ADOL-C: [1]

A Package for the Automatic Differentiation of Algorithms Written in C/C++

Version 1.5, December 1993

Andreas Griewank[2]
David Juedes[3]
Jean Utke[4]

Abstract

The C++ package ADOL-C described here facilitates the evaluation of first and higher derivatives of vector functions that are defined by computer programs written in C or C++. The resulting derivative evaluation routines may be called from C/C++, Fortran, or any other language that can be linked with C.

The numerical values of derivative vectors are obtained free of truncation errors at a small multiple of the run time and randomly accessed memory of the given function evaluation program. Derivative matrices are obtained by columns or rows. For solution curves defined by ordinary differential equations, special routines are provided that evaluate the Taylor coefficient vectors and their Jacobians with respect to the current state vector. The derivative calculations involve a possibly substantial (but always a priori predictable) amount of data that are accessed strictly sequentially and are therefore automatically paged out to files on external mass storage devices.

Keywords: Automatic Differentiation, Chain Rule, Overloading, Taylor Coefficients, Gradients, Hessians, Reverse Propagation

Abbreviated title: Automatic differentiation by overloading in C++

1 Introduction: Differentiation of Algorithms

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions with m components in n real or complex variables. This requirement arises particularly in optimization, nonlinear equation solving, numerical studies of bifurcation, and the solution of nonlinear differential or integral equations. Often these functions are defined by sequential evaluation procedures involving many intermediate variables. By eliminating

[1]This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by funding from DOE's Office of Energy Research Laboratory Technology Transfer.

[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, and Institut für Wissenschaftliches Rechnen, Technical University Dresden, D-01062 Germany

[3]Department of Computer Science, Iowa State University, Ames, IA 50010

[4]Institut für Wissenschaftliches Rechnen, Technical University Dresden, D-01062 Germany

the intermediate variables symbolically, it is theoretically always possible to express the m dependent variables directly in terms of the n independent variables. Typically, however, the attempt results in unwieldy algebraic formula, if it can be completed at all. Symbolic differentiation of the resulting formulae will usually exacerbate this problem of *expression swell* and often entails the repeated evaluation of common expressions.

An obvious way to avoid such redundant calculations is to apply an optimizing compiler to the source code that can be generated from the symbolic representation of the derivatives in question. Exactly this approach was investigated by Bert Speelpenning, a student of Bill Gear, during his Ph.D. research [18] at the University of Illinois from 1977 to 1980. Eventually he realized that at least in the cases $n = 1$ and $m = 1$, the most efficient code for the evaluation of derivatives can be obtained directly from that for the evaluation of the underlying vector function. In other words, he advocated the differentiation of evaluation algorithms rather than formulae. In his thesis he made the particularly striking observation that the gradient of a scalar-valued function (i.e., $m = 1$) can always be obtained for no more than five times the operations count of evaluating the function itself. This bound is completely independent of n, the number of variables, and allows the row-wise computation of Jacobians for at most $5\,m$ times the effort of evaluating the underlying vector function.

When m, the number of component functions, is larger than n, Jacobians can be obtained more cheaply column by column through propagating gradients forward. This classical technique of automatic differentiation goes back at least to Wengert [20] and was later popularized by Rall [16]. It was noted in [7] that in general neither the row-by-row nor the column-by-column method is optimal for the calculation of Jacobians. The potentially more efficient alternatives, however, require some combinatorial optimization and involve large data structures that are not necessarily accessed sequentially [9]. Therefore, the package ADOL-C described here was written primarily for the evaluation of derivative vectors (e.g., rows or columns of Jacobians). This approach also simplifies parameter passing between subroutines and calls in different computer languages.

The *reverse propagation of gradients* employed by Speelpenning is closely related to the *adjoint sensitivity analysis* for differential equations, which has been used at least since the late sixties, especially in nuclear engineering [4],[5], weather forecasting [19], and neural networks [21]. The discrete analog used here was apparently first discovered in the early seventies by Ostrovskii et al. [15] and Linnainmaa [14] in the context of rounding error estimates. Since then, there have been numerous rediscoveries and various software implementations. Speelpenning himself wrote a Fortran precompiler called JAKE, which was upgraded at Argonne National Laboratory to JAKEF. Currently, there exist at least three other precompilers for automatic differentiation, namely, GRESS/ADGEN [11],[5] PADRE2 [13],[6] and ADIFOR [1]. [7]

[5]Contact: Jim Horwedel, ORNL, P.O. Box X, Oak Ridge, TN 37831, e-mail: jqh%ornlstc.bitnet
[6]Contact: K. Kubota, Keio Univ., 3-14-1 Hiyoshi, Yokohama 223, Japan, e-mail: kubota@ae.keio.ac.jp
[7]Contact: Ch. Bischof, ANL-MCS, Argonne IL 60439-4844, e-mail: bischof@mcs.anl.gov

Following the work of Kedem [12] with the Fortran preprocessor AUGMENT, Rall [17] implemented in 1983 the *forward propagation of gradients* by overloading in PASCAL-SC. In contrast to precompilation, overloading requires only minor modifications of the user's evaluation program and does not generate intermediate source code. Our package ADOL-C utilizes overloading in C++, but the user has to know only C. The acronym stands for **A**utomatic **D**ifferentiation by **O**ver**L**oading in **C++**. ADOL-C facilitates the simultaneous evaluation of arbitrarily high directional derivatives and the gradients of these Taylor coefficients with respect to all independent variables. Relative to the cost of evaluating the underlying function, the cost for evaluating any such scalar-vector pair grows as the square of the degree of the derivative but is still completely independent of the numbers m and n.

For the reverse propagation of derivatives, the whole execution trace of the original evaluation program must be recorded, unless it is recalculated in pieces as advocated in [8]. In ADOL-C, this potentially very large data set is written first into a buffer array and later into a file if the buffer is full or if the user wishes a permanent record of the execution trace. In either case, we will refer to the recorded data as the *tape*. The user may create several tapes in several named arrays or files. During subsequent derivative evaluations, tapes are always accessed strictly sequentially, so that they can be paged in and out to disk without significant runtime penalties. If written into a file, the tapes are self-contained and can be used by other Fortran, C or C++ programs.

This paper is organized as follows. Section 2 explains the modifications required to convert undifferentiated code to code that compiles with ADOL-C. For better efficiency and programming convenience one may employ the vector and matrix classes described in Section 3. Section 4 covers aspects of the tape of recorded data that ADOL-C uses to evaluate arbitrarily high order derivatives. The discussion includes storage requirements and the tailoring of certain tape characteristics to fit specific user needs. Section 5 offers a more mathematical characterization of ADOL-C as well as descriptions of the calling sequences of various derivative evaluation routines. Section 6 details the installation and use of the ADOL-C package. It also provides a description of each of the C and C++ modules that the package comprises. Finally, Section 7 furnishes four example programs that incorporate the ADOL-C package to evaluate first and higher-order derivatives. These and other examples are distributed with the ADOL-C source; simply refer to them when the more abstract and general descriptions of ADOL-C provided in this document do not suffice.

2 Preparing a Section of C or C++ Code for Differentiation

ADOL-C was designed so that the user has to make only minimal changes to his undifferentiated code. The main modifications concern variable declarations and input/output operations.

2.1 Declaring Active Variables

The key ingredient of automatic differentiation by overloading is the concept of an *active variable*. All variables that may at some time during the program execution be considered differentiable quantities must be declared to be of an active type. ADOL-C uses one active scalar type, called **adouble**, whose real part is of the standard type **double**. There are corresponding types **adoublev** and **adoublem** of vectors and matrices, whose components function like **adouble**s. Typically, one will declare the independent variables and all quantities that directly or indirectly depend on them as *active*. Other variables that do not depend on the independent variables but enter, for example, as parameters, may remain one of the *passive* types **double, float**, or **int**. There is no implicit type conversion from **adouble** to any of these passive types; thus, *failure to declare variables as active when they depend on other active variables will result in a compile-time error message.* In data flow terminology, the set of active variable names must contain all its successors in the dependency graph. All components of indexed arrays must have the same activity status.

The real component of an **adouble x** can be extracted as **value(x)**. In particular, such explicit conversions are needed for the standard output procedure **printf**. The output stream operator << is overloaded such that first the real part of an **adouble** and then the string "(a)" is added to the stream. Naturally, **adouble**s may be components of vectors, matrices, and other arrays, as well as members of structures or classes. For regular arrays it may be more efficient to use the vector and matrix classes discussed in Section 3.

The C++ class **adouble**, its member functions, and the overloaded versions of all arithmetic operations, comparison operators, and ANSI C functions are contained in the file **adouble.c** and its header **adouble.h**. The latter must be included for compilation of all program files containing **adouble**s and corresponding operations.

2.2 Marking Active Sections

All calculations involving active variables that occur between the void function calls

trace_on(tag,keep) and **trace_off(file)**

are recorded on a sequential data set called *tape*. Pairs of these function calls can appear anywhere in a C++ program, but they may not overlap. The nonnegative integer argument **tag** identifies the particular tape for subsequent function or derivative evaluations. Unless several tapes need to be kept, **tag** = 0 may be used throughout. The optional integer arguments **keep** and **file** will be discussed in Section 4. We will refer to the sequence of statements executed between a particular call to **trace_on** and the following call to **trace_off** as an *active section* of the code. The same active section may be entered repeatedly, and one can successively generate several traces on distinct tapes by changing the tag.

Active sections may contain nested or even recursive calls to functions provided by the user. Naturally, their formal and actual parameters must have matching types. In particular, the functions must be compiled with their active variables declared as **adouble**s and with the header file **adouble.h** included. Variables of type **adouble** may be declared outside an active section and need not go out of scope before the end of an active section. It is not necessary—though desirable—that free-store **adouble**s allocated within an active section be deleted before its completion. The values of all **adouble**s that exist at the beginning and end of an active section are recorded by **trace_on** and **trace_off**, respectively.

2.3 Selecting Independent and Dependent Variables

One or more active variables that are read in or initialized to the values of constants or passive variables must be distinguished as independent variables. Other active variables that are similarly initialized may be considered as temporaries (e.g., a variable that accumulates the partial sums of a scalar product after being initialized to zero). In order to distinguish an active variable x as independent, ADOL-C requires an assignment of the form

$$\mathbf{x} \ll= \mathbf{px} \quad // \ \mathbf{px} \text{ of any passive numeric type}$$

This special initialization ensures that **value(x)=px**, and it should precede any other assignment to **x**. However, **x** may be reassigned other values subsequently. Similarly, one or more active variables **y** must be distinguished as dependent by an assignment of the form

$$\mathbf{y} \gg= \mathbf{py} \quad // \ \mathbf{py} \text{ of any passive type}$$

which ensures that **py=value(y)** and should not be succeeded by any other assignment to **y**. However, a dependent variable **y** may have been assigned other real values previously, and it could even be an independent variable as well. The derivative values calculated after the completion of an active section always represent **derivatives of the final values of the dependent variables with respect to the initial values of the independent variables.**

The order in which the independent and dependent variables are marked by the $\ll=$ and $\gg=$ statements matters crucially for the subsequent derivative evaluations. However, these variables do not have to be combined into continuous vectors. ADOL-C counts the number of independent and dependent variable specifications within each active section and records them in the header of the tape.

2.4 A Subprogram as an Active Section

As a generic example let us consider a C(++) function of the form shown in Figure 1.

```
void eval(int n, int m,          // number of independents and dependents
          double *x,             // independent variable vector
          double *y,             // dependent variable vector
          int *k,                // integer parameters
          double *z)             // real parameters
{                                // beginning of function body
    double t = 0;                // local variable declaration
    for( int i=0; i < n; i++)    // begin crunch
       t += z[i]*x[i];           // continue crunch
    ............                 // continue crunch
    ............                 // continue crunch
    y[m-1] = t/m;                // end crunch
}                                // end of function
```

Figure 1: Generic Example of an Active Subprogram

If **eval** is to be called from within an active C(++) section with **x** and **y** as vectors of **adouble**s and the other parameters passive, then one merely has to change the type declarations of all variables that depend on **x** from **double** or **float** to **adouble**. Subsequently, the subprogram must be compiled with the header file **adouble.h** included as described in Section 4. Now let us consider the situation when **eval** is still to be called with integer and real arguments, possibly from a program written in Fortran 77, which does not allow overloading.

To automatically compute derivatives of the dependent variables **y** with respect to the independent variables **x**, we can make the body of the function into an active section. For example, we may modify the previous program segment as in Figure 2. The renaming and doubling up of the original independent and dependent variable vectors by active counterparts may seem at first a bit clumsy. However, this transformation has the advantage that the calling sequence and the "crunchy" part of **eval** remain completely unaltered. If the temporary variable **t** had remained a **double**, the code would not compile, because of a type conflict in the assignment following the declaration. Four more detailed example codes are listed in Section 7.

2.5 Overloaded Operators and Functions

As in the subprogram discussed above, the actual computational statements of a C(++) code need not be altered for the purposes of automatic differentiation. All arithmetic operations, as well as the comparison and assignment operators, are overloaded if at least

```
void eval( int n,m,                 // number of independents and dependents
           double *px,              // independent passive variable vector
           double *py,              // dependent passive variable vector
           int *k,                  // integer parameters
           double *z)               // parameter vector
{                                   // beginning of function body
  short int tag = 0;                // tape array and/or tape file specifier
  trace_on(tag);                    // start tracing
  adouble *x, *y;                   // declare active variable pointers
  x=new adouble[n];                 // declare active independent variables
  y=new adouble[m];                 // declare active dependent variables
  for( int i=0; i < n; i++)
     x[i] <<= px[i];                // distinguish independent variables
  adouble t;                        // local variable declaration
  for( int i=0; i < n; i++)         // begin crunch
    t += z[i]*x[i];                 // continue crunch
  ............                      // continue crunch
  ............                      // continue crunch
  y[m-1] = t/m;                     // end crunch as before
  for( int j=0; j<m;j++)
    y[j] >>= py[j];                 // select dependent variables
  delete[] y;                       // destruct dependent active variables
  delete[] x;                       // destruct independent active variables
  trace_off();                      // complete tape
}                                   // end of function
```

Figure 2: The ADOL-C version of the code listed in Fig. 1

one of their arguments is an active variable. An **adouble x** occurring in a comparison operator is effectively replaced by its real value **value(x)**. Most functions contained in the ANSI C standard for the math library are overloaded for active arguments. The only exceptions are the nondifferentiable functions **fmod**, and **modf**. Otherwise, legitimate C code in active sections can remain completely unchanged, provided the direct output of active variables is avoided. Whenever derivatives are undefined or discontinuous, the derivatives values are assigned to be the limit of the derivative values at arguments to the right of the exceptional point. For example, at $\mathbf{x} = 0$ the first derivatives of the square root function **sqrt(x)** and the absolute value function **fabs(x)** are set to +1.0/0.0 and +1.0, respectively. The general power function $\mathbf{pow}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^{\mathbf{y}}$ is computed whenever it is defined for the corresponding double arguments. If the basis is negative, however, the partial with respect to an integral exponent is set to zero. Similarly, the partial of **pow** with respect to both arguments is set to zero at the origin, where both arguments vanish. The derivatives of the step functions **floor, ceil, frexp**, and **ldexp** are set to zero at all arguments **x**. Some C implementations supply other special functions, in particular the error function **erf(x)**. For the latter, we have included an **adouble** version in **adouble.c**, which has been commented out for systems on which the double valued version is not available. The increment and decrement operators ++, −− (pre- and postfix) are available for **adouble**s and also the active subscripts described in the appendix. Ambiguous statements like `a+=a++;` must be avoided because the compiler may sequence the evaluation of the overloaded expression differently form the original in terms of **double**s.

As we have indicated above, all subroutines called with active arguments must be modified or suitably overloaded. The simplest procedure is to declare the local variables of the function as active so that their internal calculations are also recorded on the tape. Unfortunately, this approach is likely to be unnecessarily inefficient and inaccurate if the original subroutine evaluates a special function that is defined as the solution of a particular mathematical problem. The most important examples are implicit functions, quadratures, and solutions of ordinary differential equations. Often the numerical methods for evaluating such special functions are elaborate, and their internal workings are not at all differentiable in the data. Rather than differentiating through such an adaptive procedure, one can obtain first and higher derivatives directly from the mathematical definition of the special function. Currently this direct approach has been implemented only for user-supplied quadratures as described in Subsection 6.2.

2.6 Step-by-Step Modification Procedure

To prepare a section of given C or C++ code for automatic differentiation as described above, one applies the following step-by-step procedure.

1. Use the statements **trace_on(tag)** [or **trace_on(tag,keep)**] and **trace_off()** [or **trace_off(file)**] to mark the beginning and the end of the active section.

2. Select the set of active variables, and change their type from **double** or **float** to **adouble** (or the array types **adoublev** and **adoublem** discussed in the next section).
3. Select a sequence of independent variables, and initialize them with $\ll=$ assignments from passive variables (or vectors).
4. Select a sequence of dependent variables among the active variables, and pass their final values to passive variable (or vectors thereof) by $\gg=$ assignments.
5. Compile the codes after including the header file **adouble.h**.

Typically, the first compilation will detect several type conflicts — usually attempts to convert from active to passive variables or to perform standard I/O of active variables. (Some C++ compilers may also disallow certain **goto**s which are legal in ANSI C, but this problem is unrelated to ADOL-C.) Since all standard C programs can be activated by a mechanical application of the procedure above, the following section is of importance only to advanced users.

3 Active and Passive Arrays and Structures

Some or all real fields of structures or members of classes may be redeclared as **adouble**s so that the differentiation can proceed by nested overloading without the explicit unroling of composite structures and operations. In this way, one may *activate* standard vector and matrix classes for numerical calculations. However, the individual activation of real fields or members may entail a significant overhead, which can be avoided by using the following active classes, if the scalars in question are arranged in regular arrays.

3.1 Active and Passive Vector Classes

To reduce the overhead in dealing with individual scalar variables and their operations, we have introduced a class of active vectors called **adoublev**s and a corresponding class of passive vectors called **doublev**s. Vectors **a** and **b** of **p** active and passive components, respectively, are declared by the statement

adoublev a(p) , doublev b(p) ,

where **p** can be an integer variable. Like all local variables, vectors are destructed when they go out of scope at the end of the block in which they were declared. Nevertheless, since their length is computed only at run time, the vector classes can often be used in lieu of dynamically allocated arrays, which can drastically increase the storage requirement of ADOL-C as discussed below. Vector elements of the form $a[i]$ or $b[i]$ can take the place of any scalar variable of type **adouble** or **double**, respectively.

3.2 Overloaded Vector Operations

Provided their lengths are compatible, vectors can be added or subtracted, yielding a third vector of the same length. Similarly, they can be multiplied by active or passive scalars, whereas the product of two vectors is a scalar, which must be active if one of the factors is. Moreover, the special operators <<= and >>= are also overloaded so that active vectors can be marked as independent or dependent, respectively. As in the scalar versions, the statement $a <<= b$ simultaneously initializes the active vector a with the values in the passive vector b, and the statement $a >>= b$ passes the values in the active vector a to the passive vector b. In the example code above, one could therefore have replaced the first loop in Fig. 2 by **x** <<= **xp**, the middle loop by **t** = **z*x**, and the last loop by **y** >>= **yp**. The use of vector operations can reduce the length of the tape and the run time of the code significantly. This advantage applies in particular for linear algebra operations, where large amounts of unnecessary intermediates are kept track of in the scalar mode.

The following binary operations are defined between active or passive vectors:

$$+ \quad , \quad - \quad , \quad *$$

where $*$ denotes the dot product. Its result is an **adouble** if at least one of the argument vectors is active, and otherwise it is a **doublev**. The same rule applies to addition and subtraction whose result is an **adoublev** or **doublev**. The assignments

$$= \quad , \quad -= \quad , \quad += \quad , \quad <<= \quad , \quad >>=$$

may also be considered as binary operations between vectors. For the first three assignments the left side must be active if the right side is. For the last two, the left side must be active and the right side passive. The binary operations

$$* \quad , \quad / \quad , \quad *= \quad , \quad /=$$

are also defined when the left argument is a vector and the right argument is a scalar. If the scalar is an **adouble**, the vector must be a **adoublev**. Note that here $*$ does not represent the dot product. Mathematically meaningless operations between vectors and scalars will produce a compiler error message or in some cases (e.g., scalar added to a vector) the run-time error message `Error in vector-oper.!` None of the operations listed above are currently defined for the active and passive matrix types described below.

3.3 Active and Passive Matrix Classes

The matrix types **adoublem** and **doublem** are used only to facilitate the automatic and contiguous allocation (and deallocation) of arrays whose elements are **adoublev**s or **doublev**s, respectively. The statements

adoublem A(q,p) , doublem B(q,p)

allocate $q \times p$ matrices of **adouble**s and **double**s, respectively. The q subscripted quantities $A[i]$ and $B[i]$ represent **adoublev**s and **doublev**s of size p, respectively. Fortran-like access by columns is not possible. As an example, consider the multiplication of a $q \times p$ matrix A by a vector b as shown in Figure 3. If one wishes to multiply A by a $p \times s$ matrix B instead of the vector b, one might use the code in Figure 4, where we have omitted initializations and independent/dependent selections. Even if no vector operations are performed, the use of the active vector and matrix types in declarations is much preferable to the declaration of **adouble** arrays, which should be avoided, especially in dynamic storage mode. The reasons for this preference are explained in the following subsection, which can be skipped unless the reader wishes to obtain some basic understanding of how the package works internally and to tailor the package to his needs.

```
doublev bp(p);
doublem A(q,p);
for( int i=0; i < p; i++)
{
      cin >> bp[i];                         // Read in values
      for( int j=0; j < q; j++)
          cin >> A[i][j];
}
adoublev b(p);
b <<= bp;                                   // Mark b as independent vector
adoublev c(q);
for( int i=0; i < q; i++)
      c[i] = A[i]*b;                        // dot product
doublev cp(q);
c >>= cp;                                   // Mark c as dependent vector
for( int i=0; i < p; i++)
      cout << cp[i];                        // Output results
```

Figure 3: Matrix-Vector Multiplication using ADOL-C Arrays

The classes **doublev** and **doublem** have been defined with implicit conversions to the types **double*** and **double****, respectively. Consequently, these vectors and matrices can be used as actual parameters in procedures, whose formal parameters are standard pointers to **double**s. Moreover, the vector-vector and vector-scalar operations listed above have also been overloaded for cases where at most one of the vector arguments is simply an **adouble*** or a **double***. Naturally, the lack of size information precludes any format compatibility

```
doublem A(q,p);
adoublem B(p,s);
adoublem C(q,s);
initializations .....
for( int i=0; i < q; i++)
{
    C[i] = 0 ;                      // Set to zero
    for( int j=0; j < p; j++)
       C[i] += A[i][j]*B[j];        // SAXPY
}
```

Figure 4: Matrix-Matrix Multiplication Using ADOL-C Arrays

checks in these cases.

3.4 Warnings and Suggestions for Improved Efficiency

Since the type **adouble** has a nontrivial constructor, the mere declaration of large **adouble** arrays may take up considerable run time. The user should be warned against the usual Fortran practice of declaring fixed-size arrays that can accommodate the largest possible case of an evaluation program with variable dimensions. If such programs are converted to or written in C, the overloading in combination with ADOL-C will lead to very large run-time increases for comparatively small values of the problem dimension, because the actual computation is completely dominated by the construction of the large **adouble** arrays. The user is advised to either use the vector and matrix types in automatic storage mode or create dynamic arrays of **adoubles** by using the C++ operator **new** and to destroy them using **delete**. For storage efficiency it is desirable that dynamic objects are created and destroyed in a last-in-first-out (LIFO) fashion. **DO NOT use malloc() and related C memory-allocating functions when declaring adoubles (see following paragraph).**

Whenever an **adouble** is declared, the constructor for the type **adouble** assigns it a nominal address, which we will refer to as its *location*. The location is of the type **locint** defined in the header file **usrparms.h**. Active vectors occupy a range of contiguous locations. As long as the program execution never involves more than 64,535 active variables, the type **locint** may be defined as **unsigned short int**. Otherwise, the range may be extended by defining **locint** as **(unsigned) int** or **(unsigned) long**, which may nearly double the overall mass storage requirement. Sometimes one can avoid exceeding the range of **unsigned shorts** by using more local variables and deleting **adouble**s or **adoublev**s created by the new operator in a last-in-first-out fashion. When memory for **adouble**s is requested through a call to **malloc()** or other related C memory-allocating functions, the storage for these **adouble**s is allocated; however, the C++ **adouble** constructor is never called. The newly defined **adouble**s are never assigned a location and are not counted in the stack of live variables. Thus, any results depending upon these pseudo-**adouble**s will be incorrect. The same point applies, of course, for active vectors. When an **adouble** or **adoublev** goes out of scope or is explicitly deleted, the destructor notices that its location(s) may be freed for subsequent (nominal) reallocation. In general, this is not done immediately but is delayed until the locations to be deallocated form a contiguous tail of all locations currently being used. At this time, a multiple death notice is recorded on the tape.

As a consequence of this allocation scheme, the currently alive **adouble** locations always form a contiguous range of integers that grows and shrinks like a stack. Newly declared **adouble**s are placed on the top so that vectors of **adouble**s obtain a contiguous range of locations. While the C++ compiler can be expected to construct and destruct automatic variables in a last-in-first-out fashion, the user may upset this desirable pattern by deleting free-store **adouble**s too early or too late. Then the **adouble** stack may grow unnecessarily, but the numerical results will still be correct, unless an exception occurs because the range of **locint** is exceeded. In general, free-store **adouble**s and **adoublev**s should be deleted in a last-in-first-out fashion toward the end of the program block in which they were created. When this pattern is maintained, the maximum number of **adouble**s alive (and, as a consequence, the core storage requirement of the derivative evaluation routines) is bounded by a small multiple of the core memory used in the relevant section of the original program. Failure to delete dynamically allocated **adouble**s may lead to longer and longer tapes if the same active section is called repeatedly.

To avoid the storage and manipulation of structurally trivial derivative values one should pay careful attention to the naming of variables. Ideally, the intermediate values generated during the evaluation of the vector function in question should be assigned to program variables that are consistently either active or passive, in that all their values either are or are not dependent on the independent variables in a nontrivial way. For example, this rule is violated if a temporary variable is successively used to accumulate inner products involving first only passive and later active arrays. Then the first inner product and all its successors in the data dependency graph become artificially active and the derivative evaluation routines described in Section 5 will waste time allocating and propagating trivial or useless derivatives. Sometimes even values that do depend on the independent variables may be of only transitory importance and not affect the dependent variables. For example, this is true for multipliers that are used to scale linear equations, but whose value does not influence the dependent variables in a mathematical sense. Such dead-end variables can be deactivated by the use of the **value** function, which converts **adouble**s to **double**s. The deleterious effects of unnecessary activity are partly alleviated by run-time activity flags in the derivative routine **hov_reverse** mentioned in Section 6.

4 Numbering the Tapes and Controlling the Buffer

The trace generated by the execution of an active section is stored by ADOL-C as either a triplet of internal arrays or as a triplet of files. We refer to these triplets as the tape array or the tape file. Either triplet may subsequently be used to evaluate the underlying function and its derivatives at the original point or at alternative arguments. Note that if the active section involves user-defined quadratures or branches conditioned on **adouble** comparisons, the function must be re-executed and retaped at each new argument. Otherwise, direct evaluation from the tape by the routine **function** (Section 5.5) may be faster, but this is not certain. (The requirement that functions with branches conditioned on **adoubles** be re-evaluated at each new argument can be circumvented by using a recent extension to ADOL-C. This extension to ADOL-C allows certain types of branches to be recorded on the tape through the use of conditional assignments and active integers. This extension is described in Appendix A.)

A tape array is used as a triplet of buffers for a tape file if the length of any of the buffers exceeds the array length of **bufsize**. The parameter **bufsize** is defined in the header file **usrparms.h** and may be adjusted by the user. Several tape files may be generated and kept simultaneously. For simple usage, **trace_on** may be called requiring only the tape **tag** as an argument, and **trace_off** may be called without an argument.

The optional integer argument **keep** of **tape_on** determines whether the numerical values of all active variables are recorded in an unnamed temporary file when they are overwritten or go out of scope. This option takes effect if **keep**$= 1$ and prepares the scene for an immediately following gradient evaluation by a call to the routine **reverse** (see Sections 5.1 and 5.4). Alternatively, gradients may be evaluated by a call to **gradient**, which includes a preparatory forward sweep for the creation of the temporary file. If omitted, the argument **keep** defaults internally to **0**, so that no temporary file is generated.

By setting the optional integer argument **file** of **trace_off** to **1**, the user may force a numbered tape file to be written even if the tape array (buffer) does not overflow. If the argument **file** is omitted, it is internally set to the default value zero, so that the tape array is written onto a tape only if the length of any of the buffers exceeds **bufsize** elements.

After the execution of an active section, if a tape file was generated (i.e., if the length some buffer exceeded **bufsize** elements or if the argument **file** of **trace_off** was set to 1), the files will be saved in the current working directory under the names **FNAME.<tag>**, **FNAME1.<tag>**, and **FNAME2.<tag>**, where **tag** is the mandatory argument to **trace_on** and **FNAME**, **FNAME1**, and **FNAME2** are the tape file names found in **usrparms.h**. Later, the problem-independent routines **forward, reverse, function, gradient, jacobian, vec_jac, jac_vec, hessian, hess_vec, lagra_hess_vec,** and **forode** accept as argument a tape's **tag** to determine the tape on which their respective computational task is to be performed. By calling **trace_on** with different tape **tag**s, one can create several tapes for various function evaluations and subsequently perform function and derivative evaluations on one or more of them.

For example, suppose one wishes to calculate for two smooth functions $f_1(x)$ and $f_2(x)$

$$f(x) = \max\{f_1(x), f_2(x)\} \quad , \quad \nabla f(x) \quad ,$$

and possibly higher derivatives where the two functions do not tie. Provided f_1 and f_2 are evaluated in two separate active sections, one can generate two different tapes by calling **trace_on** with **tag = 1** and **tag = 2** at the beginning of the respective active sections. Subsequently, one can decide whether $f(x) = f_1(x)$ or $f(x) = f_2(x)$ at the current argument and then evaluate the gradient $\nabla f(x)$ by calling **gradient** with the appropriate argument value **tag = 1** or **tag = 2**.

4.1 Examining the Tape and Predicting Storage Requirements

At any point in the program, one may call the routine

void tapestats(unsigned short tag, int* counts)

with **counts** an array of at least eleven integers. The first argument **tag** specifies the particular tape of interest. The components of **counts** represent

counts[0] the number of independents, i.e., calls to $\ll=$

counts[1] the number of dependents, i.e., calls to $\gg=$

counts[2] the maximal number of live active variables

counts[3] the number of deaths and overwrites

counts[4] the buffer size (a multiple of eight)

counts[5] the total number of operations recorded

counts[6-10] other internal information about the tape

The values **maxlive = counts[2]** and **deaths = counts[3]** determine the temporary storage requirements during calls to the workhorses **forward** and **reverse**. For a certain degree **deg** ≥ 0, the scalar verison of the routine **forward** involves (apart from the tape buffers) an array of **(deg+1)*maxlive** doubles in core and, in addition, a sequential data set of **deaths*keep revreal**s if called with the option **keep**$\geq$**0**. Here the type **revreal** is defined as **double** or **float** in the header file **usrparms.h**. The latter choice halves the storage requirement for the sequential data set, which stays in core if its length is less than **bufsize** bytes and is otherwise written out to a temporary file. The drawback of the economical **revreal=float** choice is that subsequent calls to **reverse** yields gradients and other adjoint vectors only in single-precision accuracy. This may be acceptable if the adjoint vectors represent rows of a Jacobian that is used for the calculation of Newton steps. In its scalar version, the routine **reverse** involves the same number of **doubles** and twice as many **revreals** as **forward**. The storage requirements of the vector versions of **forward** and **reverse** are equal to that of the scalar versions multiplied by the vector lenght.

4.2 Customizing ADOL-C

Based on the information provided by **tapestats**, the user may alter the following types and constant dimensions set in **usrparms.h** to suit his problem and environment.

bufsize (default: 16384) This integer determines the length of internal buffers. If the buffers are large enough to accommodate all required data, any disk access is avoided unless **trace_off** is called with a positive argument. This desirable situation can be achieved for many problem functions with an execution trace of moderate size. Primarily **bufsize** occurs as an argument to **malloc**, so that setting it unnecessarily large may have no ill effects, unless the operating system prohibits or penalizes large array allocations.

locint (default: unsigned short) The range of the integer type **locint** determines how many **adouble**s can be simultaneously alive. In extreme cases when there are more than 65,535 **adouble**s alive at any one time, the **locint** value must be changed to **long int**.

revreal (default: double) The choice of this floating-point type trades accuracy with storage during reverse sweeps. While functions and their derivatives are always evaluated in double precision during forward sweeps, gradients and other adjoint vectors are obtained with the precision determined by the type **revreal**. The more accurate choice **revreal = double** virtually doubles the storage requirement during reverse sweeps.

npr (default: 0) If this integer is positive, ADOL-C prints out some basic messages about its progress.

store (default: dontusethisuglymessplease) This is the name of another internal array, which may not be reused.

FNAME (default: "_adol-op_tape.") This is the physical name of the file that holds the operations tape when the internal buffer is exceeded. An integer **tag** is appended to the name to make it unique.

FNAME1 (default: "_adol-in_tape.") This is the physical name of the file that holds the integer tape when the internal buffer is exceeded. An integer **tag** is appended to the name to make it unique.

FNAME2 (default: "_adol-rl_tape.") This is the physical name of the file that holds the real-valued tape when the internal buffer is exceeded. An integer **tag** is appended to the name to make it unique.

The only other protected names are those of the external functions listed in the header files **adutils.h, adutilsc.h, taputil1.h, taputil2.h, taputil3.h**, and **tayutils.h** as well as the pair of functions **take_stock** and **keep_stock** declared in **adouble.h**.

5 Evaluating Derivatives from a Tape

After the execution of an active section, the corresponding tape contains a detailed record of the computational process by which the final values y of the dependent variables were obtained from the values x of the independent variables.

5.1 General Mathematical Description

Provided no arithmetic exceptions occurred and all special functions were evaluated in the interior of their domains, the functional relation between the input variables x and the output variables y, which we will denote by $y = F(x)$, is in fact analytic. In other words, we can compute arbitrarily high derivatives of the vector function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$ defined by the active section. We find it most convenient to describe and compute derivatives in terms of univariate Taylor expansions, which are truncated after the highest derivative degree d that is desired by the user. Let

$$x(t) \quad \equiv \quad \sum_{j=0}^{d} x_j t^j \quad : \quad \mathbb{R} \mapsto \mathbb{R}^n \tag{1}$$

denote any vector polynomial in the scalar variable $t \in \mathbb{R}$. In other words $x(t)$ decribes a path in $\mathbb{R}^n$ parametrized by t. The Taylor coefficient vectors

$$x_j \quad = \quad \frac{1}{j!} \frac{\partial^j}{\partial t^j} x(t) \Big|_{t=0}$$

are simply the scaled derivatives of $x(t)$ at the parameter origin $t = 0$. The first two vectors $x_1, x_2 \in \mathbb{R}^n$ can be visualized as tangent and curvature at the base point x_0, respectively. Provided that F is d times continuously differentiable it follows from the chain rule that the image path

$$y(t) \quad \equiv \quad F(x(t)) \quad : \quad \mathbb{R} \mapsto \mathbb{R}^m$$

is also smooth and has at least d Taylor coefficient vectors $y_j \in \mathbb{R}^m$ at $t = 0$ so that

$$y(t) \quad = \quad \sum_{j=0}^{d} y_j t^j + O(t^{d+1}) \quad . \tag{2}$$

Also as a consequence of the chain rule one can observe that each y_j is uniquely and smoothly determined by the x_i with $i \leq j$, i.e., by the coefficient vectors x_i of the same or lower order as y_j. In particular we have

$$y_0 = F(x_0) \quad , \quad y_1 = F'(x_0)\, x_1$$

and

$$y_2 = F'(x_0)\, x_2 + \tfrac{1}{2} F''(x_0)\, x_1\, x_1 \quad .$$

In writing down the last term we have already departed from the usual matrix-vector notation. It is well known that the number of terms that occur in these more or less 'symbolic' expressions for the y_j in terms of the first j derivative tensors of F and the 'input' coefficients x_i with $i \leq j$ grows very rapidly with j. Fortunately, this exponential growth does not occur in automatic differentiation, where the many terms are somehow implicitly combined so that storage and operations count grow only quadratically in the bound d on j.

Provided F is analytic, this property is inherited by the functions

$$y_j = y_j(x_0, x_1, \ldots, x_j) \in \mathbf{R}^m \quad ,$$

and their derivatives satisfy the identity

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i}) \tag{3}$$

which has been established in [6]. The $m \times n$ matrices $A_k, k = 0, \ldots, d$ are in fact the Taylor coefficients of the Jacobian path $F'(x(t))$, a fact that is of interest primarily in the context of ordinary differential equations and differential algebraic equations.

Given the tape of an active section and the coefficients x_j, the resulting y_j and their derivatives A_j can be evaluated by appropriate calls to the ADOL-C procedures **forward** and **reverse**. The co-called scalar version of **forward** propagates just one truncated Taylor series from the $(x_j)_{j \leq d}$ to the $(y_j)_{j \leq d}$. The vector version of **forward** propagates families of $p \geq 1$ such truncated Taylor series in order to reduce the relative cost of the overhead incurred in the tape interpretation. There are also specialized codes for the cases $d = 0$ and $d = 1$, where some unnecessary dereferencing can be avoided. Details of the appropriate calling sequences are given in Subsection 5.4.

Given a weighting vector u, the ADOL-C procedure **reverse** computes the collection of row vectors

$$z_j \equiv u^T \frac{\partial y_j}{\partial x_0} = u^T A_j \in \mathbf{R}^n \tag{4}$$

for $j = 0, 1, \ldots, d$. If $j = 0$ and u is the i-th Cartesian basis vector in $\mathbf{R}^m$, then (4) yields the i-th row of the Jacobian $F'(x)$. To produce the entire Jacobian in this mode, one may make m calls to **reverse** setting u to the i-th Cartesian basis vector for $i = 0, 1, \ldots, m$.

An alternative to this method is provided by the vector version of **reverse**, which yields a collection of matrices of the form

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in \mathbf{R}^{p \times n}, \tag{5}$$

where $U \in \mathbf{R}^{p\times n}$ represents a *weighting matrix.* When $U = I_m$ with $p = m$, one call to **reverse** yields the set of full Jacobians $\partial y_j / \partial x_0$. This choice requires more storage, but it significantly reduces the relative cost of the tape interpretation when the degree d is small.

5.2 Derivatives for Optimization Calculations

When $d = 0$ in the vector mode, we have the undifferentiated relation $y_0 = F(x_0)$, and

$$z_0 = u^T \; F'(x_0) \tag{6}$$

yields the Jacobian of F multiplied from the left by $u \in \mathbf{R}^m$. In nonlinear least squares calculations, one may use $u^T \equiv F(x_0)^T$ so that $z_0 \in \mathbf{R}^n$ is simply the gradient of the sum of squares. For the iterative computation of Newton-like steps, one may wish to calculate $u^T F'(x_0)$ for a sequence of m vectors u. Thus, **reverse** with $d = 0$ can be used to premultiply the Jacobian by one (or more) row vector u^T from the left. Similarly, one can use **forward** with $d = 1$ to calculate the matrix-vector product

$$y_1 = F'(x_0) \cdot x_1, \tag{7}$$

where x_1 is an arbitrary n vector. In contrast to **reverse**, **forward** does not currently have the capability to multiply the Jacobian simultaneously by several column vectors.

For a scalar function F (i.e., $m = 1$), one finds that with $u^T = 1 \in \mathbf{R}$, the adjoint $z_0 = F'(x_0)$ is the gradient of F, and the adjoint

$$z_1 = \frac{\partial y_1}{\partial x_0} = \frac{\partial F'(x_0)x_1}{\partial x_0} = [\nabla^2 F(x_0) x_1]^T \tag{8}$$

represents the product of the Hessian $\nabla^2 F(x_0)$ with an arbitrary vector x_1. More generally, let us consider the case where $F^T(x) \equiv [f(x), c^T(x)]$ consists of a scalar objective function $f(x)$ and an $m - 1$ vector $c(x)$ of constraint functions. Here one may choose u^T as a vector of Lagrange multiplier estimates such that approximately $u^T F'(x) = 0$ with the first component normalized to 1. Then $z_0 \in \mathbf{R}^n$ represents the gradient of the Lagrangian function $u^T F(x)$, and $z_1 \in \mathbf{R}^n$ represents its Hessian multiplied by the vector x_1.

5.3 Derivatives for Differential Equations

When F is the right-hand side of an (autonomous) ordinary differential equation

$$x'(t) \;=\; F(x(t)),$$

we must have $m = n$. Along any solution path $x(t)$ its Taylor coefficients x_j at some time, say $t = 0$, must satisfy the relation (1) with the y_j the Taylor coefficients of its derivative $y(t) = x'(t)$, namely,

$$x_{i+1} = \tfrac{1}{1+i} y_i \quad .$$

Using this relation, one can generate the coefficients x_i recursively from the current point x_0 by calling **forward** with increasing degree $i = 0, 1, \ldots, d-1$. This task is achieved by the driver routine **forode**.

For the numerical solutions of ordinary differential equations, one also may wish to calculate the Jacobians

$$B_j \;\equiv\; \frac{\mathrm{d}x_{j+1}}{\mathrm{d}x_0} \;\in\; \mathbf{R}^{n\times n}\,, \tag{9}$$

which exist provided F is sufficiently smooth. These matrices can be obtained from the partial derivatives $\partial y_i/\partial x_0$ obtained from **reverse** by an appropriate version of the chain rule. This task is performed by the utility **accode**, which involves $\frac{1}{2}d(d-1)$ matrix-matrix products. Through an optional argument of **reverse** one can find out which entries of the Jacobian $F'(x(t))$ are zero or constant with respect to t, and this sparsity information can be exploited by **accode** and other utilities. In particular, there need be no loss in computational efficiency if a time-dependent ODE is rewritten in an autonomous form.

5.4 Forward and Reverse Calls

The following calling sequences utilize the overloading capabilities of C++ and can therefore not be called directly from C or Fortran. However, the ADOL-C source (see the file driversc.c) contains C and Fortran-callable versions of **forward** and **reverse**, and the optimization drivers described in the following section are all C functions with Fortran-callable companions. Given any correct tape, one may call from within the generating program, or subsequently during another run, the following procedure:

```
void forward(tag,m,n,d,keep,X,Y)
short int tag;                    // tape identification
int m;                            // number of dependent variables
int n;                            // number of independent variables
int d;                            // highest derivative degree
int keep;                         // flag for reverse sweep
double X[n][d+1];                 // independent-variable values
double Y[m][d+1];                 // result coefficients as in (2)
```

The rows of the matrix **X** must correspond to the independent variables in the order of their initialization by the $\ll=$ operator. The columns of $\mathbf{X} = \{x_j\}_{j=0..d}$ represent Taylor coefficient vectors as in Equation (2). The rows of the matrix **Y** must correspond to the dependent variables in the order of their selection by the $=\gg$ operator. Thus the first column of **Y** contains the function value $F(x)$ itself, the next column represents the first Taylor coefficient vector of F, and the last column the **d**-th Taylor coefficient vector. The integer flag **keep** plays a similar role as in the call to **trace_on**; it determines how many Taylor coefficients of all intermediate quantities **forward** writes into a buffered temporary file in preparation for a subsequent reverse sweep. If **keep** is omitted, the variable defaults internally to 0.

The given **tag** value is used by **forward** to determine the name of the file on which the tape was written. If the tape file does not exist, **forward** assumes that the relevant tape is still in core and reads from the buffers. Provided the original code involves neither user-defined quadratures nor conditional branches, **forward** can be used to evaluate the vector-function F at arguments x other than the point a. However, if these preconditions are not met, **forward** and subsequently **reverse** may appear to function properly, but the numerical values will be incorrect.

After the execution of an active section with **keep** = 1 or a call to **forward** with **keep** $\leq$ **d+1**, one may call the function **reverse** with **d=keep-1** and the same tape identifier **tag**. When **u** is a vector and **z** an $n \times (d+1)$ matrix as in (4), **reverse** is executed in the *scalar mode* by the following calling sequence:

```
void reverse(tag,m,n,d,u,Z)
short int tag;                  // tape identification
int m;                          // number of dependent variables
int n;                          // number of independent variables
int d;                          // highest-derivative degree
double u[m];                    // weighting vector
double Z[n][d+1];               // result adjoints as in (4)
```

When U is a matrix as in (5), **reverse** is executed in the *vector mode* by the following calling sequence:

```
void reverse(tag,m,n,d,p,U,Z,nz)
short int tag;                  // tape identification
int m;                          // number of dependent variables
int n;                          // number of independent variables
int d;                          // highest derivative degree
int p;                          // number of weight vectors
double U[p][m];                 // domain weight vector
double Z[p][n][d+1];            // result adjoints as in (5)
short nz[p][n];                 // nonzero pattern of Z
```

The last argument can be omitted. Otherwise each short **nz[i][j]** has a value of 0, 1, 2, 3, or 4 which characterizes the (i,j)-th entry of the weighted Jacobian **Z** as a zero, constant, polynomial, rational, or transcendental function of the independents, respectively. When the arguments **p** and **U** are omitted, they default to m and the identity matrix of order m, respectively.

In both scalar and vector mode, the degree **d** must agree with **keep-1** for the most recent call to **forward**, or it must be equal to zero if **reverse** directly follows the taping of an active section. Otherwise, reverse will return control with a suitable error message. In order to avoid possible confusion, the first four arguments must always be present in the calling sequence. However, if **m** or **d** attain their trivial values 1 and 0, respectively, then corresponding dimensions of the arrays **X**, **Y**, **u**, **U**, or **Z** can be omitted, thus eliminating one level of indirection. For example, we may call **reverse(tag,1,n,0,1.0,g)** after **double g[n]** or **doublev g(n)** to calculate a gradient of a scalar-valued function.

Sometimes it may be useful to perform the forward sweep for families of Taylor series with the same leading term. This vector version of forward can be called in the form

```
void forward(tag,m,n,d,p,x,X,y,Y)
short int tag;             // tape identification
int m;                     // number of dependent variables
int n;                     // number of independent variables
int d;                     // highest derivative degree
int p;                     // number of Taylor series
double x[n];               //common leading terms
double X[n][p][d];         //Taylor coefficients of independent variables
double y[m];               //values of dependent variables
double Y[m][p][d];         //Taylor coefficients of dependent variables
```

where **X** and **Y** hold the Taylor coefficients of first and higher degree and **x**, **y** the common Taylor coefficients of degree 0. There is no option to keep the values of active variables that are going out of scope or that are overwritten. Therefore this function cannot prepare a subsequent reverse sweep.

Since the calculation of Jacobians is probably the most important automatic differentiation task we have provide a specialization of vector forward to the case where $d = 1$. This version can be called in the form

```
void forward(tag,m,n,p,x,X,y,Y)
short int tag;                    // tape identification
int m;                            // number of dependent variables
int n;                            // number of independent variables
int p;                            // number of partial derivatives
double x[n];                      //common leading terms
double X[n][p];                   //Seed derivatives of independent variables
double y[m];                      //values of dependent variables
double Y[m][p];                   //First derivatives of dependent variables
```

When this routine is called with $p = n$ and X the identity matrix the resulting Y is simply the Jacobian $F'(x)$. In general, one obtaines the $m \times p$ matrix $Y = F'(x)\,X$ for the chosen initialization of X. In a work station environment a value of p somewhere between 10 and 50 appears to be fairly optimal. For smaller p the interpretive overhead is not appropriately amortized and for larger p the p-fold increase in storage causes too many page faults. Therefore, large Jacobians that cannot be compressed via column coloring (as was done for example in [2]) should be *strip-mined* in that the above first-order-vector version of **forward** is called repeatedly with the successive $n \times p$ matrices X forming a partition of the identity matrix of order n.

5.5 Drivers for Optimization and Nonlinear Equations

For convenience one may use instead of **forward** and **reverse** the following C driver routines:

```
void function(tag,m,n,x,y)
short int tag;                         // tape identification
int m;                                 // number of dependent variables
int n;                                 // number of independent variables
double x[n];                           // independent vector x_0 as in (1)
double y[m];                           // result y_0 as in (2)
```

```
void gradient(tag,n,x,g)
short int tag;                         // tape identification
int n;                                 // number of independent variables
double x[n];                           // independent vector x_0 as in (1)
double g[n];                           // result z_0 as in (6) for m = 1
```

```
void vec_jac(tag,m,n,repeat,x,u,z)
```

```
short int tag;            // tape identification
int m;                    // number of dependent variables
int n;                    // number of independent variables
int repeat;               // has vec_jac been called here ?
double x[n];              // independent vector $x_0$ as in (1)
double u[m];              // range weight vector
double z[n];              // result $z_0$ as in (6)
```

```
void jacobian(tag,m,n,x,J)
short int tag;            // tape identification
int m;                    // number of dependent variables
int n;                    // number of independent variables
double x[n];              // independent vector $x_0$ as in (1)
double J[m][n];           // output Jacobian $F'(x)$
```

```
void jac_vec(tag,m,n,x,v,z)
short int tag;            // tape identification
int m;                    // number of dependent variables
int n;                    // number of independent variables
double x[n];              // independent vector $x_0$ as in (1)
double v[n];              // tangent vector
double z[m];              // result $y_1$ as in (7)
```

```
void hess_vec(tag,n,x,v,z)
short int tag;            // tape identification
int n;                    // number of independent variables
double x[n];              // independent vector $x_0$ as in (1)
double v[n];              // tangent vector $x_1$
double z[n];              // result $z_1$ as in (8)
```

```
void hessian(tag,n,x,H)
short int tag;            // tape identification
int n;                    // number of independent variables
double x[n];              // independent vector $x_0$ as in (1)
double H[n][n];           // Hessian matrix (lower half)
```

```
void lagra_hess_vec(tag,m,n,x,v,u,h)
short int tag;                    // tape identification
int m;                            // number of dependent variables
int n;                            // number of independent variables
double x[n];                      // independent vector x_0 as in (1)
double v[n];                      // tangent vector x_1
double u[m];                      // range weight vector
double h[n];                      // result z_1 as in (4)
```

These procedures are prototyped in **adutilsc.h** for compilation in a C program. This header file is included as **extern "C"** within the corresponding C++ header file **adutils.h**. All drivers have companion Fortran functions which are obtained by adding a trailing underscore to the name and replacing all short and integer arguments with integer pointers. For example, the Fortan version of gradient is declared in **adutilsc.h** as

```
int gradient_(int* tag, int* n, double* x, double* g)
```

and defined in **driversc.c**.

The scalar-mode drivers **gradient** and **hess_vec** create a temporary file by an appropriate call to **forward** with **keep=1** and then call **reverse** with the corresponding argument. The routine **vec_jac** functions in the same way, except that the internal call to **forward** is omitted if a nonzero value of the parameter **repeat** indicates that **forward** has already been called at the same argument. When **m=1** and the original evaluation code contains neither quadratures nor branches, **function**, **gradient**, and **hessian** can be used to evaluate the scalar function and its derivatives at any argument in its domain.

5.6 Drivers for ODEs

Given the basis point x_0, we can obtain the matrix $\mathbf{X}=(x_j)_{j\leq d}$ of the Taylor coefficient defined by an autonomous right-hand side recorded on the tape by the following call:

```
void forode(tag,n,tau,dol,deg,X)
short int tag;                    // tape identification
int n;                            // number of state variables
double tau;                       // scaling parameter
int dol;                          // degree on previous call
int deg;                          // degree on current call
double X[n][deg+1];               // Taylor coefficient vectors
```

If **dol** is positive, it is assumed that **forode** has been called before at the same point so that all Taylor coefficient vectors up to the **dol**-th are already correct. Subsequently one may call

hov_reverse(tag,n,n,deg-1,n,I,Z,nz)

to compute the family of square matrices **Z[n][n][deg]** defined in equation (5) of subsection 5.1. Here **double** I** must be the identity matrix of order **n**. To compute the total derivatives **B**=$(B_j)_{0\leq j<d}$ defined in (9), one may finally call the following:

```
void accode(n,tau,deg,Z,B,nz)
int n;                          // number of state variables
double tau;                     // scaling parameter
int deg;                        // degree on current call
double Z[n][n][deg];            // partials of coefficient vectors
double B[n][n][deg];            // results as defined in (9)
short nz[n][n];                 // optional sparsity pattern
```

Naturally, **nz** can be used by **accode** only if it has been set in the call to **reverse** above. The nonpositive entries of **nz** are changed by **accode** so that upon return

$$B[i][j][k] \;\equiv\; 0 \quad \text{if} \quad k \leq -nz[i][j]\ .$$

In other words, the matrices B_k=**B[ ][ ][k]** have a sparsity pattern that fills in as k grows.

6 Installing and Using ADOL-C

The ADOL-C package consists of the following sixteen modules and eleven header files. Only two header files are needed for simple applications of ADOL-C and three more for more advanced uses. The remaining sources are compiled into the library **libad.a**. Thirteen of the sixteen modules can be compiled with either C++ or ANSI C, and only the first four in the following list are written in C++.

```
adouble.c       avector.c      drivers.c       utils.c

taputil1.c      taputil2.c     taputil3.c      tayutil.c
hos_forward.c  fov_forward.c  hov_forward.c   fos_reverse.c
hos_reverse.c  fov_reverse.c  hov_reverse.c   driversc.c

adouble.h     avector.h      adutils.h      adutilsc.h     usrparms.h

dvlparms.h   taputil1.h  taputil2.h  taputil3.h  oplate.h  tayutil.h
```

The sixteen modules contain the following functions. The module **adouble.c** controls the nominal allocation and elementary operations for variables of the class **adouble** as defined in **adouble.h**. The module **avector.c** contains the routines for the elementary operations for the classes **doublev** and **adoublev** as defined in **avector.h**. The module **tayutil.c** controls the storage and retrieval of Taylor series coefficients during forward and reverse sweeps, respectively. The modules **taputil1.c**, **taputil2.c**, and **taputil3.c** contain all the functions and variables needed for the taping of arithmetic operations and function evaluations in an active section. The modules **hos_forward.c**, **fov_forward.c**, **hov_forward.c**, **fos_reverse.c**, **fov_reverse.c**, **hos_reverse.c**, and **hov_reverse.c** consist of the core procedures for evaluating derivatives from a given tape. They are called from various driver routines contained in the files **drivers.c** and **driversc.c**; these modules are natural places for additional user-defined drivers and utilities. The module **utils.c** contains the routines **trace_on** and **trace_off**.

The user may modify the header file **usrparms.h** in order to tailor the package to one's needs in the particular system environment as discussed in Section 4.2. To generate the library **libad.a** one should use one of the executables <*operating_system*>**install** (e.g. **aixinstall**) according to the given operating system. They call `make` with the appropriate makefiles; the source contains makefiles for IBM RS/6000 and various other environments. Notice that the only files that need to be compiled with a C++ compiler are **adouble.c**, **avector.c**, **drivers.c**, and **utils.c**. The user has to ensure that the suitable compilers and their corresponding libraries are in the path.

6.1 Compiling and Linking C++ Programs with Active Sections

To compile a C++ program that involves variables of type **adouble** or **adoublev**, one must add the directive **#include "adouble.h"** at the beginning of the program file. Programs that call on the various derivative evaluation routines must include the header **adutils.h**. For linking the resulting object codes, the options pointing to the header files and the library **libad.a** must be used.

6.2 Adding Quadratures as Special Functions

Let us suppose an integral

$$f(x) = \int_0^x g(t)dt$$

is evaluated numerically by a user-supplied function

double integral(double& x)

Similarly, let us suppose that the integrand itself is evaluated by a user-supplied block of C code *integrand*, which computes a variable with the fixed name **val** from a variable with the fixed name **arg**. In many cases of interest, *integrand* will simply be of the form

{ val = expression(arg) }

In general, the final assignment to **val** may be preceded by several intermediate calculations, possibly involving local automatic variables of type **adouble**, but no external or static variables of that type. However, *integrand* may involve local or global variables of type **double** or **int**, provided they do not depend on the value of **arg**. The variables **arg** and **val** are declared automatically; and as *integrand* is a block rather than a function, *integrand* should have no header line.

Now the function **integral** can be overloaded for **adouble** arguments and thus included in the library of elementary functions by the following modifications.

1. At the end of the file **adouble.c**, include the full code defining **double integral(double& x)**, and add the line

   extend_quad(integral, *integrand***);**

   this is a macro that is extended to the full definition of **adouble integral(adouble& arg)**. Then remake the library **libad.a**.

2. In the definition of the class **adouble** in **adouble.h**, add the statement

   friend adouble integral(adouble&).

In the first modification, **integral** represents the name of the double function, whereas *integrand* represents the actual block of C code.

For example, in case of the arcos of the hyperbolic cosine, we have **integral=acosh**. *Integrand* can be written as

{ val = sqrt(arg*arg-1) }

so that the line

extend_quad(acosh,val = sqrt(arg*arg-1))

can be added to the file **adouble.c**. A mathematically equivalent but longer representation of *integrand* is

$$
\begin{aligned}
\{\textbf{adouble}\quad \textbf{temp} &= \textbf{arg};\\
\textbf{temp} &= \textbf{temp} * \textbf{temp};\\
\textbf{val} &= \textbf{sqrt}(\textbf{temp});\}
\end{aligned}
$$

The integrands may call on any elementary function that has already been defined in **adouble.c**, so that one may also introduce iterated integrals.

7 Example Codes

The following listings are all simplified versions of codes that are contained in the examples subdirectory `*/SRC/DEX` of ADOL-C. In particular, we have left out timings and correctness checks, which are included in the full codes.

7.1 Product Example

The first example evaluates the gradient and a Hessian-vector product for the function

$$y = f(x) = \prod_{i=0}^{n-1} x_i$$

using the appropriate drivers **gradient** and **hessian**.

```
#include "adouble.h"
#include "adutils.h"
#include <stream.h>
main() {
  int n,i,j,counts[12];
  cout << "number of independent variables = ?  \n";
  cin >> n;
  double* xp = new double[n];          // or: doublev xp(n);
  adouble* x = new adouble[n];         // or: adoublev x(n);
  for(i=0;i<n;i++)
    xp[i] = (i+1.0)/(2.0+i);           // some initializations
  trace_on(1);                         // tag =1, keep=0 by default
  adouble y = 1;
  for(i=0;i<n;i++){
    x[i] <<= xp[i];                    // or  x<<= xp outside the loop
    y *= x[i]; }
  double yp=0.0;
  y >>= yp;
  delete[] x;                          // Not needed if x  adoublev
  trace_off();
  tapestats(1,counts);                 // Reading of tape statistics
  cout<<"maxlive "<<counts[2]<<"\n";
  // ..... print other tape stats
  double* g = new double[n];          // or: doublev g(n);
  gradient(1,n,xp,g);                 // gradient evaluation.
  doublem h(n,n);
  hessian(1,n,xp,h);                  // h equals (n-1)g since g is
  double errg =0;                     // homogeneous of degree n-1.
  double errh =0;
  for(i=0;i<n;i++)
     errg += abs(g[i]-yp/xp[i]);      // vanishes analytically.
  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      if (i>j)                        // lower half of hessian
        errh += abs(h[i][j]-g[i]/xp[j]); } }
  cout << yp-1/(1.0+n) << " error in function \n";
  cout << errg <<" error in gradient \n";
  cout << errh <<" consistency check \n";
  }
```

7.2 Scalar Example

The second example function evaluates the n-th power of a real variable x in $\log_2 n$ multiplications by recursive halving of the exponent. Since there is only one independent variable, the scalar derivative can be computed by using both **forward** and **reverse**, and the results are subsequently compared.

```
#include "adouble.h"
#include "adutils.h"
#include <stream.h>
adouble power(adouble x, int n) {
  adouble z=1;
  if (n>0) {                        // Recursion and branches
    int nh =n/2;                    // that do not depend on
    z = power(x,nh);                // adoubles are fine !!!!
    z *= z;
    if (2*nh != n) z *= x;
    return z; }
  else {
    if (n==0) return z;             // The local adouble z dies
    else return 1/power(x,-n); }    // as it goes out of scope.
}
```

The function `power(...)` above was obtained from the original undifferentiated version by simply changing the type of all doubles including the return variable to **adoubles**. The new version can now be called from within any active section, as in the following main program.

```
#include ....... as above
main() {
  int i,tag=1;
  cout<<"monomial degree=? \n";      // Input the desired degree.
  int n; cin >> n;
/*Allocations and Initializations*/
  double* Y[1];
  *Y = new double[n+2];
  double* X[1];                      // Allocate passive variables with
  *X = new double[n+4];              // extra dimension for derivatives
  X[0][0] = 0.5;                     // function value = 0. coefficient
  X[0][1] = 1.0;                     // first derivative = 1. coefficient
  for(i=0; i < n+2; i++)
    X[0][i+2]=0;                     // further coefficients.
  double* Z[1];                      // used for checking consistency
  *Z = new double[n+2];              // between forward and reverse
  adouble y,x;                       // Declare active variables
/*Beginning of Active Section*/
  trace_on(1);                       // tag = 1 and keep = 0
  x <<= X[0][0];                     // Only one independent var
  y = power(x,n);                    // Actual function call
  y >>= Y[0][0];                     // Only one dependent adouble
  trace_off();                       // No global adouble has died
/*End of Active Section */
  double u[1];                         // weighting vector
  u[0]=1;                              // for reverse call
  for(i=0; i < n+2; i++) {       // Note that keep = i+1 in call
     forward(tag,1,1,i,i+1,X,Y);     // Evaluate the i-the derivative
     if (i==0)
       cout << Y[0][i] << "=?" << value(y) << " should be the same \n";
     else
       cout << Y[0][i] << "=?" << Z[0][i] << " should be the same \n";
     reverse(tag,1,1,i,u,Z);         // Evaluate the (i+1)-st deriv.
     Z[0][i+1]=Z[0][i]/(i+1); }     // Scale derivative to Taylorcoeff.
}
```

Since this example has only one independent and one dependent variable, **forward** and **reverse** have the same complexity and calculate the same scalar derivatives, albeit with a slightly different scaling. By replacing the function **power** with any other univariate test function, one can check that **forward** and **reverse** are at least consistent. In the following example the number of independents is much larger than the number of dependents, which makes the reverse mode preferable.

7.3 Determinant Example

Now let us consider an exponentially expensive calculation, namely, the evaluation of a determinant by recursive expansion along rows. The gradient of the determinant with respect to the matrix elements is simply the adjoint (i.e., the matrix of cofactors). Hence the correctness of the numerical result is easily checked by matrix-vector multiplication. The example illustrates the use of **adouble** arrays and pointers.

```
#include ....... as above
adouble** A;                       // A is an n x n matrix
int n;                             // k <= n is the order
adouble det(int k, int m) {        // of the submatrix
  if(m == 0 ) return 1.0 ;         // its column indices
  else {                           // are encoded in m.
    adouble* pt = A[k-1];
    adouble t =0 ;
    int s, p =1;
    if (k%2) s = 1; else s = -1;
    for(int i=0;i<n;i++) {
      int p1 = 2*p;
      if ( m%p1 >= p ) {
        t += *pt*s*det(k-1, m-p);  // Recursive call to det.
        s = -s; }
      ++pt;
      p = p1; }
    return t; }
}
```

As one can see, the overloading mechanism has no problem with pointers and looks exactly the same as the original undifferentiated function except for the change of type from **double** to **adouble**. If the type of the temporary **t** or the pointer **pt** had not been changed, a compile time error would have resulted. Now consider a corresponding calling program.

```
#include ........ as above
main() {
  int i, m=1,tag=1,keep=1;
  cout << "order of matrix = ? \n";  // Select matrix size
  cin >> n;
  A = new adouble*[n];
  trace_on(tag,keep);                      //      tag=1=keep
  double detout=0.0 , diag = 1.0;   // here keep the intermediates for
  for (i=0; i<n; i++) {             // the subsequent call to reverse
    m *=2;
    A[i] = new adouble[n];
    adouble* pt = A[i];
    for (int j=0;j<n; j++)
      A[i][j] <<= j/(1.0+i);        //make all elements of A independent
    diag += value(A[i][i]);         //value(adouble) converts to double
    A[i][i] += 1.0; }
  det(n,m-1) >>= detout;            // Actual function call.
  printf("\n %f =? %f should be the same \n",detout,diag);
  trace_off();
  double u[1];
  u[0] = 1.0;
  double* B = new double[n*n];
  reverse(tag,1,n*n,1,u,B);
  cout <<" \n first base? : ";
  for (i=0;i<n;i++) {
    adouble sum = 0;
    for (int j=0;j<n;j++)           // The matrix A times the first n
      sum += A[i][j]*B[j];          // components of the gradient B
    cout<<value(sum)<<" "; }        // must be a Cartesian basis vector
  cout<<"\n";
}
```

The variable **diag** should be equal to the determinant, because the matrix **A** is defined as a rank 1 perturbation of the identity.

7.4 ODE Example

Finally, we use the right-hand side of a nonlinear ordinary differential equation.

```
#include ..... as above
void tracerhs(short int tag, double* py, double* pyprime)
{
adoublev y(3);              //This time we left the parameters
adoublev yprime(3);         // passive and use the vector types.
trace_on(tag);
y <<= py;                   //Initialize and mark independents
yprime[0] = -sin(y[2]) + 1e8*y[2]*(1-1/y[0]);
yprime[1] = -10*y[0] + 3e7*y[2]*(1-y[1]);
yprime[2] = -yprime[0] - yprime[1];
yprime >>= pyprime;        //Mark and pass dependents
trace_off(tag);
}
```

This function is a slight modification of Robertson test problem given in Hairer and Wanner's book on the numerical solution of ODEs [10]. The Jacobian of the right-hand side has large negative eigenvalues, which make the ODE quite stiff. We have added some numerically benign transcendentals to make the differentiation more interesting. The following program uses **forode** to calculate the Taylor series defined by the ODE at the given point y_0 and **reverse** as well as **accode** to compute the Jacobians of the coefficient vectors with respect to x_0.

```
#include ..... as above
main() {
int i,j,deg;
int n=3;
doublev py(3);
doublev pyp(3);
cout << "degree of Taylor series =?\n";
cin >> deg;
doublem X(n,deg+1);
double*** Z=new double**[n];
double*** B=new double**[n];
short** nz = new short*[n];
for(i=0;i<n;i++) {
    py[i] = (i == 0) ? 1.0 : 0.0;     // Initialize the base point
    X[i][0] = py[i];                  // and the Taylor coefficient
    nz[i] = new short[n];             // set up sparsity array
    Z[i]=*(new doublem(n,deg));
    B[i]=*(new doublem(n,deg));}
tracerhs(1,py,pyp);                   // trace RHS with tag = 1
forode(1,n,deg,X);                    // compute deg coefficients
reverse(1,n,n,deg-1,Z,nz);            // U defaults to the identity
accode(n,deg-1,Z,B,nz);
cout << "nonzero pattern:\n";
for(i=0;i<n;i++) {
   for(j=0;j<n;j++)
      cout << nz[i][j]<<"\t";
   cout <<"\n"; }
}
```

The pattern **nz** returned by **accode** is

```
3  -1   4
1   2   2
3   2   4
```

The original pattern **nz** returned by **reverse** is the same except that the negative entry -1 was zero.

A Extended Features

A.1 Conditional Assignments

In some situations it may be desirable to calculate the value and derivatives of a function at arbitrary arguments by using a tape of the function evaluation at one argument and reevaluating the function and its derivatives using the given routines. This approach can significantly reduce run times, and it also allows one to port problem functions, in the form of the corresponding tape files, into a computing environment that does not support C++ but does support C or Fortran.

Unfortunately, the evaluation utilities **function, gradient,** etc., may appear to work correctly but quite likely produce incorrect results if the program contains quadratures and/or branches conditioned on the arguments or their descendents in the data flow sense. The crux of the problem lies in the fact that the tape records only the operations that are executed during one evaluation of the function. It also has no way to evaluate integrals since the corresponding quadratures are never recorded on the tape.

It appears unsatisfactory that, for example a simple table lookup of some physical property forces the rerecording of a possibly much larger calculation. However, the basic philosophy of ADOL-C is to overload arithmetic, rather than to generate a new program with jumps between 'instructions', which would destroy the strictly sequential tape access and require the infusion of substantial compiler technology. Therefore, we introduced the two constructs of conditional assignments and active integers as partial remedies to the branching problem.

In many cases, the functionality of branches can be replaced by conditional assignments. For this purpose, we provide a special function called **condassign(a,b,c,d)**. Its calling sequence corresponds to the syntax of the conditional assignment $a = b\ ?\ c : d$, which C++ inherited from C. However, here the arguments are restricted to be active or passive scalar arguments, and all expression arguments are evaluated before the test on b, which is different from the usual conditional assignment or the code segment.

Suppose the original program contains the code segment

```
if (condition)
res = arg1;
else
res = arg2;
```

Here, only one of the expressions (or, more generally, program block) **arg1** and **arg2** is evaluated, which exactly constitutes the problem for ADOL-C. To obtain the correct value **res** with ADOL-C, one may first execute both branches and then pick either **arg1** or **arg2** using **condassign(a,b,c,d)**. To maintain consistency with the original code, one has to make sure that the two branches do not have any side effects that can interfere with each other or may be important for subsequent calculations. The header file **adouble.h** contains a definition of **condassign(a,b,c,d)** and **condassign(a,b,c)** for four/three passive arguments so that the modified code without any differentiation can be tested for correctness. If there is no *else* part in a conditional assignment, one may call the three argument version **condassign(a,b,c)**, which is logically equivalent to **condassign(a,b,c,a)** in that nothing happens if **b** is nonpositive.

A.2 Active Subscripts

In many important procedures such as table lookup or numerical pivoting, the result of a conditional assignment is not a real variable but an integer that is subsequently used as an index into an array of active reals. For that purpose we have introduced the class **along** of active integers, which are implemented as a derived class of active doubles, so that all arithmetic operations involving them are recorded on the tape. The key functionality is that of subscripting; that is for an **along j** the expressions

a[j] with **a** an **adoublev**
A[j] with **A** an **adoublem**

are considered and recorded as a binary operation between **a** or **A** and **j**. The resulting **a[j]** and **A[j]** behave like variables of type **adouble** and **adoublev** except that they may not occur as arguments of the operators <<= and >>=.

Using the conditional assignment of active integers, one can, for example, fully record a function that involves Gaussian elimination with pivoting on a tape. In that case the recoding effort is minimal, and there is not much overhead at run time, either.

A.3 Example Gaussian Elimination

The following example uses conditional assignments as well as active subscripts to illustrate the correct usage of these extended features of ADOL-C. In this example the elimination is performed with column pivots.

```
void gausselim(int n, adoublem& A, adoublev& bv)
{
  along i;
  adoublev temp(n);
  adouble r,rj,temps;
  int j,k,ik;
  for (k=0; k < n; k++) /* elimination loop */
  {
    i = k;
    r = fabs(A[k][k]); /* initial pivot size */
    for (j=k+1; j<n; j++)
    {
      rj = fabs(A[j][k]);           /* look for a larger element */
      condassign(i,(rj >r),j);      /* in the same column */
      condassign(r,(rj >r),rj);
    } // endfor
    temp = A[i]; /* switch rows */
    A[i] = A[k];
    A[k] = temp;
    temps = bv[i];
    bv[i]=bv[k];
    bv[k]=temps;
    if (!value(A[k][k]))
      exit(1); /* Matrix singular! */
    temps= A[k][k];
    A[k] = A[k]/temps;
    bv[k] = bv[k]/temps;
    for (j=k+1; j<n; j++)
    {
      temps= A[j][k];
      A[j] -= temps*A[k];
      bv[j] -= temps*bv[k];
    } // endfor
  } // endfor elimination loop
  temp=0.0;
  for(k=n-1; k >= 0; k--)
     temp[k] = (bv[k]-(A[k]*temp))/A[k][k];
  bv=temp;
  return;
} // end gausselim
```

This function can be called from any program that suitably initializes the components of **A** and **bv** as independents. The resulting tape can be used to solve any nonsingular linear system of the same size and to get the sensitivities of the solution with respect to the system matrix and the right hand side.

Acknowledgments

Parts of the ADOL-C source were developed by Jay Srinivasan, Chuck Tyner, and Duane Yoder. We are also indebted to Joe Chou (CADSI), George Corliss, Tom Epperly, Brad Karp, Koichi Kubota, Bob Olson, Marcela Rosemblun, Jeng Yen (CADSI), and the TOMS referees for helping in various ways with the development and documentation of ADOL-C.

References

[1] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):1–29, 1992.

[2] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. Preprint MCS–P348–0193, Argonne National Laboratory, 1993.

[3] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. Elsevier (North Holland), 1989.

[4] D. G. Cacuci. Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. *J. Math. Phys.*, 22(12):2794–2802, 1981.

[5] D. G. Cacuci. Sensitivity theory for nonlinear systems. II. Extension to additional classes of responses. *J. Math. Phys.*, 22(12):2803–2812, 1981.

[6] Bruce Christianson. Reverse accumulation and accurate rounding error estimates for taylor series. *Optimization Methods and Software*, pages 81-94, 1(92).

[7] Andreas Griewank. Direct calculation of Newton steps without accumulating Jacobians. In T. F. Coleman and Yuying Li, editors, *Large-Scale Numerical Optimization*, pages 115 – 137. SIAM, Philadelphia, Penna., 1990.

[8] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[9] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penna., 1991.

[10] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II,* Springer-Verlag, Berlin, 1991.

[11] Jim E. Horwedel, Brian A. Worley, E. M. Oblow, and F. G. Pin. GRESS version 1.0 users manual. Technical Memorandum ORNL/TM 10835, Oak Ridge National Laboratory, Oak Ridge, Tenn., 1988.

[12] G. Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150–165, June 1980.

[13] Koichi Kubota. PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penna., 251-262, 1991.

[14] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT (Nordisk Tidskrift for Informationsbehandling)*, 16(1):146–160, 1976.

[15] G. M. Ostrovskii, Yu. M. Volin, and W. W. Borisov. Über die Berechnung von Ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

[16] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[17] Louis B. Rall. Differentiation and generation of Taylor coefficients in Pascal-SC. In Ulrich W. Kulisch and Willard L. Miranker, editors, *A New Approach to Scientific Computation*, pages 291–309. Academic Press, New York, 1983.

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

[19] Olivier Talagrand and P. Courtier. Variational assimilation of meteorological observations with the adjoint vorticity equation – Part I. Theory. *Q. J. R. Meteorol. Soc.*, 113:1311–1328, 1987.

[20] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.

[21] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Ph.D/ thesis, Committee on Applied Mathematics, Harvard University, Cambridge, Mass., November 1974.