PCOMP: A FORTRAN Code for Automatic Differentiation

- Language Description and User's Guide -

Version 5.3 (November, 1996)

M. Dobmann, M. Liepelt, K. Schittkowski, C. Trassl [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 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 and the evaluation of the Hessian matrix, 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 $x \in \mathbb{R}^n$. 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 et al. (1990)), SYSTRA (Kelevedzhiev and Kirov (1989)), EMP (Schittkowski (1987a)), or EASY-OPT (Schittkowski (1993)). 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 like `goto`, `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 and the second derivatives 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). Details about the mathematical background of PCOMP is found in Dobmann, Liepelt and Schittkowski (1994), where also the results of some numerical experiments are reported.

In Section 2 of this paper, we describe the input format for data and functions that is required to execute PCOMP. All allowed operations of the proposed language are defined. Some examples are presented in Section 3. Program organization and use of the FORTRAN subroutines is outlined in Section 4, which is particularly important for those who want to implement PCOMP within their own software environment. Section 5 then shows how user-provided external functions can be linked to PCOMP and called from the underlying program. Two appendices contain a listing of the formal grammar and a list of all error messages.

2. Input Format

The symbolic input of nonlinear functions is only possible if certain syntax rules are satisfied. The PCOMP-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 for identifiers of the user and key words of the language. The length of an identifier has to be smaller than 20 tokens.

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 eleven 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.

* `INDEX`
  Definition of an index variable, which can be used in a `FUNCTION` program block.

* `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 within 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 within 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 (starting at column 7 or later).

* `VARIABLE`
  Declaration of variables with up to one index, with respect to which automatic differentiation is to be performed.

* `CONINT <identifier>`
  Declaration of a piecewise constant interpolation function.

* `LININT <identifier>`
  Declaration of a piecewise linear interpolation function.

* `SPLINE <identifier>`
  Declaration of a spline interpolation function.

* `MACRO <identifier>`
  Definition of a macro function, i.e. an arbitrary set of PCOMP statements that define an auxiliary function to be inserted into subsequent function declaration blocks. Macros are identified by a name that can be used in any right-hand side of an assignment statement.

* `FUNCTION <identifier>`
  Declaration of functions either with up to one index, for which function and derivative values are to be evaluated. The subsequent statements must assign a numerical value to the function identifier.

* `END`
  End of the program.

It is recommended to follow the order of the above program blocks. 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. for variables and functions, index sets and constant data, must begin with a letter and the number of characters, i.e. letters, digits and underscores, must not exceed 20.

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 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

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

The exponential operator ** handles integer exponents in the same way as real ones, i.e. one should prevent non-positive arguments. Available 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)
```

`INDEX`–**Variables:** In PCOMP it is possible to define indices separately to avoid unnecessary differentiation of integer variables. They have to be defined in the program block `INDEX`, e.g.

```
*     INDEX
      i,j
      l
```

It is allowed to manipulate the index e.g. by statements of the form

```
i = 1+2*4-3
i = a(1)
f = a(i+2)+i*2.0
f = SUM(a(m-i), m IN ind)
f = i
f = g(i)
```

In this case, `a` must be declared in form of in integer array. However the following assignment statements are not allowed, if `b` is a real array:

```
i = b(3)
i = 1.0
i = 4/2
f(i) = 3.0
```

Interpolation functions: The built-in constant and linear interpolation functions `CONINT` and `LININT` are scalar functions and are defined by their break points and corresponding function values, e.g.

```
*     CONINT f
      1.0  100.0
      3.0  200.0
      4.0  500.0
      7.0  300.0
*     LININT g
      1.0  100.0
      3.0  200.0
      4.0  500.0
      7.0  300.0
```

For the input of numerical values any format can be used (starting at column 7 or later). The above declaration generates the following interpolation functions f and g:

$$f(x) = \begin{cases} 0.0 & : \quad x < 1.0 \\ 100.0 & : \quad 1.0 \leq x < 3.0 \\ 200.0 & : \quad 3.0 \leq x < 4.0 \\ 500.0 & : \quad 4.0 \leq x < 7.0 \\ 300.0 & : \quad 7.0 \leq x \end{cases}$$

$$g(x) = \begin{cases} 100.0 & : \quad x < 1.0 \\ 100.0 + (x - 1.0) * (200.0 - 100.0)/(3.0 - 1.0) & : \quad 1.0 \leq x < 3.0 \\ 200.0 + (x - 3.0) * (500.0 - 200.0)/(4.0 - 3.0) & : \quad 3.0 \leq x < 4.0 \\ 500.0 + (x - 4.0) * (300.0 - 500.0)/(7.0 - 4.0) & : \quad 4.0 \leq x < 7.0 \\ 300.0 & : \quad 7.0 \leq x \end{cases}$$

Within a function block, the interpolated function is treated like a FORTRAN function, i.e. has to contain a scalar variable or constant in brackets. The interpolation functions are not differentiable at their break points. At these points PCOMP generates the right-hand side differential quotient.

The built-in spline interpolation function SPLINE is also a scalar function and is defined similar to CONINT and LININT, e.g.

```
*     SPLINE h
      -3.0   4.0
      -1.0  -1.0
       0.0  -2.0
       2.0   4.0
       3.0   0.0
       7.0  -1.0
```

```
10.0  -2.0
11.0   4.0
```

Because of the special definition of spline functions, at least four break points and function values are required.

The spline functions generated, are twice differentiable with the exception of the fourth break point. At this point there exists only the first derivative and PCOMP generates the right-hand side differential quotient for the second derivative.

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) could 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..
The GOTO– and the CONTINUE–statements are further possibilities to control the execution of a program. The syntax for these statements is

GOTO ⟨label⟩

and

⟨label⟩ CONTINUE

where label has to be a number between 0 and 9999. Since PCOMP produces labels during the generation of the FORTRAN code in the reverse mode, it is advisable to use labels between 5000 and 9999. The <label> part of the CONTINUE-statement has to be located between columns 2 and 5 of an input line. Together with an index, the GOTO-statement can be used e.g. to simulate DO-loops, which are forbidden in PCOMP, e.g.

```
      i = 1
      s = 0.0
 6000 CONTINUE
      s = s + a(i)*b(i)
      i = i+1
      IF (i.LE.n) THEN
        GOTO 6000
      ENDIF
```

Whenever indices are used within arithmetic expressions, it is allowed 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 an access to f(i) is allowed, not to f(1) or f(j), for example. In other words, PCOMP does not extend the indexed functions to a sequence of single expressions similar to the treatment of SUM and PROD statements.

In PCOMP 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 3. 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.

3. Examples

The following examples illustrate typical applications of the PCOMP language. Most of them have been used to describe mathematical optimization problems.

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
```

Example 2:

- Exponential data fitting:

$$h(x,t) = x_1 x_2 x_3 \left(\frac{x_4 - x_2}{z_1} e^{-x_2(t-\tau)} + \frac{x_4 - x_3}{z_2} e^{-x_3(t-\tau)} + \frac{x_4 - x_5}{z_3} e^{-x_5(t-\tau)} + \frac{x_4 - x_6}{z_4} e^{-x_6(t-\tau)}\right)$$

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 $x \in \mathbb{R}^n$.

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
```

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 was used for testing PCOMP with 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
```

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
```

Example 5:

- Dynamic system:

$$\begin{aligned} \dot{y} &= -a_0 \exp(-\frac{e_p}{rT(t)}) y q_0(t) \\ y_0 &= m_0 \\ y_{fit} &= y \end{aligned}$$

  Here we assume that we want to estimate parameters in an ordinary differential equation with initial value m_0 and the fitting criterion y_{fit}, that describes a chemical reaction. The temparature $T(t)$ is a piecewise linear interpolation function with respect to the time variable t and $q_0(t)$ is obtained by solving a cubic equation that depends on $T(t)$.

- Variables:

$$(a_0, e_p, m_0, y, t) = (2.0, 2.0, 2.0, 2.0, 2.0)$$

- PCOMP program:

```
c
c     Mixed rate model
c

*     REAL CONSTANT
      n=3
```

```
      r=1.987
      e=26920.0
      k0=1.572E-11
      ie=0.56779E-3

*     VARIABLE
      a0, ep, m0, y, t

*     LININT TEMP
      0.0      291.0
      1200.0   316.0
      1800.0   329.0
      1980.0   333.0
      2160.0   340.0
      2280.0   350.0
      2460.0   363.0
      2640.0   367.0
      3000.0   364.0
      3390.0   362.0

*     FUNCTION yp
      kt = k0*DEXP(e/(r*temp(t)))
      pt = 1/(3.0*n*kt)
      qt = -ie/(2.0*n*kt)
      rt = DSQRT(pt)
      wt = -qt/rt**3
      phi = DLOG(wt + DSQRT(wt**2 + 1.0))
      q0 = 2.0*rt*DSINH(phi/3.0)
      yp = -a0*DEXP(-ep/(r*temp(t)))*y*q0

*     FUNCTION y0
      y0 = m0

*     FUNCTION yfit
      yfit = y
C
C     ... Fitting condition
C
C
*     END
C
```

4. 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, WATCOM, MS-FORTRAN, LAHEY).

(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:

```
      parameter (lrsym=15000, lisym=15000)
      double precision rsym(lrsym)
      integer isym(lisym), larsym, laisym, ierr, lrow
      integer nvar,nfunc
      integer dbglev
      open(2,file='pcomp.fun', status='UNKNOWN')
      open(3,file='pcomp.sym', status='UNKNOWN')
      open(4,file='debug.fil',status='UNKNOWN')
      dbglev=3
      call SYMINP(2,3,rsym,lrsym,isym,lisym,larsym,laisym,ierr,lrow,
     F            1,nvar,nfunc,4,dbglev)
      if (ierr.gt.0) goto 900
      goto 9999
  900 call SYMERR(ierr,lrow)
 9999 continue
      close(2)
```

```
      close(3)
      close(4)
      stop
      end
```

(2) **Evaluation of Function, Gradient and Hessian Matrix:**
Proceeding from an intermediate code generated by SYMINP, the function and derivative values are computed within subroutines called SYMFUN, SYMGRA and SYMHES. They can be linked to any user program as required by the underlying application. First and second derivatives 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	-	evaluation of function, gradient and Hessian matrix
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 derivative 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=30, mmax=20, lrsym=50000, lisym=10000)
      dimension x(nmax), f(mmax), df(mmax,nmax),
     /          ddf(mmax,nmax*nmax),irsym(lrsym),rsym(lisym)
      logical act(mmax)
      integer nvar,nfunc
      integer dfx(nmax),dfxlen
      open(3,file='pcomp.sym',status='UNKNOWN')
      n=2
      m=1
      x(1)=1.0
      x(2)=-1.2
      act(1)=.true.
      dfx(1)=1
      dfx(2)=2
      dfxlen=2
      call SYMPRP(3,rsym,lrsym,isym,lisym,larsym,laisym,ierr,2,
     F            nvar,nfunc)
```

```
      if (ierr.gt.0) goto 900
      call SYMFUN(x,n,f,m,act,rsym,lrsym,isym,lisym,dfx,dfxlen,ierr)
      if (ierr.gt.0) goto 900
      call SYMGRA(x,n,f,m,df,mmax,act,rsym,lrsym,isym,lisym,dfx,dfxlen,
     F            ierr)
      if (ierr.gt.0) goto 900
      call SYMHES(x,n,f,m,df,ddf,mmax,act,rsym,lrsym,isym,lisym,dfx,
     F            dfxlen,ierr)
      if (ierr.gt.0) goto 900
      write(*,*) f(1),df(1,1),df(1,2)
      write(*,*) ddf(1,1),ddf(1,2),ddf(1,3),ddf(1,4)
      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 are two files to be linked to the user code, and the error routine as before:

PCOMP_G1.FOR	-	routines to generate FORTRAN code
PCOMP_G2.FOR	-	routines to generate FORTRAN code
PCOMP_ER.FOR	-	error messages

```
      parameter (lrsym=15000, lisym=15000)
      double precision rsym(lrsym)
      integer isym(lisym), larsym, laisym, ierr, lrow,nvar,nfunc
      open(3,file='pcomp.sym', status='UNKNOWN')
      open(4,file='pcomp.for', status='UNKNOWN')
      call SYMPRP(3,rsym,lrsym,isym,lisym,larsym,laisym,ierr,1,
     F            nvar,nfunc)
      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), Liepelt (1990) and Trassl (1993) 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,
  MODE,NVAR,NFUNC,DEBFIL,LEVDEB)

- **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. If SYMFIL is set to zero when calling SYMINP, then only the working arrays are filled with the intermediate program code.
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. In the latter case, call SYMERR for more information.
LNUM	-	In case of unsuccessful termination, LNUM contains the line number where the error was detected.
MODE	-	MODE = 0 : WA is supposed to supply sufficient working space to store function values.
	-	MODE = 1 : WA is supposed to supply sufficient working space to store function values and corresponding first derivatives.
	-	MODE = 2 : WA is supposed to supply sufficient working space to store function values and corresponding first and second derivatives.
NVAR	-	On return, NVAR contains the number of variables of the INPUT file
NFUNC	-	On return, NFUNC contains the number of functions of the INPUT file
DEBFIL	-	An integer identifying the output file number to which debug information is to be written, if required the parameter LEVDEB.
LEVDEB	-	When calling SYMINP, LEVDEB has to contain the desired debugging level: LEVDEB = 0 : No debugging information generated. LEVDEB = 1 : Debug information generated by scanner. LEVDEB = 2 : Debug information generated by parser. LEVDEB = 3 : Full debug information generated.

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,MODE,
  NVAR,NFUNC)

- **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. In the latter case, call SYMERR for more information.
MODE	-	MODE = 0 : WA is tested for sufficient space to store function values.
	-	MODE = 1 : WA is tested for sufficient space to store function values and corresponding first derivatives.
	-	MODE = 2 : WA is tested for sufficient space to store function values and corresponding first and second derivatives.
NVAR	-	On return, NVAR contains the number of variables of the INPUT file
NFUNC	-	On return, NFUNC contains the number of functions of the INPUT file

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,DFX,DFXLEN,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 $g_j(x)$. ACTIVE(J)=.FALSE. : Do not compute $g_j(x)$.
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.
DFX	-	Array for the evaluating function EVAL.
DFXLEN	-	Length of the array DFX.
IERR	-	On return, IERR shows the termination reason of SYMFUN: IERR = 0 : Successful termination. IERR > 0 : There is an error in the input file. In the latter case, 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,DFX,
  DFXLEN,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. : Compute function $g_j(x)$. ACTIVE(J)=.FALSE. : Do not compute $g_j(x)$.
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.
DFX(DFXLEN)	-	INTEGER-array of length DFXLEN that contains the numbers of the variables for which the first derivatives are to be evaluated.
DFXLEN	-	Lenght of the vector DFX.
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 SYMHES:

- **Purpose:**
  The intermediate code is passed from a SYMINP call to SYMHES in the form of a real and an integer working array. Given a variable vector x, the subroutine computes the corresponding function and derivative values $f_i(x)$, $\nabla f_i(x)$ and $\nabla^2 f_i(x)$. The functions, gradients and hessian matrices that are to be evaluated by SYMHES must be specified by a logical array.

- **Calling sequence:**
  SYMHES (X,N,F,M,DF,DDF,MMAX,ACTIVE,WA,LWA,IWA,LIWA,DFX
  DFXLEN,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.
DDF(MMAX,N∗N)	-	Two-dimensional double precision array to i take over the values of the Hessian matrix computed by SYMHES. 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. : Compute function $g_j(x)$. ACTIVE(J)=.FALSE. : Do not compute $g_j(x)$.
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.
DFX(DFXLEN)	-	INTEGER-array of length DFXLEN that contains the numbers of the variables for which the first and second derivatives are to be evaluated.
DFXLEN	-	Lenght of the vector DFX.
IERR	-	On return, IERR shows the termination reason of SYMHES: 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 and additional storage that 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. in hte latter case, call SYMERR for more information.

5. 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 derivative and insert their calling sequences in subroutines EXTFUN, EXTGRA and EXTHES.

A user can change the module system provided by the authors. The interface functions EXTFUN, EXTGRA and EXTHES 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.

Subroutine EXTHES:

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

- **Calling sequence:**
  EXTHES (EXT,X,N,DDF,EXTPAR)

- **Parameters:**

EXT	-	Integer value to identify the EXT-th external function for which the Hessian matrix 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 Hessian matrix is to be evaluated.
N	-	Dimension, i.e. number of variables.
DDF(N∗N)	-	Double precision array of length N∗N to take over the second derivative values of function EXT on return.
EXTPAR(2)	-	Integer array of length two containing up to two actual parameters when calling EXTFUN.

Example 6:

- 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 derivation evaluation, if we assume that a function evaluation always preceeds a derivation evaluation. The following PCOMP program contains four 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 EXTHES (ext, x, n, ddf, extpar)
  integer ext, n, extpar(2)
  double precision x(n), df(n)
  goto (1,2,3,4) ext
1 call HAX(x, n, ddf, extpar(1))
  return
2 call HBX(x ,n ,ddf)
  return
3 call HXLOGX(x ,n ,ddf)
  return
4 call HX1(x, n, ddf)
  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 HAX (x, n, ddf, i)
  double precision x(n), ddf(n*n), a, b
  do 1 j=1,n*n
1 ddf(j)=0.0D0
  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 HBX (x, n, ddf)
  double precision x(n), ddf(n*n), a, b
  do 1 j=1,n*n
1 ddf(j)=0.0D0
  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 HXLOGX (x, n, ddf)
      double precision x(n), ddf(n*n), logx
      do 1 j=1,n*n
    1 ddf(j)=0.0D0
      do 2 j=1,n
    2 ddf(j*j)=1.0/x(j)
      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

      subroutine HX1 (x, n, ddf)
      double precision x(n), ddf(n*n)
      do 1 j=1,n*n
    1 ddf(j)=0.0D0
      return
      end
```

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,INTERPOL
%token PARAM, INDEX, REAL, INT, TABLE, CONINT, LININT, SPLINE, VAR
%token INFUNC, FUNC, END, GOTO, LABEL, 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
                  | con_interpolation_head interpolation_declarations
                  | lin_interpolation_head interpolation_declarations
                  | spl_interpolation_head spline_declarations
                  | variable_head variable_declarations
                  | infunc_head infunc_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' {};
con_interpolation_head : CONINT ID '\n' {};
lin_interpolation_head : LININT ID '\n' {};
interpolation_declarations : interpolation_declarations
                             interpolation_declaration
                           |  ;
interpolation_declaration : RNUM RNUM '\n' {}
                          | RNUM '-' RNUM '\n' {}
                          | '-' RNUM RNUM '\n' {}
                          | '-' RNUM '-' 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 {}
       | {};
infunc_head : INFUNC '\n' {};
infunc_declarations : infunc_declarations infunc_declaration
                    |  ;
infunc_declaration : ID opt_id '\n' {};
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
     | interpolation_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 ')' {};
interpolation_function : INTERPOL '(' expr ')' {};
spl_interpolation_head : SPLINE ID '\n' {};
spline_declarations : spline_declarations
                      spline_declaration
                    |  ;
spline_declaration : RNUM RNUM '\n' {}
                   | RNUM '-' RNUM '\n' {}
                   | '-' RNUM RNUM '\n' {}
                   | '-' RNUM '-' RNUM '\n' {};
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
57 - LABEL defined more than once
58 - LABEL not found
59 - wrong index expression
60 - wrong call of the subroutine SYMINP
61 - wrong call of the subroutine SYMPRP
62 - compilation of the source file in GRAD-mode
63 - interpolation values not in right order
64 - not enough space for interpolation functions in subroutine REVCDE
65 - length of working array IWA in subroutine SYMFOR too small
66 - not enough interpolation values
67 - compilation of source file not in GRAD-mode
68 - missing macro name
69 - more than MAXMAC macros defined

70 - more than MAXBUF lines of macro statements
71 - more than MAXBUF statements in function

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

Dobmann M., Liepelt M., Schittkowski K. (1994): *PCOMP: A FORTRAN code for automatic differentiation,* to appear: ACM Transactions on Mathematical Software

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. ET 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

SCHITTKOWSKI K. (1993): *Easy-to-use optimization programs with automatic differentiation,* Report, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany

TRASSL C. (1993): *Zweite Ableitungen beim Automatischen Differenzieren in PCOMP*, Diplomarbeit, Mathematisches Institut, Universität Bayreuth, Bayreuth, Germany