Overloading Point and Interval Taylor Operators

George F. Corliss

CRPC-TR92243
1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Overloading Point and Interval Taylor Operators*

George F. Corliss†

Abstract. This volume includes papers on a variety of implementation techniques for automatic differentiation, and papers on a variety of applications. For each, the user must write a program, in some sense, for the function to be differentiated. This paper outlines the use of overloaded operators to make the necessary programming as user-friendly as possible. Most implementations of either the forward or the reverse mode of automatic differentiation generate all requested derivatives in the same pass through the computational graph. We describe an efficient implementation strategy when it is required to generate Taylor coefficients one order at a time. We then address trade-offs of memory space, time, and ease of use in the implementation of overloaded operators for use in applications which require that derivatives be generated one at a time, rather than all in the same call.

1 Introduction. The goal of this paper is to present a programmer-friendly way of expressing automatic differentiation operators. Many of the systems for automatic differentiation reported in this volume require a programmer to write code. Here, we consider the use of operator overloading to make the programming required as routine as possible. As an example, the programming style of Listing 1 is *strongly* preferred to that of Listing 2.

```
A, B, C : AD_Type;
. . .

C := B + sin (A);
. . .
```

Listing 1. Convenient user interface to automatic differentiation operators.

```
      INTEGER LENSER
      PARAMETER (LENSER = 30)
      INTEGER LA, MA, LB, MB, LC, MC, LT, MT
      REAL AL(LENSER), AR(LENSER), BL(LENSER), BR(LENSER),
     +     CL(LENSER), CR(LENSER), TL(LENSER), TR(LENSER)
```

*This work was supported in part by IBM Germany and by the National Science Foundation under grant No. CCR–8802429.

†Dept. of Mathematics, Marquette University, Milwaukee, WI 53233 USA

```
      . . .

C  C = B + SIN (A)
      CALL IT_SIN (LA, MA, AL, AR, LT, MT, TL, TR, IER)
      CALL IT_ADD (LB, MB, BL, BR, LT, MT, TL, TR,
     +             LC, MC, TL, TR, IER)
      . . .
```

Listing 2. Awkward user interface to automatic differentiation operators.

Listing 1 is the style of the Ada-based operators reported in [Corl91b]; Listing 2 is the style of the Fortran-based SVALAQ system [Corl87a].

The presentation of this paper is intended to be generic; the AD_Type in Listing 1 could represent any appropriate type such as Taylor series, gradient, Jacobian, Hessian, etc. for real, complex, interval, or complex interval objects. In Section 4, we discuss some implementation strategies which apply more specifically to Taylor series generation than to the generation of partial derivatives.

Christianson [Chri91a] computes Hessians by reverse accumulation (see [Irim84a]) using overloaded Ada operators. Second derivatives have previously been obtained using two (apparently different) algorithms, which Christianson shows to be identical. He calculates a single row (or an arbitrary linear combination of rows) of the Hessian of a function f in less than twice the time required to calculate $\{f, \nabla f\}$ by reverse accumulation.

The techniques of operator overloading reported here require language support. ACRITH-XSC [IBM90a], Ada [AdaLRM], C++ [Stro86a], Maple [Char88a], Pascal-XSC [Klat91a],and S [Beck88a] all support user defined data types and overloading of operators for those types. Many of the preprocessors used for automatic differentiation have the effect of operator overloading. Examples include Kedem [Kede80a], ADOL-C [Grie92a], ATOMFT [Corl82a], Augment [Crar79a], DAPRE [Step90a], JAKEF [Hill85a], GRESS [Horw88a], and PADRE2 [Irim84a], [Irim88a]. The examples in this paper are presented in Ada. Interval versions of the Ada programs were also used to compute guaranteed global bounds for the defect in the solution to ordinary differential equations [Corl91b].

As a further example of natural use of overloaded operators, consider the problem of coding the evaluation of the Taylor series or the Jacobian of the right hand side of the system of differential equations

$$\begin{aligned} y_1' &= y_3 \\ y_2' &= y_4 \\ y_3' &= c_1 y_1 / d \\ y_4' &= c_2 y_2 / d, \text{ where } d = \left(y_1^2 + y_2^2\right)^{3/2}. \end{aligned}$$

As an aside, we remark that it is usually better to solve a higher order system of ODE's in their original form as in [Corl82a], [Chan91a], rather than transforming to a system of first order equations as we have done for the purposes of this example. With appropriate type declarations, the code is

```
function Dist (Y1, Y2 : AD_Type) return AD_Type is
begin
   return Sqrt ((Y1 ** 2 + Y2 ** 2) ** 3);
end Dist;

function RHS (T : AD_Type;
              Y : AD_Vector_Type) return AD_Vector_Type is
   -- T is not used in this example.
   D       : AD_Type;
```

```
   Y_Prime : AD_Vector_Type;
begin
   D := Dist (Y[1], Y[2]);
   Y_Prime[1] := Y[3];
   Y_Prime[2] := Y[4];
   Y_Prime[3] := c_1 * Y[1] / D;
   Y_Prime[4] := c_4 * Y[2] / D;
   return Y_Prime;
end RHS;
```

Listing 3. Overloaded operator code for satellite orbit problem.

We find it convenient to distinguish bewteen three possible strategies for implementing overloaded operators:

1. "In-one-call" operators which generate all requested derivatives in one pass through the computational graph. Intermediate results are discarded. This alternative is convenient to use, but it is inefficient for applications which need to compute the derivatives one order at a time.
2. Sequenced procedure calls (code lists, factorable functions, etc.) which use variables for intermediate results. This alternative is awkward to program, but it can be used to generate additional derivatives efficiently because the intermediate results are saved.
3. Operator overloading with memory management as described in this paper provides the convenience of 1) with the flexibility and speed of 2)

We do not address precompilation, parsing, or methods which otherwise process the function code before automatic differentiation.

2 Specification - Interface. Listing 4 gives a portion of the specification of the user interface for a set of routines comprising a library for automatic differentiation for interval valued Taylor series. An alternative specification appears in [Davi88a].

```
with Interval_Types_Pkg;
use Interval_Types_Pkg;
package Interval_Taylor_Pkg is

                  -- Declarations --

   type Taylor_Type is private;
   type Taylor_Vector_Type
         is array (Integer range <>) of Taylor_Type;

   Max_Index : constant := 100;
   subtype Taylor_Index_Type is Integer range 0 .. Max_Index;

                  -- Operators --

   function "+" (U, V : Taylor_Type) return Taylor_Type;
   function "-" (U, V : Taylor_Type) return Taylor_Type;
   function "*" (U, V : Taylor_Type) return Taylor_Type;
   function Sqr (U    : Taylor_Type) return Taylor_Type;
   function "/" (U, V : Taylor_Type) return Taylor_Type;
   function Exp (U    : Taylor_Type) return Taylor_Type;
   -- Other elementary functions

                  -- Constants --

   function Minus_One return Taylor_Type;
```

```
   function Zero      return Taylor_Type;
   function One       return Taylor_Type;
   function E         return Taylor_Type;
   function Pi        return Taylor_Type;

                  -- Utilities --

   function Taylor (A : Interval_Type) return Taylor_Type;
      -- Convert Interval_Type into series for a constant.
   function Taylor (A   : Interval_Type;
                    Len : Taylor_Index_Type) return Taylor_Type;
      -- Build the series for an independent variable expanded at A
   function Taylor (A_0 : Interval_Vector_Type) return Taylor_Type;
      -- Build the 1-term series for an dependent variable.
   function Deriv (A : Taylor_Type;
                   N : Taylor_Index_Type) return Interval_Type;
      -- Returns the n-th derivative of a series which has
      -- been computed previously.

                  -- Exception --

   Series_Error : exception;

private
   . . .
end Interval_Taylor_Pkg;
```

Listing 4. Partial specifications for an automatic differentiation package.

3 Implementation issues. Arithmetic on Taylor series uses a declaration of the form

```
   type Series_Type is record
      Number_of_Known_Terms       : Taylor_Index_Type;
      Number_of_Significant_Terms : Taylor_Index_Type;
      Series : array (Taylor_Index_Type) of Coefficient_Type;
   end;
```

Listing 5. Series_Type

In Listing 5, `Coefficient_Type` may be real, complex, or interval. Alternatively, an object of `Series_Type` may be a pointer to an object of the type declared above. In either case, each operator has the general form shown in Listing 6.

```
   function "*" (U, V : Taylor_Type) return Taylor_Type is
      Result : Taylor_Type;
   begin
      -- Use recurrence relations to compute Result.
      -- Forward mode is easy.
      -- Reverse mode requires more effort.
      -- Operators may employ fancy memory conservation strategies.
      return Result;
   end "*";
```

Listing 6. General form of an overloaded AD operator.

We remark in passing that there are ample opportunities to exploit parallelism:

- Within operators. Most operators include scalar products and other operations which can be parallelized.

- Between operators. Separate nodes of the computational graph can be scheduled in parallel [Bisc91a].
- Course grained operators. Instead of scheduling individual arithmetic operations, schedule parallel execution at the granularity of matrix and vector operations or quadrature. This is related to Speelpenning's concept of funnel nodes [Spee80a] and Bischof's concept of hoisting nodes [Bisc91a].

This concludes our discussion of the principles of using overloaded operators for automatic differentiation. Next, we turn our attention to issues which become important for time-efficient generation of Taylor series one term at a time. The same issues arise when one generates a function value, and subsequently returns to compute partial derivatives. The reverse mode for generating partial derivatives can address the same issues by using a "tape" [Grie92a], [Grie91f].

4 Efficient one-at-a-time series generation. In some applications such as optimization, rootfinding, or quadrature, we are able to compute the entire truncated Taylor series with one call as shown in Listing 7.

```
function f (T : Taylor_Type) return Taylor_Type is
   Result : Taylor_Type;
begin
   Result := some expression in +, sin, ...
   return Result;
end f;

X, A : Taylor_Type;
. . .
X := Taylor (A => X_0, Len => 20);  -- Taylor series for X
A := f(X);
. . .
```

Listing 7. Generate an entire Taylor series with one call.

In contrast to this "in-one-call" style, many applications require that the Taylor series (or other AD_Type) be generated one term at a time:

- Ordinary differential equations [Corl82a], [Moor66a], [Moor79a], [Stet86a]
- Order adaptive quadrature [Gray75a], [Corl87a]
- Computing a function value followed by the computation of its gradient or Jacobian
- Computing some partials followed by others.

For example, given the function RHS from Listing 3, the series for the solution to the system of ODE's is computed as shown in Listing 8.

```
T : Taylor_Type;
X, X_Prime : Taylor_Vector_Type;
. . .
T := Taylor (T_0, 1);  -- One term series for t
X := Taylor (X_0);     -- Vector of initial conditions
for i in 1 .. Max_Index loop
    X_Prime := RHS (T, X);
    X := Integrate (X_0, X_Prime);
end loop;
Extract_Order_and_Location_of_Singularities (X, ...);
```

Listing 8. Taylor series solution for ODE's.

The reason why the generation of a solution to an ODE must be done one term at a time is that the solution to an ODE is given by

$$x(t) = x_0 + \int_{t_0}^{t} x'(t)dt,$$

which is not a factorable function.

The function `Integrate (X_0, X_Prime)` called in Listing 8 computes the series for

$$x(t) = x_0 + \int_{t_0}^{t} x'(t)\,dt$$

by term-by-term integration of the series for $x'(t)$.

As an aside, we remark on the procedure `Extract_Order_and_Location_of_Singularities` in Listing 8. One of the attractive features of Taylor series methods for solving ODE's is their ability to provide analytic information about the solution such as the order and location of singularities [Chan67a], [Corl82a], [Chan81a], information which is often physically significant. Chang [Chan91a] has also used analytic information to enable ATOMFT, his ODE solver, to take integration steps which are nearly as large as the radius of convergence of the solution of the differential equation.

If the automatic differentiation arithmetic operators are programmed as outlined in the preceding sections, then the cost of generating n terms of the series for $x(t)$ by Listing 8 is $O(n^3)$, instead of the $O(n^2)$ achieved by the in-one-call generation. To achieve a complexity of $O(n^2)$ requires

- Avoid recomputing previously computed results
- Saving all temporary results
- Comparing `Result.Number_of_Known_Terms` with `U` and `V.Number_of_Known_Terms`
- That each operator function must know its result
- That each operator be a ternary, not a binary, operator.

Hence, overloaded binary operators as outlined above are not sufficient. Our operators must provide memory management.

All intermediate results must be saved. For example, suppose `function RHS` contains the code

```
for i in 0 .. N loop
   A := A * X + B(i);
end loop;
```

Then each instance of `A` must be stored in a different location in order that the next time through the code, the original partial results are available. Hence, in order to implement $O(n^2)$ operators, we must perform run-time loop unrolling in space, as well as the run-time loop unrolling to form the computational graph.

Key insight:
In computing each of $x(t), x'(t), x''(t), \ldots$, the order of operations is the same.

That is, the computational graph is traversed in the same order for each derivative, although the path may differ at different points t. Hence, the order in which memory locations are allocated for storing results is the same. In computing $x(t)$, we allocate space for the results of each operation. When we compute $x'(t), x''(t), \ldots$ at the same point, we repeat the same allocations.

A memory `Allocator` maintains a sequential array

```
type Taylor_Type is range 0 .. Memory_Size;
Memory : array (Taylor_Type) of Series_Type;
```

where `Series_Type` is defined in Listing 5. Hence, an object of `Taylor_Type` is actually an integer index giving the array location at which the series is stored. The memory `Allocator` function returns the index of the next available location in the array `Memory`. That is why Listings 7 and 8 use the identifier `Taylor_Type` rather than `Series_Type` defined in Listing 5. The form of each operator is then shown in Listing 9.

```
function "*" (U, V : AD_Type) return AD_Type is
   Result : AD_Type;
begin
   Result := Allocator;
   -- Previous result, if any, should still be in Memory(Result).

   for Term_Index in Memory(Result).Number_of_Known_Terms ..
                     minimum (Memory(U).Number_of_Known_Terms,
                              Memory(V).Number_of_Known_Terms) loop
      -- Use recurrence relations to compute
      --   Memory(Result).Series(Term_Index)
   end loop;
   return Result;
end "*";
```

Listing 9. General form of an overloaded AD operator with memory management.

With operators of the form given in Listing 9, n terms of the series for the solution can be generated one term at a time as shown in Listing 8 with a cost $O(n^2)$. In order to support the $O(n^2)$ operators, we need to add two additional procedures to the specifications given in Listing 4.

```
procedure Start_New_F;
   -- No terms are known.  Initialize automatic differentiation
   -- memory structure before each new function is evaluated.
   -- Set the memory allocation pointer to the top of the
   -- internally managed memory array.
procedure Start_Same_F;
   -- We want to traverse the dependency graph again in the
   -- same way.  Initialize automatic differentiation memory
   -- structure before each evaluation of the same function.
   -- Set the memory allocation pointer to the top of the
   -- internally managed memory array, but leave the partial
   -- results already stored there unchanged.
```

Listing 10. Additional procedures for memory management.

With the procedures specified in Listing 10, the program for computing the series terms from Listing 8 becomes

```
T : Taylor_Type;
X, X_Prime : Taylor_Vector_Type;
. . .
T := Taylor (T_0, 1);  -- One term series for t
X := Taylor (X_0);     -- Vector of initial conditions
Start_New_F;
for i in 1 .. Max_Terms loop
    X_Prime := RHS (T, X);
    Integrate (X_0, X_Prime, X);
    Start_Same_F;
end loop;
Extract_Order_and_Location_of_Singularities (X, ...);
```

Listing 11. Taylor series solution for ODE's with cost $O(n^2)$.

The memory management described in this section *cannot* be done using dynamic memory allocation provided by the programming language because it is essential that the same sequence of allocation calls return exactly the same sequence of memory locations. It would be possible to use a dynamically allocated linked list structure to store the Taylor series for the intermediate results as they are initially generated. The difficulty lies in using information stored in one or two operands to locate the area where the corresponding intermediate result is stored during subsequent passes through the computational graph. For example in the expression `(A - B ) + (A * B)`, the operators `-` and `*` both are passed the same operands `A` and `B`, but their intermediate results must be stored in different locations. It would be possible to address this concern by storing the computational graph in some form. However, it is much simpler and faster to observe that each pass through the computational graph to generate each additional derivative must visit the intermediate results in the same order. Hence, sequential initial allocation and sequential re-visiting works. The principle is the same as in Griewank's "tape" (see the next section).

5 Relationship to Griewank's "tape". Griewank [Grie91f], [Grie92a] has used the concept of a tape for out of core storage of the code list and partial results. A tape is essential for large problems whose storage requirements for both the computational graph and the derivatives being computed exceed the storage capacity of any computer. The tape is created by overloaded operators as a byproduct of the evaluation of f. Subsequently, higher order ordinary or partial derivatives are evaluated by a sequential processing of the tape. Christianson [Chri91a] uses overloaded operators in a similar way.

The memory management reported here stores in core partial results in a manner equivalent to the tape. In this design, subsequent derivatives are evaluated by repeated execution of the same program which created the tape, rather than by a second program whose function is to process the tape.

6 Trade-offs. The framework presented here for automatic differentiation operators represents a classical space vs. time vs. ease of use trade-off.

1. Space efficient operators:
   - Overloaded operators are easy to use.
   - Relatively simple to program operators.
   - Storage proportional to largest "live set" size [Grie91f].
   - Run time for generating series in one call is $O(n^2)$.
   - Run time for generating series one term at a time is $O(n^3)$.
2. Time efficient operators:
   - Overloaded operators are easy to use.
   - Relatively complicated to program operators.
   - Storage proportional to run time to evaluate RHS.
   - Run time for generating series is always $O(n^2)$
3. Three operand operators: `procedure Add (A, B, C)`
   - Infix notation is replaced by procedure calls.
   - Awkward to use.
   - Provision of storage for all results becomes the users' problem.
   - Procedures are relatively simple to program.
   - Run time for generating series may be $O(n^2)$ or $O(n^3)$, depending on how the user assigns variables.

7 Code availability. I have suppressed some important implementation details for the sake of clarity of exposition. An automatic differentiation package for interval valued Taylor series written in the Ada programming language is available from the author.

References

[AdaLRM] ADA JOINT PROGRAM OFFICE, *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A*, Washington, D.C., 1983.

[Beck88a] R. A. BECKER, J. M. CHAMBERS, AND A. R. WILKS, *The New S Language*, Computer Science Series, Wadsworth and Brooks/Cole, Pacific Grove, Calif., 1988.

[Bisc91a] C. BISCHOF, *Issues in parallel automatic differentiation*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 100–113.

[Chan67a] Y. F. CHANG, *The conduction-diffusion theory of semiconductor junctions*, J. Applied Physics, 38 (1967), pp. 534–544.

[Chan91a] ——, *A variable-order (10 - ∞) Taylor series method for solving ODE's with integration steps almost equal to the radii of convergence.* Poster presented at SIAM Workshop on Automatic Differentiation of Algorithms, Breckenridge, Colo., January 1991.

[Chan81a] Y. F. CHANG, M. TABOR, J. WEISS, AND G. F. CORLISS, *On the analytic structure of the Henon-Heiles system*, Physics Letters, 85A (1981), pp. 211–213.

[Char88a] B. W. CHAR, K. O. GEDDES, G. H. GONNET, M. B. MONAGAN, AND S. M. WATT, *MAPLE Reference Manual*, Watcom Publications, Waterloo, Ontario, Canada, 1988.

[Chri91a] D. B. CHRISTIANSON, *Automatic Hessians by reverse accumulation in Ada*, IMA J. on Numerical Analysis, (1991). Presented at SIAM Workshop on Automatic Differentiation of Algorithms, Breckenridge, Colo., January 1991.

[Corl91a] G. F. CORLISS, *Overloading point and interval Taylor operators*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 139–146.

[Corl82a] G. F. CORLISS AND Y. F. CHANG, *Solving ordinary differential equations using Taylor series*, ACM Trans. Math. Software, 8 (1982), pp. 114–144.

[Corl87a] G. F. CORLISS AND L. B. RALL, *Adaptive, self-validating quadrature*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. 831–847.

[Corl91b] ——, *Computing the range of derivatives*, in Computer Arithmetic, Scientific Computation, and Mathematical Modelling, E. Kaucher, S. M. Markov, and G. Mayer, eds., vol. 12 of IMACS Annals on Computing and Applied Mathematics, J. C. Baltzer AG, Basel, 1991, pp. 195–212.

[Crar79a] F. D. CRARY, *A versatile precompiler for nonstandard arithmetics*, ACM Trans. Math. Software, 5 (1979), pp. 204–217.

[Davi88a] P. H. DAVIS, G. F. CORLISS, AND G. S. KRENZ, *A bibliography on methods and techniques in differentiation arithmetic*, Technical Report AM–88–09, School of Mathematics, University of Bristol, Bristol, U.K., 1988.

[Gray75a] J. H. GRAY AND L. B. RALL, *INTE: A UNIVAC 1108/1110 program for numerical integration with rigorous error estimation*, MRC Technical Summary Report No. 1428, Mathematics Research Center, University of Wisconsin - Madison, 1975.

[Grie89a] A. GRIEWANK, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108. Also appeared as Preprint MCS–P10–1088, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., October 1988.

[Grie92a] A. GRIEWANK, D. JUEDES, J. SRINIVASAN, AND C. TYNER, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, (to appear). Also appeared as Preprint MCS–P180–1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., November

1990.

[Grie91a] A. GRIEWANK AND S. REESE, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, Penn., 1991, pp. 126–135. Also appeared as Preprint MCS–P267–1091, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1992.

[Hill85a] K. E. HILLSTROM, *Users guide for JAKEF*, Technical Memorandum ANL/MCS–TM–16, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1985.

[Horw88a] J. E. HORWEDEL, B. A. WORLEY, E. M. OBLOW, AND F. G. PIN, *GRESS version 1.0 users manual*, Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, Tenn., 1988.

[IBM90a] IBM, *IBM High Accuracy Arithmetic - Extended Scientific Computation*, IBM, Mechanicsburg, Penn., 1990. IBM Publication No. SC33–6462–00.

[Irim84a] M. IRI, *Simultanious computation of functions, partial derivatives and estimates of rounding errors — Complexity and practicality*, Japan J. Applied Mathematics, 1 (1984), pp. 223–252.

[Irim88a] M. IRI, T. TSUCHIYA, AND M. HOSHI, *Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations*, J. Computational and Applied Mathematics, 24 (1988), pp. 365–392. Original Japanese version appeared in *J. Information Processing*, 26 (1985), pp. 1411–1420.

[Klat91a] R. KLATTE, U. KULISCH, M. NEAGA, D. RATZ, AND C. ULLRICH, *PASCAL-XSC: A PASCAL Extension for Scientific Computation*, Springer Verlag, Berlin, 1991.

[Moor66a] R. E. MOORE, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[Moor79a] ——, *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, Penn., 1979.

[Rall81a] L. B. RALL, *Automatic Differentiation: Techniques and Applications*, vol. 120 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 1981.

[Spee80a] B. SPEELPENNING, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[Step90a] B. R. STEPHENS AND J. D. PRYCE, *The DAPRE/UNIX Preprocessor Users' Guide v1.2*, Royal Military College of Science at Shrivenham, Shrivenham, U.K., 1990.

[Stet86a] H. J. STETTER, *Algorithms for the inclusion of solutions of ordinary initial value problems*, in Equadiff 6: Proceedings of the International Conference on Differential Equations and Their Applications (Brno, 1985), J. Vosmanský and M. Zlámal, eds., vol. 1192 of Lecture Notes in Mathematics, Springer Verlag, Berlin, 1986, pp. 85–94.

[Stro86a] B. STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.