ADOL-C: [1]

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

Version 1.8, December 1998

Andreas Griewank[2], David Juedes[3], Hristo Mitev[2], Jean Utke[4], Olaf Vogel[2], and Andrea Walther[2]

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. For explicitly or implicitly defined functions derivative tensors are obtained with a complexity that grows only quadratically in their degree. The derivative calculations involve a possibly substantial but always predictable amount of data that are accessed strictly sequentially and are therefore automatically paged out to external files.

Keywords: Computational Differentiation, Automatic Differentiation, Chain Rule, Overloading, Taylor Coefficients, Gradients, Hessians, Forward Mode, Reverse Mode, Implicit Function Differentiation, Inverse Function Differentiation

Abbreviated title: Automatic differentiation by overloading in C++

---

[1]The development of earlier versions was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38 and the NSF cooperative agreement No. CCR-8809615. During the development of the current version the co-authors Mitev, Vogel, and Walther were supported by the Grants Gr 705/5-1 and Gr 705/4-2,-3 of the Deutsche Forschungsgemeinschaft

[2]Institute of Scientific Computing, Technical University Dresden, D-01062 Germany

[3]Department of Computer Science, Ohio University, Athens, OH 45701, USA

[4]Framework Technologies, Inc., 10 South Riverside Plaza, Suite 1800D-01062, Chicago, IL 60606, USA

Contents

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 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 formulae, 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 [20] 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 independent 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 [22] and was later popularized by Rall [18]. It was noted in [8] 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 [11]. 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 from 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 [5],[6], weather forecasting [21], and neural networks [23]. The discrete analog used here was apparently first discovered in the early seventies by Ostrovskii et al. [17] and Linnainmaa [16] 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 five other precompilers for automatic differentiation, namely ADIFOR[5], GRESS/ADGEN[6], ODYSSEE[7], PADRE2[8], and TAMC[9].

Following the work of Kedem [14] with the Fortran preprocessor AUGMENT, Rall [19] 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**++. As starting points to retrieve further information on techniques and application of automatic differentiation, as well on other overloading based packages like AD01[10] and IMAS[11], we refer to the conference proceedings [10] and [2] and the web page `http://www.mcs.anl.gov/autodiff/AD_Tools/index.html`.

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 [9]. 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 manual 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

[5] Contact: Paul Hovland, ANL-MCS, USA, e-mail: hovland@mcs.anl.gov
[6] Contact: Jim Horwedel, ORNL, USA, e-mail: jqh@ornl.gov
[7] Contact: Christele Faure, INRIA Sophia-Antipolis, France, e-mail: odyssee@sophia.inria.fr
[8] Contact: Kiochi Kubota, CHUO University, Japan, e-mail: kubota@ise.chuo-u.ac.jp
[9] Contact: Ralf Giering, Max-Planck-Intitut für Meteorologie, Germany, e-mail: giering@dkrz.de
[10] Contact: John Reid, Atlas Centre, Rutherford Appleton Laboratory, England, e-mail: jkr@rl.ac.uk
[11] Contact: Andreas Rhodin, GKSS Research Center, Germany, e-mail: rhodin@gkss.de

mathematical characterization of ADOL-C. The basic derivative evaluation routines are explained in Section 6. Descriptions of the calling sequences of more convenient derivative evaluation routines are contained in Section 7. This section covers also the detection of sparsity using ADOL-C. Section 8 details the installation and use of the ADOL-C package. Finally, Section 9 furnishes some 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. The user should simply refer to them when the more abstract and general descriptions of ADOL-C provided in this document do not suffice.

From the users point of view Version 1.8 is almost identical to Version 1.7, which was released in September 1996. There are quite a few new drivers, including some for the detection of sparsity and the evaluation of higher order tensors of explicit or implicit functions. With the exception of **hos_forward** all formerly existing ADOL-C functions have the same calling sequences. While keeping the old header files `adutils.h` and `adutilsc.h` for compatibility reasons Version 1.8 provides a more structured hierarchy of header files supplying the user with the interfaces to all ADOL-C functions (see Section 8.3). The library sources have been completely reorganized and the taping mechanism was altered so that tapes generated with Version 1.7 are no longer interpretable. New tapes will be stamped and old tapes rejected by Version 1.8. Compatibility with the Fortran version ADOL-F developed by Dima Shiriaev in 1996 has not been maintained as the future of that project is currently uncertain. Internal changes to ADOL-C achieve reductions in tape sizes and run times of up to about 25 percent compared to Version 1.7.

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 $\ll$ is overloaded such that first the real part of an **adouble** and then the string "(a)" is added to the stream. The input stream operator $\gg$ can be used to assign a constant value to an **adouble**. 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 most 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 value of **tag**. Both functions **trace_on** and **trace_off** are prototyped in the header file `taputil.h`, which is included by the header `adouble.h` automatically.

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 automatically 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

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

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

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

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 contiguous 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.

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 2.1. 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

```
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 a subprogram to be activated

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. More detailed example codes are listed in Section 9.

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, so any or all of their operands can be 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. The rest of this subsection may be skipped by first time users who are not worried about marginal issues of differentiability and efficiency.

The modulus **fabs(x)** is everywhere Lipschitz continuous but not properly differentiable at the origin, which raises the question of how this exception ought to be handled. Fortunately, one can easily see that **fabs(x)** and all its compositions with smooth functions

```
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];                 // select independent variables
  adouble 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 as before
  for (int j=0; j < m; j++)
    y[j] >>= py[j];                  // select dependent variables
  delete[] y;                        // delete dependent active variables
  delete[] x;                        // delete independent active variables
  trace_off();                       // complete tape
}                                    // end of function
```

Figure 2: Activated version of the code listed in Figure 1

are still directionally differentiable. These directional derivatives of arbitrary order can be propagated in the forward mode without any ambiguity. In other words the routine **forward** described in Section 6 computes Gateaux derivatives in certain directions, which reduce to Fréchet derivatives only if the dependence on the direction is linear. Otherwise, the directional derivatives are merely positively homogeneous with respect to the scaling of the directions. For the reverse mode, the derivative of **fabs(x)** at the origin is set by ADOL-C somewhat arbitrarily to zero.

We have defined binary functions **fmin** and **fmax** for **adouble** arguments, so that function and derivative values are obtained consistent with those of **fabs** according to the identities

$$\min(a, b) = [a + b - |a - b|]/2 \quad \text{and} \quad \max(a, b) = [a + b + |a - b|]/2 \quad .$$

These relations cannot hold if either a or b is infinite, in which case **fmin** or **fmax** and their derivatives may still be well defined. It should be noted that the directional differentiation of **fmin** and **fmax** yields at ties $a = b$ different results from the corresponding assignment based on the sign of $a - b$. For example, the statement

$$\textbf{if (a} < \textbf{b) c} = \textbf{a; else c} = \textbf{b;}$$

yields for $\mathbf{a} = \mathbf{b}$ and $\mathbf{a}' < \mathbf{b}'$ the incorrect directional derivative value $\mathbf{c}' = \mathbf{b}'$ rather than the correct $\mathbf{c}' = \mathbf{a}'$. Therefore this form of conditional assignment should be avoided by use of the function $\mathbf{fmin}(\mathbf{a}, \mathbf{b})$. There are also versions of **fmin** and **fmax** for two passive arguments and mixed passive/active arguments are handled by implicit conversion. On the function class obtained by composing the modulus with real analytic functions, the concept of directional differentiation can be extended to the propagation of unique one-sided Taylor expansions. The branches taken by **fabs, fmin**, and **fmax**, are recorded on the tape.

The functions **sqrt**, **pow**, and some inverse trigonometric functions have infinite slopes at the boundary points of their domains. At these marginal points the derivatives are set by ADOL-C to either **±InfVal**, 0 or **NoNum**, where **InfVal** and **NoNum** are user-defined parameters, see Section 4.2. On IEEE machines **InfVal** can be set to the special value **Inf** = 1.0/0.0 and **NoNum** to **NaN** = 0.0/0.0. For example, at $\mathbf{a} = 0$ the first derivative $\mathbf{b}'$ of $\mathbf{b} = \mathbf{sqrt(a)}$ is set to

$$\mathbf{b}' = \begin{cases} \textbf{InfVal} & \text{if } \mathbf{a}' > 0 \\ 0 & \text{if } \mathbf{a}' = 0 \\ \textbf{NoNum} & \text{if } \mathbf{a}' < 0 \end{cases} .$$

In other words, we consider **a** and consequently **b** as a constant when $\mathbf{a}'$ or more generally all computed Taylor coefficients are zero.

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 **x** is negative, however, the partial with respect to an integral exponent is set to zero. The derivatives of the step functions **floor**, **ceil**, **frexp**, and **ldexp** are set to zero at all arguments **x**. The result values of the step functions are recorded on the tape and can later be checked to recognize whether a step to another level was taken during a forward sweep at different arguments than at taping time.

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 Section 3.4. Ambiguous statements like **a += a++;** must be avoided because the compiler may sequence the evaluation of the overloaded expression differently from 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 Section 8.5.

2.6 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 ADOL-C 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.

Whenever the evaluation utilities **function**, **gradient**, etc., return negative values one has to retape the active section at the current argument. The crux of the problem lies in the fact that the tape records only the operations that are executed during one particular 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

$$\mathbf{a = (b > 0)\ ?\ c : d;}$$

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

$$\mathbf{if\ (b > 0)\ a = c;\ else\ a = d;}$$

Here, only one of the expressions (or, more generally, program blocks) **c** and **d** is evaluated, which exactly constitutes the problem for ADOL-C. To obtain the correct value **a** with ADOL-C, one may first execute both branches and then pick either **c** or **d** 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. Furthermore the test parameter **b** has to be an **adouble** or an **adouble** expression. Otherwise the test condition **b** is recorded on the tape as a *constant* with its runtime value. Thus the original dependency of **b** on active variables gets lost, for instance if **b** is a comparison expression, see Section 2.5. 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. The header file `adouble.h` contains also corresponding definitions of **condassign(a,b,c,d)** and **condassign(a,b,c)** for passive **double** arguments so that the modified code without any differentiation can be tested for correctness.

2.7 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 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. 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 Arrays and Structures

Some or all real fields of structures or members of classes may be redeclared as **adouble**s so that differentiation can proceed without the explicit unrolling 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 An Active Vector Class

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

adoublev a(p);

where **p** is an integer constant or variable. Like all local variables, vectors are deallocated 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 class can often be used in lieu of dynamically allocated arrays, thereby avoiding a possibly drastic increase in the storage requirement of ADOL-C as discussed below. Vector elements of the form **a[i]** can take the place of any scalar variable of type **adouble**.

3.2 Overloaded Vector Operations

Provided their lengths are compatible, active vectors can be added or subtracted, yielding a third vector of the same length. Similarly, they can be multiplied by active or passive scalars yielding a vector, whereas the dot-product of two vectors is a scalar. Moreover, the special operators $\ll=$ and $\gg=$ are also overloaded so that active vectors can be marked as independent or dependent, respectively. Here, the second operand must be a **double***, which is assumed to represent a passive vector of the same length as the active operand. As in the scalar versions, the statement **a** $\ll=$ **b** simultaneously initializes the active vector **a** with the values in the passive vector **b**, and the statement **a** $\gg=$ **b** passes the values in the active vector **a** to the passive vector **b**.

In the example code in Figure 2, one could therefore have replaced the first loop by the statement $\mathbf{x} \ll= \mathbf{xp}$, and the last loop by $\mathbf{y} \gg= \mathbf{yp}$. The use of active 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 many unnecessary intermediates are maintained in scalar mode.

The following binary operations are defined between **adoublev**s:

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

where $*$ denotes the dot product. In the last case the result is an **adouble**; in the first two an **adoublev**. The assignments

$$-= \quad , \quad += \quad , \quad \ll= \quad , \quad \gg=$$

may also be considered as binary operations between vectors. For the first two assignments both sides must be active. For the last two, the left side must be active and the right side passive. The assignment operator

$$=$$

must have an **adoublev** as its left hand side whereas the right hand side may be either an **adoublev**, a **double*** or a passive scalar. In the last case the right hand side constant is assigned to all components of the vector. The binary operations

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

are also defined when the left argument is an **adoublev** and the right argument is an active or passive scalar. Note, that here $*$ does not represent the dot product. Mathematically meaningless operations between vectors and scalars will produce a compiler error message. None of the operations listed above is currently defined for the active matrix types described below.

3.3 An Active Matrix Class

The matrix type **adoublem** is used only to facilitate the automatic and contiguous allocation (and deallocation) of arrays whose elements are **adoublev**s . The statement

adoublem A(q,p);

allocates a $\mathbf{q}\times\mathbf{p}$ matrix of **adouble**s. The **q** quantities **A[i]** represent **adoublev**s of size **p**. Fortran-like access by columns is not possible. As an example, consider the multiplication of a $\mathbf{q}\times\mathbf{p}$ matrix **A** by a vector **b** as shown in Figure 3. If one wishes to multiply **A** by a $\mathbf{p}\times\mathbf{s}$ 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 Section 4.3, 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.

```
const p = 10; const q = 20; double bp[p];
adoublem A(q,p);
for (int j=0; j < p; j++)
{
      cin >> bp[j];                          // read in values
      for (int i=0; i < q; i++)
          cin >> A[i][j];                    // overloaded Input
}
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
double cp[q];
c >>= cp;                                    // mark c as dependent vector
for (int i=0; i < q; i++)
      cout << cp[i];                         // output results
```

Figure 3: Matrix-vector multiplication using ADOL-C arrays

3.4 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 **adouble**s, so that all arithmetic operations involving them are recorded on the tape. In the current version the class **along** is derived from class **badouble**, i.e. operations with **along** variables are performed in floating point arithmetic, which may yield different results than integer arithmetic. The key functionality is that of subscripting; that is for an **along j** the expressions

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

```
double** A;
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

are considered and recorded as a binary operation between **a** and **j** 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 $\ll=$ and $\gg=$.

Using the conditional assignment of active integers, one can, for example, fully record a function that involves Gaussian elimination with pivoting on a tape, see Section 9.5. In that case the recoding effort is minimal, and there is not much overhead at run time, either.

4 Numbering the Tapes and Controlling the Buffer

The trace generated by the execution of an active section may stay within a triplet of internal arrays or it may be written out to corresponding files. We will refer to these triplets as the tape array or tape file, which may subsequently be used to evaluate the underlying function and its derivatives at the original point or at alternative arguments. If the active section involves user-defined quadratures it must be executed and retaped at each new argument. Similarly, if conditions on **adouble** values lead to a different program branch being taken at a new argument the evaluation process also needs to be retaped at the new point. Otherwise, direct evaluation from the tape by the routine **function** (Section 7.1) is likely to be faster. The use of quadratures and the results of all comparisons on **adouble**s are recorded on the tape so that **function** and other forward routines stop and return appropriate flags if their use without prior retaping is unsafe. To avoid any retaping certain types of branches can be recorded on the tape through the use of conditional assignments and active integers described before in Section 2.6 and 3.4, respectively.

Several tape files may be generated and kept simultaneously. 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. This parameter is defined in the header file `usrparms.h` and may be adjusted by the user. For simple usage, **trace_on** may be called with only the tape **tag** as argument, and **trace_off** may be called without argument.

The optional integer argument **keep** of **trace_on** determines whether the numerical values of all active variables are recorded in a buffered temporary file called value stack before they will be overwritten. 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 6). The name of the tempory file is **FNAME3** found in `usrparms.h`. 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 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 defaults to 0, so that the tape array is written onto a tape file 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 of 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, all problem-independent routines (like **forward**, **reverse**, **gradient**, **jacobian**, **tensor_eval**, ...) expect as first argument a **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 $f_2(x)$

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

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 size of value stack (number of 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 **vssize** = **counts[3]** determine the temporary storage requirements during calls to the workhorses **forward** and **reverse**. For a certain degree **deg** ≥ 0, the scalar version 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 (the value stack) of **vssize**$*$**keep revreals** if called with the option **keep** > 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 **TBUFSIZE** bytes and is otherwise written out to a temporary file. The parameter **TBUFSIZE** is defined in the header file `usrparms.h`. The drawback of the economical **revreal** = **float** choice is that subsequent calls to **reverse** yield 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 length.

4.2 Customizing ADOL-C

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

BUFSIZE: This integer determines the length of internal buffers (default: 65 536). If the buffers are large enough to accommodate all required data, any file 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.

TBUFSIZE: This integer determines the length of internal buffer of the value stack (default: 65 536).

locint: The range of the integer type **locint** determines how many **adouble**s can be simultaneously alive (default: **unsigned short**). In extreme cases when there are more than 65 536 **adouble**s alive at any one time, the type **locint** must be changed to **unsigned int** or **unsigned long**.

revreal: The choice of this floating-point type trades accuracy with storage for reverse sweeps (default: **double**). 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 less accurate choice **revreal** = **float** nearly halves the storage requirement during reverse sweeps.

DEBUG: Definining **DEBUG** forces ADOL-C to print out messages about its progress (default: undefined).

inf_num: This together with **inf_den** sets the "vertical" slope **InfVal** = **inf_num/inf_den** of special functions at the boundaries of their domains (default: **inf_num** = 1.0). On IEEE machines the default setting produces the standard **Inf**. On non-IEEE machines change these values to produce a small **InfVal** value and compare the results of two forward sweeps with different **InfVal** settings to detect a "vertical" slope.

inf_den: See **inf_num** (default: 0.0).

non_num: This together with **non_den** sets the mathematically undefined derivative value **NoNum** = **non_num/non_den** of special functions at the boundaries of their domains (default: **non_num** = 0.0). On IEEE machines the default setting produces the standard **NaN**. On non-IEEE machines change these values to produce a small **NoNum** value and compare the results of two forward sweeps with different **NoNum** settings to detect the occurrence of undefined derivative values.

non_den: See **non_num** (default: 0.0).

FNAME: This name made unique by appending **tag** determines the file holding the operations tape when the internal buffer is exceeded (default: "_adol-op_tape.").

FNAME1: This name made unique by appending **tag** determines the file holding the integer tape when the internal buffer is exceeded (default: "_adol-in_tape.").

FNAME2: This name made unique by appending **tag** determines the file holding the real-valued tape when the internal buffer is exceeded (default: "_adol-rl_tape.").

FNAME3: This is the name of the file that holds the value stack when the internal buffer is exceeded (default: "_adol-vs_tape."). The file will be deleted automatically.

ATRIG_ERF: By removing the comment signs the overloaded versions of the inverse hyperbolic functions and the error function are enabled (default: undefined).

4.3 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 **adouble**s 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 fashion. *DO NOT use malloc() and related C memory-allocating functions when declaring adoubles (see the 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 65 536 active variables, the type **locint** may be defined as **unsigned short**. 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 accessible range of **unsigned short**s 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.

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 randomly accessed storage space of the derivative evaluation routines) is bounded by a small multiple of the memory used in the relevant section of the original program. Failure to delete dynamically allocated **adouble**s may cause that the maximal number of **adouble**s alive at one time will be exceeded 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 a vector function 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 Sections 6 and 7 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 7.2.

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 exception occurred, no comparison (including **fmax**, **fmin** or **fabs**) yielded a tie, 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

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

denote any vector polynomial in the scalar variable $t \in I\!R$. In other words, $x(t)$ describes a path in $I\!R^n$ parameterized by t. The Taylor coefficient vectors

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

are simply the scaled derivatives of $x(t)$ at the parameter origin $t = 0$. The first two vectors $x_1, x_2 \in I\!R^n$ 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

$$y(t) \equiv F(x(t)) \; : \; I\!R \mapsto I\!R^m \tag{2}$$

is also smooth and has $(d+1)$ Taylor coefficient vectors $y_j \in I\!R^m$ at $t = 0$, so that

$$y(t) = \sum_{j=0}^{d} y_j t^j + O(t^{d+1}). \tag{3}$$

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

$$y_0 = F(x_0), \qquad y_1 = F'(x_0)\, x_1, \tag{4}$$

and

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

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 "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

$$y_j = y_j(x_0, x_1, \ldots, x_j) \in I\!R^m,$$

and their derivatives satisfy the identity

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i}) \tag{5}$$

as established in [7]. The $m \times n$ matrices $A_k, k = 0, \ldots, d$, are actually 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 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 Section 6.

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

$$z_j \equiv u^T \frac{\partial y_j}{\partial x_0} = u^T A_j \in I\!R^n \tag{6}$$

for $j = 0, 1, \ldots, d$. If $j = 0$ and u is the i-th Cartesian basis vector in $I\!R^m$, then (6) 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 $i = 1, 2, \ldots, m$.

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

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in I\!R^{q\times n}, \tag{7}$$

where $U \in I\!R^{q\times m}$ represents a *weighting matrix*. When $U = I_m$ with $q = m$, 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 and Nonlinear Equations

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

$$z_0^T = u^T \; F'(x_0) \tag{8}$$

yields the Jacobian of F multiplied from the left by $u \in I\!R^m$. In nonlinear least squares calculations, one may use $u^T \equiv F(x_0)^T$ so that $z_0 \in I\!R^n$ 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

$$y_1 = F'(x_0) \; x_1, \tag{9}$$

where x_1 is an arbitrary n vector. Using the vector version of **forward** one can also multiply the Jacobian simultaneously by several column vectors.

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

$$z_1 = \frac{\partial y_1}{\partial x_0} = \frac{\partial F'(x_0)x_1}{\partial x_0} = \nabla^2 F(x_0)x_1 \tag{10}$$

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 $z_0 \in I\!R^n$ represents the gradient of the Lagrangian function $u^T F(x)$, and $z_1 \in I\!R^n$ represents its Hessian multiplied by the vector x_1.

To perform Newton-like steps in order to solve nonlinear equations one needs to find the solution w of the following equation

$$F'(x_0)w = b \tag{11}$$

for a given right-hand side b. Then the routine **jac_solv** can be applied to calculate the desired values of w.

5.3 Derivatives for Ordinary 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 (2) with the y_j the Taylor coefficients of its derivative $y(t) = x'(t)$, namely,

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

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 may also wish to calculate the Jacobians

$$B_j \;\equiv\; \frac{\mathrm{d}x_{j+1}}{\mathrm{d}x_0} \;\in\; I\!R^{n\times n}\,, \tag{12}$$

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 ordinary differential equation is rewritten in autonomous form.

5.4 Dependence Analysis

The sparsity pattern of Jacobians is often needed to set up data structures for their storage and factorization or to allow their economical evaluation by compression [3]. Compared to the evalution of the full Jacobian $F'(x_0)$ in real arithmetic computing the Boolean matrix $\tilde{P} \in \{0,1\}^{m\times n}$ representing its sparsity pattern in the obvious way requires a little less run-time and certainly a lot less memory. This is already true for the current version of ADOL-C and significant improvements can be expected from further development.

The entry $\tilde{P}_{ji}$ in the j-th row and i-th column of $\tilde{P}$ should be $1 = true$ exactly when there is a data dependence between the i-th independent variable x_i and the j-th dependent variable y_j. Just like for real arguments one would wish to compute matrix-vector and vector-matrix products of the form $\tilde{P}\tilde{v}$ or $\tilde{u}^T\tilde{P}$ by appropriate **forward** and **reverse** routines where $\tilde{v} \in \{0,1\}^n$ and $\tilde{u} \in \{0,1\}^m$. Here multiplication corresponds to logical **AND** and addition to logical **OR**, so that algebra is performed in a semi-ring.

For practical reasons it is assumed that $s = 8*$**sizeof**(**unsigned int**) such Boolean vectors $\tilde{v}$ and $\tilde{u}$ are combined to integer vectors $v \in \mathbb{N}^n$ and $u \in \mathbb{N}^m$ whose components can be interpreted as bit patterns. Moreover p or q such integer vectors may be combined column-wise or row-wise to integer matrices $X \in \mathbb{N}^{n\times p}$ and $U \in \mathbb{N}^{q\times m}$, which naturally correspond to Boolean matrices $\tilde{X} \in \{0,1\}^{n\times(sp)}$ and $\tilde{U} \in \{0,1\}^{(sq)\times m}$. The provided bit pattern versions of **forward** and **reverse** allow to compute integer matrices $Y \in \mathbb{N}^{m\times p}$ and $Z \in \mathbb{N}^{q\times m}$ corresponding to

$$\tilde{Y} = \tilde{P}\tilde{X} \qquad \text{and} \qquad \tilde{Z} = \tilde{U}\tilde{P}\,, \tag{13}$$

respectively, with $\tilde{Y} \in \{0,1\}^{m\times(sp)}$ and $\tilde{U} \in \{0,1\}^{(sq)\times n}$. In general the application of the bit pattern versions of **forward** or **reverse** can be interpreted as propagating dependences between variables forward or backward, therefore both the propagated integer matrices and the corresponding Boolean matrices are called *dependence structures*.

To determine the whole sparsity pattern $\tilde{P}$ of the Jacobian $F'(x)$ as an integer matrix P one may call **forward** or **reverse** with $p \geq n/s$ or $q \geq m/s$, respectively. For this purpose the corresponding dependence structure X or U must be defined to represent the identity matrix of the respective dimension. Due to the fact that always a multiple of s Boolean vectors are propagated there may be superfluous vectors, which can be set to zero.

6 Forward and Reverse Calls

In this section, the basic versions of the **forward** and **reverse** routines, which utilize the overloading capabilities of C++, are described in detail. With exception of the bit pattern versions all interfaces are prototyped in the header file `interfaces.h`, where also some more specialized **forward** and **reverse** routines are explained. Furthermore ADOL-C provides C and Fortran-callable versions prototyped in the same header file. The bit pattern versions of **forward** and **reverse** introduced in Section 6.3 are prototyped in the header file `SPARSE/sparse.h`, which will be included by the mentioned header file `interfaces.h` automatically.

6.1 The Scalar Case

Given any correct tape, one may call from within the generating program, or subsequently during another run, the following procedure:

```
int forward(tag,m,n,d,keep,X,Y)
short int tag;                // tape identification
int m;                        // number of dependent variables m
int n;                        // number of independent variables n
int d;                        // highest derivative degree d
int keep;                     // flag for reverse sweep
double X[n][d+1];             // Taylor coefficients X of independent variables
double Y[m][d+1];             // Taylor coefficients Y as in (3)
```

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 $X = \{x_j\}_{j=0\ldots d}$ represent Taylor coefficient vectors as in (1). The rows of the matrix Y must correspond to the dependent variables in the order of their selection by the $\gg=$ operator. The columns of $Y = \{y_j\}_{j=0\ldots d}$ represent Taylor coefficient vectors as in (3). 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 (the value stack) in preparation for a subsequent reverse sweep. If **keep** is omitted, it defaults 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. The procedure **forward** can be used to evaluate the vector function F at arguments x other than the point at which the tape was generated, provided there are no user defined quadratures and all comparisons involving **adouble**s yield the same result. The last condition implies that the control flow is unaltered by the change of the independent variable values. Therefore, this sufficient condition is tested by **forward** and if it is not met the routine indicates this contingency through its return value. Currently, there are six return values, see Table 1.

+3	The function is locally analytic.
+2	The function is locally analytic but the sparsity structure (compared to the situation at the taping point) may have changed, e.g. while at taping arguments **fmax(a,b)** returned **a** we get **b** at the argument currently used in **forward** or **reverse** routines.
+1	At least one of the functions **fmin**, **fmax** or **fabs** is evaluated at a tie or zero, respectively. Hence, F is Lipschitz-continuous but possibly nondifferentiable.
0	Some arithmetic comparison involving **adouble**s yields a tie. Hence, F may be discontinuous.
-1	An **adouble** comparison yields different results from the evaluation point at which the tape was generated.
-2	The argument of a user-defined quadrature has changed from the evaluation point at which the tape was generated.

Table 1: Description of return values

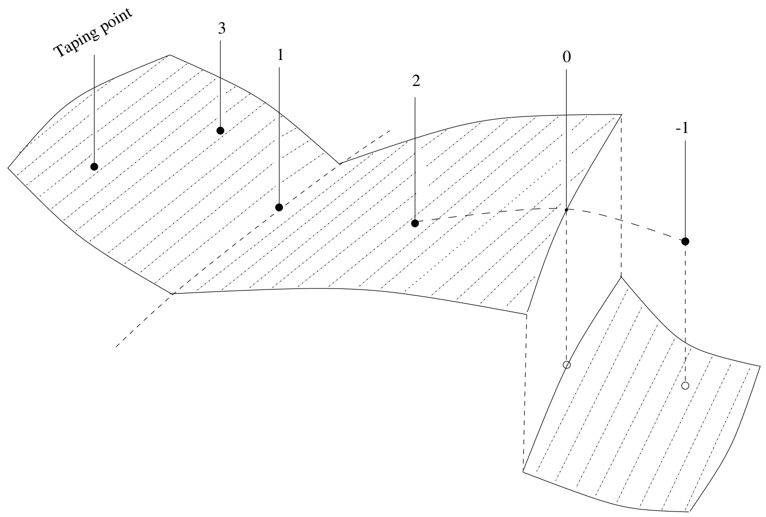

Figure 5: Return values around the taping point

In Figure 5 these return values are illustrated. If the user finds the return value to be negative he may simply repeat the taping process, by executing the active section again. When there are user-defined quadratures this is necessary at each new point. If there are only branches conditioned on **adouble** comparisons one may hope that retaping becomes unnecessary when the points settle down in some small neighborhood, as one would expect for example in an iterative equation solver. The return values of the other forward variants and some of the drivers discussed below have exactly the same meaning.

After the execution of an active section with **keep** = 1 or a call to **forward** with any **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 (6), **reverse** is executed in the *scalar mode* by the following calling sequence:

```
int reverse(tag,m,n,d,u,Z)
short int tag;                // tape identification
int m;                        // number of dependent variables m
int n;                        // number of independent variables n
int d;                        // highest derivative degree d
double u[m];                  // weighting vector u
double Z[n][d+1];             // resulting adjoints Z as in (6)
```

6.2 The Vector Case

When U is a matrix as in (7), **reverse** is executed in the *vector mode* by the following calling sequence:

```
int reverse(tag,m,n,d,q,U,Z,nz)
short int tag;                // tape identification
int m;                        // number of dependent variables m
int n;                        // number of independent variables n
int d;                        // highest derivative degree d
int q;                        // number of weight vectors q
double U[q][m];               // domain weight vector U
double Z[q][n][d+1];          // resulting adjoints Z as in (7)
short nz[q][n];               // nonzero pattern of Z
```

The last argument can be omitted. Otherwise each short **nz[i][j]** has a value that characterizes the functional relation between the i-th component of $UF'(x)$ and the j-th independent value x_j as:

0	trivial	2	polynomial	4	transcendental
1	linear	3	rational	5	nonsmooth

Here, "trivial" means that there is no dependence at all and "linear" means that the partial derivative is a constant that does not dependent on other variables either. "Nonsmooth" means that one of the functions on the path between x_i and y_j was evaluated at a point where it is not differentiable. All positive labels $1, 2, 3, 4, 5$ are pessimistic in that the actual functional relation may in fact be simpler, for example due to exact cancellations. When the arguments **p** and **U** are omitted, they default to m and the identity matrix of order m, respectively.

The return values of **reverse** calls are interpreted according to Table 1, but negative return values are not valid. The reason is the corresponding forward sweep would have stopped without completing the necessary taylor file. The return value of **reverse** may be higher than that of the preceding **forward** call because some operations that were evaluated at a critical argument during the forward sweep were found not to impact the dependents during the reverse sweep.

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 declaring **double g[n]** to calculate a gradient of a scalar-valued function.

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

```
int forward(tag,m,n,d,p,x,X,y,Y)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
int p;                  // number of Taylor series p
double x[n];            // values of independent variables x_0
double X[n][p][d];      // Taylor coefficients X of independent variables
double y[m];            // values of dependent variables y_0
double Y[m][p][d];      // Taylor coefficients Y 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. The return integer serves as a flag to indicate quadratures or altered comparisons as described for the scalar version of **forward** at the beginning of this section.

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

```
int forward(tag,m,n,p,x,X,y,Y)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int p;                  // number of partial derivatives p
double x[n];            // values of independent variables x_0
double X[n][p];         // seed derivatives of independent variables X
double y[m];            // values of dependent variables y_0
double Y[m][p];         // first derivatives of dependent variables Y
```

When this routine is called with $\mathbf{p} = \mathbf{n}$ and $\mathbf{X}$ the identity matrix, the resulting $\mathbf{Y}$ is simply the Jacobian $F'(x_0)$. In general, one obtains the $m \times p$ matrix $Y = F'(x_0)\,X$ for the chosen initialization of X. In a workstation 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 [1]) should be "strip-mined" in the sense 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.

6.3 Propagation of Bit Patterns

Suppose, one wants to analyze the dependences between dependent and independent variables or even to determine the whole sparsity structure of the Jacobian of a function given as a tape with the identifier **tag**. Then one may call the bit pattern **forward** routine

```
int forward(tag,m,n,p,x,X,y,Y)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int p;                  // number of integers propagated p
double x[n];            // values of independent variables x_0
unsigned int X[n][p];   // dependence structure X
double y[m];            // values of dependent variables y_0
unsigned int Y[m][p];   // dependence structure Y according to (13)
short int mode;         // 0 : safe mode (default), 1 : tight mode
```

to obtain the dependence structure Y for a given dependence structure X as described in Section 5.4. The dependence structures are represented as arrays of **unsigned int** the entrys of which are interpreted as bit patterns. Thus $p * 8*$**sizeof**(**unsigned int**) Boolean vectors packed into integer matrices are propagated by **forward**. For example, for $n = 3$ the identity matrix I_3 should be passed with $p = 1$ as the 3×1 array

$$\mathbf{X} = \begin{pmatrix} 10000000\,00000000\,00000000\,00000000_2 \\ 01000000\,00000000\,00000000\,00000000_2 \\ 00100000\,00000000\,00000000\,00000000_2 \end{pmatrix}$$

in the 4-byte integer format. The parameter **mode** will be explained below.

A call to the corresponding bit pattern **reverse** routine

```
int reverse(tag,m,n,q,U,Z)
short int tag;             // tape identification
int m;                     // number of dependent variables m
int n;                     // number of independent variables n
int q;                     // number of integers propagated q
unsigned int U[q][m];      // dependence structure U
unsigned int Z[q][n];      // dependence structure Z according to (13)
short int mode;            // 0 : safe mode (default), 1 : tight mode
```

yields the dependence structure Z for a given dependence structure U as explained in Section 5.4. Here **reverse** propagates the $q * 8*$**sizeof**(**unsigned int**) Boolean vectors packed into integer matrices backwards.

The return values of the bit pattern **forward** and **reverse** routines correspond to those described in Table 1. Both routines can be run in one of the two modes *safe* and *tight*. The first, more conservative option is the default. It accounts for all dependences that might occur for any value of the independent variables. For example, the intermediate **c** = **max**(**a**,**b**) is always assumed to depend on all independent variables that **a** or **b** dependent on, i.e. the bit pattern associated with **c** is set to the logical **OR** of those associated with **a** and **b**. In contrast the tight option gives this result only in the unlikely event of an exact tie **a** = **b**. Otherwise it sets the bit pattern associated with **c** either to that of **a** or to that of **b**, depending on whether **c** = **a** or **c** = **b** locally. Obviously, the sparsity pattern obtained with the tight option may contain more zeros than that obtained with the safe option. On the other hand, it will only be valid at points where the return value of **forward** and **reverse** remains 3. The safe versions do not require any reevaluation of the function and may thus be a little faster. Note that only the tight version can handle the active subscripts properly.

The sparsity pattern of Jacobians may vary as a function of the independent variable vector x for one of three reasons. First, numerical values may be incidentally zero, second the control flow may be completely changed, and finally **fabs**, **fmin**, **fmax** or other conditional assignments may flip to a different branch. In contrast to other automatic differentiation tools our bit pattern routines propagate generic dependences and disregard incidental zeros, which may be due to cancellations or special values of the independents. When sparsity pattern might be altered due to changes in the control flow the return values of all **forward** and **reverse** routines indicate this fact as described above. Then these routines must be rerun, possibly with the old sparsity pattern providing a warm start. Thus we are left with the third possibility, namely conditional assignments.

One can control the storage growth by the factor p through "strip-mining" – calling **forward** or **reverse** with successive groups of columns or respectively rows at a time, i.e. partitioning X or U appropriately as described for the computation of Jacobians in Section 6.2.

7 Easy To Use Drivers

For convenience one may use instead of (subsequent) calls to **forward** and **reverse** one of the following driver routines. These drivers are all C functions and therefore can be used within C and C++ programs. Some Fortran-callable companions can be found in the appropriate header files.

7.1 Drivers for Optimization and Nonlinear Equations

The drivers provided for solving optimization problems and nonlinear equations are prototyped in the header file `DRIVERS/drivers.h`, which is included automatically by the global header file `adolc.h` (see Section 8.3).

The routine **function** allows to evaluate the desired function from the tape instead of executing the corresponding source code:

int function(tag,m,n,x,y)	
short int tag;	// tape identification
int m;	// number of dependent variables m
int n;	// number of independent variables n
double x[n];	// independent vector x_0
double y[m];	// dependent vector $y_0 = F(x_0)$

If the original evaluation program is available this should be used to compute the function value in order to avoid the interpretative overhead.

For the calculation of whole derivative vectors and matrices up to order 2 there are the following procedures:

```
int gradient(tag,n,x,g)
short int tag;                // tape identification
int n;                        // number of independent variables n
double x[n];                  // independent vector x_0
double g[n];                  // resulting gradient ∇f(x_0)
```

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

```
int hessian(tag,n,x,H)
short int tag;                // tape identification
int n;                        // number of independent variables n
double x[n];                  // independent vector x_0
double H[n][n];               // resulting Hessian matrix ∇^2 f(x_0)
```

The driver routine **hessian** computes only the lower half of $\nabla^2 f(x_0)$ so that all values **H[i][j]** with $j > i$ of **H** allocated as a square array remain untouched during the call of **hessian**. Hence only $i + 1$ **double**s need to be allocated starting at the position **H[i]**.

To use the full capability of automatic differentiation when the product of derivatives with certain weight vectors or directions are needed, ADOL-C offers the following four drivers:

```
int vec_jac(tag,m,n,repeat,x,u,z)
short int tag;                // tape identification
int m;                        // number of dependent variables m
int n;                        // number of independent variables n
int repeat;                   // indicate repeated call at same argument
double x[n];                  // independent vector x_0
double u[m];                  // range weight vector u
double z[n];                  // result z_0 as in (8)
```

```
int jac_vec(tag,m,n,x,v,z)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
double x[n];            // independent vector x_0
double v[n];            // tangent vector x_1
double z[m];            // result y_1 as in (9)
```

```
int hess_vec(tag,n,x,v,z)
short int tag;          // tape identification
int n;                  // number of independent variables n
double x[n];            // independent vector x_0
double v[n];            // tangent vector x_1
double z[n];            // result z_1 as in (10)
```

```
int lagra_hess_vec(tag,m,n,x,v,u,h)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
double x[n];            // independent vector x_0
double v[n];            // tangent vector x_1
double u[m];            // range weight vector u
double h[n];            // result z_1 as in (6)
```

If a nonzero value of the parameter **repeat** indicates that the routine **vec_jac** has been called at the same argument immediately before, the internal **forward** call will be skipped and only **reverse** with the corresponding arguments is executed.

The next procedure allows the user to perform Newton steps only having the corresponding tape at hand:

```
jac_solv(tag,n,x,b,sparse,mode)
short int tag;          // tape identification
int n;                  // number of independent variables n
double x[n];            // independent vector x_0 as in (1)
double b[n];            // in: right-hand side b, out: result w as in (11)
int sparse;             // option to use sparsity
int mode;               // option to choose different solvers
```

On entry, parameter **b** of the routine **jac_solv** contains the right-hand side of (11). On exit, **b** equals the solution w of this equation. Currently, the parameter **sparse** is not used, but will be subject of further developments of ADOL-C in the future. It is planed to provide a sparse version of **jac_solv**. If **mode** $= 1$ only the Jacobian of the function given by the tape labeled with **tag** is provided internally. The LU-factorisation of this Jacobian is computed for **mode** $= 2$. The solution of the equation is calculated if **mode** $= 2$. Hence, it is possible to compute the LU-factorisation only once. Then the equation can be solved for several right-hand sides b without calculating the Jacobian and its factorisation again.

If the original evaluation code of a function contains neither quadratures nor branches, all drivers described above can be used to evaluate derivatives at any argument in its domain. the same still applies if there are no user defined quadratures and all comparisons involving **adouble**s have the same result as during taping. If this assumption is falsely made all drivers while internally calling forward will return the value -1 or -2 as already specified in Section 6.

7.2 Drivers for Ordinary Differential Equations

The ODE-drivers described below are prototyped in the header file `DRIVERS/odedrivers.h`. The global header file `adolc.h` includes this file automatically (see Section 8.3).

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

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

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 (7) of Section 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 (12), one may finally use the following routine:

```
int accode(n,tau,deg,Z,B,nz)
int n;                      // number of state variables n
double tau;                 // scaling parameter
int deg;                    // degree on current call
double Z[n][n][deg];        // partials of coefficient vectors
double B[n][n][deg];        // result B as defined in (12)
short nz[n][n];             // optional nonzero 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

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

In other words, the matrices $B_k = \mathbf{B[\ ][\ ][k]}$ have a sparsity pattern that fills in as k grows.

7.3 Higher Derivative Tensors

The special drivers provided for efficient calculation of higher order derivatives are prototyped in the header file `DRIVERS/taylor.h`, which is included by the global header file `adolc.h` automatically (see Section 8.3).

Many applications in scientific computing need second- and higher-order derivatives. Often, one does not require full derivative tensors but only the derivatives in certain directions $s_i \in \mathbb{R}^n$. Suppose a collection of p directions $s_i \in \mathbb{R}^n$ is given, which form a matrix

$$S \;=\; [s_1, s_2, \ldots, s_p] \;\in\; \mathbb{R}^{n\times p}.$$

One possible choice is $S = I_n$ with $p = n$, which leads to full tensors being evaluated. ADOL-C provides the function **tensor_eval** to calculate the derivative tensors

$$\nabla_S^k F(x_0) \;=\; \left.\frac{\partial^k}{\partial z^k} F(x_0 + Sz)\right|_{z=0} \in \mathbb{R}^{p^k} \quad \text{for} \quad k = 0, \ldots, d \tag{14}$$

simultaneously. The function **tensor_eval** has the following calling sequence and parameters:

```
void tensor_eval(tag,m,n,d,p,x,tensor,S)
short int tag;              // tape identification
int m;                      // number of dependent variables m
int n;                      // number of independent variables n
int d;                      // highest derivative degree d
int p;                      // number of directions p
double x[n];                // values of independent variables x0
double tensor[m][size];     // result as defined in (14) in composed form
double S[n][p];             // seed matrix S
```

Using the symmetry of the tensors defined by (14), the memory requirement can be reduced enormously. The collection of tensors up to order d comprises $\binom{p+d}{d}$ distinct elements. Hence, the second dimension of **tensor** must be greater or equal to $\binom{p+d}{d}$. To compute the derivatives, **tensor_eval** propagates internally univariate Taylor series along $\binom{n+d-1}{d}$ directions. Then the desired values are interpolated. This approach is described in the article [12].

The access of individual entries in symmetric tensors of higher order is a little tricky. We always store the derivative values in the two dimensional array **tensor** and provide two different ways of accessing them. The leading dimension of the tensor array ranges over the component index i of F_{i+1} for $i = 0, \ldots, m-1$. The subarrays pointed to by **tensor[i]** have identical structure for all i. Each of them represents the symmetric tensors up to order d of the scalar function F_{i+1} in p variables. The $\binom{p+d}{d}$ mixed partial derivatives in each of the m tensors are linearly ordered according to the tetrahedral scheme described by Knuth [15]. In the familiar quadratic case $d = 2$ the derivative with respect to z_j and z_k with z as in (14) and $j \leq k$ is stored at **tensor[i][l]** with $l = k * (k+1)/2 + j$. At $j = 0 = k$ and hence $l = 0$ we find the function value F_{i+1} itself and the gradient $\nabla F_{i+1} = \partial F_{i+1}/\partial x_k$ is stored at $l = k(k+1)/2$ with $j = 0$ for $k = 1, \ldots, p$.

For general d we combine the variable indices to a multiindex $j = (j_1, j_2, \ldots, j_d)$, where j_k indicates differentiation with respect to variable x_{j_k} with $j_k \in \{0, 1, \ldots, p\}$. The value $j_k = 0$ indicates no differentiation so that all lower derivatives are also contained in the same data structure as described above for the quadratic case. The location of the partial derivative specified by j is computed by the function

```
int address(d,j)
int d;                     // highest derivative degree d
int j[d];                  // multiindex j
```

and it may thus be referenced as

tensor[i][address(d,j)] .

Notice that the address computation does depend on the degree d but not on the number of directions p, which could theoretically be enlarged without the need to reallocate the original tensor. Also, the components of j need to be nonincreasing. To some C programmers it may appear more natural to access tensor entries by successive dereferencing in the form

tensorentry[i][j1][j2]...[jd] .

We have also provided this mode, albeit with the restriction that the indices $j_1, j_2, \ldots, j_d$ are nonincreasing. In the second order case this means that the Hessian entries must be specified in or below the diagonal. If this restriction is violated the values are almost certain to be wrong and array bounds may be violated. We emphasize that subscripting is not overloaded but that **tensorentry** is a conventional and thus moderately efficient C pointer structure. Such a pointer structure can be allocated and set up completely by the function

```
void** tensorsetup(m,p,d,tensor)
int m;                      // number of dependent variables n
int p;                      // number of directions p
int d;                      // highest derivative degree d
double tensor[m][size];     // pointer to two dimensional array
```

Here, **tensor** is the array of m pointers pointing to arrays of **size** $\geq \binom{p+d}{d}$ allocated by the user before. During the execution of **tensorsetup**, $d-1$ layers of pointers are set up so that the return value allows the direct dereferencing of individual tensor elements.

For example, suppose some active section involving $m \geq 5$ dependents and $n \geq 2$ independents has been executed and taped. We may select $p = 2$, $d = 3$ and initialize the $n \times 2$ seed matrix S with two columns s_1 and s_2. Then we are able to execute the code segment

```
double**** tensorentry = (double****) tensorsetup(m,p,d,tensor);
tensor_eval(tag,m,n,d,p,x,tensor,S);
```

This way, we evaluated all tensors defined in (14) up to degree 3 in both directions s_1 and s_2 at some argument x. To allow the access of tensor entries by dereferencing the pointer structure **tensorentry** has been created. Now, the value of the mixed partial

$$\left.\frac{\partial^3 F_5(x + s_1 z_1 + s_2 z_2)}{\partial z_1^2 \partial z_2}\right|_{z_1=0=z_2}$$

can be recovered as

tensorentry[4][2][1][1] or **tensor[4][address(d,j)]**,

where the integer array **j** may equal (1,1,2), (1,2,1) or (2,1,1). Analogously, the entry

tensorentry[2][1][0][0] or **tensor[2][address(d,j)]**

with **j** = (1,0,0) contains the first derivative of the third dependent variable F_3 with respect to the first differentiation parameter z_1.

Note, that the pointer structure **tensorentry** has to be set up only once. Changing the values of the array **tensor**, e.g. by a further call of **tensor_eval**, directly effects the values accessed by **tensorentry**. When no more derivative evaluations are desired the pointer structure **tensorentry** can be deallocated by a call to the function

```
int freetensor(m,p,d, (double **) tensorentry)
int m;                       // number of dependent variables m
int p;                       // number of independent variables p
int d;                       // highest derivative degree d
double* tensorentry[m];      // return value of tensorsetup
```

that does not deallocate the array **tensor**.

Example codes using the above procedures can be found in the files `taylorexam.C` and `accessexam.C` contained in the directory <ADOLC18_DIR>/EXA/TAYLOR.

7.4 Derivatives of Implicit and Inverse Functions

The described drivers are also prototyped in the header file `DRIVERS/taylor.h`. As indicated before this header is included by the global header file `adolc.h` automatically (see Section 8.3).

In many applications, one needs the derivatives of variables $y \in \mathbb{R}^m$ that are implicitly defined as functions of some variables $x \in \mathbb{R}^{n-m}$ by an algebraic system of equations

$$G(z) \; = \; 0 \in \mathbb{R}^m \quad \text{with} \quad z = (y, x) \in \mathbb{R}^n .$$

Naturally, the n arguments of G need not be partitioned in this regular fashion and we wish to provide flexibility for a convenient selection of the $n - m$ *truly* independent variables. Let $P \in \mathbb{R}^{(n-m)\times n}$ be a $0 - 1$ matrix that picks out these variables so that it is a column permutation of the matrix $[0, I_{n-m}] \in \mathbb{R}^{(n-m)\times n}$. Then the nonlinear system

$$G(z) \; = \; 0, \quad Pz = x,$$

has a regular Jacobian, wherever the implicit function theorem yields y as a function of x. Hence, we may also write

$$F(z) = \left(\begin{array}{c} G(z) \\ Pz \end{array} \right) \; \equiv \; \left(\begin{array}{c} 0 \\ Pz \end{array} \right) \; \equiv \; S\, x, \tag{15}$$

where $S = [0, I_p]^T \in \mathbb{R}^{n\times p}$ with $p = n - m$. Now, we have rewritten the original implicit functional relation between x and y as an inverse relation $F(z) = Sx$. In practice, we may implement the projection P simply by marking $n - m$ of the independents also dependent.

Given any $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ that is locally invertible and an arbitrary seed matrix $S \in \mathbb{R}^{n\times p}$ we may evaluate all derivatives of $z \in \mathbb{R}^n$ with respect to $x \in \mathbb{R}^p$ by calling the following routine:

```
void inverse_tensor_eval(tag,n,d,p,z,tensor,S)
short int tag;              // tape identification
int n;                      // number of variables n
int d;                      // highest derivative degree d
int p;                      // number of directions p
double z[n];                // values of independent variables z
double tensor[n][size];     // partials of z with respect to x
double S[n][p];             // seed matrix S
```

The results obtained in **tensor** are exactly the same as if we had called **tensor_eval** with **tag** pointing to a tape for the evaluation of the inverse function $z = F^{-1}(y)$ for which naturally $n = m$. Note that the columns of S belong to the domain of that function. Individual derivative components can be accessed in tensor exactly as in the explicit case described above.

It must be understood that **inverse_tensor_eval** actually computes the derivatives of z with respect to x that is defined by the equation $F(z) = F(z_0) + S\,x$. In other words the base point at which the inverse function differentiated is given by $F(z_0)$. The routine has no capability for inverting F itself as solving systems of nonlinear equations $F(z) = 0$ in the first place is not just a differentiation task. However, the routine **jac_solv** described in Section 7.1 may certainly be very useful for that purpose.

As an example consider the following two nonlinear expressions

$$\begin{aligned} G_1(z_1, z_2, z_3, z_4) &= z_1^2 + z_2^2 - z_3^2 \\ G_2(z_1, z_2, z_3, z_4) &= \cos(z_4) - z_1/z_3 \ . \end{aligned}$$

The equations $G(z) = 0$ describe the relation between the Cartesian coordinates (z_1, z_2) and the polar coordinates (z_3, z_4) in the plane. No suppose we are interested in the derivatives of the second Cartesian $y_1 = z_2$ and the second (angular) polar coordinate $y_2 = z_4$ with respect to the other two variables $x_1 = z_1$ and $x_2 = z_3$. Then the active section could look simply like

```
for (j=1; j < 5;j++)    z[j] <<= zp[j];
g[1] = z[1]*z[1]+z[2]*z[2]-z[3]*z[3];
g[2] = cos(z[4]) - z[1]/z[3];
g[1] >>= gp[1];             g[2] >>= gp[2];
z[1] >>= zd[1];             z[3] >>= zd[2];
```

where **zd[1]** and **zd[2]** are dummy arguments. In the last line the two independent variables **z[1]** and **z[3]** are made simultaneously dependent thus generating a square system that can be inverted (at most arguments). The corresponding projection and seed matrix are

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad S^T = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} .$$

Provided the vector **zp** is consistent in that its Cartesian and polar components describe the same point in the plane the resulting tupel **gp** must vanish. The call to **inverse_tensor_eval** with $n = 4$, $p = 2$ and d as desired will yield the implicit derivatives, provided **tensor** has been allocated appropriately of course and S has the value given above. The example is untypical in that the implicit function could also be obtained explicitly by symbolic manipulations. It is typical in that the subset of z components that are to be considered as truly independent can be selected and altered with next to no effort at all.

The provided example programs `inversexam.C`, `coordinates.C` and `trigger.C` in the directory `<ADOLC18_DIR>/EXA/TAYLOR` show the application of the procedures described here.

7.5 Detection of Sparsity in Jacobian Matrices

The driver routine described below is prototyped in the header file `SPARSE/jacutils.h`, which is included automatically by the global header file `adolc.h` (see Section 8.3).

ADOL-C offers a convenient way of determining the Jacobian sparsity structure. The routine **jac_pat** is based on bit pattern propagation, but yields the sparsity structure in a comprehensive compressed row format. Moreover, it allows certain independent and/or dependent variables (e.g. vectors) to be bound together as a block, building a block sparsity pattern:

```
int jac_pat(tag, m, n, x, rb, cb, crs, options)
short int tag;                // tape identification
int m;                        // number of dependent variables m
int n;                        // number of independent variables n
double x[n];                  // independent variables x_0
unsigned short *rb[1+m];      // description of the blocks of dependents
unsigned short *cb[1+n];      // description of the blocks of independents
unsigned short *crs[rb[0]][1+(non-zero ind. blocks w.r.t. current dep. block)];
                              // row compressed sparsity structure
int options[2];               // array of control parameters
```

The information how separate variables are bound together as blocks of variables is stored in the arrays ***rb** and ***cb**. **(*rb)[0]** is the number of blocks of dependent variables and

(*cb)[0] is the number of blocks of independent variables. The number of the column group to which the variable **x[j]** belongs is represented by the integer **(*cb)[j+1]** $\leq$ **(*cb)[0]**. Correspondingly **(*rb)[i+1]** $\leq$ **(*rb)[0]** represents the number of the row group to which the variable **y[i]** belongs. The derivative of **y[i]** with respect to **x[j]** may be nonzero only if the block with index (**(*rb)[i]**,**(*cb)[j]**) as a whole is nozero. The latter fact is represented in a customary compressed row format stored in the array ***crs**, where each row corresponds to a block of dependent variables. **(*crs)[][0]** is always the number of blocks of independent variables on which the current block of dependent variables depends. The components **(*crs)[][i]**, **i** $>$ 0 store the indeces of these blocks of independent variables (see Figure 6). Natural applications of this variable grouping are for example simulations in physical space,

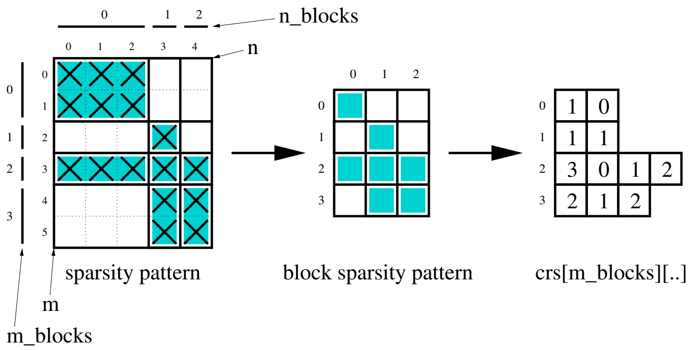

Figure 6: Jacobian (block) sparsity pattern

where the three components of position or velocity vectors naturally occur together or not at all in various relations. When ***rb** is equal to zero on entry all dependent variables are treated separately as a singleton block. The same applies to the independent variables when ***cb** equals zero on entry. It is planned that **jac_pat** will in future automatically deduce suitable nontrivial row and column groupings.

options[0]	0	mode selection (default)
	1	forward mode
	2	reverse mode
options[1]	0	safe mode (default)
	1	tight mode

Table 2: **jac_pat** parameter **options**

The elements of the array **options** control the action of **jac_pat** according to Table 2. The value of **options[0]** selects the direction of bit pattern propagation. Depending on the number of independent and of dependent variables, one would prefer the forward mode if $n \leq m$ and would otherwise use the reverse mode. The component **options[1]** determines the usage of the safe or tight mode of bit pattern propagation, both explained in Section 6.3. In the safe mode one does not need to supply the base point x_0. The return values of **jac_pat** are explained in Table 1 in Section 6.3.

The routine **jac_pat** uses "strip-mining" to cope with large matrix dimensions. If the system happens to run out of memory, one may reduce the value of the constant **PQ_STRIPMINE_MAX** following the instructions in `SPARSE/jacutils.h`.

The example code `jacpatexam.C` contained in directory `<ADOLC18_DIR>/EXA/SPARSE` demonstrates the driver routine **jac_pat**.

8 Installing and Using ADOL-C

The ADOL-C package Version 1.8 consists of the following files separated into five subdirectories:

- subdirectory `<ADOLC18_DIR>/INS` (installation scripts):

  ```
  makefile

  aix_comp        gnu_comp        sgi_comp        sun_comp
  xxx_comp

  README.INS      README.BUGS
  ```

  and makefile components:

  ```
  makefile.<subdir>.head          makefile.<subdir>.tail
  ```

- subdirectory `<ADOLC18_DIR>/SRC` (library sources):

  ```
  adalloc.h       adolc.h         adouble.h       adutils.h
  adutilsc.h      avector.h       convolut.h      dvlparms.h
  fortutils.h     interfaces.h    oplate.h        taputil.h
  tayutil.h       usrparms.h
  adallocc.c      adallocC.C      adouble.C       avector.C
  convolut.c      fo_rev.c        fortutils.c     ho_rev.c
  interfacesc.c   interfacesC.C   interfacesf.c   taputilc.c
  taputilC.C      tayutilc.c      tayutilC.C      uni5_for.c

  DRIVERS/drivers.h       DRIVERS/odedrivers.h   DRIVERS/taylor.h
  DRIVERS/driversc.c      DRIVERS/driversf.c     DRIVERS/odedriversc.c
  DRIVERS/odedriversC.C   DRIVERS/odedriversf.c  DRIVERS/taylor.c

  SPARSE/jacutils.h       SPARSE/sparse.h
  SPARSE/int_for.c        SPARSE/int_rev.c       SPARSE/jacutils.c
  ```

```
SPARSE/sparsec.c       SPARSE/sparseC.C

README.SRC
```

and the tape documentation utility destined for ADOL-C developers:

```
TAPEDOC/tapedoc.h
TAPEDOC/tapedoc.c
```

- subdirectory `<ADOLC18_DIR>/DEX` (documented examples):

```
detexam.C       gaussexam.C     odexam.C        powexam.C
speelpenning.C

README.DEX
```

- subdirectory `<ADOLC18_DIR>/EXA` (more examples):

  There are about 30 example programs partially placed in the subdirectories

```
ODE             SPARSE          TAYLOR          TIMING
CLOCK
```

  Some of them demonstrate how to make use of the capabilities of ADOL-C. Others allow to check computed derivatives for correctness or to perform run time measurements. Detailed information can be found in the file

```
README.EXA
```

- subdirectory `<ADOLC18_DIR>/DOC` (this documentation):

```
adolc18.tex    adolc18.ps      blocks.eps      tap_point.eps
```

8.1 Generating the ADOL-C Library `libad.a`

All source files needed to generate the ADOL-C library `libad.a` are placed in the source directory `<ADOLC18_DIR>/SRC`. The necessary `makefile` can be provided by the command `make <operating_system>install` from installation directory `<ADOLC18_DIR>/INS`. Before doing so appropriate changes in `<operating_system>_comp` (e.g. the choice of a different compiler and/or machine dependent options) allow best fit to the given operating system. Details and known problems can be found in the files `README.INS` and `README.BUGS`. Furthermore the user may modify the header file `usrparms.h` in order to tailor the ADOL-C package to his needs in the particular system environment as discussed in Section 4.2. After all calling `make` from the source directory generates the library `libad.a` that will be placed in the same directory. All object files and other intermediately generated files can be removed by the call `make clean`. See file `README.SRC` for further information.

8.2 Compiling and Linking the Example Programs

The installation procedure described in Section 8.1 also provides the particular `makefiles` necessary to compile the example programs placed in the directories `<ADOLC18_DIR>/DEX` and `<ADOLC18_DIR>/EXA`. As described in the files `README.DEX` and `README.EXA`, respectively, the command `make <example_name>` compiles and links a particular example program. Just calling `make` generates all examples of a directory. The examples placed in the directory `<ADOLC18_DIR>/DEX` are documented in Section 9. Detailed information about the examples in the directory `<ADOLC18_DIR>/EXA` can be found in the file `README.EXA`.

8.3 Description of Important Header Files

The application of the facilities of ADOL-C requires the user source code (program or module) to include appropriate header files where the desired data types and routines are prototyped. The new hierarchy of header files enables the user to take one of two possible ways to access the right interfaces. The first and easy way is recommended to beginners: As indicated in Table 3 the provided *global* header file `adolc.h` can be included by any user code to support all capabilities of ADOL-C depending on the particular programming language of the source. Note, that the two header files `adutils.h` and `adutilsc.h` kept for compatibility reasons to ADOL-C Version 1.7, so that existing applications do not have to be changed.

`adolc.h`	→ global header file available for easy use of ADOL-C; • includes all ADOL-C header files depending on whether the users source is C++ or C code.
`adutils.h`	→ global C++ header file kept for compatibilty to ADOL-C Version 1.7; • includes the global header `adolc.h` only.
`adutilsc.h`	→ global C header file kept for compatibilty to ADOL-C Version 1.7; • includes the global header `adolc.h` only.
`usrparms.h`	→ user customization of ADOL-C package (see Section 4.2); • after a change of user options the ADOL-C library `libad.a` has to be rebuilt (see Section 8.1); • is included by all ADOL-C header files and thus by all user programs.

Table 3: Global header files

The second way is meant for the more advanced ADOL-C user: Some source code includes only those interfaces used by the particular application. The respectivly needed header files are indicated throughout the manual. Existing application determined dependences between the provided ADOL-C routines are realized by automatic includes of headers in order to maintain easy use. The header files important to the user are described in the Tables 4 and 5.

`adouble.h`	→ provides the interface to the basic active scalar data type of ADOL-C: **class adouble** (see Section 2); • includes the headers `avector.h` and `taputil.h`.
`avector.h`	→ provides the interface to the active vector and matrix data types of ADOL-C: **class adoublev** and **class adoublem**, respectively (see Section 3); • is included by the header `adouble.h`.
`taputil.h`	→ provides functions to start/stop the tracing of active sections (see Sections 2.2) as well as utilities to obtain tape statistics (see Sections 4.1); • is included by the header `adouble.h`.

Table 4: Important header files: tracing/taping

`interfaces.h`	→ provides interfaces to the **forward** and **reverse** routines as basic versions of derivative evaluation (see Section 6); • comprises C++, C, and Fortran-callable versions; • includes the header `SPARSE/sparse.h`; • is included by the header `DRIVERS/odedrivers.h`.
`DRIVERS/drivers.h`	→ provides “easy to use” drivers for solving optimization problems and nonlinear equations (see Section 7.1); • comprises C and Fortran-callable versions.
`DRIVERS/odedrivers.h`	→ provides “easy to use” drivers for numerical solution of ordinary differential equations (see Section 7.2); • comprises C++, C, and Fortran-callable versions; • includes the header `interfaces.h`.
`DRIVERS/taylor.h`	→ provides “easy to use” drivers for evaluation of higher order derivative tensors (see Section 7.3) and inverse/implicit function differentiation (see Section 7.4); • comprises C++ and C-callable versions.
`adalloc.h`	→ provides C++ and C functions for allocation of vectors, matrices and three dimensional arrays of **double**s.

Table 5: Important header files: evaluation of derivatives

The intergation of tools for the exploration and the subsequent exploitation of the sparsity structure of Jacobians is subject of the current ADOL-C development. Presently available ADOL-C routines for the exploration of sparsity are prototyped in the header files described in Table 6.

SPARSE/sparse.h	→	provides interfaces to C++-callable versions of **forward** and **reverse** routines propagating bit patterns (see Section 6.3);
	→	provides the "easy to use" driver **jac_pat** for the exploration of the sparsity structure of Jacobians (see Section 7.5);
	•	is included by the header `interfaces.h`.
SPARSE/jacutils.h	→	provides interfaces to the underlying C-callable versions of **forward** and **reverse** routines propagating bit patterns.

Table 6: Header files for exploration of Jacobian sparsity

8.4 Compiling and Linking C/C++ Programs

To compile a C/C++ program or single module using ADOL-C data types and routines one must make sure that all necessary header files according to Section 8.3 are included. All modules involving *active* data types as **adouble**, **adoublev** and **adoublem** have to be compiled as C++. Modules that make use of a previously generated tape to evaluate derivatives can either be programmed in ANSI-C (while avoiding all C++ interfaces) or in C++. Depending on the chosen programming language the header files provide the right ADOL-C prototypes. For linking the resulting object codes the library `libad.a` must be used (see Section 8.1).

8.5 Adding Quadratures as Special Functions

Suppose an integral

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

is evaluated numerically by a user-supplied function

double myintegral(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 active 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 **myintegral** 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 myintegral(double& x)**, and add the line

   extend_quad(myintegral, integrand);

   This macro is extended to the definition of **adouble myintegral(adouble& arg)**. Then remake the library `libad.a` (see Section 8.1).

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

   friend adouble myintegral(adouble&).

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

For example, in case of the inverse hyperbolic cosine, we have **myintegral** = **acosh**. Then **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

```
{ adouble temp = arg;
          temp = temp*temp;
          val = sqrt(temp-1); }
```

The code block **integrand** may call on any elementary function that has already been defined in file `adouble.C`, so that one may also introduce iterated integrals.

9 Example Codes

The following listings are all simplified versions of codes that are contained in the example subdirectory `<ADOLC_DIR>/DEX` of ADOL-C. In particular, we have left out timings, which are included in the complete codes.

9.1 Speelpenning's Example (`speelpenning.C`)

The first example evaluates the gradient and the Hessian of the function

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

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

```
#include "adouble.h"                // use of active doubles and taping
#include "DRIVERS/drivers.h"        // use of "Easy to Use" drivers
                                    // gradient(.) and hessian(.)
...
int main() {
int n,i,j,tape_stats[11];
cout << "SPEELPENNINGS PRODUCT (ADOL-C Documented Example) \n";
cout << "number of independent variables = ?  \n";
cin >> n;
double* xp = new double[n];
double  yp = 0.0;
adouble* x = new adouble[n];        // or: adoublev x(n);
adouble  y = 1;
for(i=0;i<n;i++)
  xp[i] = (i+1.0)/(2.0+i);          // some initialization
trace_on(1);                        // tag =1, keep=0 by default
for(i=0;i<n;i++) {
  x[i] <<= xp[i];                   // or  x<<= xp outside the loop
  y *= x[i]; }                      // end for
y >>= yp;
delete[] x;                         // not needed if x adoublev
trace_off();
tapestats(1,tape_stats);            // reading of tape statistics
cout<<"maxlive "<<tape_stats[2]<<"\n";
                                    // ..... print other tape stats
double* g = new double[n];          // or: doublev g(n);
gradient(1,n,xp,g);                 // gradient evaluation
```

```
double** H=(double**)malloc(n*sizeof(double*));
for(i=0;i<n;i++)
  H[i]=(double*)malloc((i+1)*sizeof(double));
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 += fabs(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 += fabs(H[i][j]-g[i]/xp[j]);
  }                               // end for
}                                 // end for
cout << yp-1/(1.0+n) << " error in function \n";
cout << errg <<" error in gradient \n";
cout << errh <<" consistency check \n";

return 1;
}                                 // end main
```

9.2 Power Example (powexam.C)

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 "adolc.h"                // use of ALL ADOL-C interfaces

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; }                     // end if
else {
  if (n==0)                       // the local adouble z dies
    return z;                     // as it goes out of scope.
```

```
  else
    return 1/power(x,-n); }          // end else
} // end power
```

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

```
#include ...                         // as above
int main() {
int i,n,tag=1;
cout <<"COMPUTATION OF N-TH POWER (ADOL-C Documented Example)\n\n";
cout<<"monomial degree=? \n";        // input the desired degree
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) << " = " << Y[0][i]-value(y)
    << " (should be 0)\n";
  else
    cout << Y[0][i] << " - " << Z[0][i] << " = " << Y[0][i]-Z[0][i]
```

```
    << " (should be 0)\n";
  reverse(tag,1,1,i,u,Z);          // evaluate the (i+1)-st derivative
  Z[0][i+1]=Z[0][i]/(i+1); }       // scale derivative to Taylorcoeff.
return 1;
}                                  // end main
```

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.

9.3 Determinant Example (detexam.C)

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 "adouble.h"               // use of active doubles and taping
#include "interfaces.h"            // use of basic forward/reverse
                                   // interfaces of ADOL-C
adouble** A;                       // A is an n x n matrix
int i,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 = zero;                // zero is predefined
  int s, p =1;
  if (k%2) s = 1; else s = -1;
  for(i=0;i<n;i++) {
    int p1 = 2*p;
    if (m%p1 >= p) {
      if (m == p) {
        if (s>0) t += *pt; else t -= *pt; }
      else {
        if (s>0)
          t += *pt*det(k-1,m-p);   // recursive call to det
```

```
        else
          t -= *pt*det(k-1,m-p); } // recursive call to det
      s = -s;}
    ++pt;
    p = p1;}
  return t; }
}                                   // end det
```

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
int main() {
int i,j, m=1,tag=1,keep=1;
cout << "COMPUTATION OF DETERMINANTS (ADOL-C Documented Example)\n\n";
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];          // not needed for adoublem
    adouble* pt = A[i];
    for(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 = %f  (should be 0)\n",detout,diag,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 (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
return 1;
}                                   // end main
```

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

9.4 Ordinary Differential Equation Example (odexam.C)

Here, we consider a nonlinear ordinary differential equation that is a slight modification of the Robertson test problem given in Hairer and Wanner's book on the numerical solution of ODEs [13]. The following source code computes the corresponding values of $y' \in \mathbb{R}^3$:

```
#include "adouble.h"                // use of active doubles and taping
#include "DRIVERS/odedrivers.h"     // use of "Easy To use" ODE drivers
#inlude "adalloc.h"                 // use of ADOL-C allocation utilities

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);
}                                   // end tracerhs
```

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 main 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
int main() {
int i,j,deg;
int n=3;
double py[3];
double pyp[3];
```

```
cout << "MODIFIED ROBERTSON TEST PROBLEM (ADOL-C Documented Example)\n";
cout << "degree of Taylor series =?\n";
cin >> deg;
double **X;
X=(double**)malloc(n*sizeof(double*));
for(i=0;i<n;i++)
  X[i]=(double*)malloc((deg+1)*sizeof(double));
double*** Z=new double**[n];
double*** B=new double**[n];
short** nz = new short*[n];
for(i=0;i<n;i++) {
  Z[i]=new double*[n];
  B[i]=new double*[n];
  for(j=0;j<n;j++) {
    Z[i][j]=new double[deg];
    B[i][j]=new double[deg]; }     // end for
}                                  // end for
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
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"; }                   // end for
return 1;
}                                  // end main
```

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.

9.5 Gaussian Elimination Example (gaussexam.C)

The following example uses conditional assignments as well as active subscripts to show the usage of a once produced tape for evaluation at new arguments. The elimination is performed with column pivoting.

```
#include "adolc.h"                   // use of ALL ADOL-C interfaces

void gausselim(int n, adoublem& A, adoublev& bv) {
along i;                             // active integer declaration
adoublev temp(n);                    // active vector declaration
adouble r,rj,temps;
int j,k;
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]);
    condassign(i,rj-r,j);            // look for a larger element in the same
    condassign(r,rj-r,rj); }         // column with conditional assignments
  temp = A[i];                       // switch rows using active subscripting
  A[i] = A[k];                       // necessary even if i happens to equal
  A[k] = temp;                       // k during taping
  temps = bv[i];
  bv[i]=bv[k];
  bv[k]=temps;
  if (!value(A[k][k]))               // passive subscripting
    exit(1);                         // matrix singular!
  temps= A[k][k];
  A[k] /= temps;
  bv[k] /= temps;
  for(j=k+1;j<n;j++) {
    temps= A[j][k];
    A[j] -= temps*A[k];              // vector operations
    bv[j] -= temps*bv[k]; }          // endfor
}                                    // end elimination loop
temp=0.0;
for(k=n-1;k>=0;k--)                  // backsubstitution
  temp[k] = (bv[k]-(A[k]*temp))/A[k][k];
bv=temp;
}                                    // 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.

Acknowledgements

Parts of the ADOL-C source were developed by Jay Srinivasan, Chuck Tyner, and Duane Yoder. We are also indebted to George Corliss, Tom Epperly, Bruce Christianson, David Gay, Brad Karp, Koichi Kubota, Bob Olson, Marcela Rosemblun, and Dima Shiriaev for helping in various ways with the development and documentation of ADOL-C. Links to updated versions of the ADOL-C source code and this manual can be found on web page `http://www.math.tu-dresden.de/~adol-c`.

References

[1] 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.

[2] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors. *Computational Differentiation – Techniques, Applications, and Tools.* Philadelphia, 1996. SIAM Proceedings of the Second International Workshop on Computational Differentiation, Santa Fe, New Mexico, February 12–14, 1996.

[3] Christian H. Bischof, Peyvand M. Khademi, Ali Bouaricha and Alan Carle. *Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in dutomatic differentiation*, Optimization Methods and Software, vol.7, 1, 1996, pp. 1-39

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

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

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

[7] Bruce Christianson. Reverse accumulation and accurate rounding error estimates for Taylor series. *Optimization Methods and Software*, 1:81–94, 1992.

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

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

[10] Andreas Griewank and George F. Corliss, editors. *Differentiation of Algorithms: Theory, Implementation, and Application.* Philadelphia, 1991. SIAM Proceedings of the First International Workshop on Computational Differentiation, Breckenridge, Colorado, January 6-8, 1991.

[11] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In [10].

[12] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. To appear in *Mathematics of Computation.*

[13] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II.* Springer-Verlag, Berlin, 1991.

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

[15] Donald E. Knuth. The Art of Computer Programming. Second edition. Addison-Wesley, Reading, 1973.

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

[17] 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.

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

[19] 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.

[20] Bert 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.

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

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

[23] 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.