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

A. Griewank
D. Juedes
J. Srinivasan

CRPC-TR90073
December, 1990

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

ADOL-C

A package for the automatic differentiation of algorithms written in C/C++ [1]

Version 1.1, December 1990

Andreas Griewank[2]
David Juedes[3]
Jay Srinivasan[4]

Abstract

The C++ package ADOL-C described here facilitates the evaluation of first and higher derivative vectors of functions that are defined by computer programs in C. Under certain restrictions ADOL-C can be applied to C codes obtained from Fortran programs by the translator f2c distributed by AT&T Bell Laboratories. The resulting derivative evaluation routines may be called from C/C++, Fortran, or any other language that can be linked with C++. The use of the package requires the availability of the GNU compiler g++, the AT&T preprocessor cfront2.0., or one of its ports.

The numerical values of derivative vectors are obtained free of truncation errors at a small multiple of the run time and RAM requirement of the given function evaluation program. Derivative matrices can be computed column by column or row by row through repeated calls of the vector routines. Typically, the derivative calculations involve a substantial 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 ADOL-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 the areas of optimization, nonlinear equation solving, numerical 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 the intermediate variables symbolically, it is theoretically always possible to express the m dependent variables directly in terms of the n independent variables. However, the attempt typically results in unwieldy algebraic formulae, if it can be completed at all.

[1]This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 and the NSF cooperative agreement No. CCR-8809615.

[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

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

[4]Department of Computer Engineering, University of Illinois, Urbana, IL 61820

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 [12] at the University of Illinois from 1977 to 1980. Eventually he realized that 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 function (i.e., $m = 1$) can always be obtained for no more than five times the cost 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 Beda [1] and was later popularized by Rall [10]. It was noted in [5] that in general neither the row-by-row nor the column-by-column method is optimal for the calculation of Jacobians. However, the potentially more efficient alternatives require some combinatorial optimization and involve large data structures that are not necessarily accessed sequentially. Therefore, ADOL-C was written solely for the evaluation of derivative vectors (e.g., rows or columns of Jacobians). This restriction also simplifies parameter passing between subroutines and calls in different computer languages.

As it turns out, 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 [2],[3], weather forecasting [13] and neural networks [14] . The discrete analog used here was apparently first discovered in the early seventies by Linnainmaa [9] 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 two other precompilers for automatic differentiation, namely GRESS/ADGEN [6][5] and PADRE2 [8] [6] .

Following the work of Kedem [7] with the Fortran extension AUGMENT, Rall [11] implemented in 1985 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 only has to know C or Fortran. In the latter case the evaluation program must first be translated into C, which can be done with the converter f2c distributed by AT&T. 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

[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, e-mail: kubota@ae.keio.ac.jp

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 it is necessary that the whole execution trace of the original evaluation program be recorded. In ADOL-C this potentially very large data set is written first into a buffer array and later onto a file if the buffer is full or the user wants a permanent record of the execution trace. In either case we will only refer to the recorded data as the *tape*. The user may create several tapes on several named files. During subsequent derivative evaluations, tapes are always accessed strictly sequentially, so that they can be paged in and out to disk without execessive run time penalties. However, the execution is usually significantly faster, if the core memory is large enough to accommodate the whole tape as an array. Therefore, the user may cause the tape to be read back into memory if it can actually be accommodated in core after the function evaluation itself is completed. This device may speed the execution significantly when several sweeps are performed (e.g., for the evaluation of a Hessian). If written onto a file the tapes are self contained and can be utilized by other C or C++ programs.

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

ADOL-C was designed so that the user only has to make 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. Currently ADOL-C uses only one active type, called **adouble**, whose real part is of the standard type **double**. 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 of the *passive* types **double, float**, or **int**. There is no implicit type conversion from **adouble** to either of these passive types so that the failure to declare variables as active when they depend on other active variables will result in a compile-time error message. (*Actually, this is not true for g++ version 1.37.1, which apparently initializes to zero all* **doubles** *that are illegally assigned adouble values . This bug may lead to normally terminating executions with reasonbably looking, but numerically incorrect results. It has been fixed in version 1.37.2.*) 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 string "ad:" and then the real part of an **adouble** is added to the stream. Naturally, **adoubles** may be components of vectors, matrices, and other arrays, as well as members of structures or classes.

Since the type **adouble** has a nontrivial constructor the mere declaration of large **adou-**

ble arrays may take up considerable run time. In particular 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 swamped by the construction of the large **adouble** arrays. It is hoped that this undesirable effect will eventually be eliminated by the inclusion of active vector and matrix types in ADOL-C. For the time being the user is advised to create dynamic arrays of adoubles by using the C++ operator **new[]** and to destroy them using **delete[]**. The remainder of this section can be skipped, unless the reader wishes to obtain some basic understanding of how the package works internally and tailor it according to his needs.

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 **userparms.h**. As long as the program execution never involves more than 64k 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 **adoubles** created by **malloc** or the C++ operator **new** in a last-in-first-out fashion. When an **adouble** goes out of scope or is explicitly deleted the destructor notices that its location 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 used and this tail is at least **mindeath=50** elements long.

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 **adoubles** are placed on the top so that vectors of **adoubles** 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 adoubles 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 exceed. In general, free-store **adoubles** should be deleted in a last-in-first-out fashion towards the end of the program block in which they were created. If this rule is obeyed, the maximal number of **adoubles** 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. 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 **adoubles** and corresponding operations.

2.2 Marking Active Sections

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

trace_on(rev) and **trace_off(pages)**

are recorded on a sequential data set called *tape*. Pairs of these function calls can appear anywhere in a C or C++ program, but they may not overlap. The optional integer arguments **rev** and **pages** will be discussed in Section 3. 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 files.

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 (pre-)compiled with their active variables declared as **adoubles** and with the header file **adouble.h** included. **Adoubles** 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 that free store adoubles allocated within an active section be deleted before its completion. The values of all **adoubles** 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

x $\ll=$ px // px of any passive numeric type.

This special initialization ensures that $value(x) = px$ and it must 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

y $\gg=$ py // py of any passive type ,

which ensures that $py = value(y)$ and may not be succeeded by any other assignment to y. However, a dependent y may have been assigned other real values previously, and it could even be independent 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 does matter 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 typical example let us consider a C function of the form

```
void eval(n,m,x,y,k,z)
int n,m;                 // number of independents and dependents
double x[];              // independent variable vector
double y[];              // dependent variable vector
int k[5];                // integer parameters
double z[];              // parameter vector
{                        // beginning of function body
double t;                // local variable declaration
t = z[0]*x[0];           // begin crunch
............             // continue crunch
............             // continue crunch
y[m-1] = t/m;            // end crunch
}                        // end of function
```

If **eval** is to be called from within an active C section with x and y as vectors of **adoubles** but 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 files **adouble.h** and **adutils.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 Fortran77, 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 create the following modified program:

```
void eval(n,m,px,py,z)
int n,m;                   // number of independents and dependents
double px[] ;              // independent passive variable vector
double py[] ;              // dependent passive variable vector
double z[] ;               // parameter vector
{                          // beginning of function body
trace_on();                // 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] ;            // select independent variables
adouble t;                  // local variable declaration
t = z[0]*x[0];              // begin crunch as before
............                // continue crunch
............                // continue crunch
y[m-1] = t/m                // end crunch as before
for( int j=0; j<m;j++)
y[j] >>= yp[j] ;            // select dependent variables
delete[n] x;                // destruct independent active variables
delete[m] y;                // destruct dependent active variables
trace_off();                // complete tape on adoltape
}                           // end of function
```

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.

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 one of their arguments is an active variable. An **adouble** x occurring in a comparison operator is effectively replaced by its real value $val(x)$. Most functions contained in the ANSI C standard for the math library are overloaded for active arguments. The only exceptions are the nondifferentiable functions **atan2, fmod, and modf**. Otherwise, legal C code in active sections can remain completed unchanged, provided the direct output of active variables is avoided. Whenever derivatives are undefined or discontinuous, the limit of the derivative values at arguments is to the right of the exceptional point. For example, at $x = 0$ 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 $pow(x, y) = x^y$ is computed as $exp[y(log x)]$ and thus undefined when x is nonpositive. 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 off for systems on which the double valued version is not available.

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 here are implicit functions and quadratures, or, more generally, 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 5.2.

2.6 Step by Step Modification Procedure

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

1. Use the statements **trace_on()** [or **trace_on(rev)**] and **trace_off()** [or **trace_off(pages)**] to mark the beginning and end of the active section.
2. Select the set of active variables, and change their type from **double** or **float** to **adouble.**
3. Select a set of independent variables, and initialize them with $\ll=$ assignments from passive variables.
4. Select a set of dependent variables among the active variables, and pass their final values to passive variable by $\gg=$ assignments.
5. Compile the codes after including the header files **adouble.h** and **adutils.h.**

Typically the first compilation will detect several type conflicts — usually attempts to convert from *adoubles* to passive variables or to perform standard I/O of active variables. Some C++ compiler may also disallow certain **gotos**, which are legal in ANSI C, but this problem is unrelated to ADOL-C. A minor nuisance is that older compilers and preprocessors require functions to be declared as overloaded (e.g., by the statement **overload sin**) before they are defined for the first time. The current versions of **g++** and **cfront2.0** automatically regard all functions as overloaded and on some systems the header file **math.h** declares the standard math functions as overloaded. Whatever the case may be, the use may have to add overload statements in the file **adouble.h** if they are missing or ignore compiler warnings if they are repeated in one of the header files.

3 Naming the Tapefile and Controlling the Buffer

The trace generated by the execution of an active section may stay within an internal *tape array* or be written out to a named *tape file*. Either may subsequently be used to evaluate the underlying function and its derivatives at the original point or alternative arguments. If the active section involves user-defined quadratures or branches conditioned on adouble comparisons, it must be executed and retaped at each new argument. Otherwise, direct evaluation from the tape by the routine **func_eval** (Section 4.2) may be faster, but this is not certain.

While several tape files may be generated and kept simultaneously, the current implementation utilizes only one tape array, which is hidden from the user. The tape array is used as a buffer for the tape file if the length of the tape exceeds the array length of **bufsize** bytes. Since operations on the tape array avoid any costly disk I/O, tape files that contain no more than **maxbuf** bytes are automatically read back into core when they are needed. The parameter **maxbuf** and the smaller parameter **bufsize** are defined in the header file **adouble.h** and may be adjusted by the user. For simple usage, **trace_on** and **trace_off** may be called without argument, and the special file **adolbox** need not be initialized. In that case, the user may skip the remainder of this subsection.

The optional integer argument **rev** of **tape_on** determines whether the numerical values of all **adoubles** are recorded on an unnamed temporary file when they are overwritten or go out of scope. This option takes effect if **rev** =**1** and prepares the scene for an immediately following gradient evaluation by a call to the routine **reverse** (see Section 3.2). Alternatively, gradients may be evaluated by a call to **grad_eval**, which includes a preparatory forward sweep for the creation of the temporary file. If omitted, the argument **rev** defaults internally to **0**, so that no temporary file is generated.

By setting the optional integer argument **pages** of **trace_off** to **1**, the user may force a named tape file to be written even if the tape array (buffer) does not overflow. On return from **trace_off**, its argument represents the number of times the buffer array has been emptied onto the tape file. If the argument **pages** is omitted, it is internally set to its default value zero, so that the buffer is written onto a file only if the total length of the tape exceeds **bufsize** bytes.

After the execution of an active section, the name of the corresponding tape file can always be found in the file **adolbox**. If this fixed file exists and contains a nonempty string of no more that 20 characters upon entry into an active section, then **trace-on** will create a tape file of that name. Otherwise **trace_on** will use the default name **adoltape** and place it into adolbox, after creating a file with that name if necessary. Later, the problem independent routines **forward, reverse, func_eval, grad_eval, hess_eval**, and **deriv_eval** always look in adolbox for the tape on which their respective computational task is to be performed. By placing different file names into **adolbox** 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 possible 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 the strings **tape1** and **tape2** placed into **adolbox** at the beginning of the respective active sections and calling **trace_off** each time with a positive argument. 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 **grad_eval** with the appropriate name **tapel** or **tape2** placed into **adolbox**.

When a tape file of the name specified in **adolbox** does already exist upon entry of **trace_on**, it is at first merely flagged as *disabled* and actually overwritten only when the buffer must be emptied. Similarly, the tape file is disabled when it has been read back into core because its length is less than **maxbuf**. Messages announcing these and some other events are printed provided the integer parameter **npr** is defined positive in the header file **usrparms.h**. The status of a tape file is encoded in the sign of the integer **pages** represented by the first ten characters. If this number is zero, nothing has been written onto the tape file. If **pages** is positive, it represents the number of times the buffer was emptied onto the file. To disable the tape file, ADOL-C makes the sign of **pages** negative without altering its magnitude. To reactivate the tape file the user may simply edit it to make page positive again. Whenever the tape file pointed to by **adolbox** is found to have a nonpositive page value, the derivative evaluation routines assume that the relevant trace can be found in the internal tape array.

3.1 Examining the Tape and Predicting Storage Requirements.

At any point in the program one may call the routine

void tapestats(pages,length,oper,indep,depend,bufsiz,maxlive,deaths)

all of whose arguments are integer references and can therefore be called from Fortran. The routine **tapestats** looks in **adolbox** for the name of a tape file and interprets the first ten characters in the tape file as an integer. If this number is positive, it is assumed that the tape file does indeed contain the execution trace of interest, and its relevant characteristics are read from its first 80 characters. **Pages** represents the number of times the buffer of size **bufsiz** was emptied. **Length** represents the total length of the tape in bytes. **Oper** represents the number of arithmetic operations, assignments and elementary function calls recorded (including some *death notices*). **Indep, depend** represent the number of independent and dependent variables. **Maxlive** represents the largest number of live variables alive at any one time, and **deaths** represents the total number of overwrites and deaths occurring within the active section.

The last two numbers determine the temporary storage requirements during calls to the work horses **forward** and **reverse**. For a certain degree **deg** >= 0 the routine forward involves (apart from the tape) an array of **(degree+1)*maxlive** doubles in core and, in addition, a sequential data set of **deaths*(degree+1) revreals** if called with the option **rev= 1**. 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** will yield gradients and other adjoint vectors only in single-precision accuracy. This may well be acceptable if the adjoint vectors represent rows of a Jacobian that is used for the calculation of Newton steps. The routine **reverse** involves the same number of **doubles** and twice as many **revreals** as **forward**.

3.2 Customizing ADOL-C

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

bufsize (default: 1024) 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_on** 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.

maxbuf (default: 1048576) This integer must be greater or equal to **bufsize** and represents an upper bound on the size of temporary arrays actually used during forward and reverse sweeps. If the tape length is greater than **bufsize** but smaller than **maxbuf**, it will be read back into core by **forward** for subsequent sweeps. The same mechanism applies to the temporary file of *death values* and its buffer. It may make sense to make **maxbuf** much larger than **bufsize** if the program containing the active section involves much core memory that is not needed during subsequent calls to the derivative evaluation routines.

NaN (default: 0.0/0.0) This double value is used as default initialization for all **adoubles** as well as their derivatives. On machines without IEEE arithmetic, **NaN** must be reset to a some number that does not cause an arithmetic exception. If final results contain **NaN**s or are in any way dependent on an alternative value chosen for **NaN**, the calculation is of course incorrect.

locint (defaults: int) The range of the integer type **locint** determines how many **adouble** s can be simultaneously alive. Only in extreme cases should it be necessary to keep **locint** as **long int**, because there are more than 65,535 **adoubles** alive at any one time. Otherwise the length of the tape can be almost halved by redefining **locint** as **unsigned short**, unless doubles must be alligned on quadruple bytes.

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.

skip2 (default: see in usrparms.h) This macro is needed only when the machine requires the alignment of **doubles** and **ints** on quadruple bytes in character arrays. Otherwise the padding macro **skip2** can be redefined as the empty string to save some storage.

npr (default: 1) If this integer is positive, ADOL-C prints out some basic messages about its progress.

tape (default: thisuglymessthatcantbused) This is the name of the tape array, which may not be used for other purposes.

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

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

4 Evaluating Derivatives from a Tape

4.1 Mathematical Description

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 initial values x of the independent variables. Provided no arithmetic exceptions occurred and all special functions were evaluated in the interior of their domains, the relation $y = F(x)$ is in fact analytic. In other words, we can compute arbitrarily high derivatives of the vector function $F(x)$. More specifically, the ADOL-C functions **forward** and **reverse** yield derivative vectors of the form

$$F^{(d)}(x)\underbrace{v\ v\ \ldots\ v}_{d\ times} \quad \equiv \quad \frac{\partial^d}{\partial t^d}F(x+t\,v)\Big|_{t=0} \quad \in \mathbf{R}^m \tag{1}$$

and

$$u^T F^{(d+1)}(x)\underbrace{v\ v\ \ldots\ v}_{d\ times} \quad \equiv \quad \nabla_x\, u^T F^{(d)}(x) v^d \quad \in \mathbf{R}^n\ , \tag{2}$$

where $v \in \mathbf{R}^n$ and $u \in \mathbf{R}^m$ represent *weight vectors* that are held constant with respect to differentiation. By choosing a suitable *tangent vector* v and a *Lagrange vector* u, one may compute any derivative of f. In particular, (1) represents the th column of the Jacobian $F'(x)$ if $d = 1$ and v is the th Cartesian basis vector in $\mathbf{R}^n$. Similarly (2) yields the i-th row of the Jacobian $F'(x)$ if $d = 0$ and u is the i-th Cartesian basis vector in $\mathbf{R}^m$. When $d = 0$ the tangent v is redundant, and (1) reduces simply to the original vector function.

For a scalar function F (i.e., m=1), one finds that with $u = 1 \in \mathbf{R}$ the second form (2) represents the gradient $\nabla F(x)$ if $d = 0$ and the th column of the Hessian $\nabla^2 f(x)$ if $d = 1$ and v is the th Cartesian basis vector. 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 as a vector of Lagrange multiplier estimates such that approximately $u^T F'(x) = 0$ with the first component normalized to 1. Then the second vector (2) represents for $d = 1$ the Hessian of the Lagrangian function $u^T F(x)$ multiplied by some vector v.

In general, one may use the first form (1) to calculate the d-th derivative of $u^T F(x)$ in the direction v and then obtain the gradient of that quantity with respect to all independent variables in the form (2). Mixed partial derivatives (e.g., $u^T F'' v_1 v_2$) and their gradients, can be obtained by interpolating the corresponding pure values for $v = v_1$ and $v = v_2$.

Since the package works in terms of Taylor coefficients, the vectors (1) and (2) are for $d > 1$ scaled by $1/d!$ so that the actual output vectors are

$$r \equiv \tfrac{1}{d!}F^{(d)}(x)v^d \in \mathbf{R}^m \quad and \quad g \equiv \tfrac{1}{d!}u^T F^{(d+1)}(x)v^d \in \mathbf{R}^n \quad . \tag{3}$$

This scaling simplifies the internal calculations slightly and reduces the danger of overflow, while increasing the likelihood of underflow.

4.2 Forward and Reverse Calls

Given any correct tape, one may call from within the generating program, or subsequently during another run, the C function

```
void forward(y,rev,deg,x,v)
double y[];                    // results as in (3) for d up to deg
int rev ;                      // flag for reverse sweep
int deg ;                      // highest derivative degree
double x[] ;                   // independent variable values
double v[] ;                   // tangent vector in domain
```

The components of the vectors x and v must correspond to the independent variables in the order of their initialization by the $\ll=$ operator. If **deg**= 0, **forward** merely evaluates $y = F(x)$, and the last argument v may be omitted or set to the null pointer in the call. If **deg**> 0 and there is only one independent variable, then the argument v may also be omitted as it defaults internally to the unit vector with one component. The one-dimensional array **y** contains the m vectors r defined in (3) for d **=deg,deg-1,...,1,0** (i.e., in reverse order). Thus the first n components of **y** contain the **deg**-th Taylor coefficient vector of F in the direction v, the next n components represent the **deg**-1st Taylor coefficient, and the last n components the function value $F(x)$ itself. Overall bf y has $n \cdot (deg + 1)$ components. The integer flag *rev* plays a similar role as in the call to **trace_on**, namely it determines whether **forward** writes the Taylor coefficients of all intermediate quantities onto a buffered temporary file in preparation for a subsequent reverse sweep.

Forward always looks in **adolbox** for the name of the file on which the tape was written. If the tape file is found empty or disabled, **forward** assumes that the relevant tape is still in core and reads from the buffer. **Forward** can be used to evaluate the vector-function F at arguments x other than the point at which the tape was created, provided the original code involves neither user-defined quadratures nor conditional branches. If these conditions 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 or a call to **forward** with **rev** = **1** in either case, one may call the C function

```
void reverse(xbar,deg,u)
double g[] ;  //results as in (3) for d up to deg
int deg ;     // highest derivative degree
double u[] ;  // domain weight vector
```

The degree *deg* must agree with the corresponding parameter of the most recent call to forward or 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. Except when $d = 0$, the value of v used in the last forward call will enter into the value of **g**. If there is only one dependent variable, the weight vector u may be omitted and defaults internally to 1. Similarly to **y** the one-dimensional array **xbar** contains the vectors g defined in (3) for $d = deg, deg - 1, \ldots, 0$ with the last n components representing the generalized gradient $\nabla f(x)^T u$. Thus the first n components of both one-dimensional arrays contain the Taylor coefficient vectors of highest degree.

With **ej** and **ei** arrays containing the j-th and i-th Cartesian basis vector in $\mathbf{R}^m$ and $\mathbf{R}^n$, respectively, one may use in particular the calls

```
1. forward(r,0,1,x,ej)          // Compute j-th column of Jacobian

2. reverse(g,0,ei)              // Compute i-th row of Jacobian

3. forward(r,1,1,x,ej)
   reverse(g,1)                 // Yields j-th column of Hessian when m=1
```

Calls of the second form 2. must be preceded by the call **forward(r,1,0,x)** or a taping with **rev 1** equal to 1, but may then be repeated for several vectors *ei*. In contrast, the two calls in 3. must follow each other for every j.

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

```
void func_eval(r,x)
double r[];          // result as in (3) for d=0
double x[];          // independent vector value


void grad_eval(r,g,x,u)
double r[];          // result as in 1 for d=0
double g[];          // result as in 3 for d=0
double x[];          // independent vector value
double u[] ;         // range weight vector for m>1
```

```
void hess_eval(g,x,v,u)
double g[];              //result as in (3) for d=1,0
double x[] ;             // independent variable values
double v[] ;             // tangent vector in domain
double u[] ;             // range weight vector for m>1
```

In the last call, the one-dimensional array **g** must have at least $2n$ components. For higher derivatives, one may use the driver

```
void deriv_eval(y,xbar,deg,x,v,u)
double y[];              //result as in (1) for d≤ deg
double xbar[];           // result as in (1) for d up to deg
int deg ;                // highest derivative degree
double x[] ;             // independent variable values
double v[] ;             // tangent vector in domain
double u[] ;             // range weight vector for m>1
```

The components of the arrays **y** and **xbar** in the call to **deriv_eval** contain each **deg+1** Taylor coefficient vectors exactly in the same way as in the corresponding calls to **forward** and **reverse**. The last three drivers create a temporary file, initialize it with an appropriate call to forward, and then call reverse with the corresponding argument. In particular, when called repeatedly for different weights u but at the same point x, the routine **grad_eval** is less efficient than a direct call to reverse. Again, the vectors v and u may be omitted and default to 1 if n or m equal one, respectively. When $m = 1$ and the original evaluation code contains neither quadratures nor branches, then **grad_eval** can be used to simultaneously evaluate the scalar function and its gradient at any argument in its domain.

All routines described above have only integer and real scalar or vector arguments. Therefore they are callable from Fortran and other languages for most linkers.

5 Installing and Using ADOL-C

The ADOL-C package consists of the following six C++ modules and six header files.

```
adouble.c      forward.c      adouble.h      templates.h
taputil.c      reverse.c      adutils.h      taputil.h
tayutil.c      drivers.c      usrparms.h     opcodes.h
```

The six modules have basically the following functions: **adouble.c** controls the nominal allocation and elementary operations for variables of the class **adouble** defined in **adouble.h**. The file **taputil.c** contains all functions and variables needed for the taping of arithmetic operations and function evaluations in an active section. The module **tayutil.c** controls the storage and retrieval of Taylor series coefficients during forward and reverse sweeps, respectively. The files **forward.c** and **reverse.c** consist of the core procedures **forward** and **reverse** as well as their auxiliary routines. The file **drivers.c** contains various drivers that call **forward** and **reverse**; it is the natural place for additional user-defined drivers and utilities. The user may modify the header file **usrparms.h** in order to tailor the package to his needs in the particular system environment das discussed in Section 3.2. On a UNIX system one may use the following makefile to generate the library **libad.a** using the GNU compiler g++.

```
CFLAGS = -g
LIB =
CC = g++
cc = g++
lib: adouble.o taputil.o forward.o reverse.o tayutil.o drivers.o
ranlib libad.a
@echo 'Library created'
adouble.o: adouble.c adouble.h opcode.h taputil.h usrparms.h
$(CC) -c $(CFLAGS) $(LIB) adouble.c
ar rcv libad.a adouble.o
taputil.o: taputil.c opcode.h template.h taputil.h usrparms.h
$(CC) -c $(CFLAGS) $(LIB) taputil.c
ar rcv libad.a taputil.o
forward.o: forward.c adutils.h  template.h usrparms.h
$(cc) -c $(CFLAGS) $(LIB) forward.c
ar rcv libad.a forward.o
reverse.o: reverse.c adutils.h template.h usrparms.h
$(cc) -c $(CFLAGS) $(LIB) reverse.c
ar rcv libad.a reverse.o
tayutil.o: tayutil.c usrparms.h
$(CC) -c $(CFLAGS) $(LIB) tayutil.c
ar rcv libad.a tayutil.o
drivers.o: drivers.c adutils.h
$(CC) -c $(CFLAGS) $(LIB) drivers.c
ar rcv libad.a drivers.o
```

The user has to ensure that a suitable compiler and its corresponding libraries are in the path.

5.1 Compiling and Linking C++ Programs with Active Sections

To compile a C++ program that involves variables of type **adouble** one must add the directive **#include <adouble.h>** at the beginning of the program file. Programs that mark an active section with **trace_on** and **trace_off** or call on the various derivative evaluation routines must also 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. For example, the scalar problem discussed in the following section was compiled and linked by the UNIX makefile

```
AD = /Net/albireo/albireo1/griewank/adolc11
CFLAG = -g -I\$(AD)
LFLAG = -L\$(AD)
scalexam : scalexam.o \$(AD)/libad.a
g++ -o scalexam scalexam.o \$(LFLAG) -lad -lm -lg++
scalexam.o : scalexam.c \$(AD)/usrparms.h \$(AD)/adouble.h \$(AD)/adutils.h
g++ -c \$(CFLAG) scalexam.c
```

Please note that the directory that contains the ADOL-C include files **adouble.h** and **adutils.h** as well as **libad.a** will vary from system to system. The user should replace AD with the corresponding directory name in the above commands.

5.2 Interfacing ADOL-C with Fortran programs

Since many applications codes and numerical software packages are written in Fortran77 the question arises how ADOL-C can be used in connection with such codes. We consider two aspects of this problem: first the translation of subprograms that contain an active Section into C, and second the calling of the ADOL-C routines **forward, reverse etc.** from Fortran. This two-pronged approach is preferable to a translation of complete Fortran programs into C, which is usually not very practical and and makes the use of precompiled numerical packages more difficult. Since ADOL-C utilizes global structures of a nonstandard type, the main program must be written in C++ and the object files should be linked by the C or C++ loader. If the original main program is written in Fortran it may be transformed into a subroutine, say **fort_main_()**, which is called by a new **main()** program in C or C++. This can be achieved by adding the two lines

```
extern "C" void fort_main_();
main() { fort_main_(); };
```

to any one of the C++ files in use. Our own experiments were conducted using the Fortran to C converter **f2c**, which is distributed by AT&T Bell Laboratories over **netlib**[7]. This public domain software comes without any guarantees, but seems to be reasonably reliable. However, the generated C code may be hard to read and understand, especially if it involves significant I/O. Therefore, the authors of **f2c** recommend to maintain the Fortran code and to convert it repeatedly. For the purposes of ADOL-C this approach is not very convenient, because the retyping of variables, marking of active sections, etc. must be performed on the C code, and would have to be repeated after each new version. Therefore, we recommend to translate only the program parts that need to be turned into active sections, and then

[7] For information on **f2c** send the electronic mail message *Send index from f2c* to : netlib@research.att.com

to maintain them as **C++** codes. Provided, I/O is largely avoided in these routines, the corresponding f2c generated procedures may be quite readable and easy to maintain. The resulting object codes can than be linked with the remaining Fortran programs, which must be called from a C++ main as we already mentioned. In our experience the linker correctly identified common blocks in remaining Fortran routines with the corresponding structures generated by f2c in the converted part of the program.

Regarding the use of **f2c** generated code in connection with ADOL-C we offer the following definitely incomplete list of hints. Unless **f2c** is used with the **-C++** option, function declarations in the resulting C code will not conform with the more stringent prototype consistency requirement of recent **C++** compilers. If used with this option however, **f2c** encloses the whole file of generated C code in a conditional **extern "C"** declaration, which means that **C++** compilers treat it as plain **C** code. Therefore , the **extern "C"** declaration needs to be removed as active sections in the file require overloading. As a consequence of this removal, all functions defined in the file can no longer be called from C or Fortran programs as their names are embellished with the types of their arguments. If a complete program is converted this applies in particular to the function MAIN__, which is generated by **f2c** and called by a standard main routine in its library libF77. To avoid this difficuly one may simply define **main() { MAIN__(); };** in some C++ file. The same difficulty arises with the functions from **libm.a** and the two libraries **libF77** and **libI77**, which contain in particular the I/O routines used by **f2c.** As explained in the **f2c** documentation the source code for these libraries may be obtained from netlib and then compiled into a single library **libf2c**. Alternatively, one may sanitize the converted code from all **f2c** specific trappings and turn it into a standard C or C++ file. This requires in particular the expansion of macros and type definitions from the header file **f2c.h** and the rewriting of I/O operations in terms of C++ procedures. One may also take the opportunity to avoid fixed size arrays by dynamic storage allocation using **new[]** and **delete[]**, which may substantially reduce overhead in the case of active variables, as we mentioned in Subsection 2.1. Because **adoubles have a nonstandard constructor delete[] must be supplied with an index giving the number of array elements to be destroyed. It should also be noted that f2c decrements some array pointers by one (e.g., by the statement --x) and then uses integer subscripts starting from 1 (rather than 0), exactly as in the original Fortran. In those cases the pointer must be readjusted by the statement ++x before the dynamically allocated array can be deleted.**

In order to make a procedure callable from Fortran one has to ensure that all parameters are references to variables of a standard type. In particular this condition is met by our various utilities, e.g., **forward, reverse, grad_eval**, whose prototypes are listed in **adutils.h**. In order to allow their call from Fortran the source file **driver.c** contains externalized interface routines, whose names end in an extra underscore as expected by the f77 compiler.

Suppose all Fortran routines including are contained in a file ff.f and all C++ functions including the main() program are contained in a file fc.c. Some of the C++ functions may have been obtained from Fortran sources by the converter f2c and subsequently modified for automatic differentiation with ADOL-C. After the ff.f has been compiled to ff.o by f77 and fc.c to fc.o as described above, the two object files may be linked by the command

```
g++ -o foo ff.o fc.o -L$(AD) -lad -lF77 -lI77 -lm -lg++
```

The software package includes an example, where the Helmholtz energy function [4] coded in Fortran was converted to C using **f2c**, then simplified and overloaded with ADOL-C, and finally called from a Fortran test program. Subsequently, the test program directly calls on **reverse** to accumulate the gradient using the tapefile. Unfortunately, this route is somewhat circuitous, and it is hoped that a rewrite of ADOL-C into Fortran90 will eventually provide a more convenient access to derivatives of Fortran programs.

5.3 Adding Quadratures as Special Functions

Suppose an integral

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

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*, that 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*);

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

2. In the file **adouble.h** add the statements **overload integral;** (if needed) near the top and then in the definition of the class **adouble**

   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 arcus of the hyperbolic cosine we have **integral=acosh**, and *integrand* could be written as

$$\{ \textbf{val = sqrt(arg*arg-1)} \}$$

so that the line

$$\textbf{extend\_quad(acosh,val = sqrt(arg*arg-1))}$$

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

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

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

6 Three Examples Codes

The first example 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 derivatives can be computed using either **forward** or **reverse**, and the results are subsequently compared. Note the scaling of the derivatives in agreement with (3).

```
\#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.
main()                                   {
  int n;
  cout << "degree of monomial =? \n";
  cin >> n;
  double xp =0.5; double yp = NaN;
  adouble y,x;                       // Adoubles may be declared
  int rev=1;                         // before the beginning of
  trace_on(rev);                     // the active section !!!!!
  x <<= xp;                          // Only one independent var.
  y = power(x,n);
  y >>= yp;                          // Only one dependent adouble.
  trace_off();                       // No global adouble has died.
  double *res, *ypp;
  res = new double[n+2];
  ypp = new double[n+2];
  for( i=0; i < n+2; i++)      {
     forward(res,rev,i,&xp);
     cout << yp << "=?" << *res << " should be the same \n";
     reverse(ypp,i);
     yp = *ypp/(i+1);                }
   delete ypp; delete res;                 }
```

In the scalar case above, the reverse mode has no advantage in terms of complexity or accuracy, and both modes yield exactly the same numerical results. In contrast, the reverse mode comes into its own by calculating derivative vectors for the following product example.

```
\#include "adouble.h"
\#include "adutils.h"
\#include <stream.h>
\#define abs(x) ((x >= 0) ? (x) : -(x))
\#define maxabs(x,y) (((x)>abs(y)) ? (x) : abs(y))
main() {
int n,i,pager,length,oper,indep,depen,buf_size,maxlive,deaths;
cout << "number of independent variables = ?  \n";
cin >> n;
double yp ;               // Undifferentiated double code
double* xp;
xp = new double[n];
yp =1;
for(i=0;i<n;i++) {
  xp[i] = (i+1.0)/(2.0+i);
  yp *= xp[i];};
int rev=1;                // Overloaded adouble version
trace_on(rev);
```

```
adouble y ;
adouble* x;
x = new adouble[n] ;
y =1;
for(i=0;i<n;i++)        {
  x[i] <<= xp[i];
  y *= x[i];            }
double yout;
y >>= yout;
cout<< yout <<" =? "<<yp<<" function values should be the same \n";
delete[n] x;
cout <<"hello \n";
trace_off();              // Now read tape statistics
tapestats(pager,length,oper,indep,depen,buf_size,maxlive,deaths);
cout<<"pages        "<<pager<<"\n"; \\........ print others
double err; int degree; double r = yp;
double* z = new double[n];
double* g = new double[n];
double* h = new double[2*n];
for(int it=0; it <2; it++) //To come back later with perturbed xp
{deg =0;
if(it>0)                       // Forward sweep only at new argument
 { forward(&r,rev,deg,xp);     // (Lagrange multipliers) could be omitted
 cout<<r<<" = function value at perturbed point xp \n";}
                               // Reverse sweep to evaluate gradient
reverse(g,deg);                // Omitted last argument defaults to 1;
err=0;                         // Compare with deleted product
for(i =0;i<n;i++)  err = maxabs(err,xp[i]*g[i]/r - 1.0);
cout<< err  <<" = maximum relative errors in gradient \n";
// Combine previous two sweeps in gradient evaluation
grad_eval(&r,z,xp);            // Last argument lagrange is omitted
err = 0;                       // Compare with previous numerical result
for( i=0; i<n; i++) err =  maxabs(err,g[i]/z[i] - 1.0);
cout << err <<" = gradient error should be exactly zero \n";
for(i=0;i<n;i++) z[i] = 0.0 ; // Compute first row of Hessian
z[0]=1.0;
hess_eval(h,xp,z);             // Computes Hessian times direction z
err = abs(h[0]);               // Compare with doubly deleted product
for(i=1;i<n;i++) err = maxabs(err,xp[0]*h[i]/g[i]-1.0);
cout<< err  <<" = maximum relative error in Hessian column \n";
double h1n = h[n-1];           // Check for symmetry
z[0]=0; z[n-1]=1;
hess_eval(h,xp,z);             // Compute Hessian times alternate z
cout<<h1n<<" =? "<<h[0]<<" (1,n) and (n,1) entry should be the same\n";
z[0]=1.0;                      // Evaluate third directional derivatives
deg = 2; double rv[3]; double* t;
t = new double[3*n];
deriv_eval(rv,t,deg,xp,z);
cout<<r<<" =? "<<h1n<<" second partials should be the same \n";
err = abs(t[0])+abs(t[n-1]);
                                          // Compare with triply deleted product
for(i=1;i<n-1;i++) err = maxabs(err,xp[0]*t[i]/h[i]-1.0);
cout<<err<<" =maximum relative error in tensor times vector^2 \n";
for(i=1; i<n; i++) xp[i] =sqrt(xp[i]);}   // Change argument and repeat
delete h; delete g; delete z; delete t; delete xp;}
```

By comparing the results of this program when the library is made with **revreal** set to **double** and **float**, one can verify the loss of derivative accuracy caused by the second choice. It should be noticed that the run time for all routines called in the program above is proportional to n and thus of the same order as the time for evaluating the product itself.

Finally, 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 "adouble.h"
\#include "adutils.h"
\#include <stream.h>
adouble** A;
int n;                                       \\ A is an n x n matrix.
adouble det(int k, int m)                 {  \\ k <= n is the order
  if(m == 0 ) return 1.0 ;                   \\ of the submatrix,
    else                              {      \\ its column indices
    adouble* pt = A[k-1];                    \\ are encoded in m.
    adouble t =0 ;
    int p =1;
    int s;
    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);
          s = -s;                  }
       ++pt;
       p = p1;                      }
    return t;                         }  }
main()                                   {
   cout << "order of matrix = ? \n";          \\ Select matrix size
   cin >> n;
A = new adouble*[n];
double diag;
diag = 0;
int rev=1;
trace_on(rev);
for (int i=0; i<n; i++)     {
    m *=2;
    A[i] = new adouble[n];
    adouble* pt = A[i];
    for (int j=0;j<n; j++) *pt++ <<= j/(1.0+i);
    diag += val(A[i][i]);
    A[i][i] += 1.0;          }
 diag += 1;
 adouble deter;
 deter = det(n,m-1);
 double detout;
 deter >>= detout;
 printf("\n %f =? %f should be the same \n",detout,diag);
 trace_off();               // because the matrix is a rank one
 int n2 =n*n;               // perturbation of the identity matrix.
 double* B;
 B = new double[n2];
 int deg = 0;
 reverse(B,deg);
 cout <<" \n first base? : ";
 for (i=0;i<n;i++)          {
   adouble sum = 0;
   adouble* pt;
   pt = A[i];
   for (int j=0;j<n;j++)
      sum += (*pt++)*B[j];
   cout << val(sum) <<" ";  }       }
```

The storage requirement of this code indeed grows very rapidly as can be seen from the following table.

n	1	2	3	4	5	6	7
tape	172	444	1320	4936	23948	141928	990736
taylor	128	352	1008	3600	16848	98784	687504

The entries represent the length of the tape and the auxiliary numerical file in bytes. The run time is proportional to the length of these sequentially accessed data sets. The run for the 7×7 case took 4.5 seconds CPU time on a Sun4 Sparc Station.

Acknowledgments

The authors are indebted to Brad Karp, Koichi Kubota, Bob Olson, and Marcela Rosemblun for helping in various ways with the development of the code.

References

[1] L. M. Beda et al (1959). *Programs for Automatic Differentiation for the Machine BESM*, Inst. Precise Mechanics and Computation Techniques, Academy of Science, Moscow.

[2] D. G. Cacuci (1981). *Sensitivity Theory for Nonlinear Systems. I. Nonlinear Functional Analysis Approach*, Journal of Mathematical Physics, Vol. 22, No. 12, pp. 2794-2802.

[3] D. G. Cacuci (1981). *Sensitivity Theory for Nonlinear Systems. II. Extension to Additional Classes of Responses*, Journal of Mathematical Physics, Vol.22, No.12, pp. 2803-2812.

[4] A. Griewank (1989). "On Automatic Differentiation," in *Mathematical Programming: Recent Developments and Applications,* ed. M. Iri and K. Tanabe, Kluwer Academic Publishers, pp. 83–108.

[5] A. Griewank (1990). *Direct Calculation of Newton Steps without Accumulating Jacobians.* Preprint MCS-P132-0290 Mathematics and Computer Science Division, Argonne National Laboratory.

[6] J. E. Horwedel, B. A. Worley, E. M. Oblow, and F. G. Pin (1988). *GRESS Version 0.0 Users Manual*, ORNL/TM 10835 , Oak Ridge National Laboratory, Oak Ridge, Tenn.

[7] G. Kedem (1980). *Automatic Differentiation of Computer Programs*, ACM TOMS, Vol. 6, No. 2, pp. 150-165.

[8] K. Kubota and M. Iri (1990). *Padre 2, version 1—User's Manual*, Research Memorandum RMI 90-01, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo.

[9] S. Linnainmaa (1976). *Taylor expansion of the accumulated rounding error.* BIT, Vol. 16, pp. 146-160.

[10] L. B. Rall (1981). *Automatic Differentiation - Techniques and Applications*, Springer Lecture Notes in Computer Science, Vol. 120 , Springer Verlag, Berlin.

[11] L.B. Rall (1984). "Differentiation in PASCAL-SC: Type GRADIENT", *ACM TOMS* Vol.10,pp.161-184.

[12] B. Speelpenning (1980). *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana.

[13] O. Talagrand and P. Courtier (1987). *Variational assimilation of meteorological observations with the adjoint vorticity equation. I: Theory* Q.J.R. Meteorological Society, Vol. 113, pp. 1311-1328.

[14] P. Werbos (1982). *Applications of Advances in Nonlinear Sensitivity Analysis* In R. Drenick and F. Kozin (eds). Systems Modeling and Optimization, Proceedings of the 10th IFIP Conference, Springer Verlag, New York, pp. 762-777.