Algorithm 746: New Features of PCOMP, a Fortran Code for Automatic Differentiation

Michael Liepelt and Klaus Schittkowski[†]

April 18, 2014

Abstract

The software system PCOMP uses automatic differentiation to calculate derivatives of functions that are defined by the user in a modeling language similar to Fortran. This symbolical representation is converted into an intermediate code, which can be interpreted to calculate function and derivative values at run-time within machine accuracy. Furthermore, it is possible to generate Fortran code for function and gradient evaluation, which has to be compiled and linked separately. The first version of PCOMP was introduced in Dobmann *et al.* [2]. In this paper, we describe a series of extensions and additional features that have been implemented in the meantime.

1 Introduction

Let $f(x)$ be a real valued differentiable function defined for $x \in \mathbb{R}^n$. By automatic differentiation we understand the numerical computation of the derivative value $\nabla f(x)$ at a given point x without using approximation techniques or explicit formulas for the derivatives. Automatic differentiation has become an important tool for numerical algorithms that require derivatives in areas such as nonlinear programming, optimal control, parameter estimation, or differential equations.

It is beyond the scope of this paper to give a decisive description of the underlying forward and backward accumulation methods and so we assume that the reader is already familiar with Dobmann *et al.* [2]. For an explanation of the theoretical background, we refer to Griewank [4]. PCOMP is a

[†]Department of Mathematics, University of Bayreuth, 95440 Bayreuth, Germany

collection of Fortran subroutines that implement both approaches, see Dobmann *et al.* [2] for details. PCOMP uses a special modeling language similar to Fortran for the declaration of arbitrary nonlinear functions. After parsing the input file and generating an intermediate code, function, gradient, and Hessian values can be directly evaluated by special subroutines called from a user program. On the other hand, it is possible to generate Fortran code for function and gradient evaluation that has to be compiled and linked separately.

Meanwhile there exists a large variety of alternative approaches especially with the goal of differentiating Fortran or C programs, see Juedes [6] for an overview, or Griewank *et al.* [5] for the description of an implementation that uses operator overloading in C++.

The new PCOMP is upwards compatible with the 1995 version, that is, any PCOMP program written for the 1995 version may be run with the new version. The extensions that are to be outlined in detail are as follows:

- more flexible identifier names
- interpolation functions (piecewise constant, piecewise linear, splines)
- macros for groups of PCOMP statements
- index variables, e.g., for the implementation of loops
- `GOTO` statements
- evaluation of gradients w.r.t. arbitrary subsets of variables
- second derivatives in forward mode
- additional error messages, especially for run-time execution

Moreover, a few bugs have been fixed and PCOMP has been tested on a multitude of different Fortran compilers. As part of programs for parameter estimation in systems of algebraic equations, in ordinary differential equations, in differential algebraic equations, and in time-dependent one-dimensional partial differential equations, PCOMP is in permanent use in many different disciplines, for instance pharmaceutics, chemical engineering, mechanical engineering, or geology, see Schittkowski [7, 8, 11, 9, 10].

2 New Language Elements

2.1 Identifiers

In order to facilitate the problem formulation, the nomenclature of Fortran 77 has been extended and identifiers for variables or functions may now contain underscores and may consist of up to 20 characters.

2.2 Index Variables

The concept of index variables already exists in the 1995 version of PCOMP; they are used in the definition of vectors and matrices and in `SUM` and `PROD` statements. For example, in the following statement to compute the scalar product of two vectors

```
      F = SUM(A(I)*X(I), I IN INDEX)
```

the variable `I` is implicitly declared as an index variable.

In order to make the structure of a PCOMP program more consistent, these index variables can also be explicitly declared in a new program block called `INDEX`, for instance:

```
*     INDEX
      I, J
      L
```

Especially together with the enhanced control structures, it is sensible to allow for assignments to these variables. If we assume that `A` is declared as an integer array and that `F` and `G` are scalar and vector valued functions, respectively, the following statements show some typical applications:

```
      I = 1+2*4-3
      I = A(1)
      F = A(I+2)+I*2.0
      F = SUM(A(M-I), M IN INDEX)
      F = I
      F = G(I)
```

Since these index variables are primarily used as indices for vectors and matrices, it is quite obvious that they should only take integer values. Thus the following statements are not permitted, if we assume that `B` is declared as a real array:

```
I = 1.0
I = 4/2
I = B(3)
```

Because of the special role of index variables, we do not allow these assignments to avoid a misleading usage. Note also, that although PCOMP distinguishes between integer and real constants, integer variables do not exist.

2.3 Interpolation Functions

A new feature of the 1999 version of PCOMP is the possibility of interpolating user defined data, using either a piecewise constant, piecewise linear, or a cubic spline function. Given n pairs of real values (t_1, y_1), ..., (t_n, y_n), we are looking for a nonlinear function interpolating these data.

In the first case, we define a piecewise constant interpolation by

$$c(t) := \begin{cases} 0, & \text{if } t < t_1, \\ y_i, & \text{if } t_i \le t < t_{i+1} \text{ for } i = 1, \ldots, n-1, \\ y_n, & \text{if } t_n \le t. \end{cases}$$

A continuous piecewise linear interpolation function is given by

$$l(t) := \begin{cases} y_1, & \text{if } t < t_1, \\ y_i + \frac{t-t_i}{t_{i+1}-t_i}(y_{i+1} - y_i), & \text{if } t_i \le t < t_{i+1} \text{ for } i = 1, \ldots, n-1, \\ y_n, & \text{if } t_n \le t, \end{cases}$$

and a piecewise cubic spline is given by

$$s(t) := \begin{cases} p(t; t_1, t_2, t_3, t_4, y_1, y_2, y_3, y_4), & \text{if } t < t_4, \\ \bar{s}(t; t_4, \ldots, t_n, y_4, \ldots, y_n, \frac{d}{dt}p(t_4; \ldots), 0), & \text{if } t_4 \le t, \end{cases}$$

where $p(t; t_1, t_2, t_3, t_4, y_1, y_2, y_3, y_4)$ is a cubic polynomial with

$$p(t_i; t_1, t_2, t_3, t_4, y_1, y_2, y_3, y_4) = y_i, \qquad i = 1, \ldots, 4,$$

and $\bar{s}(t; \bar{t}_1, \ldots, \bar{t}_m, \bar{y}_1, \ldots, \bar{y}_m, \bar{y}'_1, \bar{y}'_m)$ is a cubic spline function interpolating $(\bar{t}_1, \bar{y}_1)$, ..., $(\bar{t}_m, \bar{y}_m)$ subject to boundary conditions

$$\frac{d}{dt}\bar{s}(\bar{t}_i; \bar{t}_1, \ldots, \bar{t}_m, \bar{y}_1, \ldots, \bar{y}_m, \bar{y}'_1, \bar{y}'_m) = \bar{y}'_i, \qquad i = 1 \quad \text{and} \quad i = m.$$

It is essential to understand that the constant and spline interpolation functions are not symmetric. As noted before, a very typical class of application problems consists of dynamic systems, say ordinary or partial differential equations, where the initial value is set to 0 without loss of generality, leading to a non-symmetric domain. In these cases, derivatives of the right-hand side of the differential equations and the initial conditions are required for two reasons: First the Jacobian of the right-hand side w.r.t. the state variables is needed for applying implicit integration routines, in case of stiff systems or additional algebraic equations. Secondly, dynamic systems are often embedded in optimization problems, for example in data fitting or optimal control applications. Thus one has to compute derivatives of the solution w.r.t. certain design parameters, for example by integrating the sensitivity or variational equations, respectively, or by any related technique. Even worser, if we integrate the resulting system by an implicit method, we have to compute also second mixed partial derivatives w.r.t. state and design variables.

Moreover interpolated data are often based on experiments that reach a steady state, that is, a constant value. Thus a zero derivative is chosen at the right end point for spline interpolation to facilitate the input of interpolated steady-state data. On the other hand, any other boundary conditions can be enforced by adding artificial interpolation data.

To give an example, we assume that we want to interpolate the nonlinear function $f(t)$ given by the discrete values $f(t_i) = y_i$ from Table 2.1, using the different techniques mentioned above.

i	t_i	y_i
1	0.0	0.00
2	1.0	4.91
3	2.0	4.43
4	3.0	3.57
5	4.0	2.80
6	5.0	2.19
7	6.0	1.73
8	7.0	1.39
9	8.0	1.16
10	9.0	1.04
11	10.0	1.00

Table 2.1: Interpolation data

Interpolation functions are defined by a program block starting with the

keyword `CONINT` for piecewise constant functions, `LININT` for piecewise linear functions, or `SPLINE` for piecewise cubic splines, followed by the name of the function. The numerical values of the break points and the function values are given on the subsequent lines, using any standard format starting at column 7 or later. Using piecewise constant approximations, we get for our example:

```
*     CONINT F
      0.0  0.00
      1.0  4.91
      2.0  4.43
      3.0  3.57
      4.0  2.80
      5.0  2.19
      6.0  1.73
      7.0  1.39
      8.0  1.16
      9.0  1.04
      10.0 1.00
```

Within a function definition block, the interpolation functions are treated as intrinsic Fortran functions, that is, they have to contain a variable or constant as a parameter. If we assume that `T` has previously been declared as a variable, a valid statement could look like

```
*     FUNCTION KT
      TEMP = F(T) + 273
      KT = K0*EXP(E/(R*TEMP))
```

The resulting approximations for piecewise constant functions, piecewise linear functions, or piecewise cubic spline functions are depicted in Figures 2.1–2.3. Whereas the cubic spline approximation is twice differentiable on the whole interval, the other two approximations are not differentiable at the break points and PCOMP uses the right-hand sided derivatives instead.

2.4 Control Statements

Using `GOTO` and corresponding `CONTINUE` statements is another possibility to control the execution of a program. The syntax of these statements is

`GOTO` ⟨label⟩

and

⟨label⟩ `CONTINUE`

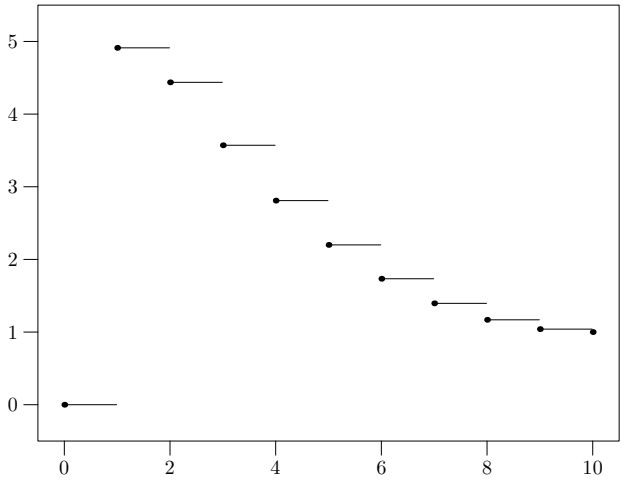

Figure 2.1: Piecewise constant interpolation

where ⟨label⟩ has to be a number between 1 and 9999. Since PCOMP uses labels in the Fortran code that is generated in reverse accumulation mode, the user defined labels must be between 5000 and 9999 in this case to avoid unnecessary clashes. The ⟨label⟩ part of the `CONTINUE` statement has to be located between columns 2 and 5 of an input line. Together with an index variable, the `GOTO` statement can also be used to implement `DO` loops, which are not yet available in PCOMP:

```
      I = 1
      S = 0.0
 6000 CONTINUE
      S = S+A(I)*B(I)
      I = I+1
      IF (I .LE. N) THEN
        GOTO 6000
      ENDIF
```

However, it is recommended to avoid `GOTO` statements whenever possible, and to replace them by `SUM` and `PROD` statements, if applicable. If not implemented very carefully, one has to be afraid that a function to be defined becomes non-differentiable.

2.5 Macros

PCOMP does not allow the declaration of subroutines. However, it is now

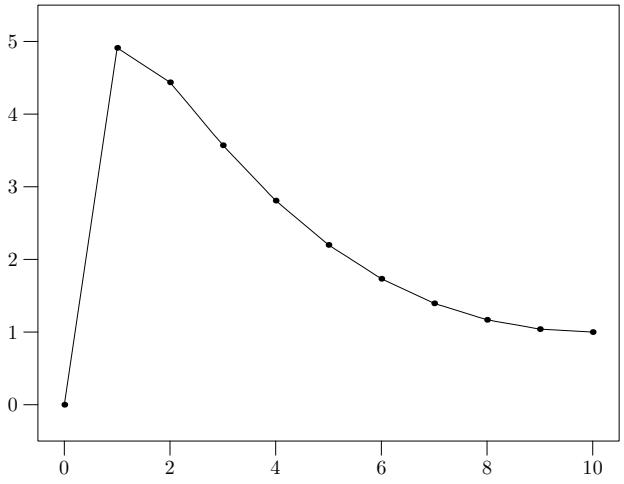

Figure 2.2: Piecewise linear interpolation

possible to define macros, that is, arbitrary sequences of PCOMP statements that define an auxiliary variable to be inserted into the beginning of subsequent function declaration blocks. Macros are identified by a name that can be used in any right-hand side of an assignment statement,

```
*       MACRO ⟨identifier⟩
```

followed by a group of PCOMP statements that assign a numerical value to the given identifier. This group of statements is inserted into the source code block, that contains the macro name. Macros have no arguments, but they may access all variables, constants, or functions that have been declared up to their first usage. Any values assigned to local variables within a macro, are also available outside in the corresponding function block.

If we assume that `X` is a variable and we want to define a macro that computes the square of `X`, we would write something like

```
*       MACRO SQRX
        SQRX = X*X
```

Now it is possible to replace each occurence of the term `X*X` with an invocation of the macro that we have just defined, for example

```
        F = SQRX-5.2
```

It should be noted that empty lines and all lines after the final `END` statement are ignored by PCOMP.

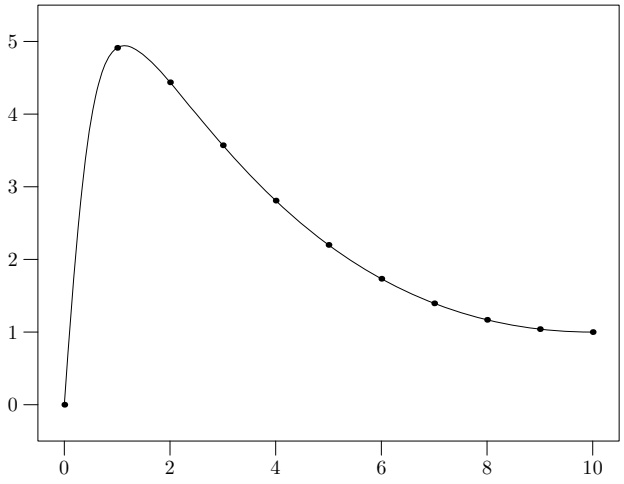

Figure 2.3: Piecewise cubic spline interpolation

3 Derivatives w.r.t. Subsets of the Variables

In the previous version of PCOMP, derivatives are always evaluated for the full set of all declared variables. However, there are situations where we need Jacobians or Hessians only with respect to a certain subset of parameters that could change within the outer algorithm.

Typical examples are parameter estimation and optimal control of dynamic systems, see Schittkowski [8, 11] or Blatt and Schittkowski [1]. In these cases we need to calculate derivatives with respect to parameters that occur in the initial values or the right-hand side of an ordinary differential equation of the form

$$\begin{aligned} \dot{y}_1(t) &= h_1(y,x,t), & y_1(0) &= y_1^0(x), \\ &\vdots & &\vdots \\ \dot{y}_m(t) &= h_m(y,x,t), & y_m(0) &= y_m^0(x). \end{aligned}$$

In order to avoid the approximation of derivatives by divided differences, we can solve the variational equations to compute $\nabla y(x,t)$, where $y(x,t)$ denotes the solution of the ODE. To formulate these equations, we need the partial derivatives

$$\frac{\partial}{\partial y_j} h_i(y,x,t), \qquad \frac{\partial}{\partial x_k} y_i^0(x), \qquad \frac{\partial}{\partial x_k} h_i(y,x,t)$$

for $i,j = 1, \ldots, m$ and $k = 1, \ldots, n$.

Moreover, the outer optimization algorithm might require the evaluation of gradients of an objective function that is defined in the same input file, with respect to the optimization variables x only, or a separate integration of the ODE by implicit methods, for which we need only the derivatives with respect to y. But we do not need any derivatives for the additional variable t. Note that also in the previous PCOMP version it is possible to evaluate derivatives for certain subsets of all declared functions.

The new version of PCOMP uses an index array to determine for which indices the first or second derivatives are to be evaluated.

4 Second Derivatives

Second derivatives are often highly desirable to achieve reliable and efficient algorithms. If we need to apply an implicit method to solve the variational equations of an ordinary differential equation for which derivatives of the solution are to be computed, then second derivatives of the original system are required. Another example is the usage of Newton's method for the solution of systems of nonlinear equations or nonlinear programming problems.

Second derivatives are easily obtained by an extension of the forward accumulation method that has been implemented in PCOMP. In order to apply any automatic differentiation technique to calculate the derivatives of a function $f(x)$ for $x \in \mathbb{R}^n$, the first step is to convert f into a sequence of basis functions $\{f_i\}$, using auxiliary variables for intermediate values. We assume that there are a finite sequence of basis functions f_{n+1}, ..., f_m and index sets $\mathcal{J}_i \subset \{1, \ldots, i-1\}$, such that the function value of f at any given point x can be obtained by the following program,

for $i := n + 1$ **to** m **do**
  $x_i := f_i(x_l)_{l \in \mathcal{J}_i}$;
$f(x) := x_m$;

see Dobmann *et al.* [2] for details. Basis functions are elementary arithmetic operations and intrinsic or external functions, where the number of operands is limited by a constant independent of n.

Proceeding from a given factorization, derivative values up to second order can be obtained by inserting the corresponding derivatives of the basis functions into the algorithm above:

for $i := 1$ **to** m **do begin**
  $\nabla x_i := e_i$;
  $\nabla^2 x_i := 0$;
end;

for $i := n+1$ **to** n **do begin**
$x_i := f_i(x_l)_{l \in \mathcal{J}_i}$;
$\nabla x_i := \sum_{j \in \mathcal{J}_i} \frac{\partial}{\partial x_j} f_i(x_l)_{l \in \mathcal{J}_i} \nabla x_j$;
$\nabla^2 x_i := \sum_{j \in \mathcal{J}_i} \left(\frac{\partial}{\partial x_j} f_i(x_l)_{l \in \mathcal{J}_i} \nabla^2 x_j + \sum_{k \in \mathcal{J}_i} \nabla x_j \frac{\partial^2}{\partial x_j \partial x_k} f_i(x_l)_{l \in \mathcal{J}_i} (\nabla x_k)^T \right)$;
end;
$f(x) := x_m$;
$\nabla f(x) := \nabla x_m$;
$\nabla^2 f(x) := \nabla^2 x_m$;

It is important to understand that function and derivative values are simultaneously computed in forward accumulation mode, i.e., PCOMP uses alternative subroutines for function evaluation only, for function and gradient evaluation, and for function, gradient, and Hessian evaluation.

Although generated Fortran code for the evaluation of functions and gradients based on the reverse mode, is substantially faster than interpreting the intermediate code, this approach cannot be used to calculate second derivatives. This is a fundamental drawback of the implemented reverse accumulation algorithm, see Griewank [4] for a more rigorous treatment of the problem.

5 Program Organization

PCOMP is a collection of Fortran subroutines that can be subdivided into four different categories. In this section we want to give only a short overview of the way PCOMP works; for more details we refer to the user's guide `pcompdoc.ps` contained in the subdirectory `./doc` of the PCOMP package.

5.1 Generation of Intermediate Code

First of all, the PCOMP source code is analyzed by the subroutine SYMINP. Intermediate code is generated for subsequent evaluation by SYMFUN, SYMGRA, or SYMHES, or alternatively for transformation into executable Fortran code by SYMFOR. If there is any error during the processing of the input file, SYMINP is interrupted and the error code can be retrieved from the integer parameter IERR. This error code and the corresponding line number can be passed to SYMERR to generate an error message on standard output. Note that the parser stops at the first error found.

If no errors have been detected, the generated intermediate code and additional data are stored in a double precision working array WA and an integer working array IWA. These two working arrays must be passed to all

subsequently called subroutines for function and derivative evaluation or for code generation, respectively.

Since PCOMP supports a modular concept, the generated intermediate code and the additional data are stored in an external file SYMFIL, from where it can be retrieved using the subroutine SYMPRP. A new feature of the 1999 version of PCOMP is the possibility to generate additional debug information, see Dobmann *et al.* [3] for details of this option.

5.2 Runtime Evaluation of Functions and Derivatives

As outlined in the previous subsection, the intermediate code generated by SYMINP is passed to SYMFUN in the form of a real and an integer working array. Given any variable vector x, this subroutine computes the corresponding function values $f_i(x)$ by interpreting the intermediate code. The subroutines SYMGRA and SYMHES are very similar to SYMFUN. The only distinction is that they also compute the gradients or gradients and Hessians of the symbolically defined functions, respectively.

The 1999 version of PCOMP requires an additional index array DFX to indicate for which variables the first or second derivatives are to be evaluated. Note that already in the previous PCOMP version it is possible to calculate function values and derivatives only for some of the declared functions.

5.3 Generation of Fortran Code

Proceeding from the intermediate code generated by the parser SYMINP, the subroutine SYMFOR generates two Fortran subroutines for function and gradient evaluation on a given output file. These routines have to be compiled and linked separately. The calling sequences of the generated subroutines XFUN and XGRA are analogous to SYMFUN and SYMGRA, except that no working arrays are required.

5.4 Interface to External Subroutines

The basic idea of PCOMP is to factorize a given function $f(x)$ with respect to a set of library functions, for which derivatives can easily be computed, for instance elementary arithmetic and intrinsic functions. For a complete list of available operations, we refer to Dobmann *et al.* [2].

In some applications it is desirable to extend the list of library functions and to allow for user-provided symbols in the source code. Thus PCOMP can easily be extended to accept additional functions by inserting information about structure, type, and symbolic name into the parser and by defining

subroutines for function, gradient, and Hessian evaluation called EXTFUN, EXTGRA, and EXTHES. For more details and an example see Dobmann *et al.* [3].

References

[1] M. Blatt and K. Schittkowski. Optimal control of one-dimensional partial differential algebraic equations with applications. *Annals of Operations Research*, to appear.

[2] M. Dobmann, M. Liepelt, and K. Schittkowski. Algorithm 746: PCOMP: A Fortran code for automatic differentiation. *ACM Transactions on Mathematical Software*, 21:233–266, 1995.

[3] M. Dobmann, M. Liepelt, K. Schittkowski, and C. Trassl. PCOMP: A Fortran code for automatic differentiation – language description and user's guide (version 5.3). Technical report, University of Bayreuth, Bayreuth, Germany, 1996.

[4] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–107. Kluwer Academic Publishers, Boston, 1989.

[5] A. Griewank, D. W. Juedes, and J. Srinivasan. 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, Illinois, 1991.

[6] D. W. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation and Applications*, pages 315–330. SIAM, Breckenridge, Colorado, 1991.

[7] K. Schittkowski. Parameter estimation in systems of nonlinear equations. *Numerische Mathematik*, 68:129–142, 1994.

[8] K. Schittkowski. Parameter estimation in differential equations. In R. P. Agarwal, editor, *Recent Trends in Optimization Theory and Applications*, pages 353–370. World Scientific Publishing Company, Singapore, 1995.

[9] K. Schittkowski. Parameter estimation in one-dimensional time-dependent partial differential equations. *Optimization Methods Software*, 7:165–210, 1997.

[10] K. Schittkowski. PDEFIT: A Fortran code for parameter estimation in partial differential equations. *Optimization Methods Software*, 10:539–582, 1999.

[11] K. Schittkowski. EASY-FIT: A software system for data fitting in dynamic systems. *Journal of Design Optimization*, to appear.