUX(28+OFS)**(2)
      XAUX(30+OFS)=XAUX(26+OFS)+XAUX(29+OFS)
      XAUX(31+OFS)=XAUX(21+OFS)+XAUX(30+OFS)
      OFS=OFS+10
   14 CONTINUE
      OFS=OFS-190
      F(1)=XAUX(211)
      ENDIF
      RETURN
      END
C
C
C
      SUBROUTINE XGRA (X,N,F,M,DF,MMAX,ACTIVE,IERR)
      INTEGER N,M,MMAX
      DOUBLE PRECISION X(N),F(M),DF(MMAX,N)
      LOGICAL ACTIVE(M)
      INTEGER IERR
C
      DOUBLE PRECISION XAUX(21:211),YAUX(21:211)
      INTEGER I0,IX0
      INTEGER I,OFS
C
      INTEGER VINDEX(39)
      INTEGER VICONS(6)
      DOUBLE PRECISION VRCONS(1)
      DATA (VINDEX(I), I=1,39)
     1     /1,2,3,4,5,
     2      6,7,8,9,10,
     3      11,12,13,14,15,
     4      16,17,18,19,20,
     5      1,2,3,4,5,
     6      6,7,8,9,10,
     7      11,12,13,14,15,
     8      16,17,18,19/
      DATA (VICONS(I), I=1,6)
     1     /20,20,1,19,100,
     2      2/
      DATA (VRCONS(I), I=1,1)
     1     /.00000000000000000D+00/
C
      IF (N .NE. 20) THEN
      IERR=43
      RETURN
      ENDIF
      IF (M .NE. 1) THEN
      IERR=44
      RETURN
      ENDIF
      OFS=0
      IF (ACTIVE(1)) THEN
      XAUX(21)=0.0D0
      DO 14 I0=0,18
      IX0=VINDEX(21+I0)
      XAUX(22+OFS)=DBLE(100)
      XAUX(23+OFS)=X(IX0)**(2)
      XAUX(24+OFS)=X(1+IX0)-XAUX(23+OFS)
      XAUX(25+OFS)=XAUX(24+OFS)**(2)
      XAUX(26+OFS)=XAUX(22+OFS)*XAUX(25+OFS)
      XAUX(27+OFS)=DBLE(1)
      XAUX(28+OFS)=XAUX(27+OFS)-X(IX0)
      XAUX(29+OFS)=XAUX(28+OFS)**(2)
      XAUX(30+OFS)=XAUX(26+OFS)+XAUX(29+OFS)
      XAUX(31+OFS)=XAUX(21+OFS)+XAUX(30+OFS)
      OFS=OFS+10
   14 CONTINUE
      OFS=OFS-190
      F(1)=XAUX(211)
      DO 15 I=1,20
      DF(1,I)=0.0D0
   15 CONTINUE
      DO 16 I=21,210
      YAUX(I)=0.0D0
   16 CONTINUE
      YAUX(211)=1.0D0
      OFS=OFS+190
      DO 3 I0=18,0,-1
      IX0=VINDEX(21+I0)
      OFS=OFS-10
      YAUX(21+OFS)=YAUX(21+OFS)+YAUX(31+OFS)
      YAUX(30+OFS)=YAUX(30+OFS)+YAUX(31+OFS)
      YAUX(26+OFS)=YAUX(26+OFS)+YAUX(30+OFS)
      YAUX(29+OFS)=YAUX(29+OFS)+YAUX(30+OFS)
      YAUX(28+OFS)=YAUX(28+OFS)+2*XAUX(28+OFS)**1*YAUX(29+OFS)
      YAUX(27+OFS)=YAUX(27+OFS)+YAUX(28+OFS)
      DF(1,IX0)=DF(1,IX0)-YAUX(28+OFS)
      YAUX(22+OFS)=YAUX(22+OFS)+XAUX(25+OFS)*YAUX(26+OFS)
      YAUX(25+OFS)=YAUX(25+OFS)+XAUX(22+OFS)*YAUX(26+OFS)
      YAUX(24+OFS)=YAUX(24+OFS)+2*XAUX(24+OFS)**1*YAUX(25+OFS)
      DF(1,1+IX0)=DF(1,1+IX0)+YAUX(24+OFS)
      YAUX(23+OFS)=YAUX(23+OFS)-YAUX(24+OFS)
      DF(1,IX0)=DF(1,IX0)+2*X(IX0)**1*YAUX(23+OFS)
    3 CONTINUE
      ENDIF
      RETURN
      END
```

References:

Char B.W., Geddes K.O., Gonnet G.H., Monegan M.B., Watt S.M. (1988): *MAPLE Reference Manual, Fifth Edition,* Symbolic Computing Group, Dept. of Computer Science, University of Waterloo, Waterloo, Canada

Dobmann, M. (1993): *Erweiterungen zum Automatischen Differenzieren,* Diplomarbeit, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

Fischer H. (1991): *Special problems in automatic differentiation,* in: Proceedings of the Workshop on Automatic Differentiation: Theory, Implementation and Application, A. Griewank, G. Corliss eds., Breckenridge, CO

Griewank A., Corliss G. (eds.) (1991): *Automatic Differentiation of Algorithms: Theory, Implementation and Application,* Proceedings of a Workshop, Breckenridge, CO

Griewank A., Juedes D., Srinivasan J. (1991): *ADOL-C: A package for the automatic differentiation of algorithms written in C/C++,* Preprint MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

Griewank A. (1989): *On automatic differentiation,* in: Mathematical Programming: Recent Developments and Applications, ed. M. Iri, K. Tanabe, Kluwer Academic Publishers, Boston, 83-107

Hillstrom K.E. (1985): *Users guide for JAKEF,* Technical Memorandum ANL/ MCS-TM-16, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

Hock W., Schittkowski K. (1981): *Test Examples for Nonlinear Programming Codes,* Lecture Notes in Economics and Mathematical Systems, Vol. 187, Springer

Idnani A. (1987): *NLPSOLVER: User guide,* 3i Corp., Park Ridge, NY

Juedes D.W. (1991): *A taxonomy of automatic differentiation tools,* in: Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation and Application, A. Griewank, G. Corliss eds., Breckenridge, CO

Kagiwada H., Kalaba R., Rosakhoo N., Spingarn K. (1986): *Numerical Derivatives and Nonlinear Analysis,* Plenum Press, New York and London

KEDEM G. (1980): *Automatic differentiation of computer programs,* ACM Transactions on Mathematical Software, Vol.6, No.2, 150-165

KELEVEDZHIEV E., KIROV N. (1989): *Interactive optimization systems,* Working Paper, IIASA, Laxenburg, Austria

KIM K.V. E.AL. (1984): *An efficient algorithm for computing derivatives and extremal problems,* English translation, Ekonomika i matematicheskie metody, Vol.20, No.2, 309-318

KNEPPE G. (1990): *MBB-LAGRANGE: Structural optimization system for space and aircraft structures,* Report, S-PUB-0406, MBB, Munich, Germany

KREDLER C., GREINER M., KÖLBL A., PURSCHE T. (1990): *User's guide for PADMOS,* Report, Institut für Angewandte Mathematik und Statistik, Universität München, Munich, Germany

LIEPELT M. (1990): *Automatisches Differenzieren,* Diplomarbeit, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

RALL L.B. (1981): *Automatic Differentiation - Techniques and Applications,* Lecture Notes in Computer Science, Vol.120, Springer

ROSENBROCK H.H. (1969): *An automatic method for finding the greatest and least value of a function,* Computer Journal, Vol. 3, 175-183

SCHITTKOWSKI K. (1987A): *EMP: An expert system for mathematical programming,* Report, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

SCHITTKOWSKI K. (1987B): *More Test Examples for Nonlinear Programming,* Lecture Notes in Economics and Mathematical Systems, Vol. 182, Springer

CHAR B.W., GEDDES K.O., GONNET G.H., MONEGAN M.B., WATT S.M. (1988): *MAPLE Reference Manual, Fifth Edition,* Symbolic Computing Group, Dept. of Computer Science, University of Waterloo, Waterloo, Canada

DOBMANN, M. (1993): *Erweiterungen zum Automatischen Differenzieren,* Diplomarbeit, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

FISCHER H. (1991): *Special problems in automatic differentiation,* in: Proceedings of the Workshop on Automatic Differentiation: Theory, Implementation and Application, A. Griewank, G. Corliss eds., Breckenridge, CO

Griewank A., Corliss G. (eds.) (1991): *Automatic Differentiation of Algorithms: Theory, Implementation and Application,* Proceedings of a Workshop, Breckenridge, CO

Griewank A., Juedes D., Srinivasan J. (1991): *ADOL-C: A package for the automatic differentiation of algorithms written in C/C++,* Preprint MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

Griewank A. (1989): *On automatic differentiation,* in: Mathematical Programming: Recent Developments and Applications, ed. M. Iri, K. Tanabe, Kluwer Academic Publishers, Boston, 83-107

Hillstrom K.E. (1985): *Users guide for JAKEF,* Technical Memorandum ANL/ MCS-TM-16, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL

Hock W., Schittkowski K. (1981): *Test Examples for Nonlinear Programming Codes,* Lecture Notes in Economics and Mathematical Systems, Vol. 187, Springer

Idnani A. (1987): *NLPSOLVER: User guide,* 3i Corp., Park Ridge, NY

Juedes D.W. (1991): *A taxonomy of automatic differentiation tools,* in: Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation and Application, A. Griewank, G. Corliss eds., Breckenridge, CO

Kagiwada H., Kalaba R., Rosakhoo N., Spingarn K. (1986): *Numerical Derivatives and Nonlinear Analysis,* Plenum Press, New York and London

Kedem G. (1980): *Automatic differentiation of computer programs,* ACM Transactions on Mathematical Software, Vol.6, No.2, 150-165

Kelevedzhiev E., Kirov N. (1989): *Interactive optimization systems,* Working Paper, IIASA, Laxenburg, Austria

Kim K.V. e.al. (1984): *An efficient algorithm for computing derivatives and extremal problems,* English translation, Ekonomika i matematicheskie metody, Vol.20, No.2, 309-318

Kneppe G. (1990): *MBB-LAGRANGE: Structural optimization system for space and aircraft structures,* Report, S-PUB-0406, MBB, Munich, Germany

Kredler C., Greiner M., Kölbl A., Pursche T. (1990): *User's guide for PADMOS,* Report, Institut für Angewandte Mathematik und Statistik, Universität München, Munich, Germany

LIEPELT M. (1990): *Automatisches Differenzieren,* Diplomarbeit, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

RALL L.B. (1981): *Automatic Differentiation - Techniques and Applications,* Lecture Notes in Computer Science, Vol.120, Springer

ROSENBROCK H.H. (1969): *An automatic method for finding the greatest and least value of a function,* Computer Journal, Vol. 3, 175-183

SCHITTKOWSKI K. (1987A): *EMP: An expert system for mathematical programming,* Report, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

SCHITTKOWSKI K. (1987B): *More Test Examples for Nonlinear Programming,* Lecture Notes in Economics and Mathematical Systems, Vol. 182, Springer